

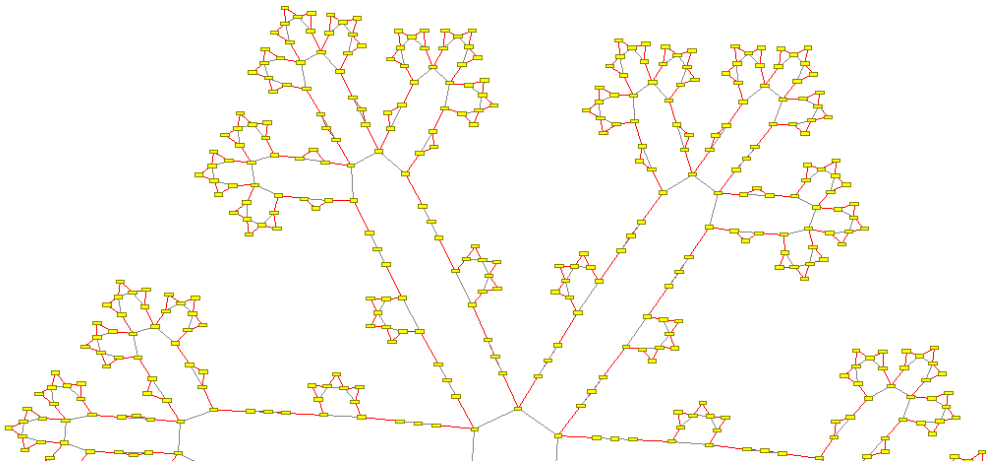
Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Goos

The GrGen.NET User Manual

Refers to GrGen.NET Release 8.6

www.grgen.de / www.grgen.net



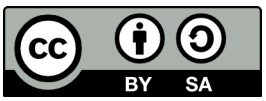
Edgar Jakumeit
Jakob Blomer Rubino Geiß

November 20, 2026

ABSTRACT



GrGen.NET: transformation of structures made easy – with languages for graph modeling, pattern matching and rewriting, as well as rule control; brought to life by a compiler and a rapid prototyping environment offering graphical and stepwise debugging. The Graph Rewrite Generator allows you to develop at the abstraction level of graph representations, leading to applications performing comparably to conventionally developed ones. This user manual contains both, normative statements in the sense of a reference manual as well as an informal guide to the features and usage of GrGen.NET.



This manual is licensed under the terms of the *Creative Commons Attribution-Share Alike 3.0 Germany* license. The license is available at http://creativecommons.org/licenses/by-sa/3.0/de/

FOREWORD FOR RELEASE 1.4

First of all a word about the term "graph rewriting". Some would rather say "graph transformation"; some even think there is a difference between these two. We don't see such differences and use graph rewriting for consistency.

The GrGen project started in spring 2003 with the diploma thesis of Sebastian Hack under supervision of Rubino Geiß. At that time we needed a tool to find patterns in graph based intermediate representations used in compiler construction. We imagined a tool that is fast, expressive, and easy to integrate into our compiler infrastructure. So far Optimix[Ass00] was the only tool that brought together the areas of compiler construction and graph rewriting. However its approach is to feature many provable properties of the system per se, such as termination, confluence of derivations, and complete coverage of graphs. This is achieved by restricting the expressiveness of the whole formalism below Turing-completeness. Our tool GrGen in contrast should be Turing-complete. Thus GrGen.NET provides the user with strong expressiveness but leaves the task of proving such properties to the user.

To get a prototype quickly, we delegated the costly task of subgraph matching to a relational database system [Hac03]. Albeit the performance of this implementation could be improved substantially over the years, we believed that there was more to come. Inspired by the PhD thesis of Heiko Dörr [Dör95] we reimplemented our tool to use search plan driven graph pattern matching of its own. This matching algorithm evolved over time [Sza05, Bat05b, Bat05a, Bat06, BKG08] and has been ported from C to C# [KG07, Kro07]. In the year 2005 Varró [VVF06] independently proposed a similar search plan based approach.

Though we started four years ago to facilitate some compiler construction problems, in the meantime GrGen.NET has grown into a general purpose tool for graph rewriting.

We want to thank all co-workers and students that helped during the design and implementation of GrGen.NET as well as the writing of this manual. Especially we want to thank Dr. Sebastian Hack, G. Veit Batz, Michael Beck, Tom Gelhausen, Moritz Kroll, Dr. Andreas Ludwig, and Dr. Markus Noga. Finally, all this would not happened without the productive atmosphere and the generous support that Prof. Goos provides at his chair.

We wish all readers of the manual—and especially all users of GrGen.NET—a pleasant graph rewrite experience. We hope you enjoy using GrGen.NET as much as we enjoy developing it.

Thank you for using GrGen.NET.

Karlsruhe in July 2007, Rubino Geiß on behalf of the GrGen.NET-Team

FOREWORD FOR RELEASE 4.4

Since the preceding foreword was written a lot has happened: Dr. Rubino Geiß finished his dissertation [Gei08] and left; Prof. Goos retired. The succeeding Professor had no commitment to graph rewriting, so GrGen switched from a university project developed by students and assistants to an open source project (which is still hosted at the IPD, reachable from www.grgen.net).

The first steps of development from version 1.4 onwards, at that time still at Karlsruhe University, were the porting of GrGen.NET from C to C# [Kro07] by Moritz Kroll, which allowed for a faster pace of development, and the addition of support for undirected edges and fine grain pattern conditions [Buc08] by Sebastian Buchwald. Many thanks to them for their work towards the milestone version 2.5 [JBK10] and for their continued support.

Another first step was the addition of alternative patterns and subpatterns by me in the context of my master's thesis, allowing for structural recursion [Jak08, HJG08]. While working on this thesis I fell in love with GrGen.NET. It did several things in just the most elegant way possible – but it had weak spots regarding not-statically-fixed patterns as well as programmability and extensibility. People in love do foolish things: after graduating I devoted much of my free time of the upcoming years to continue developing, in order to close the gaps. I was dragged along by the feeling that graph rewriting is an underrated programming paradigm, just lacking a general-purpose language and a development environment advanced enough to materialize that vision. (Besides programming language design is a highly interesting activity.) With version 4.4 of GrGen.NET this is accomplished.

We want to thank the organizers of GraBaTs[RVG08]/TTC, the annual meeting of the graph rewrite tool community, which gave us the possibility to ruthlessly steal the best ideas of the competing tools. Thanks to Berthold Hoffmann, and the discussions in "French". And many thanks to several early users giving valuable feedback, a lot of bug reports, and even code (which is of course the best contribution you can give to an open source project), by name: Bugra Derre, Paul Bedaride, Normen Müller, Pieter van Gorp and Nicholas Tung. Many thanks to all the others that contributed since then. If *you* want to contribute or if you have a question, don't hesitate to contact the GrGen.NET-Team via email to `grgen` at the host given by `grgen.de`.

The rule-and-pattern-based transformation of graph-based representations is a perfect fit for knowledge-intensive tasks, for tasks built on models where the relations play a central part, and for linguistic applications. A much better fit than predicate logic or the lambda calculus typically misused for modelling them, or relational algebra that funnily fails when the relations become the important part, or tedious traditional programming. Go and build a great application (e.g. an AI that enslaves humanity :) with GrGen.NET for them.

We wish you a pleasant graph rewriting experience.

Karlsruhe in July 2014, Edgar Jakumeit on behalf of the GrGen.NET-Team

FOREWORD

Again a lot has happened since the preceeding foreword was written, the university domain `uni-karlsruhe.de` does not exist anymore, thus `info.uni-karlsruhe.de/software/grgen` has gone for good, GrGen.NET is now hosted at `www.grgen.de` and `https://github.com/ejaku/grgen`, at the time of GrGen.NET version 7.0 you only find yComp still at IPD Snelting (see `https://pp.ipd.kit.edu/firm/yComp.html`).

GrGen.NET was extended with database-like graph query result processing (based on pattern matches), quite a lot of general purpose programming advancements, and now features a new builtin graph viewer/debugger based on the MSAGL library[NPDH07] you could use inside your own .NET-application – unpolished as of version 7.0, but the same holds for this foreword that requires still some refinement ;).

As of 8.0 it's still somewhat unpolished, but an SQLite[GPB+22]-based persistent graph is available as of now in exchange.

The system found quite some applications in computational design synthesis, luckily with fewer bug reports as beforehand.

Are you ready for takeoff with GrGen.NET?

Karlsruhe in December 2025, Edgar Jakumeit on behalf of the GrGen.NET-Team

CONTENTS

1 Introduction 1

1.1 What Is GrGen.NET? . . . 1
1.2 When to Use GrGen.NET . . . 1
1.3 An Example Graph-Based Representation . . . 1
1.4 When Not to Use GrGen.NET . . . 4
1.5 What Is Graph Rewriting? . . . 4
1.6 An Example for Rewriting . . . 5
1.7 An Example Rule Application on the Representation . . . 6
1.8 Why to use GrGen.NET . . . 8

2 Overview 9

2.1 System Overview . . . 9
2.2 The Tools . . . 10
2.2.1 `GrGen.exe` . . . 11
2.2.2 `GrShell.exe` and `GGrShell.exe` . . . 11
2.2.3 `yComp.jar` and `MSAGL` . . . 11
2.2.4 Several Dynamic Link Libraries that can be used at API level . . . 11
2.3 The Languages and their Features . . . 12
2.3.1 Graph Model Language . . . 12
2.3.2 Rule and Computations Language . . . 12
2.3.3 Rule Application Control Language . . . 14
2.3.4 Shell Language . . . 15

3 Quickstart 17

3.1 Downloading & Installing . . . 17
3.2 Creating a Graph Model . . . 18
3.3 Creating Graphs . . . 18
3.4 The Rewrite Rules . . . 20
3.5 Applying the Rules . . . 22
3.6 Debugging and Output . . . 22

4 Graph Model Language 25

4.1 Building Blocks . . . 26
4.1.1 Base Types . . . 27
4.2 Type Declarations . . . 27
4.2.1 Node and Edge Types . . . 28
4.2.2 Attributes and Attribute Types . . . 32
4.2.3 Enumeration Types . . . 33
4.2.4 Hints on Modeling . . . 34

5 Rule Language 37
5.1 Building Blocks . . . 38
5.1.1 Graphlets . . . 38
5.2 Rules and Tests . . . 41
5.3 Pattern Part . . . 45
5.3.1 Isomorphic and Homomorphic Matching . . . 48
5.4 Rewrite Part with Replace and Modify Modes . . . 49
5.4.1 Implicit Definition of the Preservation Morphism r . . . 49
5.4.2 Specification Modes for Graph Transformation . . . 50
5.4.3 Syntax . . . 51

6 Basic Types and Attribute Evaluation Expressions 55
6.1 Built-In Types . . . 55
6.2 Expressions . . . 56
6.3 Boolean Expressions . . . 57
6.4 Relational Expressions . . . 57
6.5 Arithmetic and Bitwise Expressions . . . 59
6.6 String Expressions . . . 60
6.7 Type Expressions . . . 62
6.8 Primary Expressions . . . 63
6.9 Operator Priorities . . . 68

7 Nested Patterns 69
7.1 Negative Application Condition (NAC) . . . 70
7.2 Positive Application Condition (PAC) . . . 72
7.3 Pattern Cardinality (iterated / multiple / optional) . . . 73
7.4 Alternative Patterns . . . 75
7.5 Named nested patterns . . . 79

8 Subpatterns 81
8.1 Subpattern Declaration . . . 81
8.2 Subpattern Entity Declaration . . . 82
8.2.1 Recursive Patterns . . . 82
8.3 Regular Expression Syntax . . . 86
8.4 Pattern Element Locking . . . 87

9 Nested and Subpattern Rewriting 89
9.1 Nested Pattern Rewriting . . . 89
9.2 Subpattern Rewriting . . . 91
9.2.1 Deletion and Preservation of Subpatterns . . . 94

10 Nested and Subpattern Yielding 97
10.1 Yielding Outwards After Pattern Matching . . . 97
10.1.1 LHS def variables . . . 98
10.1.2 Yielding and Iterated Filtering . . . 98
10.2 Yielding Outwards During Rewriting . . . 104
10.2.1 RHS def variables . . . 104
10.2.2 Yielding . . . 105
10.3 Flow Example . . . 106
10.4 Ordered Evaluation . . . 109

11 Rule Application Control Language (Sequences) 113
11.1 Rule Application . . . 114
11.2 Logical and Sequential Connectives . . . 116
11.3 Simple Variable Handling . . . 117
11.4 Decisions and Loops . . . 119
11.5 Quick reference table . . . 121

12 Advanced Matching and Rewriting 123
12.1 Rule and Pattern Modifiers . . . 124
12.2 Static Type Constraint . . . 125
12.3 Exact Dynamic Type . . . 127
12.4 Retyping . . . 128
12.5 Copy and Clone . . . 129
12.6 Node Merging . . . 130
12.7 Edge Redirection . . . 131
12.8 Attribute Initialization . . . 132

13 Embedded Sequences and Textual Output/Input 133
13.1 Exec and Emit in Rules . . . 133
13.2 Deferred Exec and Emithere in Nested and Subpatterns . . . 135
13.3 Scan and Tryscan . . . 137

14 Computations (Attribute Evaluation Statements) 139
14.1 Assignments . . . 140
14.2 Local Variable Declarations . . . 141
14.3 Control flow . . . 141
14.4 Embedded Exec . . . 145
14.5 Computation Definition and Call . . . 145
14.5.1 Function Definition and Call (and Quick Reference Tables) . . . 145
14.5.2 Procedure Definition And Call (and Quick Reference Tables) . . . 147
14.6 The Big Picture . . . 153

15 Container Types and Computations 159
15.1 Built-In Types and Concept of Containers . . . 159
15.2 Set Operations . . . 160
15.3 Map Operations . . . 163
15.4 Array Operations . . . 166
15.5 Deque Operations . . . 170
15.6 Storage Access in the Rules . . . 174
15.7 Hints on container usage . . . 176

16 Graph Type and Computations 179
16.1 Built-In Types . . . 179
16.2 Graph Functions And Procedures . . . 179
16.2.1 Graph Updates / Basic Graph Manipulation . . . 179
16.2.2 Graph Query by Types . . . 180
16.2.3 Graph Query by Neighbourhood . . . 181
16.2.4 Subtle Points in the Semantics . . . 188
16.3 Subgraph Operations . . . 189
16.4 File Operations . . . 190

16.5 Graph comparison . . . 192
16.6 Visited Flags. . . . 195
16.7 Graph Processing Environment Procedures. . . . 197
16.7.1 Transaction Handling . . . 197
16.7.2 Assertions . . . 198
16.7.3 Misc. Global Procedures . . . 198

17 Filtering and Sorting of Matches 201

17.1 Match types . . . 201
17.2 Filter Functions . . . 202
17.3 Auto-Generated Filters . . . 203
17.4 Auto-Supplied Filters . . . 205
17.5 Filter Calls . . . 205
17.6 Lambda Expression Filter Calls . . . 207
17.7 Quick Reference Table . . . 208

18 Advanced Modelling (Object-Oriented and Graph-Oriented Programming) 211

18.1 Object Types and Transient Object Types . . . 212
18.1.1 Object Types . . . 212
18.1.2 Transient Object Types (preliminary) . . . 213
18.1.3 Object Creation . . . 214
18.1.3.1 new Operator . . . 214
18.1.3.2 Structure examples . . . 215
18.1.3.3 Copy and Clone functions . . . 217
18.1.3.4 Equality comparison . . . 218
18.2 Methods . . . 219
18.2.1 Function Method Definition and Call . . . 219
18.2.2 Procedure Method Definition And Call . . . 219
18.3 Packages . . . 224
18.3.1 Package Definition in the Model . . . 225
18.3.2 Package Definition in the Actions . . . 226
18.4 Graph References / Graph Nesting / Hierarchical Graphs . . . 228

19 Sequence Computations and Container Type 231

19.1 Sequence Statements . . . 231
19.2 Sequence Expression. . . . 235
19.3 Storages – Container Type Handling in the Sequences . . . 239
19.4 Graph and Subgraph Based Queries and Updates . . . 242
19.5 Quick Reference Table . . . 244

20 Graph Queries 247

20.1 Advanced Array Operations (Querying Support) . . . 248
20.1.1 Array Accumulation Methods . . . 248
20.1.2 Array By-Member Methods . . . 249
20.1.3 Array Per-Element Methods with Lambda Expressions . . . 250
20.2 Pattern Matching Based Queries . . . 251
20.3 Query Result Combination (Auto-generated functions) (preliminary) . . . 256
20.4 Query Looping and Counting Rule Execution . . . 260

21 Multi Rule Application and Match Classes 261

21.1 Rule Prefixed Sequence . . . 261
21.2 Multi Rule Application . . . 261
21.3 Indeterministic Choice . . . 262
21.4 Match Classes . . . 264
21.5 Match Class Filter Calls . . . 265
21.6 Quick Reference Table . . . 267

22 Advanced Control with Backtracking 269

22.1 Sequence Definitions (Procedural Abstraction) . . . 269
22.2 Transactions, Backtracking, and Pause Insertions . . . 270
22.3 Multi-Rule Backtracking . . . 273
22.4 For Loops . . . 274
22.5 Mapping Clause . . . 275
22.6 Persistence Provider Transactions . . . 275
22.7 Quick Reference Table . . . 276

23 Parallelization 279

23.1 Internal parallelization . . . 279
23.2 Sequence parallelization (preliminary) . . . 280
23.3 Locking (preliminary) . . . 282
23.4 Quick Reference Table . . . 284

24 Transformation Techniques 285

24.1 Merge and Split Nodes . . . 286
24.2 Node Replacement Grammars . . . 290
24.3 Mapping in a Rewriting Tool . . . 292
24.4 Comparing Structures . . . 294
24.5 Copying Structures . . . 295
24.5.1 Built-In Graph Copying . . . 297
24.6 Data Flow Analysis for Computing Reachability . . . 297
24.7 State Space Enumeration . . . 302

25 GrShell Language 305

25.1 Building Blocks . . . 306
25.2 Common Commands . . . 308
25.3 Graph and Graph Element Creation . . . 309
25.4 Sequence Execution and Profiles . . . 312
25.5 Variables . . . 313
25.6 Attribute Assignment and Graph Manipulation . . . 314
25.7 Inclusion and Conditional Execution . . . 316
25.8 File System Commands and External Shell Execution . . . 316
25.9 Shell and Environment Configuration . . . 317
25.10 Compilation Configuration . . . 318
25.11 Backend, Graph, and Actions Selection . . . 319
25.12 LGSPBackend Custom Commands . . . 320

26 Persistent Storage 323
26.1 Persistent Graph Creation . . . 323
26.1.1 Model Update . . . 325
26.2 Graph Input and Output . . . 327
26.3 Graph Change Recording and Replaying . . . 331
26.4 Model Migration . . . 332

27 Visualization and Debugging 335
27.1 Graph Visualization Commands (Nested Layout) . . . 335
27.2 yComp Usage . . . 345
27.3 MSAGL and MSAGLExt Usage . . . 345
27.4 extMSAGLExt Usage . . . 349
27.5 Debugging Related Commands . . . 350
27.6 Using the Debugger . . . 352
27.7 Subrule Debugging and Programmed Halts . . . 356
27.8 Watchpoint configuration . . . 357
27.9 Debugging related functionality . . . 359

28 Validation and Inspection 361
28.1 Model and Graph Queries . . . 362
28.2 Validation Commands . . . 363
28.3 Queries with Graphical Output . . . 365

29 Components and Applications 367
29.1 `GrGen.exe` . . . 367
29.2 `GrShell.exe` . . . 369
29.3 `GGrShell.exe` . . . 370
29.4 `libGrShell.dll` . . . 370
29.5 `graphViewerAndSequenceDebugger.dll` . . . 370
29.6 `graphViewerAndSequenceDebuggerWindowsForms.dll` . . . 370
29.7 `lgspBackend.dll` . . . 371
29.8 `libGrPersistenceProviderSQLite.dll` . . . 371
29.9 `libGr.dll` . . . 371
29.10 `libConsoleAndOS.dll` . . . 371
29.11 `libConsoleAndOSWindowsForms.dll` . . . 371
29.12 MSAGL assemblies/libraries . . . 371
29.13 SQLite assemblies/libraries . . . 372
29.14 `extMSAGLExt.exe` . . . 372
29.15 `yComp.jar` . . . 372
29.16 Birds eye view . . . 373

30 Indices 377
30.1 Type Indices . . . 377
30.2 Neighbourhood Indices . . . 377
30.3 The Costs . . . 378
30.4 Attribute Indices . . . 379
30.5 Incidence Count Indices . . . 392
30.6 Name Index . . . 394
30.7 Uniqueness Index . . . 396

31 Performance Optimization 399
31.1 Search Plans . . . 399
31.2 Find, Don't Search . . . 401
31.2.1 Consequences For Optimization . . . 401
31.2.2 Search Planning On Request . . . 404
31.2.3 Consequences For Optimization, II . . . 405
31.3 Location Passing and Memorization . . . 406
31.4 Profile and Parallelize . . . 408
31.5 Compilation and Static Knowledge . . . 409
31.6 Miscellaneous Things . . . 409

32 Examples 419
32.1 Fractals . . . 419
32.2 Busy Beaver . . . 421
32.2.1 Graph Model . . . 421
32.2.2 Rule Set . . . 421
32.2.3 Rule Execution with GrShell . . . 423

33 Application Programming Interface 427
33.1 Interface to the Host Graph . . . 428
33.2 Interface to the Rules . . . 430
33.3 Interface of the Graph Processing Environment . . . 433
33.4 Import/Export and Miscellaneous Stuff . . . 434
33.5 Graph Viewer and Debugger . . . 438
33.6 Shell functionality . . . 439
33.7 Persistent Graph and Persistence Provider . . . 439
33.8 External Class, Function and Procedure Implementation . . . 440
33.9 External Filter and Sequence Implementation . . . 441
33.10 Graph Events . . . 442
33.11 Action Events . . . 444

34 Extensions 447
34.1 External Attribute Types . . . 447
34.2 External Function Types . . . 448
34.3 External Procedure Types . . . 448
34.4 External Filter Functions . . . 449
34.5 External Sequences . . . 449
34.6 External Emitting and Parsing . . . 450
34.7 External Cloning and Comparison . . . 450
34.8 Shell Commands . . . 450
34.9 Shell and Compiler Parameters . . . 451
34.10 Annotations . . . 451

35 Understanding and Extending GrGen.NET 453
35.1 How to Build . . . 453
35.2 How to Test . . . 455
35.3 The Generated Code . . . 455
35.4 Search Planning in Code Generation . . . 468
35.5 The Code Generator . . . 471
35.5.1 Frontend . . . 471

35.5.2 Backend 476
35.6 Further Backend Parts and Applications 479

36 Development Goals and Design Decisions 483

36.1 Expressiveness 483
36.2 General-Purpose Graph Rewriting 484
36.3 Performance 486
36.4 Understandability and Learnability 487
36.5 Development Convenience 487
36.6 Well Founded Semantics 488
36.7 Platform Independence 488
36.8 General-Purpose Graph Transformation 489

Bibliography 491

Index 498

CHAPTER 1

INTRODUCTION

1.1 What Is GrGen.NET?

GrGen (Graph Rewrite GENerator) is a software *development tool* that offers *programming languages* optimized for graph-structured data, with declarative *pattern matching* and *rewriting* at their core, with support for imperative and object-oriented programming, and database-like query-result processing. The generative programming system featuring a *graphical debugger* eases the *transformation* of *graph-based representations*, as they are needed in e.g. engineering, model transformation, computer linguistics, or compiler construction.

1.2 When to Use GrGen.NET

You may be interested in using GrGen.NET if you have to tackle the task of changing meshes of linked objects, i.e. *graph-like data structures*; anytime the focus is on the *relationship* in between your data entities, and esp. if your algorithms operate upon data entities that stand in multiple relations to each other.

These tasks are traditionally handled by pointer structures and pointer structure navigation-, search-, and replacement routines written by hand – this low-level, pointer-fiddling code can be generated automatically by GrGen.NET for you. You specify your transformation task on a *higher level of abstraction* with nodes connected by edges, and rewrite *rules* consisting of *patterns* to be searched as well as modifications to be carried out. GrGen.NET then generates the algorithmic core of your application. You specify the *what* with *declarative rules*, GrGen.NET takes care of the *how*; similar to SQL expressions implemented by database engines, or to grammar rules implemented by parser generators. Development is supported by *visual debugging*, you work on a visualization of your network and the rule applications inside it. The debugger and the pattern based languages boost your *productivity* for graph-representation-based tasks way beyond traditional programming. Due to all the optimizations implemented in the pattern matching engine you still gain *high-performance* solutions. Altogether, GrGen.NET offers the highest combined speed of development and execution you can find for those kinds of tasks.

1.3 An Example Graph-Based Representation

Let's have a look at an example of a graph-based representation GrGen.NET is well suited for – a simplified version of the graph-based compiler intermediate representation Firm[1] that GrGen was originally developed for. Nodes represent machine instructions, e.g. a Load fetches a value from an address or an Add sums two values. Edges indicate dependencies between nodes, e.g. an Add requires that its two operands are available before it can be executed. Each instruction is located within a basic block, all nodes within the Block are executed if the Block is executed (in an order allowed by the dependencies).

Figure 1.1 shows the program graph for the following C-function:

[1] www.libfirm.org

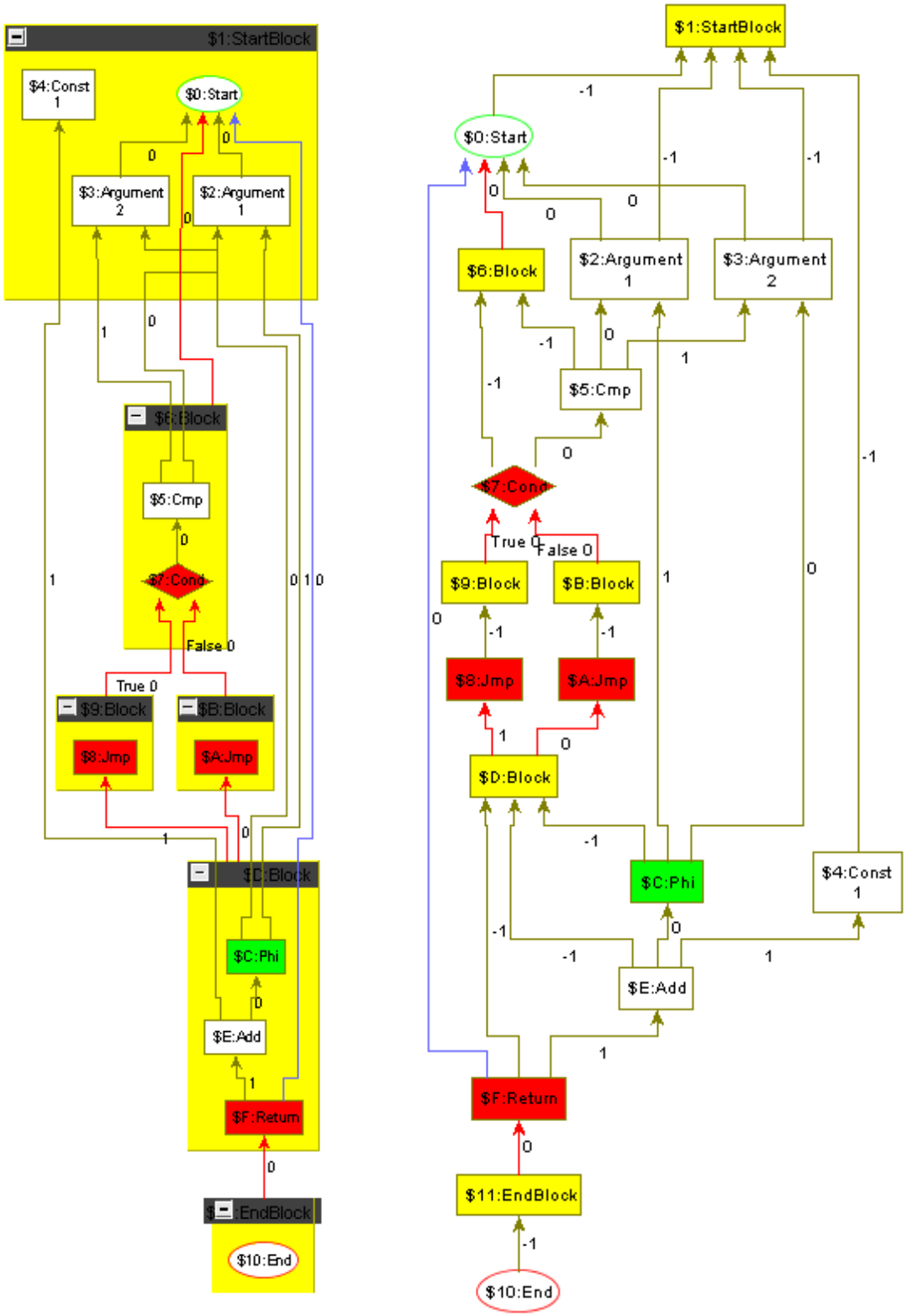


Figure 1.1: Program graph of a minimum plus one function, with block containment visualized as containment left, and the plain graph right.

```
int MinPlus(int x, int y)
{
    int min;

    if (x < y) {
        min = x;
    } else {
        min = y;
    }

    return min + 1;
}
```

Program execution begins at the Start node, which also produces the initial memory state and the given Arguments. The Start and the Arguments belong to the special StartBlock. The Cmp compares the arguments, its result is used by the Cond to carry out a conditional jump depending on the result of the comparison. After following the Jmp of the then- or the else-Block, the program execution is continued at Block $D. The Phi[2] chooses one of its operands depending on the previously executed block, here Argument 1 if the Cond was evaluated to True, or Argument 2 if Cond was evaluated to False. The Constant 1 is added to the value selected by the Phi, before the result is returned by the Return. The end of the program execution is represented by the special EndBlock which by convention contains exactly one End. If you want to learn more about this representation or typical transformations over it, have a look at the compiler case[BJ11a] of the TTC 2011 where this representation was used or our solution [BJ11c].

The key points here are that

1. this is a mesh of objects (denoting operations),
2. that stand to each other in 3 relations (control flow (red), data flow (brown), and memory flow (blue) [3]),
3. and a major part of the information is encoded in the connection structure (topology), which is explicitly and directly represented with edges.

You find such representations in other domains than *programming languages* and *compilers*, too. The *knowledge* of an agent about the external world can be *represented* intuitively like this with *semantic nets*, you could use GrGen.NET for reasoning over them. *Models* as understood by *model driven engineering* are typically graphs, with GrGen.NET you can implement a model transformation between models, or into a lower-level program representation. When working in *computational linguistics* you may be interested in modeling an *abstract syntax graph* in GrGen.NET; the recursive and iterated patterns that allow to process tree-like data structures declaratively render GrGen.NET an excellent match for such tasks. But also in *engineering* or *architecture* where you need to describe the buildup of the modeled system from components, here you can derive context-sensitively system *blueprints* or simulate their behaviour. You find help not only in modelling them, but also in deriving all interesting configurations or simulating all interesting execution traces, in the form of built-in support for a backtracking search through a search space or even the enumeration of a state space with isomorphic state pruning, guided by match ordering and filtering.

[2] a helper node from the static single assignment form employed; in [BBH+13] you find more information and a simple algorithm for constructing SSA from an abstract syntax tree

[3] we use dependencies, i.e. the direction is reversed compared to the flow

1.4 When Not to Use GrGen.NET

There is nothing to gain from GrGen.NET if scalars, lists or arrays are sufficient to model your domain, which is the case for a lot of tasks in computing indeed. (But which is not the case for others which would be better modeled with trees and especially graphs, but aren't because of the cost of maintaining pointer structures by hand.) GrGen.NET is likely too heavyweight if you are just interested in computing shortest paths in simple graphs. You're better off with a traditional graph library and its set of pre-implemented algorithms – the library can be learned quicker and the algorithms can be reused directly. GrGen.NET with its domain-specific languages in contrast requires time to learn and some effort to specify or code a solution in (only afterwards are you rewarded with a much higher pace of development).

The graph rewrite generator is not the right tool for you if you're searching for a visual environment to teach children programming – it's a powerful tool for (software) engineers. Simpler and less powerful languages like story diagrams[FNTZ00] that can be learned quicker and understood more intuitively may be a better match here. Neither is it what you need if your graph-structured data is to be interactively edited by an end user instead of being automatically transformed by rules, patterns, and algorithms (the editor generator DiaGen[Dia] may be of interest in the former case).

Each object-oriented program contains a heap of objects pointing to each other that can be understood as a graph-like representation, with objects being the nodes and the references being the (fixed) edges. In contrast to object-oriented programming, the topology in graph-oriented programming as offered by GrGen.NET is not hidden and encapsulated in the objects, but globally open for inspection and modification; open and readily accessible for pattern matching and rewriting. Prefer the closed heaps if you don't need a global view on the data, they are cheaper and the information hiding cuts dependencies.

1.5 What Is Graph Rewriting?

The notion of graph rewriting as understood by GrGen.NET is a method for declaratively specifying "changes" to a graph. This is comparable to term rewriting (and thus partly to functional programming). Normally you use one or more *graph rewrite rules* to accomplish a certain task. GrGen.NET implements an SPO-based approach (as default). In the simplest case such a graph rewrite rule consists of a tuple $L \rightarrow R$, where L—the *left-hand side* of the rule—is called *pattern graph* and R—the *right-hand side* of the rule—is the *rewrite graph*.

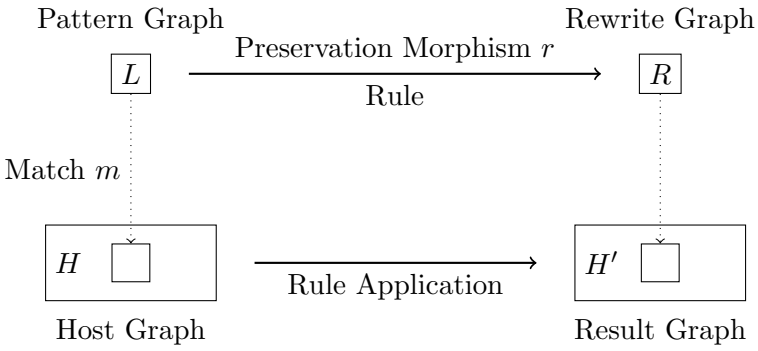


Figure 1.2: Basic Idea of Graph Rewriting

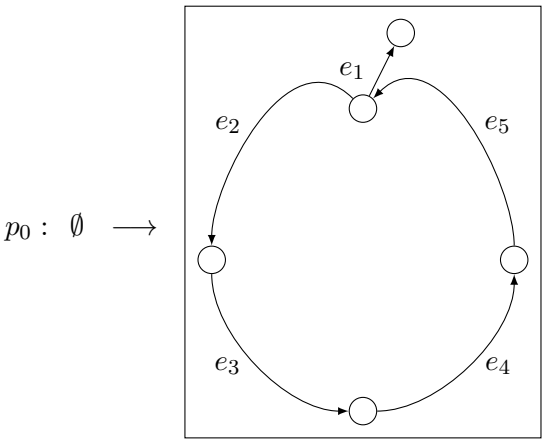
Moreover we need to identify graph elements (nodes or edges) of L and R for preserving them during rewrite. This is done by a *preservation morphism* r mapping elements from L to R; the morphism r is injective, but needs to be neither surjective nor total. Together with a rule name p we have $p : L \xrightarrow{r} R$.

The transformation is done by *application* of a rule to a *host graph* H. To do so, we have to find an occurrence of the pattern graph in the host graph. Mathematically speaking, such

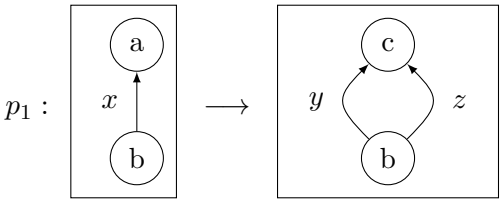
a *match* m is an isomorphism from L to a subgraph of H. This morphism may not be unique, i.e. there may be several matches. Afterwards we change the matched spot $m(L)$ of the host graph, such that it becomes an isomorphic subgraph of the rewrite graph R. Elements of L not mapped by r are deleted from $m(L)$ during rewrite. Elements of R not in the image of r are inserted into H, all others (elements that are mapped by r) are retained. The outcome of these steps is the resulting graph H'. In symbolic language: $H \overset{m,p}{\Longrightarrow} H'$.

1.6 An Example for Rewriting

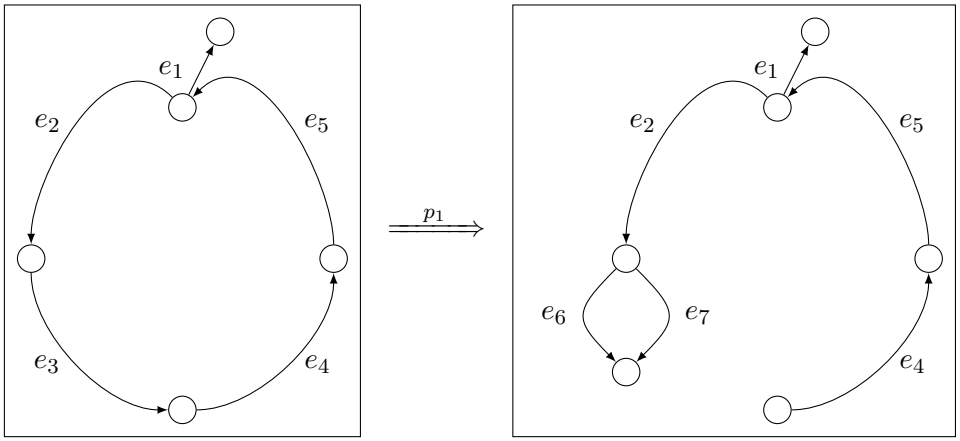
We'll have a look at a small example. Graph elements (nodes and edges) are labeled with an identifier. If a type is necessary then it is stated after a colon. We start using a special case to construct our host graph: an empty pattern always produces exactly one[4] match (independent of the host graph). So we construct an apple by applying



to the empty host graph. As the result we get an apple as new host graph H. Now we want to rewrite our apple with stem to an apple with a leaflet. So we apply



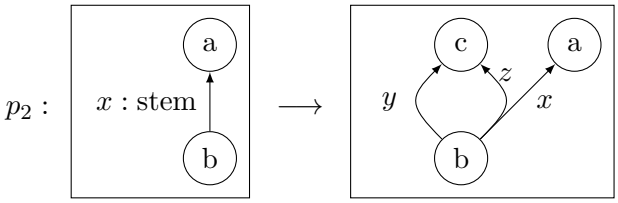
to H and get the new host graph H_1, something like this:



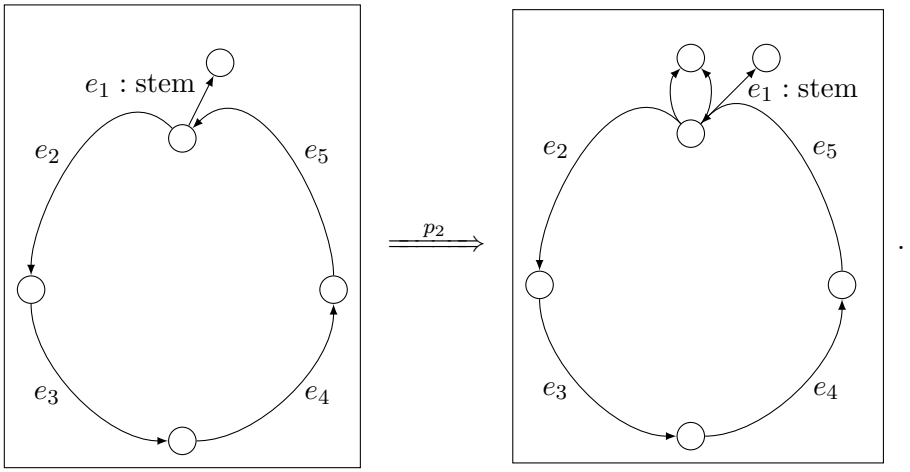
What happened? GrGen.NET has arbitrarily chosen one match out of the set of possible matches, because x matches edge e_3 as well as e_1. A correct solution could make use of edge

[4] Because of the uniqueness of the total and totally undefined morphism.

type information. We have to change rule p_0 to generate the edge e_1 with a special type "stem". And this time we will even keep the stem. So let



If we apply p_2 to the modified H_1 this leads to



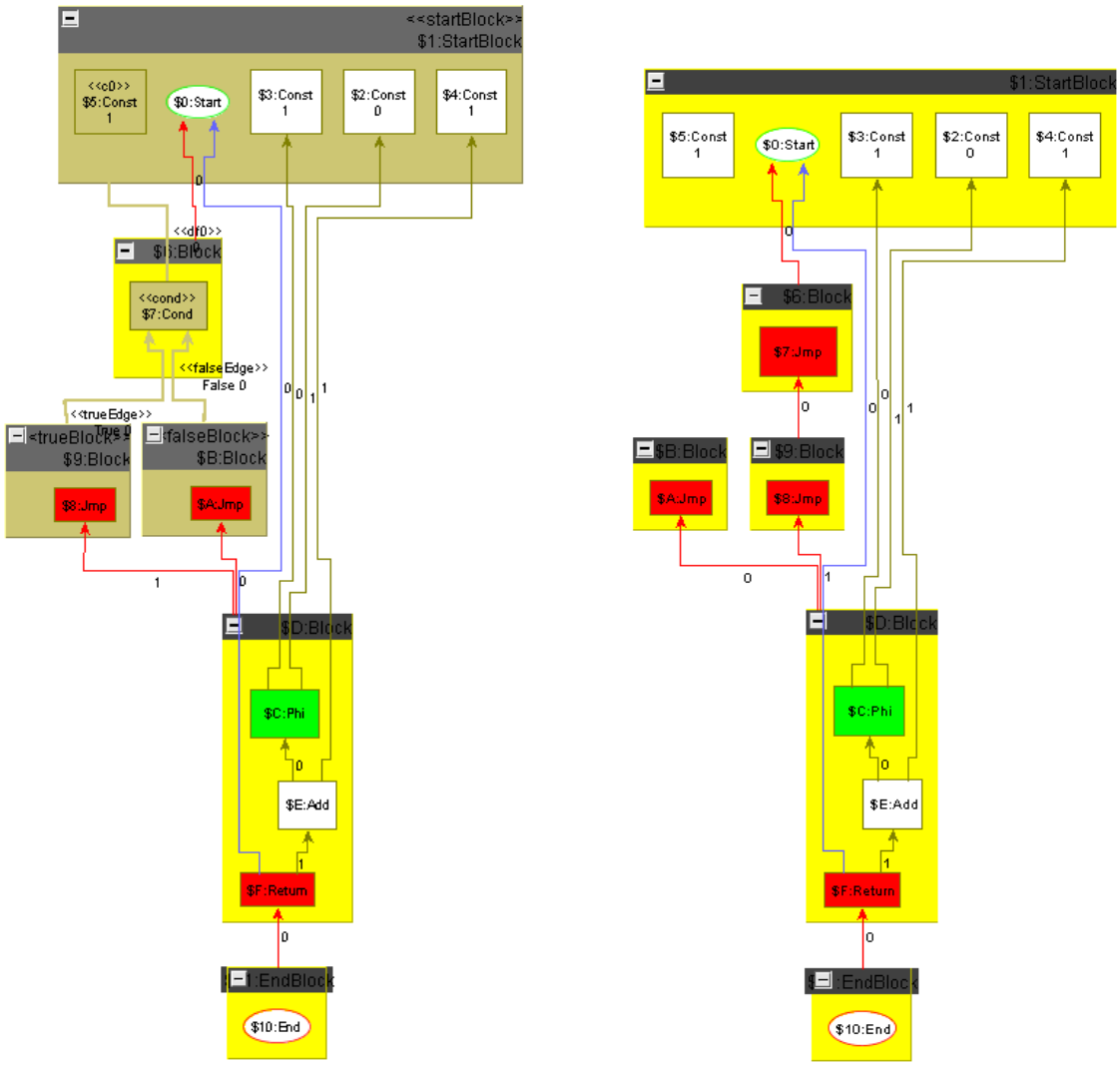
1.7 An Example Rule Application on the Representation

An interesting transformation that can be applied to the compiler intermediate representation introduced in section 1.3 is constant folding. Figure 1.3 highlights the effect of applying the following example rule by showing the situation before it is applied and afterwards:

```
1  rule foldCond {
2     cond:Cond -df0:Dataflow-> c0:Const;
3     falseBlock:Block -falseEdge:False-> cond;
4     trueBlock:Block -trueEdge:True-> cond;
5     alternative {
6        TrueCond {
7           if { c0.value == 1; }
8           modify {
9              delete(falseEdge);
10             -jmpEdge:Controlflow<trueEdge>->;
11          }
12       }
13       FalseCond {
14          if { c0.value == 0; }
15          modify {
16             delete(trueEdge);
17             -jmpEdge:Controlflow<falseEdge>->;
18          }
19       }
20    }
21    modify {
22       delete(df0);
23       jmp:Jmp<cond>;
24    }
25 }
```

The rule `foldCond` is used to replace a conditional jump by an unconditional jump, iff it depends just on a constant value. To this end, the `cond` is retyped to `Jmp`, and the dependency on the constant value `c0` is `deleted`. The syntax `name:type` declares a node of the given type. An edge is declared with the same syntax, just inscribed into an edge `-->` representation. An `<original>` suffix denotes retyping, specifying the original element to be retyped. The pattern matches furtheron the blocks where execution continues: `trueBlock` and `falseBlock`. In the true case (`if { c0.value == 1; }`), the `falseEdge` to the `falseBlock` is deleted and the `trueEdge` to the `trueBlock` retyped to a normal `Controlflow` edge; in the false case, the opposite is carried out.

Here the true case applies as you can see from the value `1` inscribed in the matched constant `$5`. The elements from the graph that were bound to the pattern elements are highlighted in light-brown, with their pattern name suffixed in double angles, e.g. the `cond` was matched to `$7` . Blocks that become unreachable (without control flow predecessors) are assumed to get deleted in a later step of the transformation, as are duplicate constants. The benefits of using GrGen.NET can be already felt from this example, they increase at an accelerating speed as the patterns (and rewrites) grow in size and complexity. Note that the rule is *modular*, i.e. independent of other functionality — traditionally, functionality is smeared over multiple graph representation traversal passes, and multiple steps of a pass.



(a) Before folding the Cond, match highlighted.

(b) After folding the Cond.

Figure 1.3: Situation reached during folding the program graph of the minimum plus one function when applied to constant arguments (rendered by the debugger).

1.8 Why to use GrGen.NET

GrGen.NET offers *processing* of *graph representations* at their *natural level of abstraction*.

Graph representations are typically employed in engineering (blueprints, designs), model transformation (models), computer linguistics (syntax graphs, semantic nets), or compiler construction (intermediate representations, program graphs) – but a graph is maybe the most natural representation of the data of your domain, too?

GrGen.NET is built on a *rich and efficient metamodel* implementing multi-graphs, with multiple inheritance on node and edge types. The nodes and edges are wired in *scalable* ringlists, which give access to their incident elements in constant time, to all elements of a type in constant time, and allow to add and delete elements in constant time.

GrGen.NET features *modular rules* that don't smear functionality into traversal code as is the case in traditional pointer structure passes. It offers declarative *graph patterns* of high expressiveness, saving you a lot of boilerplate code that would be needed for coding them manually. And it implements efficient graph change rollback, thus allowing you to easily crawl search spaces without your own bookkeeping (for undoing the changes).

GrGen.NET contains a C#-like programming language, so you don't run against walls in case of subtasks where the rules and patterns don't work well. The *general-purpose* system is highly extensible and customizable, you can express solutions fitting well to your task at hand.

GrGen.NET offers the convenience of *dedicated languages* with well-readable syntax and static type checking, instead of clunky internal DSLs, limited annotations, or annoying XML. Its languages are brought to life by an *optimizing compiler* that prunes unneeded generality and generates pattern matchers adapted to the characteristics of the host graph at hand.

The system ships with a readily available shell application for file mapping tasks, which especially offers step-wise and *visual debugging*, saving you from chasing reference chains in the debugger of your programming language. Further mature development support is offered with search plan explanation and profiling instrumentation. Moreover, GrGen.NET is properly documented in an extensive *user manual*.

The underlying framework supports automatic persistence and automatic visualization of network-like data (based on event-sourcing - besides, also built-in serialization / deserialization is offered). A runtime library is supplied (in the form of several DLLs) that implements an easy-to-use API to these features, allowing you to build your own .NET (or native) applications based on it.

Traditional programming of graph representation processing is tedious, it requires careful orchestration of functionality attached to passes, maintenance of helper state, boilerplate code for patterns, and low-level pointer fiddling. You are more *productive* with GrGen.NET in one-shot-coding a solution, but especially so during a continuous development process or when changes are needed afterwards, a GrGen.NET-specification can be *adapted* at a much higher pace.

Altogether, GrGen.NET offers the *highest combined speed of development and execution* for graph representation processing you can find.

CHAPTER 2

OVERVIEW

GrGen.NET is combined from two groups of components: The first consists of the compiler `grgen` and some runtime libraries, which offer the basic functionality of the system; the compiler transforms specifications of declarative graph rewrite rules into highly efficient .NET-assemblies. The second consists of the interactive command line `(G)GrShell` and the graph viewers `yComp` and MSAGL with some further runtime libraries, which offer a rapid prototyping environment supporting graphical and stepwise debugging of rules that are controlled by sequences.

2.1 System Overview

Figure 2.1 gives an overview of the GrGen.NET system components and the involved files.

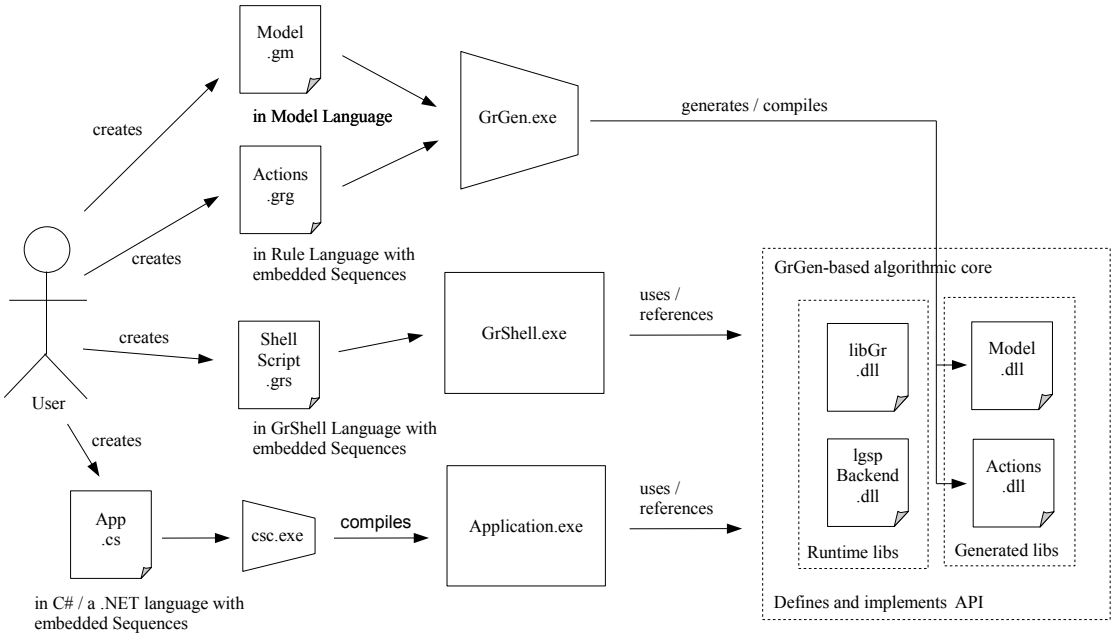


Figure 2.1: GrGen.NET system components and artifacts

A graph rewrite system[1] is *defined* by a rule set file (`*.grg`, written in the rule and computations language), which may include further rule set files, and use zero or more graph model description files (`*.gm`, written in the model language). Executable code is *generated* from these specifications by `GrGen.exe`; the generated parts are assisted by the *runtime libraries* `libGr` and `lgspBackend`. Together they form the system that can be used via

[1] In this context, system is less a grammar rewrite system, but rather a set of interacting software components.

an *API* by arbitrary .NET-applications, for the processing of a graph-based representation. One application employing the API is shipped with GrGen.NET, the *shell* application GrShell. It may be used interactively, esp. for debugging, or may be scripted (in the shell language, which allows via `libGr` to execute sequences, a concise language offered to control rule applications).

Figure 2.2 depicts the major runtime objects.

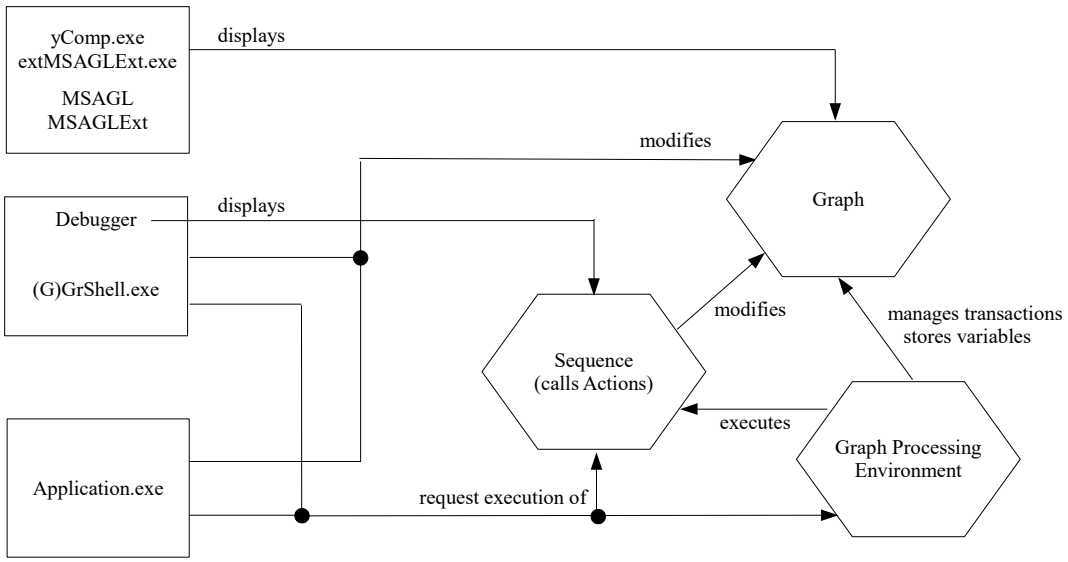


Figure 2.2: GrGen.NET major runtime objects

The central object is the *graph*, it adheres to the specified graph model. (In general you have to distinguish between a graph model on the meta level, a host graph created as an instance of the graph model, and a statically specified pattern graph of a rule that matches a portion of the host graph at runtime). It may be modified directly by GrShell, or by a user application. But it is typically changed by the application of one of the generated *actions*, most often from within a *sequence*, which is parsed into an operator tree and then interpreted, by the graph *processing environment*. The processing environment maintains the global variables in addition, and modifies the graph directly in case of transaction rollback (based on the graph change events recorded since transaction start).

The GrShell is a readily available execution environment that is sufficient for mapping tasks that require only console or file output. But even when you want to integrate a graph-based algorithmic core into your existing .NET application, or are forced to write one because you must supply a long-running application reacting to events from the environment, esp. to user input, you may be interested in additionally using the GrShell because of its debugging abilities. Employing the graph viewer yComp(or alternatively MSAGL) it can visualize the graph at any time during execution. Moreover, it can visualize the application of a rule, highlighting the matched pattern in the host graph, and the changes carried out. It can do so for actions executed from a sequence, which can be executed step-by-step, under debugger control, highlighting the currently executed rule in the sequence.

2.2 The Tools

Here we give a short introduction to the tools. You find a more detailed explanation in 29, also explaining the components/assemblies.

The programs and libraries of GrGen.NET are open source licensed under LGPL[Fou07].

The graph viewer yComp is closed source and ships with its own license granted by yFiles for academic use. The MSAGL graph viewer library comes with a way more permissive MIT license. The persistent graph is implemented utilizing the SQLite database, which is in the public domain.

Executing a generated graph rewrite system requires .NET Framework[Mic07] version 4.7.2 or mono[Tea07] version 5.10, or later (maybe complete packaging); the same holds for compiling it, building GrGen.NET itself and debugging a graph rewrite system with yComp in addition requires JAVA[Ora96] version 1.8 or later. You find the tools in the `bin` subdirectory of your GrGen.NET installation.

2.2.1 `GrGen.exe`



The `GrGen.exe` assembly implements the GrGen.NET generator. The GrGen.NET generator parses a rule set and its model files and compiles them into .NET assemblies. The compiled assemblies form a specific graph rewriting system together with the GrGen.NET backend.

It is typically very helpful during development, to use the `-keep` and `-debug` compiler options, you find more on them in Chapter 29.

2.2.2 `GrShell.exe` and `GGrShell.exe`



The `GrShell.exe` and the `GGrShell.exe` version are shell applications on top of the libGr. They are capable of creating, manipulating, and dumping graphs as well as performing graph rewriting with graphical debug support. For further information about the GrShell language see Chapter 25.

The GrShell is a console application, while the GGrShell is a WindowsForms application, using a rich text box console control.

2.2.3 `yComp.jar` and `MSAGL`

yComp [KBG+07] is a graph visualization tool based on yFiles [yWo07]. It is well integrated and shipped with GrGen.NET, but it's not a part of GrGen.NET; it is supplied by IPD Snelting, the successor institute of IPD Goos.yComp implements several graph layout algorithms and has file format support for VCG, GML and YGF among others.

You may use the internal MSAGL graph viewer or its extended version MSAGLExt instead, which come as a .NET component. Alternatively, you may use the external graph viewer application extMSAGLExt. They are a more natural fit for a .NET application, and can be freely included in your own .NET applications (to visualize the graph(s) for the user). For more on yComp, see Section 29.15, for more on MSAGL and MSAGLExt, see Section 27.3, and for more on extMSAGLExt, see Section 27.4.

2.2.4 Several Dynamic Link Libraries that can be used at API level

Several DLLs are used in implementing the functionality of the system, that is offered by the GrShell or the GGrShell in the form of an application. The base system functionality by `libGr.dll`, `lgspBackend.dll`, and `libConsoleAndOS.dll`, plus `libConsoleAndOSWindowsForms.dll` for the GUI extension (defining the base runtime environment). Persistence support is realized by a persistence provider `libGrPersistenceProviderSQLite.dll`. Graph display and sequence debugging are implemented by the `graphViewerAndSequenceDebugger.dll`, and its `graphViewerAndSequenceDebuggerWindowsForms.dll` extension. The shell functionality itself is contained in the `libGrShell.dll`. These assemblies are also available on API level, they can be or have to be included in order to use their corresponding functionalities – besides the model and actions DLLs generated by the compiler.

2.3 The Languages and their Features

The process of graph rewriting at runtime can be divided into four steps: Creating an instance graph according to a model, searching a pattern aka finding a match, performing changes to the matched spot in the host graph, and, finally, selecting which rule(s) to apply where next.

This process is programmed at specification time utilizing a graph model language, a rule and computations language, and the sequences language for controlling rule applications.

2.3.1 Graph Model Language

At the foundation, you find the graph model (meta-model) language, consisting of a series of `class` definitions in the style of an object-oriented programming language, which shows some influence of data definition languages as offered by databases.

It allows you to specify *node* and *edge types*, with *multiple inheritance* on the types (see Chapter 4). They are used in building a typed, directed multigraph (supporting multiple edges of the same type between two nodes). In addition, you may declare undirected and arbitrarily directed edges; or may even register external types with GrGen.NET. Furthermore, connection assertions allow you to restrict the "shape" of the graphs.

Node and edge types can be equipped with *typed attributes*, of the commonly known elementary types (integer and floating-point numbers, strings, booleans, and enums) or of some container types (sets and maps based on hashmaps, as well as arrays and deques based on a field of contiguous memory cells). In addition, methods supporting dynamic dispatch may be given.

Moreover, you may configure indices (see Chapter 31): attribute indices for a quick lookup of a graph element based on the attribute value, or incidence count indices for a quick lookup based on the number of incident edges. Several indices are already built-in and automatically used by the pattern matchers, notably the type index for quick lookups of graph elements based on element type, and the incident elements index giving access to the neighbouring elements in constant time (also available are a name-based and an unique-id based index).

2.3.2 Rule and Computations Language

On top of the graph model language and at the heart of GrGen.NET, you find the rule and computations language.

It supports pattern-based *rules*, which are built by a *pattern to be matched* and a *rewrite* to be carried out; or put differently: a left-hand-side precondition pattern and a right-hand-side postcondition pattern (see Chapter 5). A pattern is described by *graphlets*, a specification of nodes connected by edges in an intuitive, easily writable, easily readable, and *easily changeable* syntax. Complex patterns are combined from elementary patterns with *nested patterns* (defining alternative, optional, multiple and iterated structures, or negative and independent application conditions, see Chapter 7) and *subpattern* definitions and usages (see Chapter 8). Besides those parts for *declarative* processing of structural information, *attribute conditions* (see Chapter 6) and *attribute evaluations* are available, the former are expressions computing an output value by reading attributes of matched elements, the latter are statements changing attribute values with assignments.

The attribute processing expressions and statements are in fact parts of a full-fledged *imperative programming language* integrated with the declarative pattern-based rule language (see Chapter 14). The *expressions* can query the graph with a multitude of available functions in addition to reading the attributes of matched elements and carrying out computations with the attributes; they compute a value and can be abstracted into *functions*. The *statements* support direct graph changes and supply control flow statements and def variable declarations in addition to the attribute assignments; they change the state and can be abstracted into

procedures. You especially find subgraph operations for the processing of nested graphs, with subgraph extraction, subgraph comparison, and subgraph insertion (see Chapter 16).

Pattern matching can be adjusted fine-grained with homomorphic matching for selected elements (with `hom` declarations, so they can match the same graph elements), overriding the default isomorphic matching, and with type constraints (forbidding certain static types or requiring equivalence of dynamic `typeof`s).

The rewrite part of a rule keeps, adds and deletes graph elements according to the SPO approach, as specified by element declarations and references in the LHS and RHS patterns (there are three additional rule application semantics available: DPO or exact patterns only or induced subgraphs only). The rewrite part supports two modes of specification: A rule can either express the changes to the match (`modify`-mode, requiring deletion to be specified explicitly) or specify a whole new subgraph (`replace`-mode).

A rich bouquet of single element rewrite operations is available on top of the basic ones with *retyping* aka relabeling, that allows to change the type of a graph element while keeping its incident elements, with the creation of new nodes/edges of only dynamically known types or as exact copies of other nodes/edges, and with node merging and edge redirection constructs (see Chapter 12). Besides changing the graph, you can `emit` user-defined text to `stdout` or files (supporting model-to-text transformations; see Chapter 13).

A special class of computations is available with the post-match filter functions (see Chapter 17), that allow to inspect and manipulate the array of matches found when applying a rule (with all bracketing). This way you can *accumulate* information over the matches, or *filter* the matches array for the most interesting matches. Several filters for sorting and clipping are already supplied or can be generated automatically on request. Symmetry reduction is available with a filter for duplicate matches due to permuted elements from automorphic patterns.

The notion of pure graph rewriting (graph grammars) comes without processing state – the entire state is contained in the graph, which is declaratively queried by rule patterns. In their simple form, GrGen.NET rules follow the idea of pure graph rewriting, and are *modular* or *local*: you just note down the functionality. This limits the changes needed during development and maintenance, and saves you together with the concise and easily editable patterns from many large-scale tedious modifications that are normally needed in those cases. You are still free to combine rules and tests (tests are rules with only a LHS pattern) utilizing processing state in the form of *parameters*: both support input and output parameters (the same holds for subpatterns, as well as functions and procedures). They are typically used to define and advance the location of processing in the graph, but they are also used for attribute computations. In addition visited flags can be queried and written, for marking already processed elements.

Subpatterns allow for pattern reuse, and allow via subpattern *recursion* to match substructures extending into *depth* (e.g. iterated paths – but for just checking an iterated path condition, i.e. the transitive closure of a relation, there are predefined and even more efficient functions available that can be simply called.) Iterated patterns allow to match substructures splitting into *breadth* (e.g. multinodes). Both combined allow to match *entire tree-like structures* within a single rule application. Subpatterns and nested patterns are matched from the outermost to the innermost patterns with *recursive descent*, handing already matched elements down; on *ascend*, they can `yield` elements found out to the `def` elements of their containing pattern. The nested patterns and subpatterns also support *nested rewrite parts*, so that complex structures are not only matched (parsing), but can also get rewritten (transduction) — yielding *structure directed transformation*, alongside structures defined by graphlets combined in an EBNF-grammar-like way[Jak11].

Those features and their optimized implementation – offering e.g. pattern matcher parallelization with a simple annotation – crown GrGen.NET the king of graph pattern matching and rewriting, and allow you to process graph representations at their natural level of abstrac-

tion, with a flexibility and performance coming near to what you can achieve by programming things by hand.

A graph rewrite system described in these languages can be employed via an *API* by .NET-applications, typically written in C# (see Chapter 33). The other way round you can include C#-code into your specifications by using external types or calling external functions and procedures, as well as sequences (see Chapter 34).

2.3.3 Rule Application Control Language

On top of the rules and computations language do you find the *sequences* language, for controlling the application of the rules (and procedures or functions), determining *which rule* to apply *where* next.

This strategy language allows to determine the rule that is called next by its control operators and constructs (see Chapter 11). The basic ones available are the *sequential* and *logical operators*, the *decisions*, and the *loops* (defining the control-flow orchestration). The location where to apply a rule may be defined with *parameters* handed into the called rules, and received back from them. The sequences support *variables* to store the locations (for data-flow orchestration). A rule may be applied for the *first match* found, or for *all matches* that are available with all-bracketing.

The sequences offer a *computations* and *expressions* sublanguage, for one to carry out small computation tasks in a rule application control scenario without having to descend to the rules, but esp. and for the other to *directly query* the graph with pattern-based queries (see Chapter 20), whose results get then refined by filters, distilled by array accumulation methods, and combined by operators.

The sequence computations support nearly all operations of the computations language. This includes esp. function and procedure calls, basic graph queries and modifications, extraction, insertion and comparison of *subgraphs*, visited flags management, and storage manipulation (see Chapter 19). Visited flags used in marking processed elements must be allocated and freed, the sequence computations are the right place to do so. Many usages are related to *storages*, which are variables of container type storing graph elements, they esp. allow to build transformations following a wavefront running over the graph (see Chapter 15 for more on containers).

The sequences can be abstracted into a *sequence definition* that can then be called in the place of a rule. This way common parts can be reused, but it is especially possible to program a recursive strategy. For simulation runs, several indeterministic choice operators are available.

Advanced constructs are available with *transactions* angles and *backtracking* double angles capable of *rolling back changes* carried out on the graph to get back to the state of the graph again from before their nested content was applied (see Chapter 22). With them you can easily try things out without having to invest into programming the bookkeeping needed to revert to an old state. They allow to systematically run through a search space, or even to enumerate a state space, materializing interesting states reached in time out into space.

Another less stateful way of enumerating a state space is offered by the *mapping clause*, that yields target graphs created by applying a rule on a source graph, to be stored in a top-level graph, or a data structure built from internal *object classes*. This more functional way of enumerating a state space can be sped up by applying the *parallel execution* construct of the sequences.

Those sequences are available in an integrated form in the rule language (see Chapter 13). After applying the direct effects of a rule it is possible to apply an *embedded sequence* for follow-up tasks that just can't be expressed with a single rule application; those sequences have access to the elements of their containing rule, they allow to build complex transformations with rule-sequence-rule call chains.

2.3.4 Shell Language

Those were the features of the languages at the core of the GrGen.NET-System (implemented by the generator `grgen.exe` and the runtime libraries `libGr` and `lgspBackend`). In addition, the GrGen.NET system supplies a shell application, the GrShell, which offers besides some common commands, variable handling constructs, and file system commands several constructs for *graph handling* and *visualization*, as well as constructs for *sequence execution* and *debugging*. Furthermore, some switches are available to configure the shell and the graph processing environment, as well as the `grgen` compiler.

Regarding graph handling you find commands for graph creation and manipulation, graph export and import, and graph change recording plus replaying (see Chapter 25). For graph creation 3 simple `new` commands are available, one for creating an empty graph, one for creating a node, and one for creating an edge (between two nodes). The GRS exporter serializes with the same syntax, which is also expected by the GRS importer. Write a file in this simple format if the available import formats are not supported by your data source.

For direct graph manipulation, you find deletion and retyping commands in addition to the creation commands, and commands for assigning attributes of graph elements. Moreover, `export` and `import` commands are available for serializing a graph to a file, and for unserializing a graph from a file (in `GRS`, `GXL` and `XMI/ecore` formats). Changes that are occurring to a graph may be `record`ed to a `GRS` and later `replay`ed or simply `import`ed. Change recording allows for post-problem debugging, but allows also to use GrGen.NET as a kind of embedded database that persists changes as they appear.

The graph and the model can be queried furtheron for e.g. the defined types (utilizing a reflection-like interface) or the attributes of a graph element. The graph may be validated against the connection assertions specified in the model, or against a sequence that gets executed on a clone of the host graph.

Regarding sequence execution, you find the central `exec` command that is *interpreting* the following sequence (in fact by delegating the parsing and execution to `libGr`), and the `eval` command that is interpreting the following sequence expression (which allows to employ pattern-based queries). In addition, action `profile`s may be `show`n. They are available when *profiling* instrumentation was switched on, telling about the number of search steps or graph accesses carried out.

The actions can be replaced at runtime by loading another actions library; the backend could be, but currently there is only one available, the `lgspBackend`. This backend supports further custom commands, that start with a `custom graph` or `custom actions` prefix. They allow to `analyze` the graph in order to compile some statistics about the characteristics of the graph that are of relevance for the pattern matcher. And in the following to regenerate the pattern matchers at runtime (`gen_searchplans`) given the knowledge about the characteristics of the graph, yielding matchers which are *adapted to the host graph* and thus typically faster than the default pattern matchers generated statically into the blue. You can inspect the search plans in use with the `explain` custom command.

The debugger component of the GrShell allows to `debug` a sequence `exec`ution step-by-step, highlighting the currently executed rule in the sequence, and displaying the current graph in the graph viewer yComp; in detail mode even the match found and the rewrite carried out by the rule are highlighted in the graph (see Chapter 27).

The graph visualization is highly *customizable*. You can choose one from several available layout algorithms. You can define e.g. the colors, the shapes of nodes, or the linestyle of edges, based on the type of the graph elements, or exclude uninteresting types altogether from the display. You can pick the attributes that are to be shown directly (normally they are only displayed on mouse-over). Furthermore, you can configure *visual graph nesting* by declaring edges as containment edges, thus keeping large graphs with containment or tree relations *understandable*. For large graphs that are beyond the capabilities of the graph viewer you can define that only the matches plus their context up to a certain depth are

displayed. Altogether, you can tailor the graph display to what you need, transforming an unreadable haystack which is what large graphs tend to become (if not rendered utilizing the means from above) into a well-readable representation.

CHAPTER 3

QUICKSTART

In this chapter we'll build a graph rewrite system from scratch. We will use GrGen.NET to construct non-deterministic state machines, and to remove ε-transitions from them. This chapter gives a quick tour of GrGen.NET and the process of using it; it esp. highlights its look and feel. For comprehensive specifications, please take a look at the succeeding chapters.

3.1 Downloading & Installing

If you are reading this document, you probably have already downloaded the GrGen.NET software from our website (https://www.grgen.de or http://www.grgen.net). Make sure you have the following system requirements installed and available in the search path:

- Mono 5.10 or above / Microsoft .NET Framework 4.7.2 or above
- Java 1.8 or above for yComp (not needed when you use MSAGL instead)

If you're using Linux: Unpack the package to a directory of your choice, for example into `/opt/grgen`:

```
mkdir /opt/grgen
tar xvfj GrGenNET-V1_3_1-2007-12-06.tar.bz2
mv GrGenNET-V1_3_1-2007-12-06/* /opt/grgen/
rmdir GrGenNET-V1_3_1-2007-12-06
```

Add the `/opt/grgen/bin` directory to your search paths, for instance if you use `bash` add a line to your `/home/.profile` file.

```
export PATH=/opt/grgen/bin:$PATH
```

Furthermore we create a directory for our GrGen.NET data, for instance by `mkdir /home/grgen`.

If you're using Microsoft Windows: Extract the .zip archive to a directory of your choice and add the `bin` subdirectory to your search path via *control panel* → *system properties* / environment variables. Execute the GrGen.NET assemblies from a command line window (*Start* → *Run...* → `cmd`). For MS .NET the `mono` prefix is neither applicable nor needed.

NOTE (1)

You might be interested in the syntax highlighting specifications of the GrGen.NET-languages supplied for the vim, Emacs, and Notepad++ editors in the `syntaxhighlighting` subdirectory.

3.2 Creating a Graph Model

In the directory `/home/grgen` we create a text file `StateMachine.gm` that contains the graph meta model for our state machine[1]. By graph meta model we mean a set of node types and edge types which are available for building state machine graphs (see Chapter 4). Figure 3.1 shows the meta model.

```
node class State {
    id: int;
}

abstract node class SpecialState extends State;
node class StartState extends SpecialState;
node class FinalState extends SpecialState;
node class StartFinalState extends StartState, FinalState;

edge class Transition {
    Trigger: string;
}

const edge class EpsilonTransition extends Transition;
```

Figure 3.1: Meta Model for State Machines

What have we done? We specified two base types, `State` for state nodes and `Transition` for transition edges that will connect the state nodes. `State` has an integer attribute `id`, and `Transition` has a string attribute `Trigger` which indicates the character sequence for switching from the source state node to the destination state node. The other types inherit from those base types (keyword `extends`). Inheritance works basically like inheritance in object-oriented languages, as does the `abstract` modifier for `SpecialState`, meaning that you cannot create a node of that precise type (only nodes of non-abstract subtypes). With `StartFinalState` we have constructed a "diamond" type hierarchy, highlighting the support for multiple inheritance.

3.3 Creating Graphs

Let's test our graph meta model by creating a state machine graph. We will use the GrShell (see Chapter 25) and—for visualization—yComp. To get everything working we need a rule set file, too. For the moment we just create an almost empty file `removeEpsilons.grg` in the `/home/grgen` directory, containing only the line

```
#using "StateMachine.gm"
```

Now, we could start by launching the GrShell and by typing the commands interactively. However, in most cases this is not the preferred way, as we want to execute the same steps multiple times. So instead, we create a GrShell script `removeEpsilons.grs`, in the `/home/grgen` directory. Figure 3.2 shows this script. Run the script by executing `grshell removeEpsilons.grs`. The first time you execute the script, it might take a while because GrGen.NET has to compile the meta model and the rule set into .NET assemblies.

The graph viewer yComp opens and after clicking the blue play "layout graph" button on the very right side of the button bar, you get a window similar to figure 3.3 (see also Section 29.15). Quit yComp and exit the GrShell by typing `exit`.

[1] You'll find the source code of this quick start example shipped with the GrGen.NET package in the `examples/FiniteStateMachine/` directory.

```
1  new graph removeEpsilons "StateMachineGraph"
2
3  new :StartState($=S, id=0)
4  new :FinalState($=F, id=3)
5  new :State($="1", id=1)
6  new :State($="2", id=2)
7  new @(S)-:Transition(Trigger="a")-> @("1")
8  new @("1")-:Transition(Trigger="b")-> @("2")
9  new @("2")-:Transition(Trigger="c")-> @(F)
10 new @(S)-:EpsilonTransition-> @("2")
11 new @("1")-:EpsilonTransition-> @(F)
12 new @(S)-:EpsilonTransition-> @(F)
13
14 show graph ycomp
```

Figure 3.2: Constructing a state machine graph in GrShell

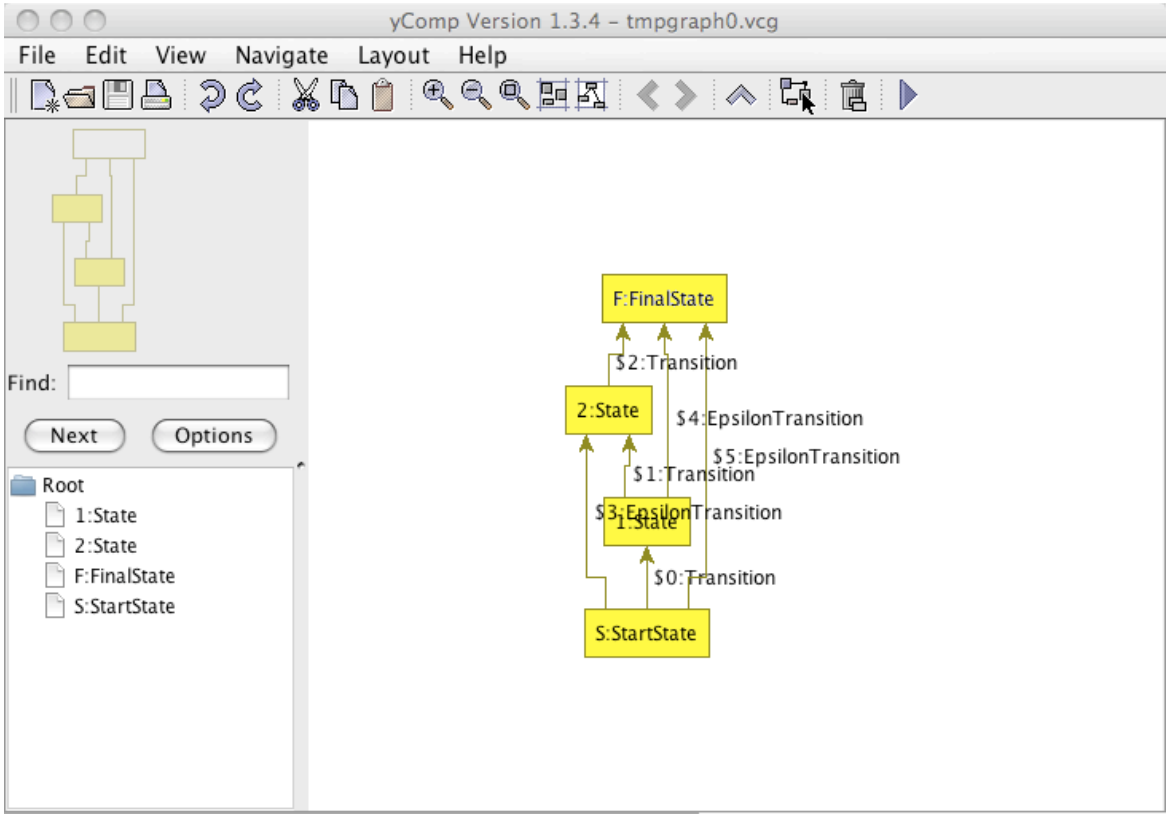


Figure 3.3: A first state machine (with ε-transitions)

Our script first creates an empty graph of the meta model `StateMachine` (which is referenced by the rule set `removeEpsilons.grg`) with the name `StateMachineGraph`. Thereafter, it creates the nodes and edges, with a `name:type` colon notation – the names are optional and missing here. Note the in-place arrow notation for edges (`-Edge->` resp. `-:EdgeType->`). As you can see, attributes of graph elements can be set during creation with a call-like syntax. The `$` and `@` notation is due to the usage of *persistent names*. Persistent names are set on creation by `$=Identifier` and used later on by `@(Identifier)` to retrieve the graph element bound to them. In addition to persistent names, another kind of "names" is available in the GrShell with the *global variables*, which would appear before the colon, but we did not use them in this script. Persistent names must be identifier strings and not numbers, that's why we have to use the quote chars around `"1"` and `"2"`.

3.4 The Rewrite Rules

We will now add the real rewrite rules to the rule set file `removeEpsilons.grg`. The idea is to "forward" normal transitions over ε-transitions one after another, i.e. if we have a pattern like `a:State -:EpsilonTransition-> b:State -ne:Transition-> c:State` we add another `a -ne-> c`. After all transitions have been forwarded we can remove the ε-transitions altogether. The complete rule set is shown in figure 3.4. See Chapter 5 for the rule set language reference.

Let's inspect the rule set specification in detail: The rule set file consists of a number of `rule`s and `test`s, each of them bearing a name, like `forwardTransition`. Rules contain a pattern specified by semicolon-terminated graphlets, and a rewrite part given with a nested modify or replace block. Tests contain only a pattern; they are used to check for a certain pattern without doing any rewrite operations. If a rule is applied, GrGen.NET tries to find the pattern within the host graph, for instance within the graph we created in Section 3.3. Of course there could be several matches for a pattern—GrGen.NET will choose one of them arbitrarily (technically the first match in an implementation defined order).

Figure 3.4 also displays the syntax `name:NodeType` for nodes and `-name:EdgeType->` for edges, which we have already seen in Section 3.3, again with an optional name. But here the meaning of a *declaration* `name:Type` depends on whether it appears in the pattern or the rewrite part. A declaration in the pattern defines a pattern element that is to be bound by searching, resp. matching in the host graph, whereas a declaration in the rewrite part defines an element that is to be created (so only the latter does the same as its counterpart from the shell script). An element with a name assigned to can be *referenced* from another graphlet, or from the rewrite block, by noting down simply the name for a node, or using the syntax `-name->` for an edge (a name can be referenced in the same or nested blocks). If you want to "do something" with your specified graph element "in another location" (which may mean simply linking a node to another edge), define a name; otherwise an anonymous graph element will work fine (like the `:FinalState` or `-:EpsilonTransition->` declarations that will be bound to a node of type `FinalState` resp. an edge of type `EpsilonTransition`, but are inaccessible for other operations). Also have a look at example 7 on page 42 for additional pattern specifications.

The rewrite part can be given in one of two modes, `modify` mode or `replace` mode. In replace mode, every graph element of the pattern which is not referenced within the `replace` block is deleted. The modify mode, in contrast, deletes nothing (by default), but just adds or keeps graph elements. Instead, the modify part allows for *explicit* deletion of graph elements by using a `delete` command inside the `modify` block.

What else do we use? We employ `negative` patterns; they prevent their containing rule from getting applied if the negative pattern is found. We also use attribute conditions noted down inside an `if {...}` to express non-structural constraints; a rule is only applicable if all such conditions are evaluated to true. Attributes of graph elements are accessed with dot

```
#using "StateMachine.gm"

test checkStartState {
    x:StartState;
    negative {
        x;
        y:StartState;
    }
}

test checkDoublettes {
    negative {
        x:State -e:Transition-> y:State;
        hom(x,y);
        x -doublette:Transition-> y;
        if {typeof(doublette) == typeof(e);}
        if { ((typeof(e) == EpsilonTransition) || (e.Trigger == doublette.Trigger)); }
    }
}

rule forwardTransition {
    x:State -:EpsilonTransition-> y:State -e:Transition-> z:State;
    hom(x,y,z);
    negative {
        x -exists:Transition-> z;
        if {typeof(exists) == typeof(e);}
        if { ((typeof(e) == EpsilonTransition) || (e.Trigger == exists.Trigger)); }
    }
    modify {
        x -forward:typeof(e)-> z;
        eval {forward.Trigger = e.Trigger;}
    }
}

rule addStartFinalState {
    x:StartState -:EpsilonTransition-> :FinalState;
    modify {
        y:StartFinalState<x>;
    ---
        emit("Start␣state␣(", x.id, ")␣mutated␣into␣a␣start-and-final␣state");
    }
}

rule addFinalState {
    x:State -:EpsilonTransition-> :FinalState;
    if {typeof(x) < SpecialState;}
    modify {
        y:FinalState<x>;
    }
}

rule removeEpsilonTransition {
    -:EpsilonTransition->;
    replace {}
}
```

Figure 3.4: Rule set for removing ε-transitions

notation (as in `e.Trigger`). The `hom(x,y)` and `hom(x,y,z)` statements mean "match the embraced nodes homomorphically", i.e. they can (but don't have to) be matched to the same node within the host graph.

The `eval {...}` statement in the rewrite part is used to evaluate or recalculate attributes of graph elements. The `emit` statement prints a string to `stdout`. Have a look at the statement `y:StartFinalState<x>` in `addStartFinalState`: we *retype* the node `x`. That means that the newly created node `y` can be found at the position of `x` in the graph, that it bears its persistent name, and that the attributes common to the new and the previous type keep their old values – only the node type is changed or cast to `StartFinalState`.

3.5 Applying the Rules

The created rewrite rules are transformation primitives, we must compose them in order to implement more complex functionality. For instance we don't want to forward just one ε-transition as `forwardTransition` would do; we want to forward them all. Such a rule composition is called a *graph rewrite sequence* (see Chapter 11). We add the following line to our shell script `removeEpsilons.grs`:

```
1 debug exec checkStartState && !checkDoublettes && (forwardTransition* | addStartFinalState |
    addFinalState* | removeEpsilonTransition* | true)
```

This looks like a boolean expression and in fact it behaves similarly. The whole expression is evaluated from left to right. A rule is successfully evaluated if a match could be found (and is rewritten, but this won't change the execution result). The negation operator `!` just toggles the result. We first check for a valid state machine, i.e. if the host graph contains exactly one start state and no redundant transitions. Thereafter we do the actual rewriting. These three steps are connected by lazy-evaluation-ands (`&&`), i.e. if one of them fails the evaluation will be canceled. We continue with disjunctively connected rules (by `|`). The eager operator causes all rules to get executed, but only one must find a match to count as a success; the final `true` ensures overall success. The `*` is used to apply the rule repeatedly as long as a match can be found. This includes applying the rule zero times. Even in this case `Rule*` is still successful.

3.6 Debugging and Output

If you execute the modified GrShell script, GrGen.NET starts its debugger. This way you can follow the execution of the graph rewrite sequence step by step in yComp. Just play around with the keys `d`, `s`, and `r` in GrShell: the `d` key lets you follow a single rule application in a matching and a rewriting step, highlighting the found spot in the graph; the `s` key jumps to the next rule; and the `r` key runs to the end of the graph rewrite sequence. After sequence execution, you should see a graph like the one in Figure 3.5.

If everything is working fine you can delete the `debug` keyword in front of the `exec` and just (re)use this sequence as you need to. Maybe you want to save a visualization of the resulting graph. This is possible by typing `dump graph mygraph.vcg` in GrShell, which is then writing `mygraph.vcg` into the current directory in the VCG format readable by yComp. Or you want to save the resulting graph as such, to continue processing later on. Say `export mygraph.grs` in GrShell then, it will write a file composed of `new` commands as is our input file from Figure 3.2, but containing the ε-free graph.

Feel free to browse the `examples` folder shipped with GrGen.NET; have a look at the succession of examples in the `FiniteStateMachine` and `ProgramGraphs` subfolders, they give an overview over most of the capabilities of the software.

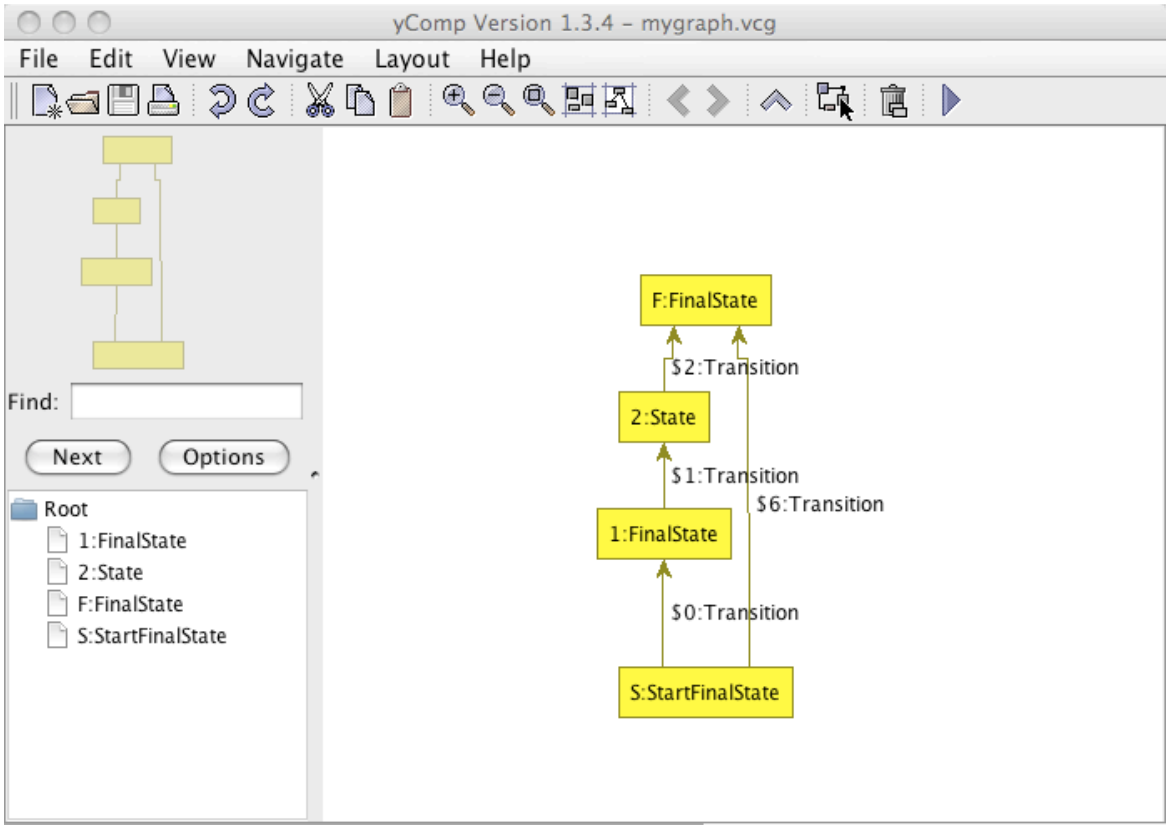


Figure 3.5: A state machine without ε-transitions

CHAPTER 4

GRAPH MODEL LANGUAGE

The graph model language lays the foundation for GrGen.NET. It allows specifying *node* and *edge classes* bearing typed *attributes* that are used in building and wiring the graph. Additionally, *enumeration types* can be declared, to be used as *attribute types* (the other available attribute types are built-in and don't need to be declared, they can just be used). A graph (instance) adheres to a graph (meta) model as defined in the following.

EXAMPLE (1)

The following toy example of a model of road maps gives a rough picture of the language:

```
1  enum Resident {VILLAGE = 500, TOWN = 5000, CITY = 50000}
2
3  node class Sight;
4
5  node class City {
6    Size:Resident;
7  }
8
9  const node class Metropolis extends City {
10   River:String;
11 }
12
13 abstract node class AbandonedCity extends City;
14 node class GhostTown extends AbandonedCity;
15
16 edge class Street;
17 edge class Trail extends Street;
18 edge class Highway extends Street
19     connect Metropolis[+] --> Metropolis[+]
20 {
21     Jam:boolean = false;
22 }
```

In this chapter as well as in Chapter 25 (GrShell) we use excerpts of Example 1 (the `Map` model) for illustration purposes.

The key features of GrGen.NET *graph models* as described by Geiß et al. [GBG+06, KG07] are given below:

Types
: Nodes and edges are typed. This is similar to classes in common (object-oriented) programming languages. GrGen.NET edge types can be directed or undirected.

Attributes
: Nodes and edges may possess attributes. The set of attributes assigned to a node or edge is determined by its type. The attributes themselves are typed, too.

Inheritance
: Node and edge types (classes) can be composed by multiple inheritance. `Node` and `Edge` are built-in root types of node and edge types, respectively. Inheritance eases the specification of attributes because subtypes inherit the attributes of their super types.

Connection Assertions
: To specify that certain edge types should only connect specific node types a given number of times, we include connection assertions.

Furthermore, node and edge types may be extended with *methods* in addition to attributes. They may get bundled into *packages*, preventing name collisions. And they support the concept of *graph nesting* with attributes of type `graph`. You find more regarding this in Chapter 18. In Chapter 31, you find more on *indices*, which allow quickly accessing the elements of the graph that are of interest to you.

4.1 Building Blocks

NOTE (2)

The following syntax specifications make heavy use of *syntax diagrams* (also known as rail diagrams). Syntax diagrams provide a visualization of EBNF[a] grammars. Follow a path along the arrows through a diagram to get a valid sentence of the nonterminal defined by the diagram (and in combination the language). Ellipses represent terminals whereas rectangles represent non-terminal uses. For further information on syntax diagrams see [MMJW91].

[a] Extended Backus–Naur Form.

Basic elements of the GrGen.NET graph model language are identifiers to denominate node types, edge types, and attributes. The model's name itself is given by its file name. The GrGen.NET graph model language is case sensitive.

Ident, *IdentDecl*
A non-empty character sequence of arbitrary length consisting of letters, digits, or underscores. The first character must not be a digit. *Ident* and *IdentDecl* differ in their role: While *IdentDecl* is a *defining* occurrence of an identifier, *Ident* is a *using* occurrence. An *IdentDecl* may be annotated. See Section 34.10 for annotations of declarations.

NOTE (3)

The GrGen.NET model language does not distinguish between declarations and definitions. More precisely, every declaration is also a definition. For instance, the following C-like pseudo GrGen.NET model language code is illegal:

```
node class t_node;
node class t_node {
  ...
}
```

Using an identifier before defining it is allowed. Every used identifier has to be defined exactly once.

NodeType, *EdgeType*, *EnumType*
These are (semantic) specializations of *Ident* to restrict an identifier to denote a node type, an edge type, or an enum type, respectively.

4.1.1 Base Types

The GrGen.NET model language has built-in types for nodes and edges. All nodes have the attribute-less, built-in type `Node` as their ancestor. All edges have the abstract (see Section 4.2), attribute-less, built-in type `AEdge` as their ancestor.

The `AEdge` has two non-abstract built-in children: `UEdge` as base type for undirected edges and `Edge` as base type for directed edges. The direction for `AEdge` and its ancestors that do not inherit from `Edge` or `UEdge` is undefined or *arbitrary*. Because there is the "magic of direction" linked to the edge base types, it is recommended to use the keywords `directed`, `undirected`, and `arbitrary` in order to specify inheritance (see Section 4.2). As soon as you decided for directed or undirected edge classes within your type hierarchy, you are not able to let ancestor classes inherit from a contradicting base type, of course. That is, no edge may be directed *and* undirected. This is an exception to the concept of multi-inheritance. Figure 4.1 shows the edge type hierarchy.

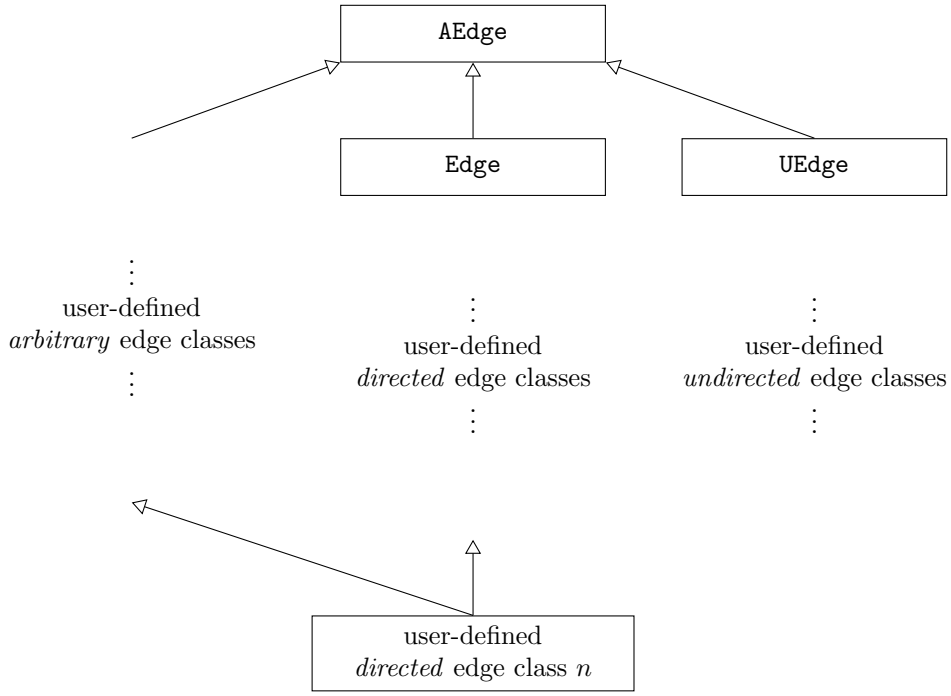
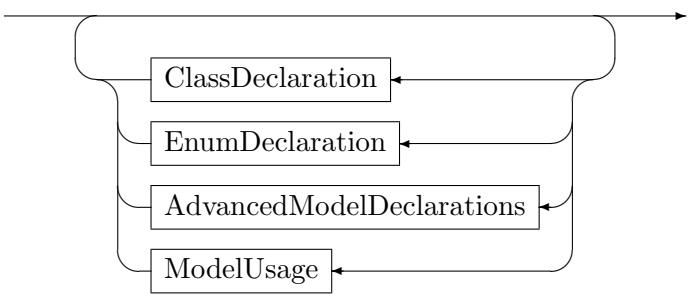


Figure 4.1: Type Hierarchy of GrGen.NET Edges

4.2 Type Declarations

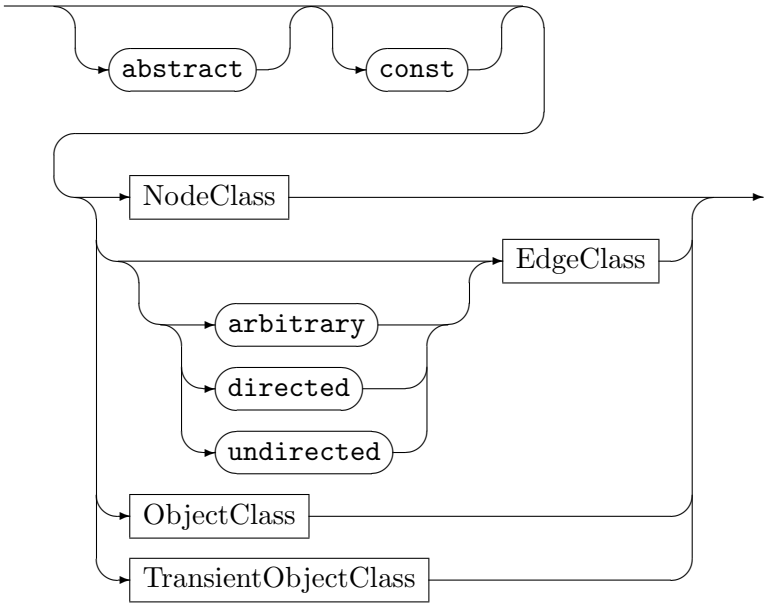
GraphModel



The graph model consists of zero or multiple type declarations. A *ClassDeclaration* defines a node type or an edge type. An *EnumDeclaration* defines an enum type to be used as a type for attributes of nodes or edges. Like all identifier definitions, types do not need to be declared before they are used (but of course must be declared somewhere). The *AdvancedModelDeclarations* are differentiated in Chapter 18 and Chapter 30. In addition, other models may be used (esp. supplying base classes to extend); this happens in the same way as models are used from the rules, see section 5.2 and the `#using` directive for more on this.

4.2.1 Node and Edge Types

ClassDeclaration



A *ClassDeclaration* defines a new node type or edge type. The keyword `abstract` indicates that you cannot instantiate graph elements of this type. Instead non-abstract types must be derived from the type in order to create graph elements. The abstract-property will not be inherited by subclasses.

EXAMPLE (2)

We adjust our map model and make `city` abstract:

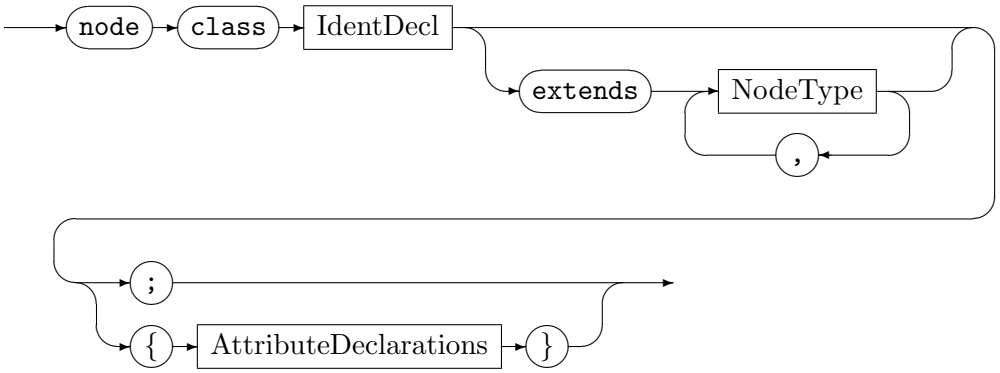
```
1 abstract node class City {
2   Size:int;
3 }
4 abstract node class AbandonedCity extends City;
5 node class GhostTown extends AbandonedCity;
```

You will be able to create nodes of type `GhostTown`, but not of type `City` or `AbandonedCity`. However, nodes of type `GhostTown` are also of type `AbandonedCity` as well as of type `City` and they have the attribute `Size`, hence.

The keyword `const` indicates that rules may not write to attributes (see also Section 5.4, `eval`). However, such attributes are still writable by LIBGR and GRSHELL directly. This property applies to attributes defined in the current class, only. It does not apply to inherited attributes. The `const` property will not be inherited by subclasses, either. If you want a subclass to have the `const` property, you have to set the `const` modifier explicitly.

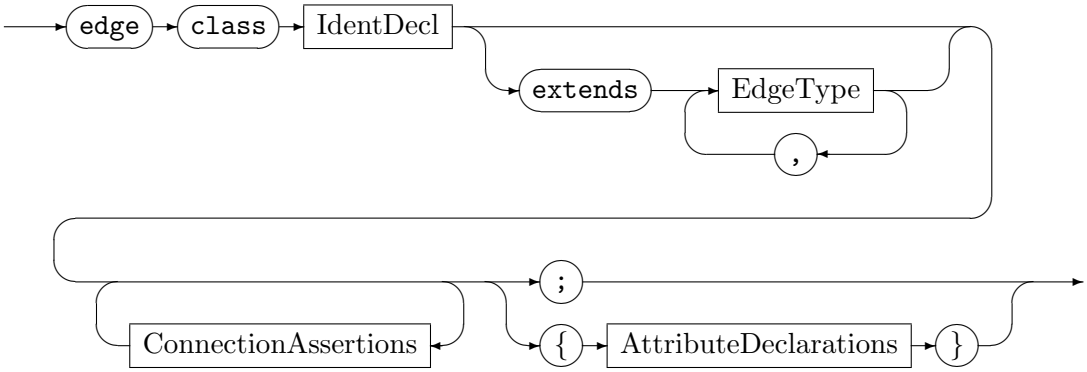
The keywords `arbitrary`, `directed`, and `undirected` specify the direction "attribute" of an edge class and thus its inheritance. An `arbitrary` edge inherits from `AEdge`, it is always abstract and neither directed nor undirected. A `directed` edge inherits from `Edge`. An `undirected` edge inherits from `UEdge`. If you do not specify any of those keywords, a `directed` edge is chosen by default. See also Section 4.1.1

NodeClass



A *NodeClass* defines a new node type. Node types can inherit from other node types (defined within the same file). If the `extends` clause is omitted, *NodeType* will inherit from the built-in type `Node`. Optionally, nodes can possess attributes.

EdgeClass



An *EdgeClass* defines a new edge type. Edge types can inherit from other edge types (defined within the same file). If the `extends` clause is omitted, *EdgeType* will inherit from the built-in type `Edge`. Optionally, edges can possess attributes. A *connection assertion* specifies that certain edge types should only connect specific nodes a given number of times. (see Section 4.1.1)

NOTE (4)

GrGen.NET supports multiple inheritance on nodes and edges – use it!
• Your nodes have something in common? Then factor it out into some base class. This works very well because of the support for multiple inheritance; you don't have to decide what is the primary hierarchy, forcing you to fall back to alternative patterns in the situations you need a different classification. Fine grain type hierarchies not only allow for concise matching patterns and rules, but deliver good matching performance, too (the search space is reduced early).
• Your edges have something in common? Then just do the same, edges are first class citizens.

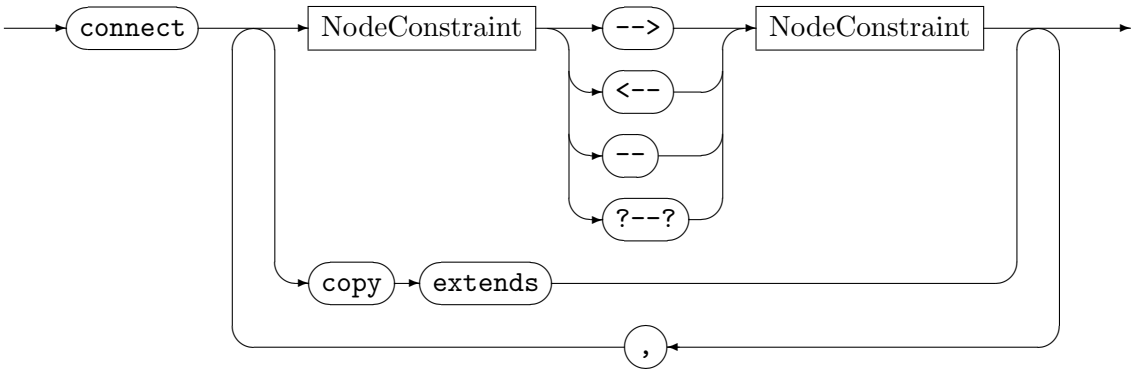
Connection Assertions

A *connection assertion* allows specifying a *constraint* on the topology. It allows specifying the node types an edge may connect to, and the multiplicities with which such edges appear on the nodes. It is denoted as a pair of node types, optionally with their multiplicities.

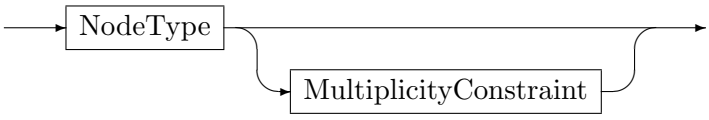
NOTE (5)

It is not forbidden to create graphs that are invalid according to connection assertions. GrGen.NET just enables you to check whether a graph is valid or not. See also Section 28.2, `validate`. This is in contrast to type graphs as implemented by tools leaning further towards theory, and more practically useful – often, constraints need to be violated temporarily during transformation.

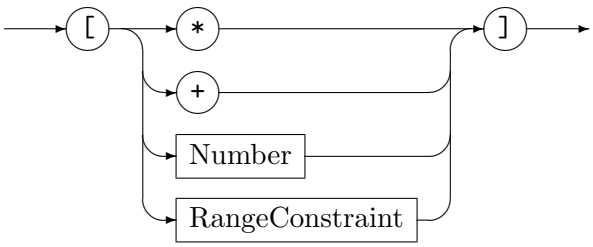
ConnectionAssertions



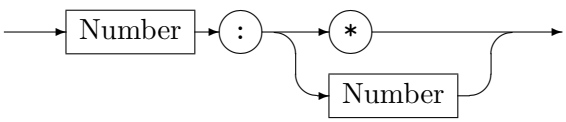
NodeConstraint



MultiplicityConstraint



RangeConstraint



A connection assertion is best understood as a simple pattern of the form (cf. graphlets 5.1.1) `:SourceNodeType -:EdgeType-> :TargetNodeType`, of which every occurrence in the graph is searched. In contrast to a real such pattern and the node types only edges of exactly the given edge type are taken into account. Per node of `SourceNodeType` (or a subtype) it is counted how often it was covered by a match of the pattern starting at it, and per node of `TargetNodeType` (or a subtype) it is counted how often it was covered by a match of the pattern ending at it. The numbers must be in the range specified at the `SourceNodeType` and the `TargetNodeType` for the connection assertion to be fulfilled. Giving no multiplicity constraint is equivalent to `[*]`, i.e. $[0, \infty[$, i.e. unconstrained.

Please take care of non-disjoint source/target types/subtypes in the case of undirected and especially arbitrary edges. In the case of multiple connection assertions, all are checked and errors for each one reported; for strict validation to succeed at least one of them must match. It might happen that none of the connection assertions of an `EdgeType` are matching an edge of this type in the graph. This is accepted in the case of normal validation (throwing connection assertions without multiplicities effectively back to nops); but as you normally want to see *only* the specified connections occurring in the graph, there is the additional mode of strict validation: if an edge is not covered by a single matching connection, validation fails. Furthermore there is the strict-only-specified mode, which only does strict validation of the edges for which connection assertions are given. See Section 28.2, `validate`, for an example.

The arrow syntax is based on the GrGen.NET graphlet specification (see Section 5.1.1). The different kinds of arrows distinguish between directed, undirected, and arbitrary edges. The `-->` arrow means a directed edge aiming towards a node of the target node type (or one of its subtypes). The `A<--B` connection assertion is equivalent to the `B-->A` connection assertion. The `--` arrow is used for undirected edges. The `?--?` arrow means an arbitrary edge, i.e. directed as well as undirected is possible (fixed by the concrete type inheriting from it); in case of a directed edge the connection pattern gets matched in both directions. *Number* is an `int` constant as defined in Chapter 6. Table 4.1 describes the multiplicity definitions.

`[`*n*`:*]`	The number of edges incident to a node of that type is unbounded. At least *n* edges must be incident to nodes of that type.
`[`*n*`:`*m*`]`	At least *n* edges must be incident to nodes of that type, but at most *m* edges may be incident to nodes of that type ($m \geq n \geq 0$ must hold).
`[*]`	Abbreviation for `[0:*]`.
`[+]`	Abbreviation for `[1:*]`.
`[`*n*`]`	Abbreviation for `[`*n*`:`*n*`]`.
	Abbreviation for `[1]`.

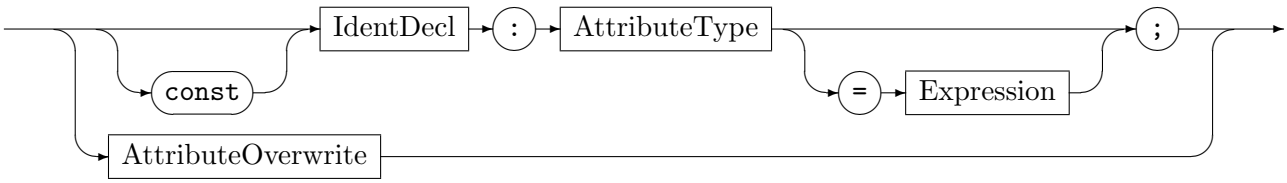
Table 4.1: GrGen.NET node constraint multiplicities

In order to apply the connection assertions of the supertypes to an `EdgeType`, you may use the keywords `copy extends`. The `copy extends` assertion "imports" the connection assertions of the *direct* ancestors of the declaring edge. This is a purely syntactical simplification, i. e. the effect of using `copy extends` is the same as copying the connection assertions from the direct ancestors by hand.

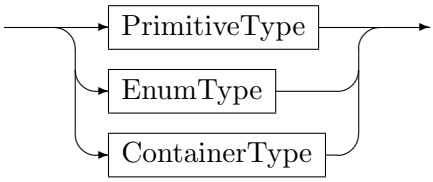
4.2.2 Attributes and Attribute Types

With attributes you model the specific properties of your node and edge classes. They are typed variables contained in the class, which are instantiated together with the node or edge.

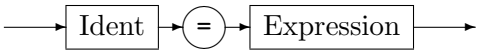
AttributeDeclaration



AttributeType



AttributeOverwrite



An *AttributeDeclaration* defines an attribute of a node or edge class. Possible types are *primitive types*, *enumeration types* (`enum`, cf. Section 4.2.3), and *container types*. See Section 6.1 for a list of built-in primitive types, and Section 16.1 for a list of built-in container types. Furthermore, attributes may be of node or edge or graph type, so they may be directly pointing to a node or an edge (seldom useful and discouraged), or to a subgraph (containers may contain those types, too, this holds also for container-typed attributes).

Optionally, attributes may be initialized with a *constant* expression (meaning only constants and other attributes are allowed). The expression has to be of a type compatible to the type of the declared attribute. See Chapter 6 for the GrGen.NET types and expressions reference. The *AttributeOverwrite* clause lets you overwrite initialization values of attributes of super classes. The initialization values are evaluated in the order in which they appear in the graph model file.

NOTE (6)

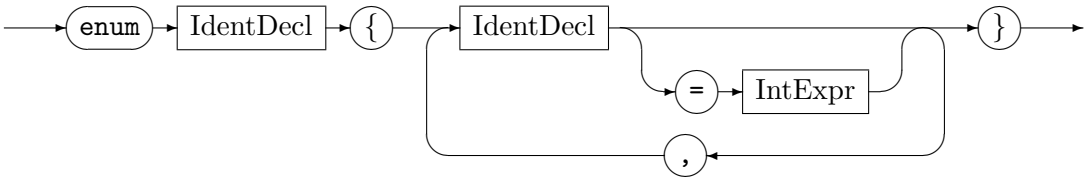
The following attribute declarations are *illegal* because of the order of evaluation of initialization values:

```
1 x:int = y;
2 y:int = 42;
```

Note that GrGen.NET lacks a concept of overwriting for attributes (in contrast to their values). On a path in the type hierarchy graph from a type up to the built-in root type there must be exactly one declaration for each attribute identifier. Furthermore, if multiple paths from a type up to the built-in root type exist, the declaring types for an attribute identifier must be the same on all such paths. This is equivalent to virtual inheritance in C++ parlance.

4.2.3 Enumeration Types

EnumDeclaration



Defines an enum type. An `enum` type is a collection of so called *enum items* (or members, or cases), which are used to define and name the values you want to distinguish (when using this type within your model). Those are internally associated with integral numbers, which may be specified explicitly. Accordingly, a GrGen.NET enum can be cast to an `int` (see Section 6.1). The following notes and examples highlight the relationship of `enum` items to `int`egers, but most often you are only interested in the main functionality: the *case distinction* with names encoded in the enum (and are then free to ignore these finer points).

NOTE (7)

An enum type and an `int` are different things, but in expressions enum values are implicitly cast to `int` values as needed (see Section 6.1). The other direction is not available at all, cf. the following Note.

NOTE (8)

Assignments of `int` values to something that has an enum type are forbidden (see Section 6.1). Only inside a declaration of an enum type an int value may be assigned to the enum item that is currently declared.

EXAMPLE (3)

```
1 enum Color {RED, GREEN, BLUE}
2 enum Resident {VILLAGE = 500, TOWN = 5000, CITY = 50000}
3 enum AsInC {A = 2, B, C = 1, D, E = (int)Resident::VILLAGE + C}
```

Consider, e.g., the declaration of the enum item `E`: By explicit and implicit casts of `Resident::VILLAGE` and `C` to `int` we get the `int` value 501, which is assigned to `E`. Moreover, the semantics is as in C [SAI+90]. So, the following holds: `RED` = 0, `GREEN` = 1, `BLUE` = 2, `A` = 2, `B` = 3, `C` = 1, `D` = 2, and `E` = 501.

NOTE (9)

The C-like semantics of enum item declarations implies that multiple items of one enum type can be associated with the same `int` value, and esp. implies that such items/cases cannot be distinguished anymore in-memory after assignment to a variable. Moreover, it implies that an enum item must not be used *before* its definition. This also holds for items of other enum types, meaning that the items of another enum type can only be used in the definition of an enum item, when the other enum type is defined *before* the enum type currently defined. The GrGen.NET-`enum` is directly mapped to a C#-`enum`.

4.2.4 Hints on Modeling

The modeling of the example from the introduction (cf. 1.3) that was chosen for the compiler case[BJ11a] is suboptimal regarding its usage of integer attributes to denote positions. This stems from the usage of indices into arrays in its originating compiler intermediate representation FIRM, and esp. its implementation. The right way to handle this in GRGEN.NET are edges explicitly denoting the successor relationship.

The basic points in modeling with GRGEN.NET are:

- Use types and inheritance as much as possible, on nodes as well as edges.
- Represent relationships directly and explicitly with typed edges, one for each relationship (modeling entities as nodes and relationships with edges in between them).
- Use edges of type `contains` to denote containment, pointing from the containing element to its children. If you have distinguished types of containment, use `contains` as abstract base class, from which the concrete ones that get instantiated are inheriting.
- Use edges of type `next` to denote orderings or successor relationships, pointing from the predecessor to the successor element in the ordering. If you have distinguished types of ordering, use `next` as abstract base class, from which the concrete ones that get instantiated are inheriting.
- If you have an ordered containment, up to a statically not known depth, use edges of both kinds at the same time. With `contains` to denote the containment in the ordered tree, and `next` to order the contained children of an element (so a node is incident to both kinds of edges).
- If you have an ordered containment with a statically known depth, use containment edges of dedicated types, with the type representing the order. Example: if you implement binary trees, use an edge of type `left` and an edge of type `right` to point to the left and right sibling. This is easier to process and less memory-hungry than the modeling needed for the full case.

NOTE (10)

If you implement an if-then-else in a syntax graph, use a node of type `IfThenElse` that points with a `condition` edge to the starting node of the condition expression, and a `then` edge to the starting node of the true case statements, and an `else` edge to the starting node of the false case statements (with statements inheriting from a `Statement` node class, being linked by `next` edges, and expressions being typically binary trees, inheriting from an `Expression` node class, being linked by edges of a type representing an ordered containment). This modeling of the syntax tree backbone is complemented with edges from the uses (nodes representing entities using other entities) to the definitions (nodes representing entities that can be used, in a textual syntax typically defined by a name that is mentioned when used) to build the full syntax graph.

NOTE (11)

Don't materialize transitive relationships. Point only to the immediate/direct successor, and compute transitive reachability as needed (with the `isReachable` predicates, or the `reachable` functions, or recursive subpatterns).

NOTE (12)

Don't materialize bidirectional relationships with two edges. Use undirected edges, or pretend they are unidirectional.

NOTE (13)

If you need to mark nodes in the graph, but only very few of them at the same time, think of reflexive edges of a special type. If the marking is sparse, they are the most efficient solution, before attributes, even before visited flags.

As a concluding remark: the graph model is the thing everything else is built upon. It is worth your continued attention. When you model things for the first time, you may be lucky in getting it fully right from the start, but it is more likely you find possibilities to make it better and increase its amenability for processing later on – don't hesitate to improve it then. The real star when working with GrGen.NET is the adequately-modeled graph representation.

CHAPTER 5

RULE LANGUAGE

The rule set language forms the core of GrGen.NET. Rule files refer to zero [1] or more graph models, and specify a set of rewrite rules. The rule language covers `test`s and `rule`s, with a pattern specification built mainly from *graphlets*, potentially with `hom`omorphic matching. Rewrite specifications may be given nested inside a rule in the form of a `replace` or `modify` block. Attributes of graph elements can be checked within `if` clauses and assigned within `eval` blocks.

The following rewrite rule gives an impression of the basic constructs that we will cover in this chapter (attribute expressions and assignments are covered in Chapters 6 and 14).

EXAMPLE (4)

```
1  #using "SomeModel.gm"
2
3  test t(n:Node, k:Node) : (Node) {
4      n --> . --> l:Node --> k;
5      hom(l, k);
6      return(l);
7  }
8
9  rule r {
10     n:N -e:E-> n;
11     if{ n.a <= e.a; }
12
13     replace {
14         n --> m:N;
15         eval { m.a = n.a + 1; }
16     }
17 }
18
19 rule s(var i:int) : (Node, N, int) {
20     :N <-- n:N --> m:Node;
21     m <--> l:Node -:E-> n;
22     if{ n.a <= i; }
23
24     modify {
25         delete(n);
26         return(m, l, n.a + 1);
27     }
28 }
```

[1] Omitting a graph meta model means that GrGen.NET uses a default graph model. The default model consists of the base type `Node` for vertices and the base type `Edge` for edges.

5.1 Building Blocks

The GrGen.NET rule set language is case sensitive. The language makes use of several identifier specializations in order to denominate all the GrGen.NET entities.

Ident, *IdentDecl*
A non-empty character sequence of arbitrary length consisting of letters, digits, or underscores. The first character must not be a digit. *Ident* may be an identifier defined in a graph model (see Section 4.1). *Ident* and *IdentDecl* differ in their role: While *IdentDecl* is a *defining* occurrence of an identifier, *Ident* is a *using* occurrence. An *IdentDecl* non-terminal can be annotated. See Section 34.10 for annotations of declarations.

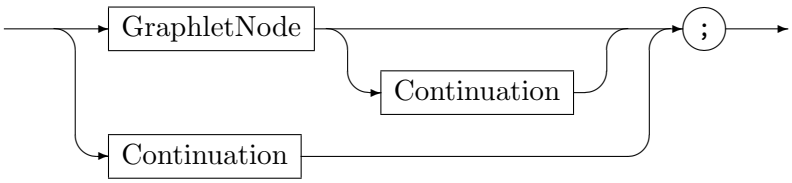
NOTE (14)

As in the GrGen.NET model language (see note 3) every declaration is also a definition. Using an identifier before defining it is allowed. Every used identifier has to be defined exactly once.

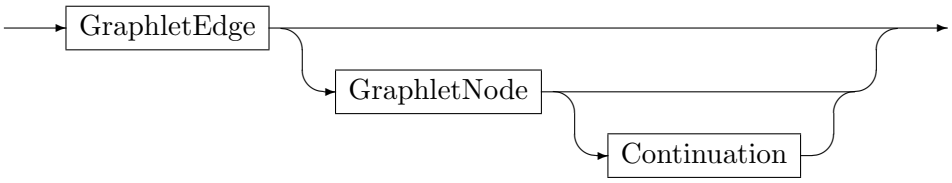
ModelIdent, *TypeIdent*, *NodeType*, *EdgeType*
These are (semantic) specializations of *Ident*. *TypeIdent* matches every type identifier, i.e. a node type, an edge type, an enumeration type or a primitive type. All the type identifiers are actually type *expressions*. See Section 6.7 for the use of type expressions.

5.1.1 Graphlets

Graphlet



Continuation



A graphlet specifies a connected subgraph. GrGen.NET provides graphlets as a descriptive notation to define both, patterns to search for as well as the subgraphs that replace or modify matched spots in a host graph. Any graph can be specified piecewise by a set of graphlets. In Example 4, line 10, the graphlet `n:N -e:E-> n` begins with a node declaration `n:N`, followed by the continuation graphlet `-e:E-> n`, declaring an edge `e:E`, leading again to the node `n`, referenced by its name.

All the graph elements of a graphlet have *names*. The name is either user-assigned or a unique internal, non-accessible name. In the second case the graph element is called *anonymous*. For illustration purposes we use a `$<number>` notation to denote anonymous graph elements in this document. For example the graphlet `n:N --> m:Node` contains an anonymous edge; thus can be understood as `n -$1:Edge-> m`. Names must not be redefined; once defined, a name is *bound* to a graph element. We use the term “binding of names”

because a name not only denotes a graph element of a graphlet but also denotes the mapping of the abstract graph element of a graphlet to a concrete graph element of a host graph. So graph elements of different names are pairwise distinct except for homomorphically matched graph elements (see Section 5.3). For instance `v:NodeType1 -e:EdgeType-> w:NodeType2` selects some node of type `NodeType1` that is connected to a node of type `NodeType2` by an edge of type `EdgeType` and binds the names `v`, `w`, and `e`. If `v` and `w` are not explicitly marked as homomorphic, the graph elements they bind to are distinct. The binding of names allows for splitting a single graphlet into multiple graphlets as well as defining cyclic structures.

EXAMPLE (5)

The following graphlet (`n1`, `n2`, and `n3` are defined somewhere else)

```
1 n1 --> n2 --> n3 <-- n1;
```

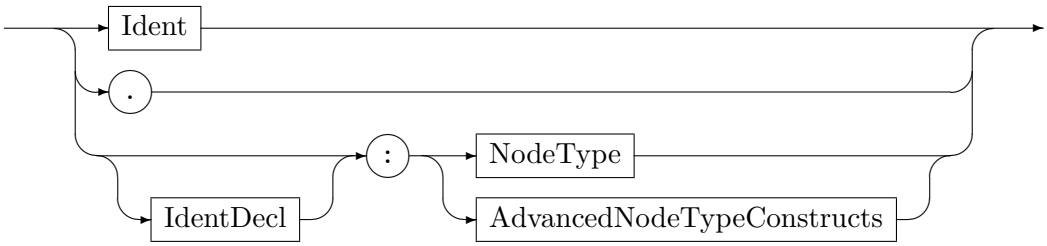
is equivalent to

```
1 n2 --> n3;
2 n1 --> n2;
3 n3 <-- n1;
```

and `n1 --> n3` is equivalent to `n3 <-- n1`, of course.

The visibility of names is determined by scopes. Scopes can be nested. Names of surrounding scopes are visible in inner scopes. Usually a scope is defined by { and }. In Example 4, lines 13 to 16, the rewrite part uses `n` from the surrounding scope, and declares `m` that is not available in the nesting pattern. And in Example 62, lines 13 to 17, the negative condition uses `n3` from the surrounding scope and declares `n4` and `e1`.

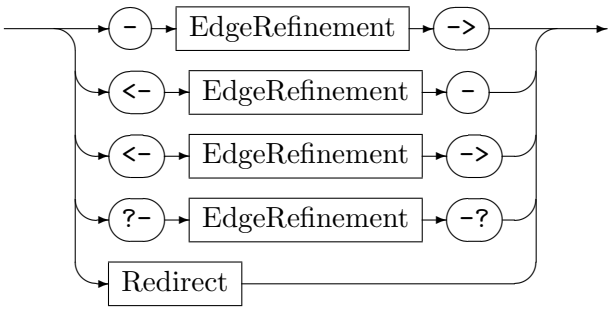
GraphletNode



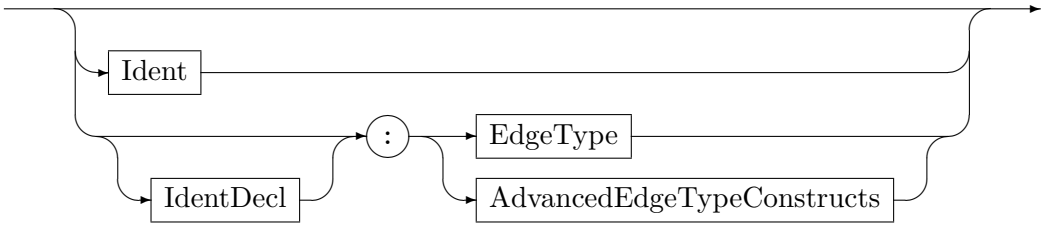
Specifies a node of type *NodeType*; the alternative advanced node constructs are explained in chapter 12. The `.` is an anonymous node of the base type `Node`; remember that every node type has `Node` as super type.

Graphlet	Meaning
`x:NodeType;`	The name `x` is bound to a node of type `NodeType` or one of its subtypes.
`:NodeType;`	`$1:NodeType`
`.;`	`$1:Node`
`x;`	The node to which `x` is bound.

GraphletEdge



EdgeRefinement



A *GraphletEdge* specifies an edge. Anonymous edges are specified by an empty *EdgeRefinement* clause, i. e. `-->`, `<--`, `<-->`, `--`, `?--?` or `-:T->`, `<-:T-`, ... for an edge type `T`, respectively. A non-empty *EdgeRefinement* clause allows for detailed edge type specification. The alternative advanced edge constructs as well as the *Redirect* clause are explained in chapter 12.

The different kind of arrow tips distinguish between directed, undirected, and arbitrary edges (see also Section 4.1.1). The arrows `-->` and `<--` are used for directed edges with a defined source and target. The arrow `--` is used for undirected edges. The pattern part allows for further arrow tips, namely `?--?` for arbitrary edges and `<-->` for directed edges with undefined direction. Note that `<-->` is *not* equivalent to the `--> ; <-- ;` statements. In order to produce a match for the arrow `<-->`, it is sufficient that one of the statements `-->`, `<--` matches. If an edge type is specified (through the *EdgeRefinement* clause), this type has to correspond to the arrow tips, of course.

Graphlet	**Meaning**
`-e:EdgeType-> ;`	The name `e` is bound to an edge of type `EdgeType` or one of its subtypes.
`-:EdgeType-> ;`	`-$1:EdgeType-> ;`
`--> ;`	`-$1:Edge-> ;`
`<--> ;`	`-$1:Edge-> ;` or `<-$1:Edge- ;`
`-- ;`	`-$1:UEdge-> ;`
`?--? ;`	`-$1:AEdge-> ;`
`-e-> ;`	The edge `e` is bound to.

As the above table shows, edges can be defined and used separately, i.e. without their incident nodes. Accidentally "redirecting" an edge is prevented by compiler checks (you must explicitly use the *Redirect* clause to achieve this): The graphlets

```
-e:Edge-> .;
x:Node -e-> y:Node;
```

are illegal, because the edge `e` would have two destinations: an anonymous node and `y`. However, the graphlets

```
-e-> ;
x:Node -e:Edge-> y:Node;
```

are allowed, but the first graphlet `-e->` is superfluous. In particular this graphlet does not identify or create any "copies", neither if the graphlet occurs in the pattern part nor if it occurs in the rewrite part.

EXAMPLE (6)

Some attempts to specify a loop edge:

Graphlet	Meaning
`x:Node -e:Edge-> x;`	The edge `e` is a loop.
`x:Node -e:Edge-> ; -e-> x;`	The edge `e` is a loop.
`-e:Edge-> x:Node;`	The edge `e` may or may not be a loop.
`. -e:Edge-> .;`	The edge `e` is certainly not a loop.

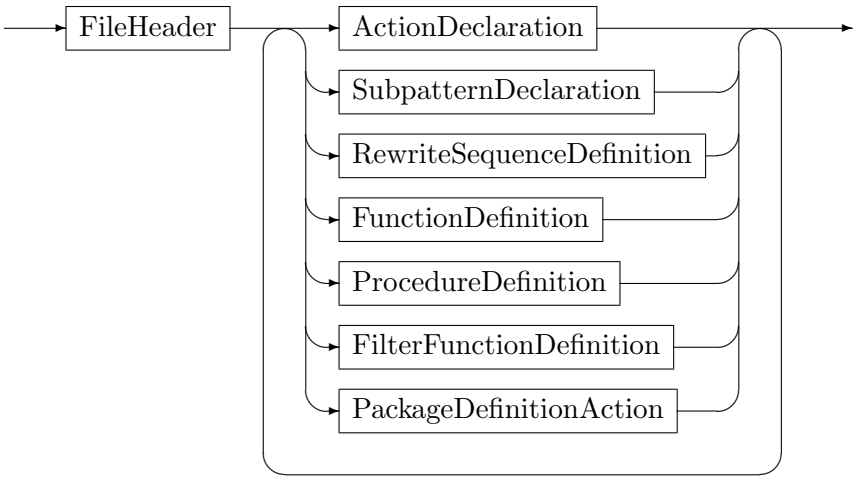
NOTE (15)

Although both the pattern part and the rewrite part use graphlets, there are subtle differences between them. Most of the differences can be seen in chapter 12 where the advanced constructs are explained.

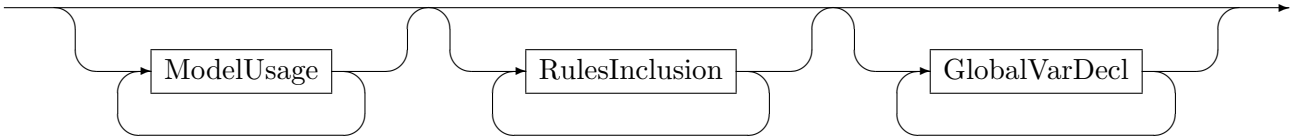
5.2 Rules and Tests

The structure of a rule set file is as follows:

RuleSet

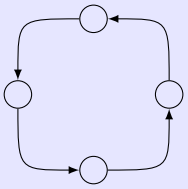


FileHeader

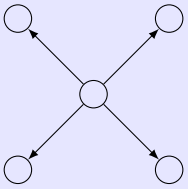


EXAMPLE (7)

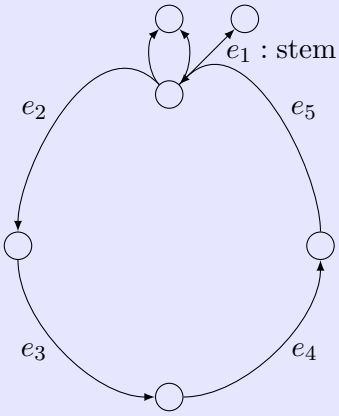
Some graphlets:



```
x:Node --> . --> . --> . --> x;
```



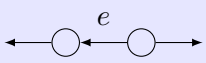
```
. <-- x:Node --> .;
. <-- x --> .;
```




```
. <-e1:stem- n1:Node -e2:Edge-> . -e3:Edge-> .
  -e4:Edge-> . -e5:Edge-> n1;
n1 --> n2:Node;
n1 --> n2;
```



```
. --> . <-- . <-- . --> .;
```

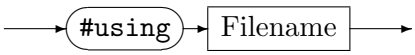



```
-e:Edge->
<-- . <-e- . --> ;
```

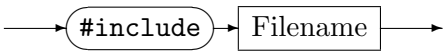
And some illegal graphlets:

`. -e:Edge-> .; . -e-> .;`	Would affect redirecting of edge `e`.
`x -e:T-> y; x -e-> x;`	Would affect redirecting of edge `e`.
`<-- --> ;`	There must be at least a node between the edges.

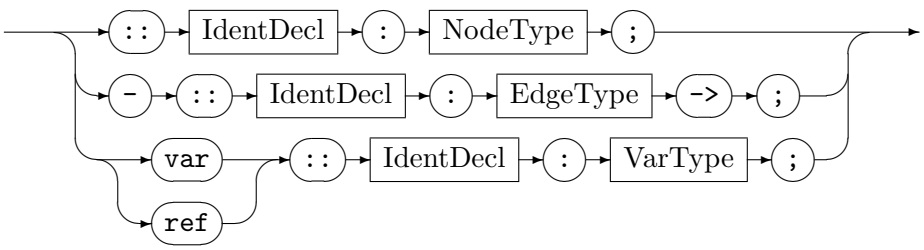
ModelUsage



RulesInclusion



GlobalVarDecl

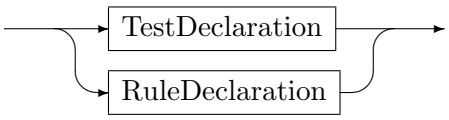


A rule set built on the graph models you are `using` consists primarily of actions, i.e. rewrite rules and tests, plus further constructs that will be introduced in later chapters. Additionally you may `include` further rule language files. (As a hint regarding the syntax diagrams: please note that the bottom rail in the *RuleSet* diagram departs before the end and joins in after the *FileHeader*, i.e. it denotes a looping back edge (a fast forward edge would have split and join points at the same positions but of the opposite direction)).

Furthermore, you may declare graph global variables; this is a pure declaration that they will exist with the given type during execution. It renders them accessible in the rules (esp. the attribute condition and attribute evaluation), but you must define and assign them before rule execution outside of the rules. They are made available to allow for an easy parameterization of entire transformations, defining the environment; other uses are discouraged. See Chapter 11 for more on this. (Use `ref` for container types and `var` for the other (non-node or edge) types.)

In case of multiple graph models, GrGen.NET uses the union of these models. In case of multiple includes, GrGen.NET uses the union of these actions. In this case beware of conflicting declarations. You may use packages as a remedy, in the model for conflicting data units, or in the actions for conflicting computation units – or you may use name prefixes as you would do in the C programming language.

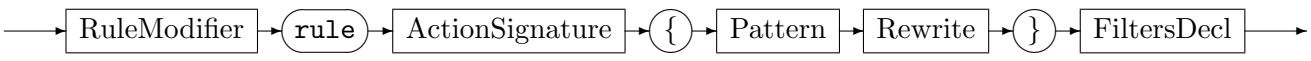
ActionDeclaration



TestDeclaration



RuleDeclaration



Declares a single rewrite rule such as `SomeRule`. It consists of a pattern part (see Section 5.3) in conjunction with its rewrite part (see Section 5.4). A *test* has no rewrite specification. It's intended to check whether (and maybe how many times) a pattern occurs (see example 8). For an explanation of the available modifiers see Section 12.1, for an explanation of the filters see Section 17 and Section 34.4.

EXAMPLE (8)

We define a test `SomeCond`

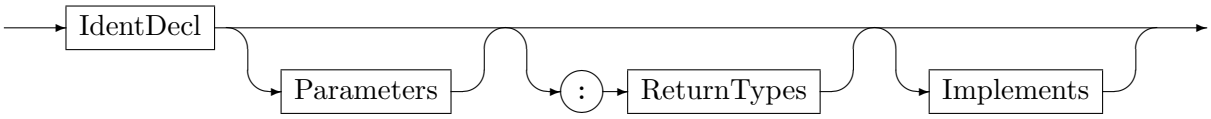
```
1 test SomeCond {
2   n:SeldomNodeType;
3 }
```

and execute in GrShell:

```
1   exec SomeCond & SomeRule
```

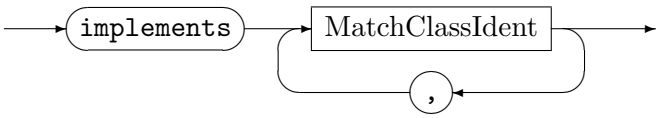
SomeRule will only be executed, if a node of type `SeldomNodeType` exists. For graph rewrite sequences in GrShell see Section 25.4.

ActionSignature



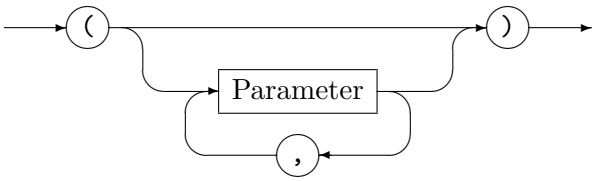
The signature sets the name of a rewrite rule to *IdentDecl* and optionally names and types of formal parameters as well as a list of return types. Parameters and return types provide users with the ability to exchange graph elements between rules, similar to parameters of procedural languages. This way it is possible to specify *where* a rule should be applied.

Implements

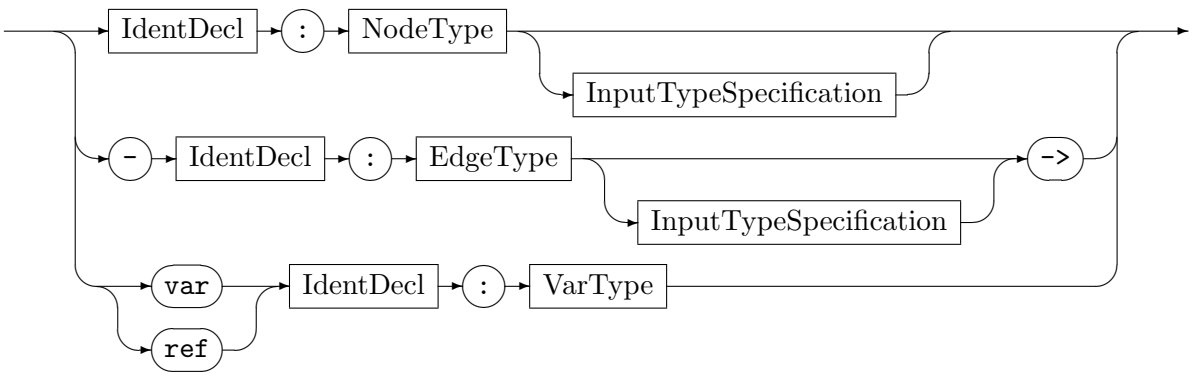


The implements clause allows specifying match classes that must be implemented by the rules, see 21.4 for more on this.

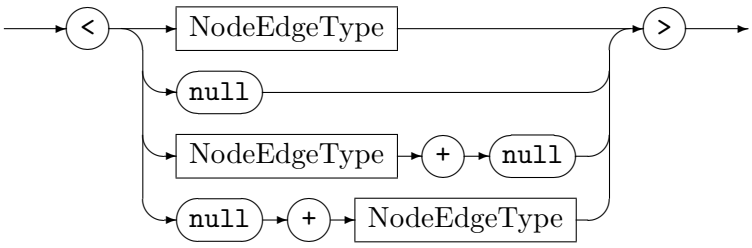
Parameters



Parameter

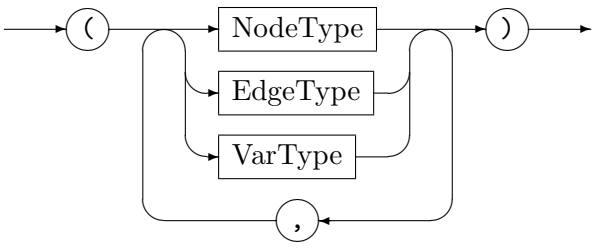


InputTypeSpecification



Within a rule, graph element parameters are treated as graph elements of the pattern - just predefined. It is your task to ensure the elements handed in satisfy the interface, i.e. parameters must not be null and must be of the type specified or a subtype of the type specified (in contrast to some old versions). If you need more flexibility and want to call the rule with parameters not fulfilling the interface you can append an input type specification to the relevant parameters, which consists of the type to accept at the action interface, or null, or both, enclosed in left and right angles. If the input type specification type is given, but the more specific pattern element type is not satisfied, matching simply fails. If null is declared in the input type specification and given at runtime, the element is searched in the host graph. Don't use null parameters unless you need them, because every null parameter doubles the number of matcher routines which get generated. Non-graph element parameters must be prefixed by the `var` or `ref`-keyword; VarType is one of the attribute types supported by GrGen.NET (cf. 6.1 and 16.1 and 18.1). The primitive types require the `var` prefix and are handed in by-value; the container types and internal (transient) class object types require the `ref` prefix and are handed in by-reference. Please note that the effect of assigning to a var/ref parameter in `eval` (see 5.4) is undefined (concerning the parameters as well as the argument); they are only available for reading, the by-ref parameters additionally for container-addition and removal (cf. 13), or attribute assignment.

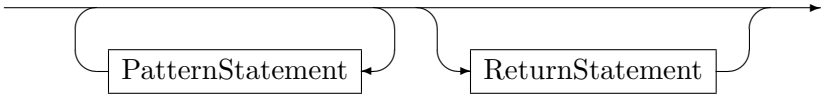
ReturnTypes



The return types specify edge and node types of graph elements that are returned by the replace/modify part. If return types are specified, the `return` statement is mandatory. Otherwise no `return` statement may occur. See also Section 5.4, `return`.

5.3 Pattern Part

Pattern



A pattern consists of zero or more pattern statements and, (only) in case of a test, an optional return statement. All the pattern statements must be fulfilled by a subgraph of the host graph in order to form a match. An empty pattern always produces exactly one (empty) match. This is caused by the uniqueness of the total and totally undefined function. For an

EXAMPLE (9)

The test `t` that checks whether node `n1` is adjacent to `n2` (connected by an undirected edge or incoming directed edge or outgoing directed edge)

```
test t(n1:Node<null>, n2:Node<null>) {
  n1 ?--? n2;
}
```

is equivalent to the tests `t1`-`t4` which are chosen dependent on what parameters are defined.

```
test t1(n1:Node, n2:Node) {
  n1 ?--? n2;
}
test t2(n1:Node) {
  n1 ?--? n2:Node;
}
test t3(n2:Node) {
  n1:Node ?--? n2;
}
test t4 {
  n1:Node ?--? n2:Node;
}
```

So if both parameters are not defined, `t4` is chosen, which succeeds as soon as there are two distinct nodes in the graph connected by some edge.

EXAMPLE (10)

We extend `SomeRule` (Example 62) with a user defined node to match and we want it to return the rewritten graph elements `n5` and `e1`.

```
rule SomeRuleExt(varnode:Node):(Node, EdgeTypeB) {
  n1:NodeTypeA;
  ...

  replace {
    varnode;
    ...
    return(n5, e1);
    eval {
      ...
```

We do not define `varnode` within the pattern part because this is already covered by the parameter specification itself.

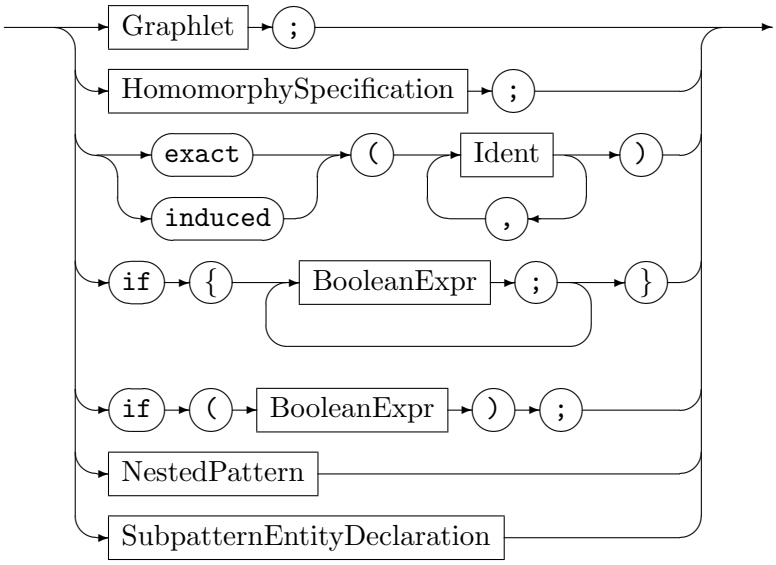
explanation of the pattern modifiers `dpo`, `identification`, `dangling`, `induced`, and `exact` see Section 12.1.

Names defined for graph elements may be used by other pattern statements as well as by replace/modify statements. Like all identifier definitions, such names may be used before their declaration. See Section 5.1 for a detailed explanation of names and graphlets.

NOTE (16)

The application of a rule is not deterministic (remember the example of the introduction in Section 1.6), there may be more than one subgraph that matches the pattern. In this case the location which is rewritten is chosen arbitrarily (in fact matching is just stopped after the first match following an implementation-defined order is found). But when you compute all matches, you can select deterministically one of them with a post-matches filter, cf. chapter 17, or you can do so on API level, see chapter 33.
The rule from the rule set to apply, in contrast, is chosen deterministically by default by the graph rewrite *sequence* (GrGen.NET follows the programmed in contrast to the graph grammar approach), but you can introduce non-determinism there with the `$` flag prepended to a sequence operator, cf. 11.1, or the dedicated indeterministic choices specified in section 21.3.

PatternStatement



The semantics of the various pattern statements are given below:

Graphlet
: Graphlets specify connected subgraphs. See Section 5.1 for a detailed explanation of graphlets.

Isomorphic/Homomorphic Matching
: See Subsection 5.3.1 for a discussion of this.

Attribute Conditions
: The Java-like attribute conditions (keyword `if`) in the pattern part allow for further restriction of the applicability of a rule. The pattern can only match if the *BooleanExpression* (see chapter 6) is evaluated to `true`.

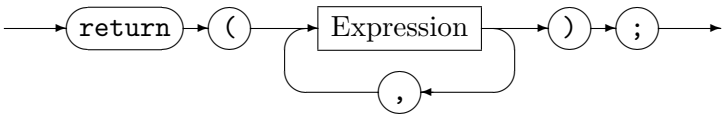
Pattern Modifiers
: Additionally to modifiers that apply to a pattern as a whole, you may also specify pattern modifiers for a specific set of nodes. Accordingly the list of identifiers for a pattern modifier must not contain any edge identifier. See Section 12.1 for an explanation of the `exact` and `induced` modifiers.

NestedPattern
: will be explained in 7.1,7.2,7.3,7.4.

SubpatternEntityDeclaration
: will be explained in 8.1.

Keep in mind that using type constraints or the `typeof` operator might be helpful. See Section 6.7 for further information.

ReturnStatement



Returned graph elements (given by their name) and value entities (given by an expression computing them) must appear in the same order as defined by the return types in the signature (see Section 5.2, `ActionSignature`). Their types must be compatible to the return types specified.

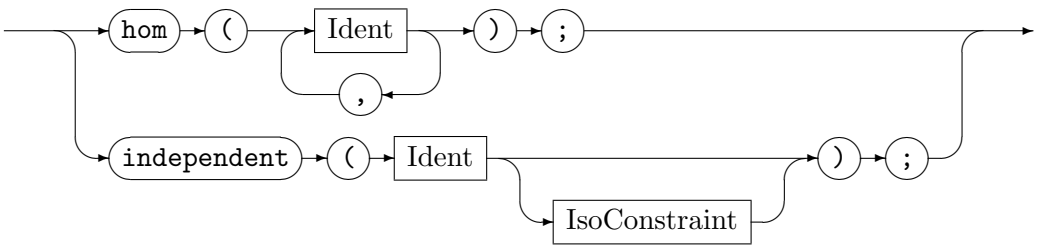
NOTE (17)

If you are using a graph at the API level without shell-provided names accessible by the `nameof`-operator, you may want to number the graph elements for dumping like this (alternatively you may use the unique index, cf. 30.7; it does not allow manual handling, but it also altogether spares you from manual handling):

```
1  rule numberNode(var id:int) : (int)
2  {
3    n:NodeWithIntId;
4    if { n.id == 0; }
5
6    modify {
7      eval {
8        n.id = id;
9      }
10     return (id + 1);
11   }
12 }
```

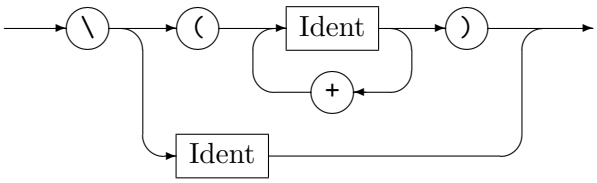
5.3.1 Isomorphic and Homomorphic Matching

HomomorphySpecification



The matching of pattern elements to host graph elements in GrGen.NET is isomorphic (injective) by default, i.e. two pattern elements can *not* be bound to the same host graph element. The `hom` operator specifies the nodes or edges that may be matched homomorphically. In contrast to the default isomorphic matching, the specified graph elements *may* be mapped to the same graph element in the host graph. Note that the graph elements must have a common subtype. Several homomorphically matched graph elements will be mapped to a graph element of a common subtype. In Example 62 nodes `n1` and `n2` may be the same node. This is possible because they are of the same type (`NodeTypeA`). Inside a NAC/PAC the `hom` operator may only operate on graph elements that are either defined or used in the NAC/PAC (cf. 7.1/7.2). Nested `negative`/`independent` blocks inherit the `hom` declarations of their nesting pattern. In contrast to previous versions of GrGen `hom` declarations are non-transitive, i.e `hom(a,b)` and `hom(b,c)` don't cause `hom(a,c)` unless specified.

IsoConstraint



The `independent` operator specifies the node or edge given within the parentheses to be homomorphic to all the other pattern elements. With the iso constraint following, exceptions can be given defining the pattern elements it must be distinct to. Thus we have a specification mode requesting homomorphic matching with additional isomorphy exceptions in contrast to the default mode, requesting isomorphic matching with additional homomorphy exception. It is recommended to *not* use the `independent` operator, it is potentially dangerous allowing to carry out conflicting rewrites, with an element `a` to be deleted, element `b` to be kept, and element `c` to be retyped, all mapping to the same graph element (you will experience funny effects and/or crashes in this case; the `hom` operator offers some static checks against this). The operator is available as a last resort for some situations in matching complex structures with iterated and subpatterns, in which it is unfeasible or not possible to explicitly give elements the pattern element may be homomorphic to, because they were matched in a pattern at an arbitrary distance in the derivation path which only dynamically called the pattern of interest, i.e. they can't be referenced by name, cf. 8.4.

5.4 Rewrite Part with Replace and Modify Modes

A rule not only specifies a pattern to be matched, but a transformation from the pattern aka left-hand side (LHS) graph to the right-hand side (RHS) graph (i.e. a dependent replacement, dependent insofar that the rewrite is specified based on the pattern). On applying the rule, an instance of the pattern is searched for in the host graph (yielding a matched subgraph), and alongside the transformation specification, graph elements of the match are kept, graph elements of the match are deleted, and graph elements are newly created.

5.4.1 Implicit Definition of the Preservation Morphism r

In graph rewriting theory, the transformation specification is done by defining a preservation morphism r. In GrGen.NET the preservation morphism r is defined implicitly by using names both in pattern graphlets and rewrite graphlets. Remember that a name is bound to each of the graph elements (either user defined or internally defined). If such a name is used in a rewrite graphlet, the denoted graph element will be kept. Otherwise the graph element will be deleted (unless modify mode is applied). By defining a name in a rewrite graphlet a corresponding graph element will be newly created. So in a rewrite pattern anonymous

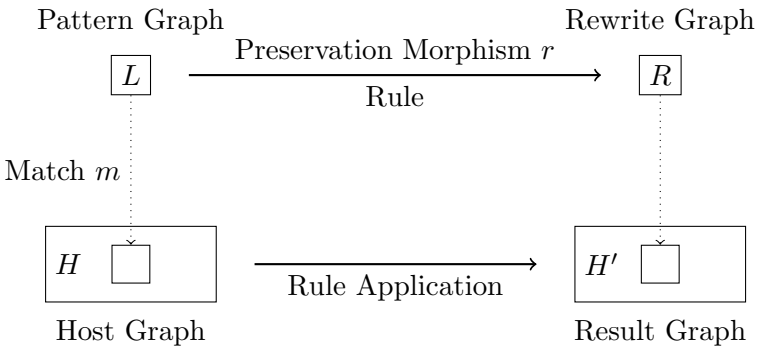


Figure 5.1: Process of Graph Transformation

graph elements will always be created. Using a name multiple times has the same effect as a single using occurrence. In case of a conflict between deletion and preservation, deletion is prioritized. If an incident node of an edge gets deleted, the edge will be deleted as well (in compliance to the SPO semantics).

Pattern (LHS)	**Rewrite (RHS)**	$r : L \longrightarrow R$	**Meaning**
`x:T;`	`x;`	$r : \text{lhs}.x \mapsto \text{rhs}.x$	Preservation
`x:T;`		$\text{lhs}.x \notin \text{def}(r)$	Deletion
	`x:T;`	$\text{rhs}.x \notin \text{ran}(r)$	Creation
`x:T;`	`x:T;`	—	Illegal, redefinition of `x`
`-e:T-> ;`	`-e-> x:Node;`	—	Illegal, redirection of `e`
`x:N -e:E-> y:N;`	`x -e-> ;`	$r : \{\text{lhs}.x\} \mapsto \{\text{rhs}.x\}$	Deletion of `y`. Hence deletion of `e`.

Table 5.1: Definition of the preservation morphism r

5.4.2 Specification Modes for Graph Transformation

For the task of rewriting, GrGen.NET provides two different modes: A *replace mode* and a *modify mode*.

Replace mode

The semantics of this mode is to delete every graph element of the pattern that is not used (referenced) in the rewrite part, keep every graph element that is used, and create every additionally defined graph element. "Using" means that the name of a graph element occurs in a rewrite graphlet. Attribute calculations are not using occurrences. In Example 10 the nodes `varnode` and `n3` will be kept. The node `n1` is replaced by the node `n5` preserving `n1`'s edges. The anonymous edge instance between `n1` and `n2` only occurs in the pattern and therefore gets deleted.
See Section 5.4.1 for a detailed explanation of the transformation semantics.

Modify mode

In modify mode, all elements are kept, unless it is explicitly specified that they are to be deleted. The modify mode can be regarded as a rewrite part in replace mode, where every pattern graph element is added (occurs) before the first rewrite statement. In particular all the anonymous graph elements are kept. This mode supports a `delete` operator that deletes every element given as an argument. Deletion takes place after all other rewrite operations. Multiple deletion of the same graph element is allowed (but pointless) as well as deletion of just created elements (pointless, too).

NOTE (18)

In general modify mode should be preferred as it allows to read the rewrite part as a diff of the changes to be made to the pattern part, whereas replace mode requires comparing the LHS and RHS pattern while reading to find out about the changes. Only if most of the pattern is to be deleted is the replace mode advantageous, pinpointing what should stay. (Furthermore it might be simpler to generate code for, just dumping both patterns.)

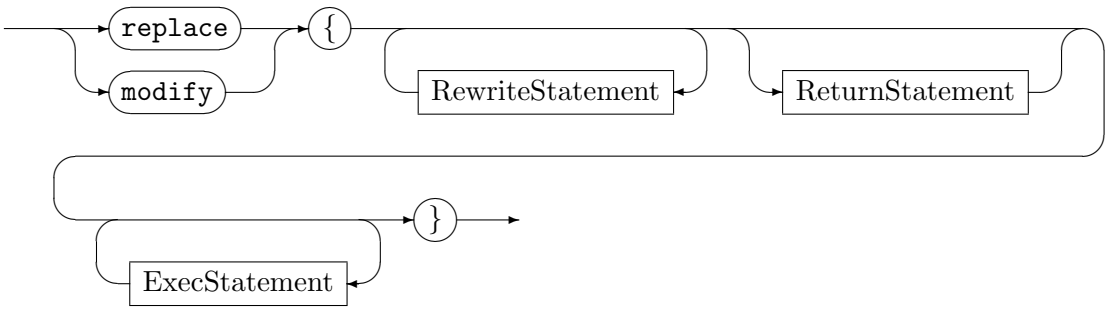
EXAMPLE (11)

How might Example 10 look in modify mode? We have to denominate the anonymous edge between `n1` and `n2` in order to delete it. The node `varnode` should be kept and does not need to appear in the modify part. So we have

```
1  rule SomeRuleExtModify(varnode: Node): (Node, EdgeTypeB)  {
2    ...
3    n1 -e0:Edge-> n2;
4    ...
5    modify {
6      n5:NodeTypeC<n1>;
7      n3 -e1:EdgeTypeB-> n5;
8      delete(e0);
9      eval {
10       ...
```

5.4.3 Syntax

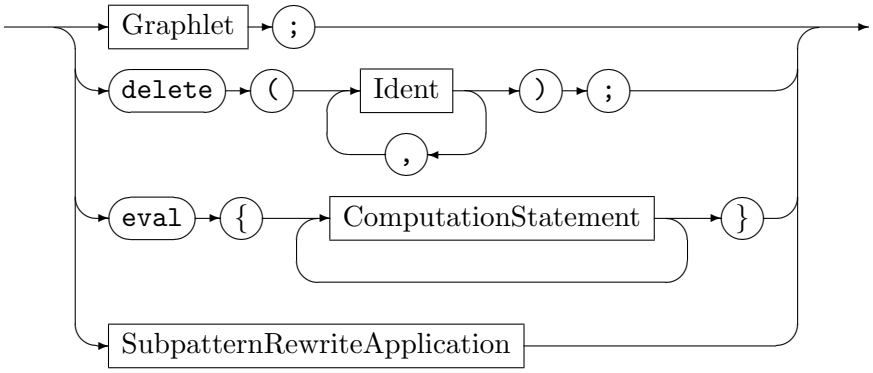
Rewrite



Selects whether replace mode or modify mode is to be used, opening a block as it is known from other block nesting languages and used in other GrGen.NET-constructs, with elements declared inside not being visible from the outside.

Several rewrite statements describe the transformation from the pattern subgraph to the destination subgraph. The *ReturnStatement* was already introduced, for tests it can appear in the pattern part. Inside rules it can only be given in the rewrite part. The *ExecStatement* will be introduced in chapter 13.

RewriteStatement



The semantics of the various rewrite statements are:

Graphlet
: Analogous to a pattern graphlet; a specification of a connected subgraph. Its graph elements are either kept because they are elements of the pattern or added otherwise. Names defined in the pattern part must not be redefined in the rewrite graphlet. See Section 5.1 for a detailed explanation of graphlets.

Deletion
: The `delete` operator is only available in modify mode. It deletes the specified pattern graph elements. Multiple occurrences of `delete` statements are allowed. Deletion statements are executed after all other rewrite statements. Multiple deletion of the same graph element is allowed (but pointless) as well as deletion of just created elements (pointless, too).

Computation (Attribute Evaluation)
: If a rule is applied, then the attributes of matched and inserted graph elements will be recalculated according to the `eval` statements. Besides attribute evaluations, further computations may be executed and side effects applied, see Chapter 14 for more on this.

SubpatternRewriteApplication
: will be explained in 9.2.

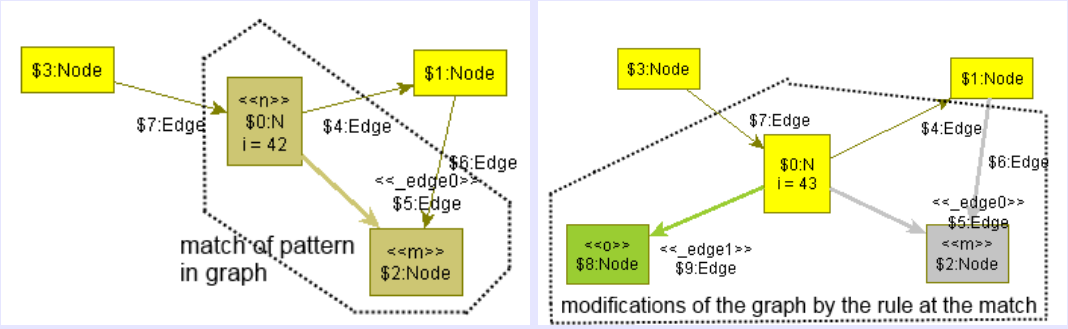
Several attribute evaluation parts may be given within the rewrite block. Multiple evaluation statements will be internally concatenated, preserving their order. Evaluation takes place before any graph element specified to be deleted by the rule gets deleted and after all the new elements (according to the rule rewrite part) have been created. You may read (and write, although this is pointless) attributes of graph elements to be deleted.

EXAMPLE (12)

```
1  ...
2  modify {
3    ...
4    eval { y.i = 40; }
5    eval { y.j = 0;  }
6    x:IJNode;
7    y:IJNode;
8    delete(x);
9    eval {
10     x.i = 1;
11     y.j = x.i;
12     x.i = x.i + 1;
13     y.i = y.i + x.i;
14   }
15 }
```

This toy example yields `y.i` = 42, `y.j` = 1.

EXAMPLE (13)



The images from above visualize the basic graph operations introduced in this chapter, rendered by the debugger (cf. Chapter 27).
The figure to the left shows how the pattern of example rule `r` from below is matched in a tiny example graph (built from nodes of types `Node` and `N`, the latter bearing an attribute `i` of type `int`, and edges of type `Edge`). Matched nodes are colored in light-brown, in contrast to the default yellow of unmatched nodes `$1` and `$3`, and have the name of the pattern element that was mapped to them inscribed, here we see pattern node `n` matched to `$0`, and pattern node `m` matched to `$2`. The unmatched edges `$4` and `$7` are shown with a thin arrow, the matched edge `$5` with a thick arrow, in addition it has the auto-assigned name `_edge0` inscribed. The attribute condition `n.i == 42` is satisfied as can be seen from the infotag shown at node `$0`. Note that `m` could have been matched to `$1` instead.
The figure to the right shows how the modifications of the rewrite part are applied at the matched spot. Node `$2` bound to `m` is deleted, this is shown with gray color ("fade to gray"); according to SPO semantics, it drags edges `$4` and `$5` with it. Node `$8` is created for pattern node `o`, and edge `$9` for pattern edge `_edge1` (which is again an auto-generated name, which is what GrGen.NET does under the covers for anonymous pattern elements.) Graph element creation is visualized in green color. Additionally, the attribute assignment `n.i = n.i + 1` was carried out.

```
1  rule r
2  {
3    n:N --> m:Node;
4    if{ n.i == 42; }
5
6    modify {
7      delete(m);
8      n --> o:Node;
9      eval { n.i = n.i + 1; }
10   }
11 }
```

CHAPTER 6

BASIC TYPES AND ATTRIBUTE EVALUATION EXPRESSIONS

6.1 Built-In Types

Besides user-defined node types, edge types, and enumeration types (as introduced in Chapter 4), GrGen.NET supports the built-in primitive types in Table 6.1 (and built-in generic types, cf. 15). The exact type format is backend specific. The LGSPBackend maps the GrGen.NET primitive types to the corresponding C# primitive types.

`boolean`	Covers the values `true` and `false`
`byte, short, int, long`	A signed integer, with 8 bits, with 16 bits, with 32 bits, with 64 bits
`float, double`	A floating-point number, with single precision, with double precision
`string`	A character sequence of arbitrary length
`object`	Contains a .NET object

Table 6.1: GrGen.NET built-in primitive types

to \ from	`enum`	`boolean`	`int`	`double`	`string`	`object`
`enum`	=/—					
`boolean`		=				
`int`	implicit		=	`(int)`		
`double`	implicit		implicit	=		
`string`	implicit	implicit	implicit	implicit	=	implicit
`object`	`(object)`	`(object)`	`(object)`	`(object)`	`(object)`	=

Table 6.2: GrGen.NET type casts

Table 6.2 lists GrGen.NET's implicit type casts and the allowed explicit type casts. Of course you are free to express an implicit type cast by an explicit type cast as well as "cast" a type to itself. The `int` is the default integer type, in the table it stands for all the integer types. The `double` is the default floating point type, in the table it stands for all the floating-point types. The integer types are implicitly cast upwards from smaller to larger types (`byte < short < int < long`), whereas a downcast requires an explicit cast. The `byte` and `short` types are not used in computations, they are cast up to `int` (or `long` if required by the context.) The floating point types are implicitly cast upwards (`float < double`), and require an explicit cast downwards, too. As specified by the table, integer numbers are automatically cast to floating point numbers, and castable with an explicit cast vice versa. According to the table neither implicit nor explicit casts from `int` to any enum type are allowed. This is because the range of an enum type is very sparse in general. For the

same reason implicit and explicit casts between enum types are also forbidden. Thus, enum values can only be assigned to attributes having the same enum type. A cast of an enum value to a string value will return the declared name of the enum value. A cast of an object value to a string value will return "null" or it will call the `toString()` method of the .NET object. Everything is implicitly cast to `string` to enable concise text output without the need for boilerplate casting. Be careful with assignments of objects: GrGen.NET does not know your .NET type hierarchy and therefore it cannot check two objects for type compatibility. Objects of type object are not very useful for GrGen.NET processing and the im/exporters can't handle them, but they can be used on the API level.

EXAMPLE (14)

- Allowed:
  `x.myfloat = x.myint; x.mydouble = (float) x.myint;`
  `x.mystring = (string) x.mybool;`
- Forbidden:
  `x.myfloat = x.mydouble;` and `x.myint = (int) x.mybool;`
  `MyEnum1 = (MyEnum1Type) int;` and `MyEnum2 = (MyEnum2Type) MyEnum1;` where `myenum1` and `myenum2` are different enum types.

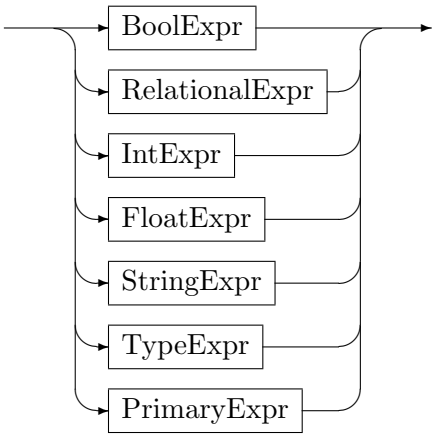
NOTE (19)

Unlike an `eval` part (which must not contain assignments to node or edge attributes) the declaration of an enum type can contain assignments of `int` values to enum items (see Section 4.2). The reason is that the range of an enum type is just defined in that context.

6.2 Expressions

GrGen.NET supports numerous operations on the entities of the types introduced above, which are organized into left associative expressions (so $a \otimes b \otimes c$ is evaluated as $(a \otimes b) \otimes c$). In the following they will be explained with their semantics and relative priorities one type after another in the order of the rail diagram below.

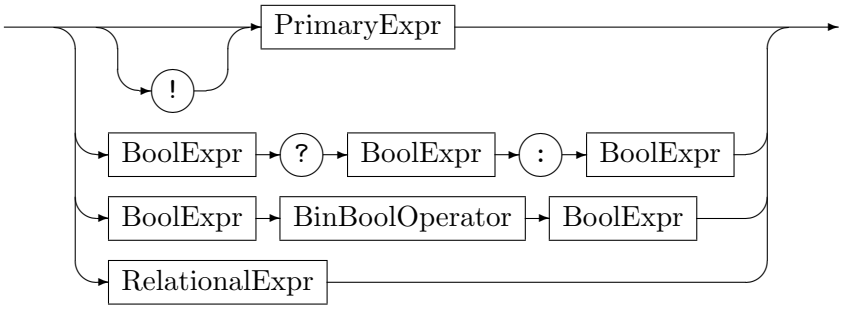
Expression



6.3 Boolean Expressions

The boolean expressions combine boolean values with logical operations. They bind more weakly than the relational expressions which bind more weakly than the other expressions.

BoolExpr



The unary `!` operator negates a Boolean. The binary *BinBoolOperator* is one of the operators in Table 6.3. The ternary `?` operator is a simple if-then-else: If the first *BoolExpr* is evaluated

`^`	Logical XOR. True, iff either the first or the second Boolean expression is true.
`&&` `||`	Logical AND and OR. Lazy evaluation.
`&` `|`	Logical AND and OR. Strict evaluation.

Table 6.3: Binary Boolean operators, in ascending order of precedence

to `true`, the operator returns the second *BoolExpr*, otherwise it returns the third *BoolExpr*.

6.4 Relational Expressions

The relational expressions compare entities of different kinds, mapping them to the type boolean. They are an intermediary that allows obtaining the boolean type required by the decisions from the other types. They bind stronger than the boolean expressions but weaker than all the other non-boolean expressions.

RelationalExpr



The *CompareOperator* is one of the following operators:

`<` `<=` `==` `!=` `>=` `>` `~~` `in`

Their semantics are type dependent.

For arithmetic expressions on `int` and `float` or `double` types the semantics is given by Table 18.1 (by implicit casting they can also be used with all enum types).

For expressions on `string` types lexicographic order is used for comparisons, the exact semantics is given by Table 6.5.

`Boolean` types and `object` types support only the `==` and the `!=` operators; on boolean values they denote equivalence and antivalence, and on object types they tell whether the references are the same, thus the objects identical.

For type expressions the semantics of compare operators are given by table 6.6, the rule to remember is: types grow larger with extension/refinement. An example is given in 6.7.

`A == B`	True, iff A is the same number as B.
`A != B`	True, iff A is a different number than B.
`A <  B`	True, iff A is smaller than and not equal B.
`A >  B`	True, iff A is greater than and not equal B.
`A <= B`	True, iff A is smaller than (or equal) B.
`A >= B`	True, iff A is greater than (or equal) B.

Table 6.4: Compare operators on arithmetic expressions

`A == B`	True, iff A is the same string as B.
`A != B`	True, iff A is not the same string as B.
`A <  B`	True, iff the first character where A and B differ is smaller for A, or A is a prefix of B.
`A >  B`	True, iff the first character where A and B differ is smaller for B, or B is a prefix of A.
`A <= B`	True, iff the first character where A and B differ is smaller for A, or A is a prefix of B, or A is the same as B.
`A >= B`	True, iff the first character where A and B differ is smaller for B, or B is a prefix of A, or B is the same as A.

Table 6.5: Compare operators on string expressions

`A == B`	True, iff A and B are identical. Different types in a type hierarchy are *not* identical.
`A != B`	True, iff A and B are not identical.
`A <  B`	True, iff A is a supertype of B, but A and B are not identical.
`A >  B`	True, iff A is a subtype of B, but A and B are not identical.
`A <= B`	True, iff A is a supertype of B or A and B are identical.
`A >= B`	True, iff A is a subtype of B or A and B are identical.

Table 6.6: Compare operators on type expressions

NOTE (20)

`A < B` corresponds to the direction of the arrow in an UML class diagram.

NOTE (21)

`Node` and `Edge` are the least specific, thus bottom elements $\perp$ of the type hierarchy, i.e. the following holds:

- $\forall n \in Types_{Node} : Node <= n$
- $\forall e \in Types_{Edge} : Edge <= e$

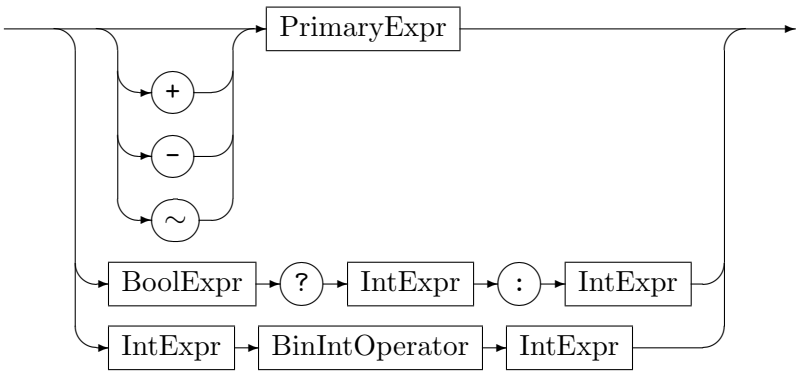
The main use of the `in` operator is for checking membership in a container type, see Chapter 15.

The `~~` operator allows to compare graphs, supplementing `==` and `!=` in this role, see Chapter 16. Furthermore, it also allows to compare class objects and transient class objects for deep attribute value equality (preliminary) instead of only reference identity, see Subsection 18.1.3.4, and is available as an extension in this rôle for graph element and container types (as well as external object types).

6.5 Arithmetic and Bitwise Expressions

The arithmetic and bitwise expressions combine integer and floating point values with the arithmetic operations usually available in programming languages and integer values with bitwise logical operations (interpreting integer values as bit-vectors).

IntExpr



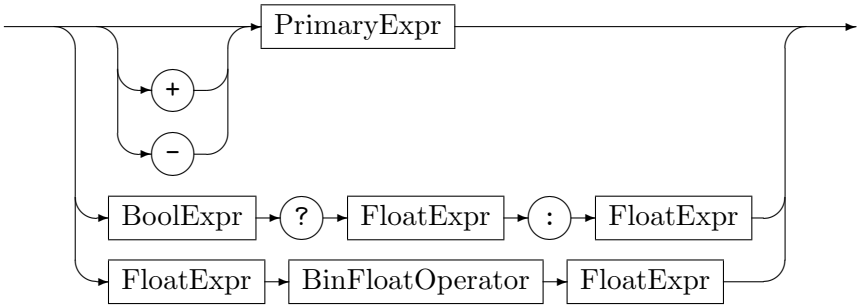
The $\sim$ operator is the bitwise complement. That means every bit of an integer value will be flipped. The `?` operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *IntExpr*, otherwise it returns the second *IntExpr*. The *BinIntOperator* is one of the operators in Table 6.7.

If one operand is `long` the operation is carried out with 64 Bits, otherwise the operation is carried out with 32 Bits, i.e. `int`-sized — even if all the operands are of type `byte` or `short`.

^ & |	Bitwise XOR, AND and OR
<< >> >>>	Bitwise shift left, bitwise shift right and bitwise shift right prepending zero bits (unsigned mode)
+ -	Addition and subtraction
* / %	Multiplication, integer division, and modulo

Table 6.7: Binary integer operators, in ascending order of precedence

FloatExpr



The `?` operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *FloatExpr*, otherwise it returns the second *FloatExpr*. The *BinFloatOperator* is one of the operators in Table 6.8.

+ -	Addition and subtraction
* / %	Multiplication, division and modulo

Table 6.8: Binary float operators, in ascending order of precedence

NOTE (22)

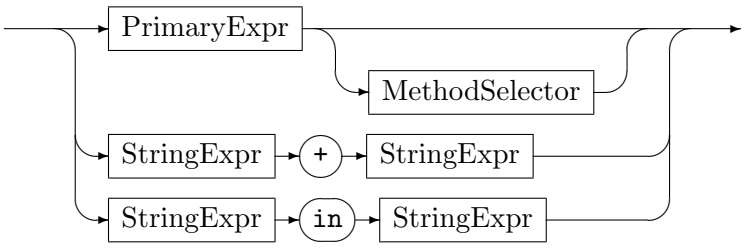
The `%` operator on float values works analogous to the integer modulo operator. For instance `4.5 % 2.3 == 2.2`.

If one operand is `double` the operation is carried out with 64 Bits, otherwise the operation is carried out with 32 Bits, i.e. `float`-sized.

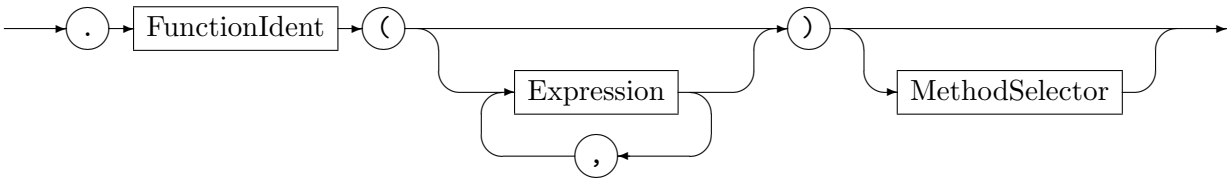
6.6 String Expressions

String expressions combine string values by string operations, with integer numbers used as helpers to denote positions in the strings (and giving the result of length counting).

StringExpr



MethodSelector



The operator `+` concatenates two strings. The operator `in` returns `true` if the left string is contained in the right string (`false` otherwise).

Several operations on strings – and other types – are available in method call notation. The methods given with the *MethodSelector* may be chained (the function methods – all string methods are function methods, see 18.2.1 for more on this topic we preview here, but the container types introduced in Chapter 16.1 also offer procedure methods (cf. 18.2.2), those are statements and not expressions and cannot be chained). The available string methods are:

`.length()`
: returns length of string, as `int`

`.startsWith(strToSearchFor)`
: returns whether the string begins with `strToSearchFor:string`, as `boolean`

`.endsWith(strToSearchFor)`
: returns whether the string ends with `strToSearchFor:string`, as `boolean`

`.indexOf(strToSearchFor)`
: returns first position where `strToSearchFor:string` appears at, as `int`, or -1 if not found

`.indexOf(strToSearchFor, startIndex)`
: returns first position where `strToSearchFor:string` appears at (moving to the end), when we start the search for it at string position `startIndex:int`, as `int`, or -1 if not found

`.lastIndexOf(strToSearchFor)`
: returns last position `strToSearchFor:string` appears at, as `int`, or -1 if not found

`.lastIndexOf(strToSearchFor, startIndex)`
: returns last position `strToSearchFor:string` appears at (moving to the begin), when we start the search for it at string position `startIndex:int`, as `int`, or -1 if not found

`.substring(startIndex, length)`
: returns substring of given `length:int` from `startIndex:int` on

`.substring(startIndex)`
: returns substring from `startIndex:int` on (of full remaining length)

.replace(startIndex, length, replaceStr)
: returns string with substring from `startIndex:int` on of given `length:int` replaced by `replaceStr:int`

.toLower()
: returns a lowercase version of the string

.toUpper()
: returns an uppercase version of the string

.asArray(separator)
: returns the original string exploded to an array of substrings (`array<string>`), at each occurrence of the `separator:string`; the separators are not included in the array. If separator is empty, you get the original string exploded to an array of single-character strings (an `array<string>` contains an `asString` method for reversal).

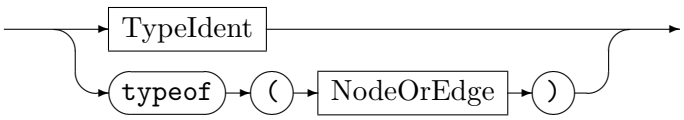
EXAMPLE (15)

For `n.str == "foo bar foo"` the operations yield

```
n.str.length()==11
n.str.startsWith("foo")==true
n.str.endsWith("foo")==true
n.str.indexOf("foo")==0
n.str.indexOf("foo", 1)==8
n.str.lastIndexOf("foo")==8
n.str.substring(4,3)=="bar"
n.str.substring(4)=="bar foo"
n.str.replace(4,3,"foo")=="foo foo foo"
n.str.toUpper()=="FOO BAR FOO"
n.str.asArray(" ")==array<string>["foo","bar","foo"]
```

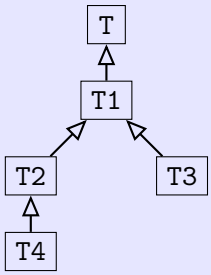
6.7 Type Expressions

TypeExpr



A type expression identifies a type (and—in terms of matching—also its subtypes). A type expression is either a type identifier itself or the type of a graph element. The type expression `typeof(x)` stands for the type of the host graph element `x` is actually bound to.

EXAMPLE (16)

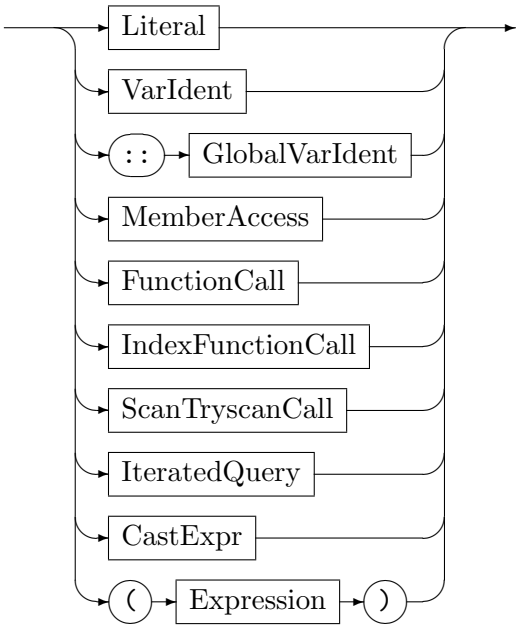


The expression `typeof(x)<=T2` applied to the type hierarchy on the left side yields `true` if `x` is a graph element of type `T` or `T1` or `T2`. The expression `typeof(x)>T2` only yields `true` for `x` being a graph element of type `T4`. The expression `T1<T3` always yields `true`.

6.8 Primary Expressions

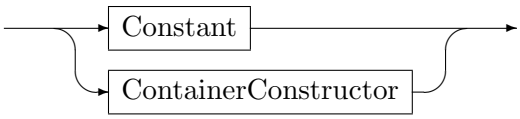
After we've seen all the ways to combine expressions, finally we'll have a look at the atoms the expressions are built of.

PrimaryExpr

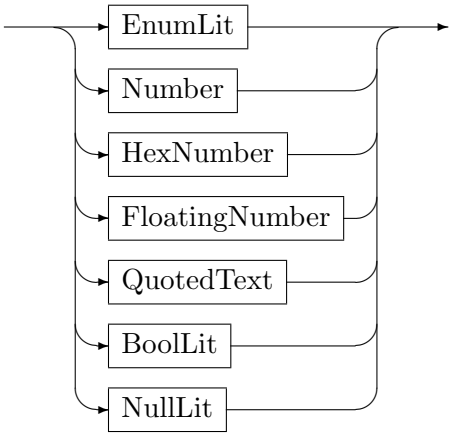


The *Literal*s comprise the *Constants* of basic type and the *ContainerConstructors*, more exactly the *SetConstructor*, the *MapConstructor*, the *ArrayConstructor* and the *DequeConstructor*, cf. Chapter 15.

Literal



Constant



The Constants are:

EnumLit
: Is the value of an enum type, given in notation `EnumType '::' EnumValue`.

Number
: Is a `byte` or `short` or `int` or `long` number in decimal notation without decimal point, postfixed by `y` or `Y` for `byte`, postfixed by `s` or `S` for `short`, postfixed by `l` or `L` for `long`, or of `int` type if not postfixed.

HexNumber
: Is a `byte` or `short` or `int` or `long` number in hexadecimal notation starting with `0x`, the different types are distinguished by the suffix as for a decimal notation number.

FloatingNumber
: Is a `float` or `double` number in decimal notation with decimal point, postfixed by `f` or `F` for `float`, maybe postfixed by `d` or `D` for `double`.

QuotedText
: Is a string constant. It consists of a sequence of characters, enclosed by double quotes.

BoolLit
: Is a constant of boolean type, i.e. one of the literals `true` or `false`.

NullLit
: Is the one constant of object type, the literal `null`.

EXAMPLE (17)

Some examples of literals:

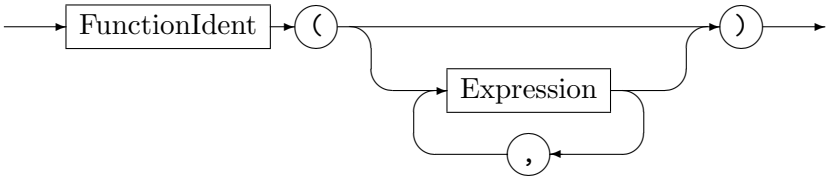
```
Apple::ToffeeApple // an enum literal
42y // an integer number in decimal notation of byte type
42s // an integer number in decimal notation of short type
42 // an integer number in decimal notation of int type
42L // an integer number in decimal notation of long type
0x7eadbeef // an integer number in hexadecimal notation of int type
0xdeadbeefL // an integer number in hexadecimal notation of long type
3.14159 // a double number
3.14159f // a float number
"ve␣rule␣and␣0wn␣ze␣world" // a text literal
true // a bool literal
null // the object literal
```

MemberAccess



The member access `n.a` returns the value of the attribute `a` of the graph element `n`. A *VarIdent* returns the value of the denoted variable, a *GlobalVarIdent* `::x` returns the value of the graph-global variable `x`.

FunctionCall



A function call employs an external (attribute evaluation) function (cf. 34.2), or an (internal) user-defined function (see 14.5.1, there you find the syntax definition we replicate here as a preview), or a built-in function (you find more on the *IndexFunctionCall* in 30.4 and on the *ScanTryscanCall* in 13.3). Available built-in functions are e.g. `nameof`, `uniqueof`, and `random`.

The `nameof` function returns the name of the given graph element as `string` (persistent name, see example 158, and Section 30.6 for an explanation of the name index; graphs elements of pure `LGSPGraph`s bear no name, then the query fails, but normally you are using `LGSPNamedGraph`s). Besides nodes or edges, subgraphs may be given, then the name of the subgraph is returned. If no argument is given, the name of the graph (the current host graph) is returned.

The `uniqueof` function returns the unique id of the given graph element as `int`; see Section 30.7 for an explanation of the uniqueness constraint/index – you only receive a unique id in case uniqueness was declared in the model. Besides nodes or edges, class objects may be given, the `uniqueof` function then returns the unique id of the given class object as `long` – in contrast to the former types, no declaration in the model is needed. Furthermore, subgraphs may be given, then the unique id of the subgraph is returned (as `int`). If no argument is given, the unique id of the graph (the current host graph) is returned.

The `random` function returns a double random value in between 0.0 and 1.0 if called without an argument, or, if an integer argument value is given, an integer random value in between 0 and the given value, excluding the value itself.

A function call may also employ a function from a package, the name `FunctionIdent` is then prefixed by the `Ident` of the package, separated by a double colon `::`. A user-defined function from a user-defined package may be called, or one of the following built-in functions contained in the built-in package `Math` or in the built-in package `Time` (for more on packages see 18.3.2; other built-in packages are `File` and `Transaction`, cf. Chapter 16, as well as `Debug`, cf. Chapter 27, the latter two only contain procedures, though):

`Math::min(.,.)`
: returns the smaller of the two argument values, which must be of the same numeric type (i.e. both either `byte` or `short` or `int` or `long` or `float` or `double`)

`Math::max(.,.)`
: returns the greater of the two argument values, which must be of the same numeric type (i.e. both either `byte` or `short` or `int` or `long` or `float` or `double`)

`Math::abs(.)`
: returns the absolute value of the argument, which must be of numeric type (i.e. `byte` or `short` or `int` or `long` or `float` or `double`)

Math::ceil(.)
: returns the ceiling of the argument, which must be of type `double`

Math::floor(.)
: returns the floor of the argument, which must be of type `double`

Math::round(.)
: returns the argument rounded, which must be of type `double`

Math::truncate(.)
: returns the argument truncated, which must be of type `double`

Math::sqr(.)
: returns the square of the first argument, it must be of type `double`

Math::sqrt(.)
: returns the square root of the first argument, it must be of type `double`

Math::pow(.,.)
: returns the first argument value to the power of the second value; both must be of type `double`

Math::pow(.)
: returns e to the power of the argument value, which must be of type `double`

Math::log(.,.)
: returns the logarithm of the first argument value regarding the base given by the second value; both must be of type `double`

Math::log(.)
: returns the logarithm of the argument value that must be of type `double` regarding the base e

Math::sgn(.)
: returns the signum of the argument, which must be of type `double` (-1 if negative, 1 if positive, 0 if zero)

Math::sin(.)
: returns the sine of the argument, which must be of type `double`

Math::cos(.)
: returns the cosine of the argument, which must be of type `double`

Math::tan(.)
: returns the tangent of the argument, which must be of type `double`

Math::arcsin(.)
: returns the inverse of the sine for the argument, which must be of type `double`

Math::arccos(.)
: returns the inverse of the cosine for the argument, which must be of type `double`

Math::arctan(.)
: returns the inverse of the tangent for the argument, which must be of type `double`

Further null-ary functions, i.e. constants are:

Math::pi()
: returns the constant π, of type `double`

Math::e()
: returns the constant e, of type `double`

Math::byteMin()
: returns the smallest number supported by type `byte`

Math::byteMax()
: returns the largest number supported by type `byte`

Math::shortMin()
: returns the smallest number supported by type `short`

Math::shortMax()
: returns the largest number supported by type `short`

Math::intMin()
: returns the smallest number supported by type `int`

Math::intMax()
: returns the largest number supported by type `int`

Math::longMin()
: returns the smallest number supported by type `long`

Math::longMax()
: returns the largest number supported by type `long`

Math::floatMin()
: returns the smallest number supported by type `float`

Math::floatMax()
: returns the largest number supported by type `float`

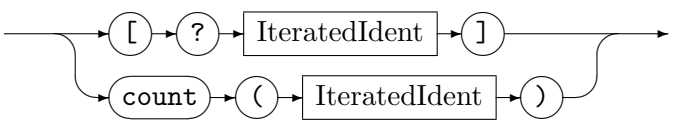
Math::doubleMin()
: returns the smallest number supported by type `double`

Math::doubleMax()
: returns the largest number supported by type `double`

Another null-ary but not constant function (an environment query) is:

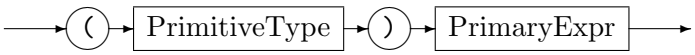
Time::now()
: returns the current UTC time as `long`, given as windows file time (i.e. 100ns ticks since 1601-01-01)

IteratedQuery



The iterated query expression `[?it]` returns an `array` of `match<r.it>` where `it` is the name of a named iterated block (cf. 7.5) and `r` the name of the rule that contains it. You may apply the `extract<member>` method and then the array accumulation methods on it to get a value for further processing (cf. 20.1). Alternatively, you may directly obtain the `count` of iterated matches.

CastExpr



The cast expression returns a value cast to a new type, it is specified by a type in parentheses before the expression computing the original value. Another primary expression is an arbitrary expression enclosed in parentheses.

6.9 Operator Priorities

The priorities of all available operators are shown in ascending order in the table below, the dots mark the positions of the operands, the commas separate the operators available on the respective priority level.

01	`. ? . : .`
02	`. || .`
03	`. && .`
04	`. | .`
05	`. ^ .`
06	`. & .`
07	`. \ .`
08	`. ==,!=,~~ .`
09	`. <,<=,>,>=,in .`
10	`. <<,>>,>>> .`
11	`. +,- .`
12	`. *,%,/ .`
13	`~,!,-,+ .`

Table 6.9: All operators, in ascending order of precedence

CHAPTER 7

NESTED PATTERNS

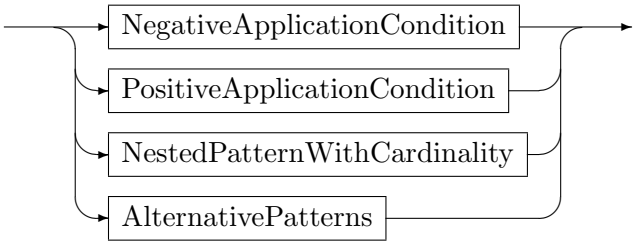
The extension of the rule set language described in chapter 5 by nested patterns described in this chapter 7 and the following chapter 8 greatly enhances the flexibility and expressiveness of pattern matching and rewriting. The following patterns that allow matching a simplified abstract syntax tree give a rough picture of the language of nested and subpatterns:

EXAMPLE (18)

```
1  test method
2  {
3    m:Method <-- n:Name; // signature of method consisting of name
4    iterated { // and 0-n parameters
5      m <-- :Variable;
6    }
7
8    :AssignmentList(m); // body consisting of a list of assignment statements
9  }
10
11 pattern AssignmentList(prev:Node)
12 {
13   optional { // nothing or a linked assignment and again a list
14     prev --> a:Assign; // assignment node
15     a -:target-> v:Variable; // which has a variable as target
16     :Expression(a);  // and an expression which defines the left hand side
17     :AssignmentList(a); // next one, plz
18   }
19 }
20
21 pattern Expression(root:Expr)
22 {
23   alternative { // expression may be
24     Binary { // a binary expression of an operator and two expresions
25       root <-- expr1:Expr;
26       :Expression(expr1);
27       root <-- expr2:Expr;
28       :Expression(expr2);
29       root <-- :Operator;
30     }
31     Unary { // or a unary expression which is a variable (reading it)
32       root <-- v:Variable;
33     }
34   }
35 }
```

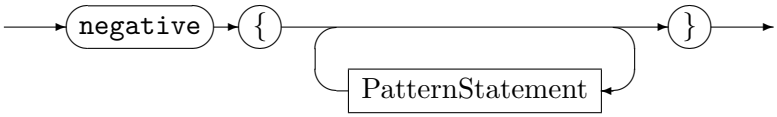
Until now we have seen rules and tests with one left hand side static pattern specification in a direct 1:1 correspondence with its dynamic match in the host graph on a successful application. From now on we will increase the expressiveness of the pattern language, and dependent on it the rewrite language, to describe much more finely and flexibly what patterns to accept. This will be done by pattern specifications built up from multiple static pattern piece specifications, where the pieces may be matched dynamically zero, one, or multiple times, or are forbidden to exist for the entire pattern to be matched. These rule set language constructs can be split into nested patterns (negative application condition, positive application condition, nested pattern with cardinality, alternative patterns) and subpatterns (subpattern declaration and subpattern entity declaration, subrule declaration and usage), in this chapter we will focus on the nested patterns:

NestedPattern



7.1 Negative Application Condition (NAC)

NegativeApplicationCondition



With negative application conditions (keyword `negative`) we can specify graph patterns which forbid the application of a rule if any of them is present in the host graph (cf. [Sza05]). NACs possess a scope of their own, i.e. names defined within a NAC do not exist outside the NAC. Identifiers from surrounding scopes must not be redefined. If they are not explicitly mentioned, the NAC gets matched independent from them, i.e. elements inside a negative are `hom(everything from the outside)` by default. But referencing the element from the outside within the negative pattern causes it to get matched isomorphically/distinct to the other elements in the negative pattern. This is a bit unintuitive if you think of extending the pattern by negative elements, but cleaner and more powerful: just think of NACs to simply specify a pattern which should not be in the graph, with the possibility of forcing elements to be the same as in the enclosing pattern by name equality.

EXAMPLE (19)

We specify a variable which is not badly typed, i.e. a node `x` of type `Variable` which must not be the target of an edge of type `type` with a source node of type `BadType`:

```
1  x:Variable;
2  negative {
3    x <-:type- :BadType;
4  }
```

Because NACs have their "own" binding, using NACs leads to specifications which might look a bit redundant.

EXAMPLE (20)

Let's check the singleton condition, meaning there's exactly one node of the type to check, for the type `RootNamespace`. The following specification is *wrong* (it will never return a match):

```
1  x:RootNamespace;
2  negative {
3    y:RootNamespace;
4  }
```

Instead we have to specify the *complete* forbidden pattern inside the NAC. This is done by:

```
1  x:RootNamespace;
2  negative {
3    x;
4    y:RootNamespace; // now it is ensured that y must be different from x
5  }
```

Btw: the `x;` is not a special construct, it's a normal graphlet (cf. 5.1.1).

If there are several patterns which should not occur, use several negatives. If there are exceptions to the forbidden pattern, use nested negatives. As a straight-forward generalization of negatives within positive patterns, negatives may get nested to an arbitrary depth. Matching of the nested negative pattern causes the matching of the nesting pattern to fail.

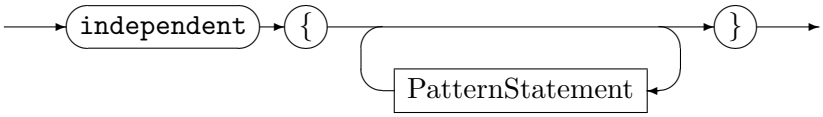
EXAMPLE (21)

A fabricated example using parallel as well as nested `negatives`:

```
1  test onlyOneChildOrAllChildrenHaveExactlyOneCommonChild
2  {
3    root:Class;
4    negative {
5      root -:extending-> :Class; // root does not extend another class
6    }
7    root <-:extending- c1:Class; // a class c1 extends root
8    negative {
9      c1;
10     root <-:extending- c2:Class; // there is no c2 which extends root
11     negative {
12       c1 <-:extending- child:Class -:extending-> c2; // except c1 and c2 have a common child
13       negative { // and c1 has no further children
14         child;
15         c1 <-:extending- :Class;
16       }
17       negative { // and c2 has no further children
18         child;
19         c2 <-:extending- :Class;
20       }
21     }
22   }
23 }
```

7.2 Positive Application Condition (PAC)

PositiveApplicationCondition



With positive application conditions (keyword `independent`) we can specify graph patterns which, in contrast to negative application conditions, must be present in the host graph to cause the matching of the enclosing pattern to succeed. Together with NACs they share the property of opening a scope, with elements being independent from the surrounding scope (i.e. a host graph element can easily get matched to a pattern element and a PAC element with a different name, unless the pattern element is referenced in the PAC). They are used to improve the logical structure of rules by separating a pure condition from the main pattern of the rule amenable to rewriting.

They are used when all matches of a pattern are wanted, and a part of that pattern is available in the graph multiple times, but should not cause combinatorially additional matches; then the `independent` can be used to check only for the existence of that part, limiting the all-matching to the core pattern (when only one match of a pattern is wanted, with a part being available multiple times in the graph, its not of importance which of the parts being available multiple times gets matched). They are needed if subpatterns want to match elements which were already matched during the subpattern derivation (and are thus isomorphy-constraint-locked, which would prevent them from getting matched another time – a situation which could easily occur because the full pattern is not statically visible).

An independent can be seen as a conjunction with an existence-quantified pattern, a negative as a conjunction with the negation of an existence-quantified pattern (a main pattern can be used as an existence-quantified or all-quantified pattern, which is decided by sequence all-bracketing, and it rewrites the matched spots, in contrast to the application condition patterns); the multiple/iterated patterns introduced below are also a kind of all-bracketing, but with isomorphy-constraint locking in between the matches.

EXAMPLE (22)

A further fabricated example rather giving the intention using `independent` patterns to check some conditions with only the main pattern available to rewriting:

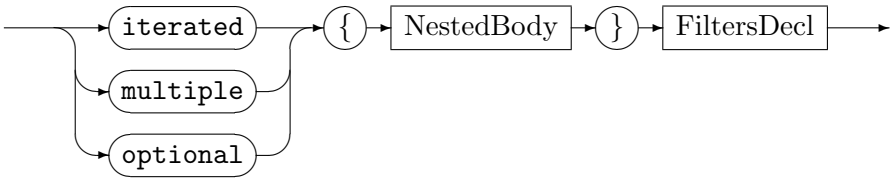
```
rule moveMethod
{
  c:Class --> m:Method;
  csub -:extending-> c;
  csub:Class -e:Edge-> msub:Method;

  independent {
    // a complicated pattern to find out that m and msub have same signatures
  }
  independent {
    // a complicated pattern to find out that msub is only using variables available in c
  }
  independent {
    // a complicated pattern to find out that m is not used
  }

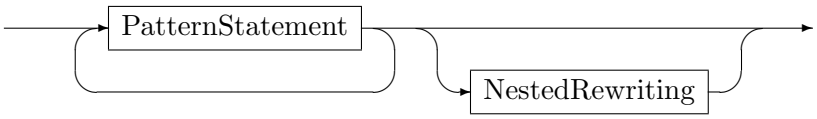
  modify { // move method upwards
    delete(m);
    delete(e);
    c --> msub;
  }
}
```

7.3 Pattern Cardinality (iterated / multiple / optional)

NestedPatternWithCardinality



NestedBody



The blocks allow specifying how often the nested pattern – opening a scope – is to be matched. Matching will be carried out eagerly, i.e. if the construct is not limiting the number of matches and a further match is possible it will be done. The *NestedRewriting* will be explained in Section 9.1. You find more on the *FiltersDecl* in Chapter 17, usage of the filters is explained in 10.1.2.

The Iterated Block

The iterated block is matching the contained subpattern as often as possible, succeeding even in the case the contained pattern is not available (thus it will never fail). It was included in the language to allow for matching breadth-splitting structures, as in capturing all methods of a class in a program graph.

EXAMPLE (23)

```
1 test methods
2 {
3   c:Class;
4   iterated {
5     c --> m:Method;
6   }
7 }
```

The Multiple Block

The multiple block is working like the iterated block, but expects the contained subpattern to be available at least once; if it is not, matching of the multiple block and thus its enclosing pattern fails.

EXAMPLE (24)

```
1 test oneOrMoreMethods
2 {
3   c:Class;
4   multiple {
5     c --> m:Method;
6   }
7 }
```

The Optional Block

The optional block is working like the iterated block, but matches the contained subpattern at most once; further occurrences of the subpattern are left unmatched. If the nested pattern is available, it will get matched, otherwise it won't; matching of the optional block will succeed either way.

EXAMPLE (25)

```
1 test variableMaybeInitialized
2 {
3   v:Variable; // match variable
4   optional { // and an initialization with a different one if available
5     v <-- otherV:Variable;
6   }
7 }
```

Iteration Breaking

If an application condition inside an iteration block fails, then that potential match of the iterated pattern is thrown away and matching continues trying to find further matches. Sometimes a different behaviour is wanted, with an application condition terminating the iteration and causing it to fail. This would allow to check with a single rule that "every

pattern X must also satisfy Y" holds. This behaviour is supported with the `break` keyword prepended to an application condition, transforming it into an iteration breaking condition.

EXAMPLE (26)

If the `negative` matches, not only the current iteration instance is prevented from matching, but the entire `iterated` (and thus the `test`) is failing to match:

```
test forEachXMustNotBeTheCaseY
{
  iterated {
    <X>;
    break negative {
      <Y>;
    }
  }
}
```

If the `independent` does not match, not only the current iteration instance is prevented from matching, but the entire `iterated` (and thus the `test`) is failing to match:

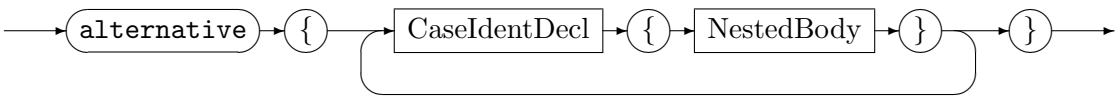
```
test forEachXMustBeTheCaseY
{
  iterated {
    <X>;
    break independent {
      <Y>;
    }
  }
}
```

NOTE (23)

Pattern cardinality constructs are match/rewrite-all enumeration blockers. For every pattern instance, the iterated/... yields only one match, even if in all mode (used in/from all-bracketed rules).

7.4 Alternative Patterns

AlternativePatterns



With the alternative block you can specify several nested alternative patterns. One of them must get matched for the matching of the alternative (and thus its directly nesting pattern) to succeed, and only one of them is matched per match of the alternative / overall pattern. The order of matching the alternative patterns is unspecified, especially it is not guaranteed that a case gets matched before the case textually following – if you want to ensure that a case cannot get matched if another case could be matched, you must explicitly prevent that from happening by adding negatives to the cases. In contrast to the iterated which locally matches everything available and inserts this combined match into the current

match, the alternative decides for one case match which it inserts into the current match tree, ignoring other possible matches by other cases.

EXAMPLE (27)

```
1  test feature(c:Class)
2  {
3    alternative // a feature of the class is either
4    {
5      FeatureMethod { // a method
6        c --> :Method;
7      }
8      FeatureVariable { // or a variable
9        c --> :Variable;
10     }
11     FeatureConstant { // or a constant
12       c ---> :Constant;
13     }
14   }
15 }
```

EXAMPLE (28)

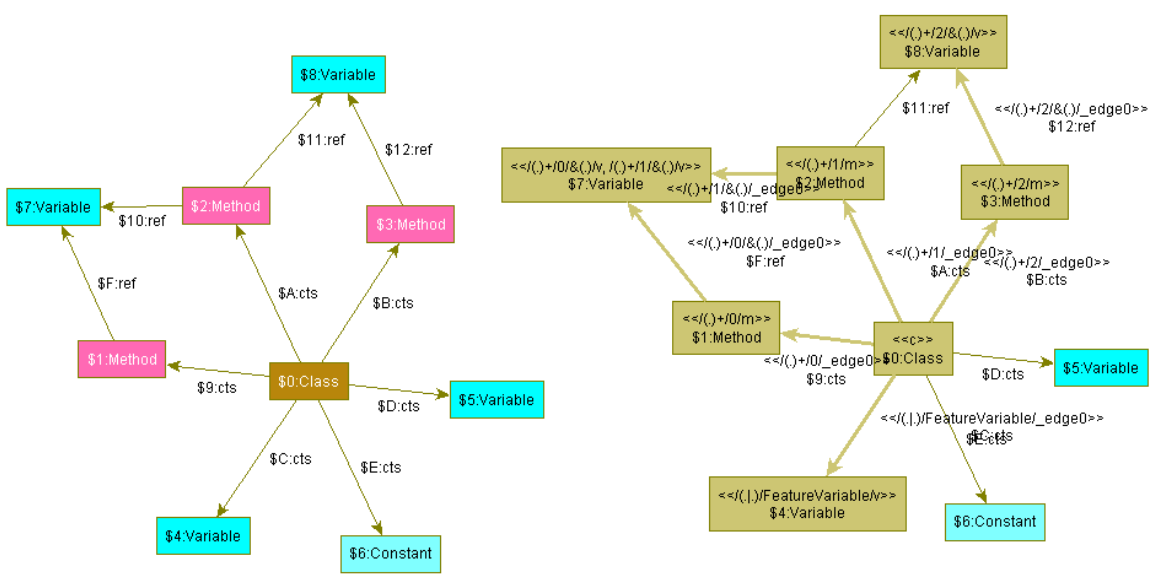
```
1  test variableMaybeInitialized
2  {
3    v:Variable; // match variable
4    alternative { // and an initialization with a different one if available
5      Empty {
6        // the empty pattern matches always
7        negative { // so prevent it to match if initialization is available
8          v <-- otherV:Variable;
9        }
10     }
11     Initialized { // initialization
12       v <-- otherV:Variable;
13     }
14   }
15 }
```

EXAMPLE (29)

When working with the subtyping hierarchy one may be interested in matching in a first step an abstract base class, specifying the rewriting behaviour for this base class once, and in a refinement step in an alternative the different possible subtypes, then being able to access their specific attributes and being able to give different additional rewrite parts.

```
1  test refineFeature
2  {
3    f:Feature; // match abstract base class Feature
4
5    alternative {
6      Variable {
7        v:Variable<f>; // try to cast to concrete Variable, if succeeds we can access the
             Variable attributes
8      }
9      Method {
10       m:Method<f>; // try to cast to a concrete Method, if succeeds we can access the Method
             attributes
11     }
12   }
13
14   modify {
15     // do stuff common to a Feature here
16   }
17 }
```

EXAMPLE (30)



The image from above shows an example of how a structure is matched by several pieces. At the left, the plain host graph, at the right with the match of `classWithStuff` inscribed.

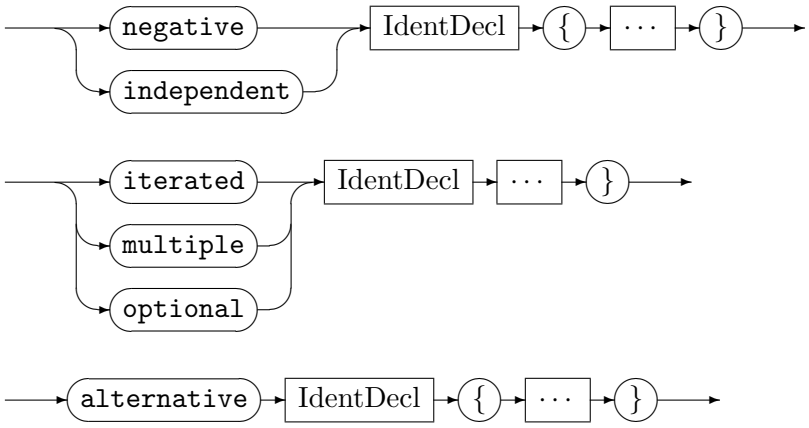
```
1  test classWithStuff
2  {
3    c:Class; // get a class
4
5    multiple { // get all of its methods, expect at least one
6      c --> m:Method;
7        independent { // expect an argument variable
8          m -:ref-> v:Variable;
9        }
10   }
11
12   alternative // additionally, match one of either
13   {
14     FeatureVariable { // a variable
15       c --> v:Variable;
16     }
17     FeatureConstant { // or a constant
18       c --> ct:Constant;
19     }
20   }
21 }
```

The example matches a class node `c` and its contained methods with a `multiple` pattern – note the 3 instances of the pattern in the host graph getting captured by the single pattern description (`/(.)+/1/m` reads as multiple pattern (see Table 8.1 explaining the regular expression syntax), match instance `1` (starting at `0`), pattern element `m`.)
Additionally, one of the alternative cases of a variable or a constant is matched – here we got two potential patterns, from which only one is matched, capturing node `$4` out of the potential targets `$4`, `$5`, `$6` (`/(.|.)/FeatureVariable/v` reads as alternative, case `FeatureVariable`, pattern element `v`.)
The independent that is used to reference a parameter variable `v` from method `m` allows to re-match an already matched variable that would be otherwise shut out from getting matched again (happened here with node `$7`, matched from instances 0 and 1 of the multiple pattern; a graph element is annotated with all matching pattern elements).

7.5 Named nested patterns

Nested patterns can be given names, this is done with the following syntax:

NamedNestedPattern



As the already introduced syntax remains valid, in aggregate we extend the nested patterns with optional names to form named nested pattern.

Iterateds named this way can then be referenced in the result accumulation part with an iterated filtering specification. Alternatives and iterateds named this way can then be referenced in the rewrite part with an alternative rewrite order specification or an iterated rewrite order specification.

CHAPTER 8

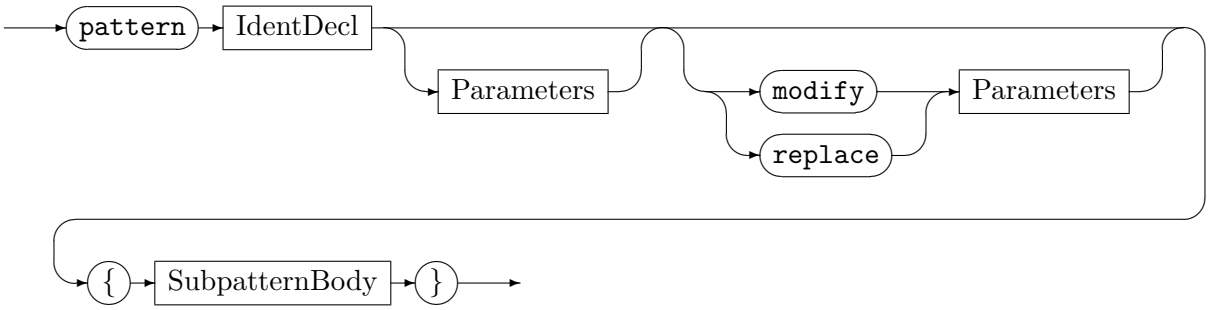
SUBPATTERNS

After the introduction of the nested patterns in chapter 7 we will now have a look at the subpatterns, the other means to greatly enhance the flexibility and expressiveness of pattern matching and rewriting. Subpattern employment is split into subpattern declaration plus subpattern entity declaration, and subrule declaration plus usage, the latter is described in the following chapter.

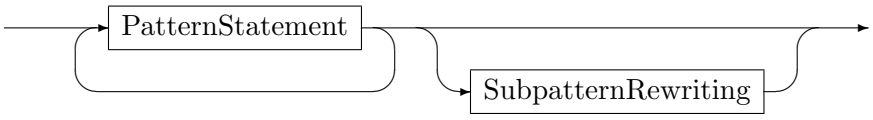
8.1 Subpattern Declaration

Subpatterns were introduced to factor out a common recurring pattern – a shape – into a named subpattern type, ready to be reused at points the pattern should get matched. The common recurring pattern is specified in a subpattern declaration and used by a subpattern entity declaration.

SubpatternDeclaration



SubpatternBody

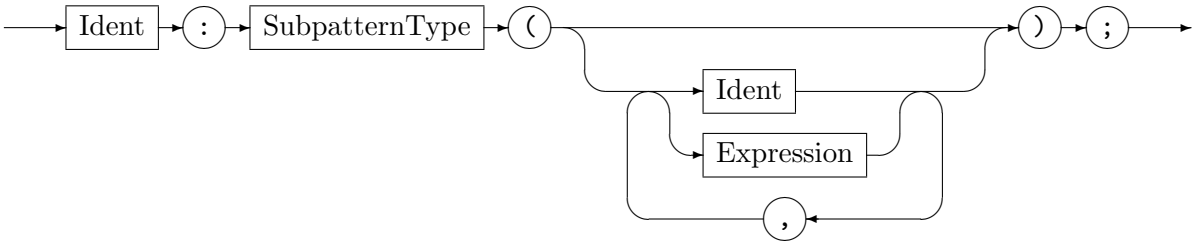


Subpattern declarations define a subpattern type denoting the specified shape in the global namespace; the parameters specify some context elements the pattern may refer to, but which are not part of the pattern itself.

So they are only syntactically the same as test/rule-parameters (which are members of the pattern part). A further difference is the lack of *ReturnTypes*; they are not actions, just a helper in constructing complex patterns. In order to get values out they employ the language construct of def entities which are yielded to (cf. 10). Subpatterns can receive additional rewrite parameters, in addition to the pattern parameters, and in contrast to the actions; they can be used to hand in nodes which are created in the rewrite part of the action or subpattern which contains the subpattern entity. They apply to the nested subpattern rewriting part, which will be explained in Section 9.2.

8.2 Subpattern Entity Declaration

SubpatternEntityDeclaration



Subpattern entity declarations (subpattern usages) instantiate an entity of the subpattern type (i.e. specified shape), which means the subpattern must get matched for the matching of the enclosing pattern to succeed. The arguments given are bound to the corresponding parameters of the subpattern. If you prefer a syntactical point of view, you may see the subpattern entity as a placeholder, which gets substituted in place by the textual body of the subpattern declaration under renaming of the parameters to the arguments. If you prefer an operational point of view, you may see the subpattern entity as a call to the matcher routine searching for the specified pattern from the given arguments on (constructing a match piece, which is the base for rewriting of that part).

EXAMPLE (31)

```
1  pattern TwoParameters(mp:Method)
2  {
3    mp <-- :Variable;
4    mp <-- :Variable;
5  }
6  test methodAndFurther
7  {
8    m:Method <-- n:Name;
9    tp:TwoParameters(m);
10 }
```

In the given example a subpattern `TwoParameters` is declared, connecting the context element `mp` via two edges to two variable nodes. The test `methodAndFurther` is using the subpattern via the declaration of the entity `tp` of type `TwoParameters`, binding the context element to its local node `m`. The resulting test after subpattern derivation is equivalent to the test `methodWithTwoParameters`.

```
1  test methodWithTwoParameters
2  {
3    m:Method <-- n:Name;
4    m <-- :Variable;
5    m <-- :Variable;
6  }
```

8.2.1 Recursive Patterns

Subpatterns can be combined with alternative patterns or the cardinality patterns into recursive subpatterns, i.e. subpatterns which may contain themselves. Subpatterns containing themselves alone – directly or indirectly – would never yield a match as an infinite pattern can't be found in a limited graph.

EXAMPLE (32)

```
test iteratedPath
{
  root:Assign;
  negative { --> root; }
  :IteratedPath(root); // match iterated path = assignment list
}

pattern IteratedPath(prev:Node)
{
  optional { // nothing or a linked assignment and again a list
    prev --> a:Assign; // assignment node
    :IteratedPath(a); // next one, plz
  }
}
```

The code above searches an iterated path from the root node on, here an assignment list. The iterated path with the optional is equivalent to the code below. Note the negative which ensures you get a longest match – without it the empty case may be chosen lazily just in the beginning. Please note that if you only need to check for the existence of such a simple iterated path you can use the `reachable` functions, or even better `isReachable` predicates introduced in 16.2.3.

```
pattern IteratedPath(prev:Node)
{
  alternative {
    Empty {
      negative {
        prev --> a:Assign;
      }
    }
    Further {
      prev --> a:Assign;
      :IteratedPath(a);
    }
  }
}
```

The code below searches an iterated path like the code above, just that it stops when a maximum length is reached. The bounded iterated path is realized by calling a pattern with the requested maximum depth, counting the depth parameter down with each recursion step, until the limit checked by a condition is reached. The `boundedReachable` functions introduced in 16.2.3 should be preferred in a simple case like this.

```
test boundedIteratedPath(root:Assign)
{
  :BoundedIteratedPath(root, 3); // match iterated path of depth 3
}

pattern BoundedIteratedPath(prev:Node, var depth:int)
{
  optional { // nothing or a linked assignment and again a list
    prev --> a:Assign; // assignment node
    :IteratedPath(a, depth-1); // next one, plz
    if{ depth >= 1; } // stop when we have reached max depth
  }
}
```

EXAMPLE (33)

```
1  rule removeMiddleAssignment
2  {
3    a1:Assign --> a2:Assign --> a3:Assign;
4    independent {
5      :IteratedPath(a1,a3)
6    }
7
8    replace {
9      a1; a3;
10   }
11 }
12
13 pattern IteratedPath(begin:Assign, end:Assign)
14 {
15   alternative { // an iterated path from begin to end is either
16     Found { // the begin assignment directly linked to the end assignment (base case)
17       begin --> end;
18     }
19     Further { // or an iterated path from the node after begin to end (recursive case)
20       begin --> intermediate:Assign;
21       :IteratedPath(intermediate, end);
22     }
23   }
24 }
```

This is once more a fabricated example, for an iterated path from a source node to a distinctive target node, and an example for the interplay of subpatterns and positive application conditions to check complex conditions independent from the pattern already matched. Here, three nodes `a1`,`a2`,`a3` of type `Assign` forming a list connected by edges are searched, and if found, `a2` gets deleted, but only if there is an iterated path of directed edges from `a1` to `a3`. The path may contain the host graph node matched to `a2` again. Without the `independent` this would not be possible, as all pattern elements – including the ones originating from subpatterns – get matched isomorphically. The same path specified in the pattern of the rule – not in the independent – would not get matched if it would go through the host graph node matched to b, as it is locked by the isomorphy constraint. Again, the `reachable` functions and `isReachable` predicates introduced in 16.2.3 are the better choice here. They always match independently from the patterns (so when you need locking, you cannot employ them), can be used directly, and are implemented (more) efficiently.

With recursive subpatterns you can already capture neatly structures extending into depth (as iterated paths) and also structures extending into breadth (as forking patterns, although the cardinality statements are often much better suited to this task). But combined with an iterated block, you may even match structures extending into breadth and depth, like e.g. an inheritance hierarchy of classes in our example domain of program graphs, see example 34, i.e. you can match a spanning tree in the graph. This gives you a very powerful and flexible notation to capture large, complex patterns built up in a structured way from simple, connected pieces (as e.g. abstract syntax trees of programming languages).

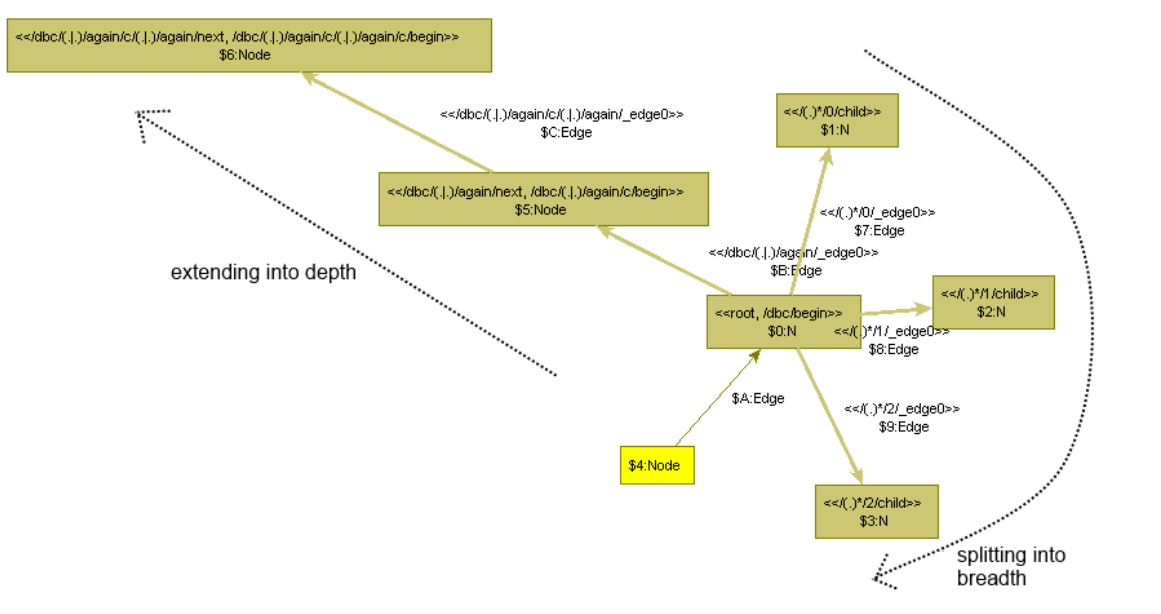
NOTE (24)

If you are working with hierarchic structures like that, you might be interested in the capabilities of GrShell/yComp for nested layout as described and shown in 27.1/56).

EXAMPLE (34)

```
pattern SpanningTree(root:Class)
{
  iterated {
    root <-:extending- next:Class;
    :SpanningTree(next);
  }
}
```

EXAMPLE (35)



The image from above visualizes the matching of structures extending into depth and splitting into breadth, employing the pattern of test `db` from below.

```
test db(root:Node)
{
  iterated {
    root --> child:N;
  }
  dbc:chain(root);
}
pattern chain(begin:Node)
{
  alternative {
    again {
      begin --> next:Node;
      c:chain(next);
    }
    end {
      negative { begin --> .; }
    }
  }
}
```

EXAMPLE (36)

The example from the previous page utilizes the nested and subpatterns introduced in this and the previous chapter; it matches `db` from a `root` node handed in on, in a tiny example graph (built from nodes of types `Node` and `N`, and edges of type `Edge`).
The breadth-splitting structure is collected with an `iterated` pattern around the center node bound to the pattern node `root`. The graph elements are annotated in the debugger (cf. Chapter 27) as usual with their pattern element name – plus the (sub)pattern nesting path that lead to the instance of this pattern.
The `(.)*/1/child` at the middle-right node tells that the node is contained in an iterated pattern (using the regular expression syntax for the iterated, see Table 8.1), that this is instance `1` (starting at `0`, here we got 3 matches of the iterated, so ending at instance count `2`, and that the name of the element in the pattern is `child`.
The depth-extending structure is collected with a subpattern `chain` employed from the test with the instance `dbc` starting at the node bound to the pattern node `root`. It uses itself recursively with the subpattern instance `c`, from the node `next` on, which is reached with one edge-following-step from the input node `begin` on.
The `/dbc/(.|.)/again/next` at the middle node of the chain tells that the node was matched on the subpattern instance named `dbc`, in an alternative (using regular expression syntax for the alternative), in the pattern of the `again` case, to the pattern node named `next`. The same node is annotated with an`/dbc/(.|.)/again/c/begin`, which stems from the fact that the node appears also in the next nesting step (subpattern call), in the instance `c` of the subpattern instantiated in the branch `again`; as `begin` node handed in to `chain`.
Breadth-splitting structure-matching is typically following `contains` edges, while depth-extending structure-matching is typically following `next` edges, please take a look at the Hints on Modeling 4.2.4.
The (sub)pattern nesting paths can get long, cluttering display, so from a certain size on, only the size of the path is printed in the debugger.

8.3 Regular Expression Syntax

In addition to the already introduced syntax for the nested patterns with the keywords `negative`, `independent`, `alternative`, `iterated`, `multiple` and `optional`, there is a more lightweight syntax resembling regular expressions available; when used together with the subpatterns it yields graph rewrite specifications which look like EBNF-grammars with embedded actions. Besides reducing syntactical weight, they offer constructs for matching a pattern a bounded number of times (same notation as the one for the bounded iteration in the sequences).

`iterated { P }`	`(P)*`
`multiple { P }`	`(P)+`
`optional { P }`	`(P)?`
`alternative { l1 { P1 } .. lk { Pk } }`	`(P1|..|Pk)`
`negative { P }`	`~(P)`
`independent { P }`	`&(P)`
`modify { R }`	`{+ R }`
`replace { R }`	`{- R }`
`-`	`(P)[k] / (P)[k:l] / (P)[k:*]`

Table 8.1: Map of nested patterns in keyword syntax to regular expression syntax

Understanding GrGen.NET-subpatterns may be easier given knowledge about EBNF-

grammars when we compare them to those. We find then that rules resemble grammar axioms, subpatterns resemble nonterminals, and graphlets resemble terminal symbols; nested patterns are similar to EBNF operators, and the rewrite part corresponds to the semantic actions of syntax directed translation. Negative and independent patterns are used to explicitly check context constraints (every graphlet as such is already able to match pieces that one would or could classify as context, graph rewriting allows for derivations that are highly adaptable to the surrounding parts). See [Jak11] for more on this.

EXAMPLE (37)

```
1  test method
2  {
3    m:Method <-- n:Name; // signature of method consisting of name
4    ( m <-- :Variable; )* // and 0-n parameters
5
6    :AssignmentList(m); // body consisting of a list of assignment statements
7  }
8
9  pattern AssignmentList(prev:Node)
10 {
11   ( // nothing or a linked assignment and again a list
12     prev --> a:Assign; // assignment node
13     a -:target-> v:Variable; // which has a variable as target
14     :Expression(a);  // and an expression which defines the left hand side
15     :AssignmentList(a); // next one, plz
16   )?
17 }
18
19 pattern Expression(root:Expr)
20 {
21   ( // expression may be a binary expression of an operator and two expresions
22       root <-- expr1:Expr;
23       :Expression(expr1);
24       root <-- expr2:Expr;
25       :Expression(expr2);
26       root <-- :Operator;
27   | // or a unary expression which is a variable (reading it)
28       root <-- v:Variable;
29   )
30 }
```

8.4 Pattern Element Locking

Isomorphy Locking

When matching a program graph as in the introductory example 18 one might be satisfied with matching a tree structure. But on other occasions one wants to match *backlinks* and especially the targets of the backlinks, too, from *uses* nested somewhere in the syntax graph to *definitions* whose nodes were already matched earlier in the subpattern derivation (subpatterns can be seen as an equivalent of grammar rules known from parser generators). Unfortunately these elements are already matched and thus isomorphy locked following the default semantics of isomorphic matching. And unfortunately these elements can't be declared `homomorphic` as they are unknown in the nested subpattern. Handing them in as

parameters and then declaring them `homomorphic` is only possible if they are of a statically fixed number (as the number of parameters is fixed at compile time), which is normally not the case for e.g. the attributes of a class in a syntax graph. In order to handle this case the `independent` *operator* (cf. 5.3.1) was added to the rule language — when you declare the backlink target node `n` as `independent(n)` it can be matched once again. Thus it is possible to match e.g. a class attribute definition node which was already matched when collecting the attributes of the class again later on in a subpattern when matching an expression containing a usage of that attribute, allowing to e.g. add further edges to it.

Patternpath Locking (preliminary)

As stated in the sections on the negative and independent constructs (7.1, 7.2), they get matched homomorphically to all already matched elements. By referencing an element from outside you can isomorphy lock that element to prevent it from getting matched again.

Maybe you want to lock all elements from the directly enclosing pattern, in this case you can just insert `pattern;` in the position of a graphlet into the NAC or PAC.

Maybe you want to lock all elements from the patterns dynamically containing the NAC/-PAC of interest, i.e. all subpattern usages and nesting patterns on the path leading to the NAC/PAC of interest (but not their siblings). In this case you can insert `patternpath;` in the position of a graphlet into the NAC or PAC. You might be interested in this construct when matching a piecewise constructed pattern, e.g. a chain, which requires to check for another chain (iterated path) which is not allowed to cross (include an element of) the original one.

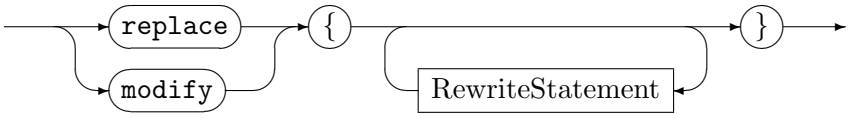
CHAPTER 9

NESTED AND SUBPATTERN REWRITING

9.1 Nested Pattern Rewriting

Until now we focused on the pattern matching of nested and subpatterns – but we're not only interested in finding patterns combined from several pattern pieces, we want to rewrite the pattern pieces, too. So we will extend the language of the structure parser introduced so far into a language for a structure transducer. This does not hold for the application conditions, which are pure conditions, but for all the other language constructs introduced in this chapter.

NestedRewriting



Syntactically the rewrite is specified by a modify or replace clause nested directly within the scope of each nested pattern; in addition to the rewrite clause nested within the top level pattern. Semantically for every instance of a pattern piece matched its dependent rewrite is applied. So in the same manner the complete pattern is assembled from pattern pieces, the complete rewrite gets assembled from rewrite pieces (or operationally: rewriting is done along the match tree by rewriting one pattern piece after the other). Note that `return` statements are not available as in the top level rewrite part of a rule, and the `exec` statements are slightly different.

For a static pattern specification like the iterated block yielding dynamically a combined match of zero to many pattern matches, every submatch is rewritten, according to the rewrite specification applied to the host graph elements of the match bound to the pattern elements (if the pattern was matched zero times, no dependent rewrite will be triggered - but note that zero matches still means success for an iterated, so the dependent rewrite piece of the enclosing pattern will be applied). This allows e.g. for reversing all edges in the iterated-example (denoting containment in the class), as it is shown in the first of the following two examples. For the alternative construct the rewrite is specified directly at every nested pattern, i.e. alternative case (as shown in the second of the following two examples); the rewrite of the matched case will be applied.

Nodes and edges from the pattern containing the nested pattern containing the nested rewrite are only available for deletion or retyping inside the nested rewrite if it can be statically determined that this is unambiguous, i.e. only happening once. So only the rewrites of alternative cases, optional patterns or subpatterns may contain deletions or retypings of elements not declared in their pattern (in contrast to iterated and multiple pattern rewrites).

EXAMPLE (38)

```
rule methods
{
  c:Class;
  iterated {
    c --> m:Method;

    replace {
      c <-- m;
    }
  }

  replace {
    c;
  }
}
```

EXAMPLE (39)

```
rule methodWithTwoOrThreeParameters(m:Method)
{
  alternative {
    Two {
      m <-- n:Name;
      m <-e1:Edge- v1:Variable;
      m <-e2:Edge- v2:Variable;
      negative {
        v1; v2; m <-- :Variable;
      }

      modify {
        delete(e1); m --> v1;
        delete(e2); m --> v2;
      }
    }
    Three {
      m <-- n:Name;
      m <-e1:Edge- v1:Variable;
      m <-e2:Edge- v2:Variable;
      m <-e3:Edge- v3:Variable;

      modify {
        delete(e1); m --> v1;
        delete(e2); m --> v2;
        delete(e3); m --> v3;
      }
    }

  //modify { can be omitted - see below
  //}
}
```

NOTE (25)

In case you got a `rule` or `pattern` with an empty `modify` clause, with all the real work going on in an `alternative` or an `iterated`, you can omit the empty `modify` clause. This is a small syntactic convenience reducing noise which is strictly restricted to the top level pattern — omitting rewrite parts of nested patterns specifies the entire pattern to be match-only (like a `test`; this must be consistent for all nested patterns).

EXAMPLE (40)

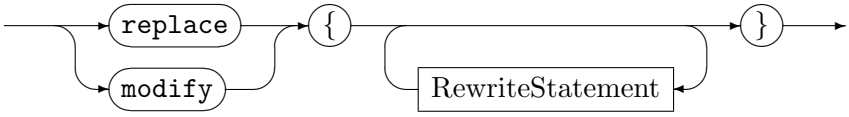
This is an example which shows how to decide with an alternative on the target type of a retyping depending on the context. Please note the omitted rewrite (cf. 25).

```
1  rule alternativeRelabeling
2  {
3    m:Method;
4
5    alternative {
6      private {
7        if { m.access == Access::private; }
8
9        modify {
10         pm:PrivateMethod<m>;
11       }
12     }
13     static {
14       negative {
15         m <-- c;
16       }
17
18       modify {
19         sm:StaticMethod<m>;
20       }
21     }
22   }
23 }
```

9.2 Subpattern Rewriting

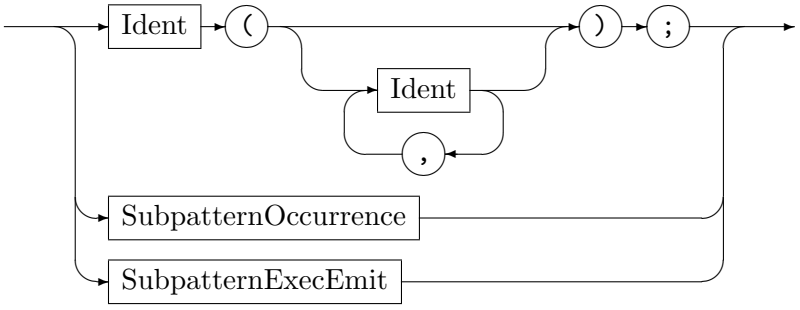
Alongside the separation into subpattern declaration and subpattern entity declaration, subpattern rewriting is separated into a nested rewrite specification given within the subpattern declaration defining how the rewrite looks like and a subpattern rewrite application given within the rewrite part of the pattern containing the subpattern entity declaration requesting the rewrite to be actually applied.

SubpatternRewriting



The subpattern rewriting specifications within the subpattern declaration look like a nested rewriting specification, but additionally there may be rewrite parameters given in the subpattern header (cf. 8.1) which can be referenced in the rewrite body. (Most elements can be handed in with normal parameters, but elements created in the rewrite part of the user of the subpattern can only be handed in at rewrite time.)

SubpatternRewriteApplication



The *SubpatternRewriteApplication* is part of the *RewriteStatement* already introduced (cf. 5.4.3). The subpattern rewrite application is given within the rewrite part of the pattern containing the subpattern entity declaration, in call notation on the declared subpattern identifier. It causes the rewrite part of the subpattern to get used; if you leave it out, the subpattern is simply kept untouched. The *SubpatternOccurrence* is explained in the next subsection 9.2.1. The *SubpatternExecEmit* is explained in chapter 13.

EXAMPLE (41)

This is an example for a subpattern rewrite application.

```
pattern TwoParametersAddDelete(mp:Method)
{
  mp <-- v1:Variable;
  mp <-- :Variable;

  modify {
    delete(v1);
    mp <-- :Variable;
  }
}
rule methodAndFurtherAddDelete
{
  m:Method <-- n:Name;
  tp:TwoParametersAddDelete(m);

  modify {
    tp(); // trigger rewriting of the TwoParametersAddDelete instance
  }
}
```

EXAMPLE (42)

This is another example for a subpattern rewrite application, reversing the direction of the edges on an iterated path.

```
pattern IteratedPathReverse(prev:Node)
{
  optional {
    prev --> next:Node;
    ipr:IteratedPathReverse(next);

    replace {
      prev <-- next;
      ipr();
    }
  }

  replace {
  }
}
```

EXAMPLE (43)

This is an example for rewrite parameters, connecting every node on an iterated path to a common node (i.e. the local rewrite graph to the containing rewrite graph). It can't be simulated by subpattern parameters which get defined at matching time because the common element is only created later on, at rewrite time.

```
pattern ChainFromToReverseToCommon(from:Node, to:Node) replace(common:Node)
{
  alternative {
    rec {
      from --> intermediate:Node;
      cftrtc:ChainFromToReverseToCommon(intermediate, to);

      replace {
        from <-- intermediate;
        from --> common;
        cftrtc(common);
      }
    }
    base {
      from --> to;

      replace {
        from <-- to;
        from --> common;
        to --> common;
      }
    }
  }

  replace {
    from; to;
  }
}
```

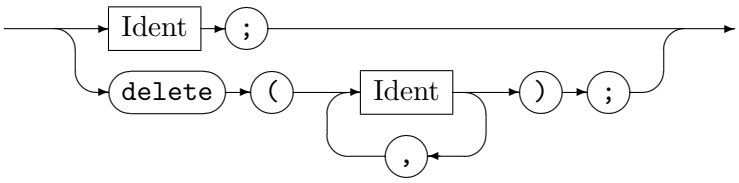
```
rule chainFromToReverseToCommon()
{
  from:Node; to:Node;
  cftrtc:ChainFromToReverseToCommon(from, to);

  modify {
    common:Node;
    cftrtc(common);
  }
}
```

9.2.1 Deletion and Preservation of Subpatterns

In addition to the fine-grained dependent rewrite, subpatterns may get deleted or kept as a whole.

SubpatternOccurrence



In modify mode, they are kept by default, but deleted if the name of the declared subpattern entity is mentioned within a delete statement. In replace mode, they are deleted by default, but kept if the name of the declared subpattern entity is mentioned (using occurrence, same as with nodes or edges).

EXAMPLE (44)

```
rule R {
  m1:Method; m2:Method;
  tp1:TwoParameters(m1);
  tp2:TwoParameters(m2);

  replace {
    tp1; // is kept
    // tp2 not included here - will be deleted
    // tp1(); or tp2(); -- would apply dependent replacement
    m1; m2;
  }
}
```

NOTE (26)

You may even give a SubpatternEntityDeclaration within a rewrite part which causes the subpattern to be created; but this employment has several issues which can only be overcome by introducing explicit creation-only subpatterns – so you better only use it if you think it should obviously work (examples for the issues are alternatives – which case to instantiate? – and abstract node or edge types – what concrete type to choose?).

```
pattern ForCreationOnly(mp:Method)
{
  // some complex pattern you want to instantiate several times
  // connecting it to the mp handed in
}
rule createSubpattern
{
  m:Method;

  modify {
    :ForCreationOnly(m); // instantiate pattern ForCreationOnly
  }
}
```

CHAPTER 10

NESTED AND SUBPATTERN YIELDING

In this chapter we explain an extension of the LHS and RHS patterns by yielding parts. The constructs seen so far are executed top-down alongside the contained nested and subpatterns, for one during matching and for the other during rewriting. The yielding parts are executed bottom-up for one after matching and for the other while rewriting, they allow accumulating results. Syntactically, the split is denoted by a separator line `---`. The content before the split is mostly related to declarative pattern matching and rewriting (following the idea of graph rewriting), while the content after the split supplements them by more powerful imperative features.

	top-down (declarative)	bottom-up (imperative extensions)
LHS pattern	pattern matching	result accumulation
RHS pattern	rewriting	result accumulation (computations)

Table 10.1: Pattern separator

As a refinement regarding the execution semantics: there are two runs, the first matching run constructs a match object tree (reflecting the LHS pattern parts and their nesting structure), the second run walks the match object tree and carries out rewriting alongside it. After the rewriting run are the embedded execs executed (they get encapsulated and stored in closures during the walk, the top-level ones are executed directly).

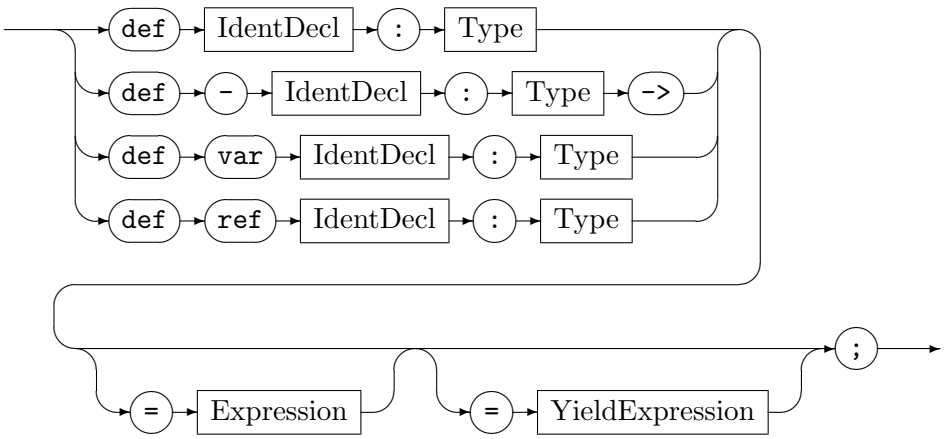
The match objects consist of the pattern elements bound by top-down pattern matching, and of new LHS def elements that receive the data from bottom-up yielding (these amount to inherited and synthesized attributes in compiler construction parlance).

10.1 Yielding Outwards After Pattern Matching

Sometimes one needs to bring something matched within a nested or subpattern to an outer pattern containing it (in case of nested patterns) or calling it (in case of subpatterns). So one can return it out, or accumulate information from matched iterated pattern instances.

10.1.1 LHS def variables

LHSPatternDefVariableDecl



The first thing one needs to bring something outwards is a target in a nesting or calling pattern. This is achieved by nodes, edges, and variables declared with the `def` keyword in the pattern, marking them as output entities. Furthermore subpattern parameters may be declared as def parameters in the subpattern definition header, marking them as output parameters.

10.1.2 Yielding and Iterated Filtering

Yield blocks

These elements can then be yielded to by eval statements from within a `yield` block (maybe with iterated accumulation) and subpattern usages. A `yield` block is a constrained `eval` block that can be given in the pattern part; it does not allow assigning to or changing non-`def` variables, or carrying out graph-changing commands. Yielding is specified by prepending the `yield` keyword to the entity yielded to, in the assignment or method call. The `yield` must be prepended to the argument for a subpattern def parameter, too.

Yielding, i.e. bubbling up of the elements in the match object tree occurs at the end of pattern matching when the match object tree gets constructed.

NOTE (27)

A def entity from the pattern part can't be yielded to from the rewrite part, they are constant after matching.

Let's have a look at two examples for yielding:

EXAMPLE (45)

```
pattern Chain(begin:Node --- def end:Node)
{
  alternative {
    further {
      begin --> next:Node;
      :Chain(next --- yield end);
    }
    done {
      negative {
        begin --> ;
      }
      ---
      yield {
        yield end = begin;
      }
    }
  }
}
pattern LinkChainTo(begin:Node) modify(n:Node)
{
  alternative {
    further {
      begin --> next:Node;
      o:LinkChainTo(next);

      modify {
        next --> n;
        o(n);
      }
    }
    done {
      negative {
        begin --> ;
      }

      modify {
      }
    }
  }

  modify { }
}
rule linkChainEndToStartIndependent(begin:Node) : (Node)
{
  independent {
    c:Chain(begin --- yield end);
  }
  o:LinkChainTo(begin);
---
  def end:Node;

  modify {
    o(end);
    return(end);
  }
}
```

The first example for LHS yielding follows within an independent a chain piece by piece to some a priori unknown end node, and yields this end node chain piece by chain piece again outwards to the chain start. There it is used as input to another chain (maybe the same chain, maybe overlapping due to the independent), linking all the nodes of this chain to the end node of the former.

When yielding from an iterated pattern there's the issue that a simple bottom-up assignment to a def variable from the outside would only overwrite the variable value multiple times, so we would end up with one of the values from the iterated pattern instances (non-deterministically chosen – multiple iterated variable values exist for a single nesting pattern variable). But we can also read entities from the nesting pattern, and thus use them as accumulation variables, for e.g. summing integers or concatenating strings. The start value of accumulation can be set with the initialization in the nesting pattern (and thus travels top-down).

EXAMPLE (46)

```
1  test sumOfWeight(start:Node) : (int,int)
2  {
3    iterated {
4      start --> n:N; // node class N { a:int; }
5    ---
6      yield {
7        yield sum = sum + n.a;
8        yield v = n.a; // v is in the end the n.a of one iterated instance
9      }
10   }
11 ---
12   def var sum:int = 0;
13   def var v:int = 0;
14   return (sum,v);
15 }
```

Alternatively, a `for` iterated accumulation loop can be used, iterating a def variable inside an iterated for all the matches of the iterated pattern referenced by name, it allows to assign to an outside def variable a value computed from the def variable and the value of the iterated def variable.

This is shown in the third example for LHS yielding, summing the integer attribute `a` of nodes of type `N` adjacent to a start node, matched with an iterated.

EXAMPLE (47)

```
test sumOfWeight(start:Node) : (int)
{
  iterated it {
    start --> n:N; // node class N { a:int; }
  ---
    def var i:int = n.a;
  }
---
  def var sum:int = 0;
  yield {
    for(i in it)
    {
      yield sum = sum + i;
    }
  }

  return (sum);
}
```

In case no accumulation is needed but a simple count of the iterated matches is sufficient, one can employ the `count` operator on an iterated (that must have been named before), as displayed in the following example (the count is only available in a `yield` or `eval` block).

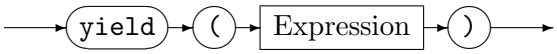
EXAMPLE (48)

```
test countOfEdges(start:Node) : (int)
{
   iterated it {
      start -->;
   }
---
   def var sum:int = 0;
   yield {
      yield sum = count(it);
   }

   return (sum);
}
```

Yield expressions

YieldExpression



For more convenience, the notational shorthand of in-place yielding at a def variable declaration is available. This syntactic sugar with the syntax `def var v:T = yield(expr);` is de-sugared to a plain `def var v:T;` declaration and an assignment in an implicitly added `yield { v = expr; }` block. It is not to be confused with separate initialization, combined: `def var v:T = init-expr = yield(expr)` — the init-expr is evaluated before the yields of nested patterns and can thus be used to give a start value for bottom-up accumulation, the in-place yielding is evaluated after nested and subpatterns were matched and yielded (only then are iterated matches available).

The example from above can be formulated succinctly with the yield expression.

EXAMPLE (49)

```
test countOfEdges(start:Node) : (int)
{
   iterated it {
      start -->;
   }
---
   def var sum:int = yield(count(it));
   return (sum);
}
```

A more general language device that benefits from the yield expression is the iterated query. It is available in the rule language expressions within a rule (but only during yielding), and returns an array of the matches of the iterated (`array<match<r.it>>` for iterated `it` in rule `r`), cf. Section 6.8, in contrast to the rule query from Chapter 20 that is available in the sequence expressions, and returns an array of matches of the rule (`array<match<r>>`). Similar to the `for` iterated accumulation loop, it accesses the matches of the iterated from the outside (in contrast to default bottom-up yielding from inside). Employing the iterated query, the example looks as follows; for a larger example, see Example 124.

EXAMPLE (50)

```
test countOfEdges(start:Node) : (int)
{
   iterated it {
      start -->;
   }
---
   def var sum:int = yield([?it].size());
   return (sum);
}
```

Iterated Filtering

A helper in iterated accumulation is iterated filtering, it allows to execute filters on the iterated matches array, reordering matches or even removing them (thus changing the result matches tree of rule application). For this to work, the iterated must be firstly given a name, as described in 7.5, so it can be referenced, and secondly, a filter must be declared on it, cf. Section 7.3 (unless only auto-supplied filters are to be applied, cf. 17.4).

IteratedFiltering



The syntax of the iterated filtering specification past the `iterated` *IteratedIdent* part is the same as for filter calls on a rule, i.e. e.g. `r\f`, see Chapter 17 for more on this. Note that the feature is limited to top-level iterateds (for now).

EXAMPLE (51)

```
1  test iteratedFiltering(start:Node) : (int)
2  {
3     iterated it {
4        start --> n:N;
5     ---
6        def var i:int = n.i;
7     }
8  ---
9     iterated it \ orderDescendingBy<i> \ keepFirst(3);
10    def var sum:int = yield([?it].extract<i>().sum());
11    return (sum);
12 }
```

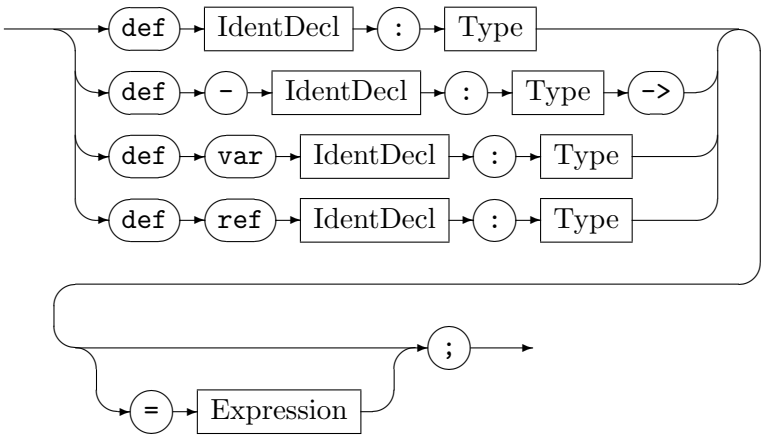
10.2 Yielding Outwards During Rewriting

Sometimes one needs to bring something obtained during rewriting within a nested or subpattern to an outer pattern containing it (nested patterns) or calling it (subpatterns). So that one can do there (in the using pattern) operations on it, e.g. attaching a further edge to an end node of a chain matched with recursive patterns (thus modularizing the graph rewrite specification into chain matching patterns and patterns using chains doing things on the chain ends), or accumulating attributes obtained in iterated pattern instances.

10.2.1 RHS def variables

The first thing one needs to bring something outwards is a target in a nesting or calling pattern. This is achieved by nodes, edges, and variables declared with the `def` keyword in a rewrite part, marking them as output entities. In addition, on their introduction an initializing expression may be given. Furthermore subpattern rewrite parameters may be declared as def parameters, marking them as output parameters.

RHSPatternDefVariableDecl



Another use besides bringing something upwards/to the outside is as a helper in attribute evaluation, as temporary variables (but local variables introduced in the computation are preferable in case they are only used in their very computation).

EXAMPLE (52)

```
1  rule r : (double) {
2    n1:N; n2:N; n3:N;
3
4    modify {
5    ---
6      def var mean:double = array<double>(n1.v, n2.v, n3.v).avg();
7      eval {
8        n1.variance = Math::sqr(n1.v - mean);
9        n2.variance = Math::sqr(n2.v - mean);
10       n3.variance = Math::sqr(n3.v - mean);
11     }
12     return(mean);
13   }
14 }
```

10.2.2 Yielding

These elements can then be yielded to from within `eval` or `evalhere` statements, subpattern rewrite usages, or `exec` statements. While the latter will be covered in chapter 13, the former will be explained in the following.

Yielding is specified by prepending the `yield` keyword to the entity yielded to, in an assignment or a change assignment to a variable, or a method call on a variable, inside an `eval` or `evalhere`-block; the target of the assignment may be a node or edge (if declared as output variable). The yield must be prepended to the argument for a subpattern def rewrite parameter, too.

EXAMPLE (53)

```
1  pattern Chain(begin:Node) modify(--- def end:Node)
2  {
3    alternative {
4      Further {
5        begin --> intermediate:Node;
6        c:Chain(intermediate);
7
8        modify {
9          c(--- yield end);
10       }
11     }
12     Done {
13       negative {
14         begin --> ;
15       }
16
17       modify {
18       ---
19         eval {
20           yield end = begin;
21         }
22       }
23     }
24   }
25
26   modify { }
27 }
28
29 rule R(begin:Node) : (Node) {
30   c:Chain(begin);
31
32   modify {
33     c(--- yield end); // end is filled with chain end
34   ---
35     def end:Node;
36     return(end);
37   }
38 }
```

First example for RHS yielding: returning the end node of a chain.

EXAMPLE (54)

```
rule outCount(head:Node) : (int)
{
  iterated {
    head --> .;
    modify {
    ---
      eval { yield cnt = cnt + 1; }
    }
  }

  modify {
  ---
    def var cnt:int = 0;
    return (cnt);
  }
}
```

Second example for RHS yielding: counting the number of edges matched with an iterated.

10.3 Flow Example

Pattern matching and rewriting occurs in two completely distinct passes, separated by the matches tree of the sub- and nested patterns. During each pass, input and output parameters may be computed and passed; with explicit passing for subpatterns, and implicit passing for nested patterns (which can directly access the content of their contained pattern).

First the pattern is matched with recursive descent alongside pattern nesting and subpattern calling, employing some helper stacks in addition to the call stack (the pushdown machine is explained in more detail in 35.3). Matching occurs strictly top-down (or from the outside to the nested inside patterns), first the graphlets of the current pattern are matched, then control descends to the nested and subpatterns. During matching, input parameters may be passed, esp. forwarding just matched elements or attributes from them; they allow to influence the matching process.

When a complete match is found, while ascending again, unwinding the call stack, output pattern-`def` parameters are passed bottom-up (or from the inside to the containing outside patterns) with `yield` blocks and `yield` assignments within those blocks, or `yield` bindings of `def` parameters in a subpattern call. During this ascent, the matches tree is assembled, each match of a pattern contains the matches of its nested patterns and used subpatterns when it is left.

In the rewrite step, the matches tree is visited recursively again, creating the new nodes of the current pattern, then descending to the nested patterns and subpatterns, executing their changes, and after they returned, executing all other changes of the pattern. During this visit, rewrite input parameters may be passed, for forwarding just created elements (or computed values).

On ascending again from a pattern, the rewrite-`def` elements are assigned from `yield` assignments in the `eval` blocks, or `yield` bindings of `def` rewrite parameters in a subpattern rewrite call. (Technically, nested patterns are handled like subpatterns, with parameter passing for accessed elements of the containing pattern automatically inserted by the compiler.)

Let's take a look at an example, with Figure 10.1 depicting the input and output parameter passing during the matching of Example 10.3, and Figure 10.2 depicting the input and output parameter passing during the rewriting of Example 10.3 (which is an extended version of Example 10.1).

EXAMPLE (55)

```
1  rule r(a:Node) : (Node, int)
2  {
3    a --> b:Node;
4    p:P(b --- yield d);
5
6    optional o {
7      a --> . --> c:N;
8    ---
9      yield { yield i = c.i; }
10
11     modify {
12     }
13   }
14 ---
15   def var i:int;
16   def d:Node;
17
18   modify {
19     b --> u:Node;
20     return(d, i);
21   }
22 }
23 pattern P(n:Node --- def rm:Node)
24 {
25   n --> . --> m:Node;
26 ---
27   yield { yield rm = m; }
28 }
```

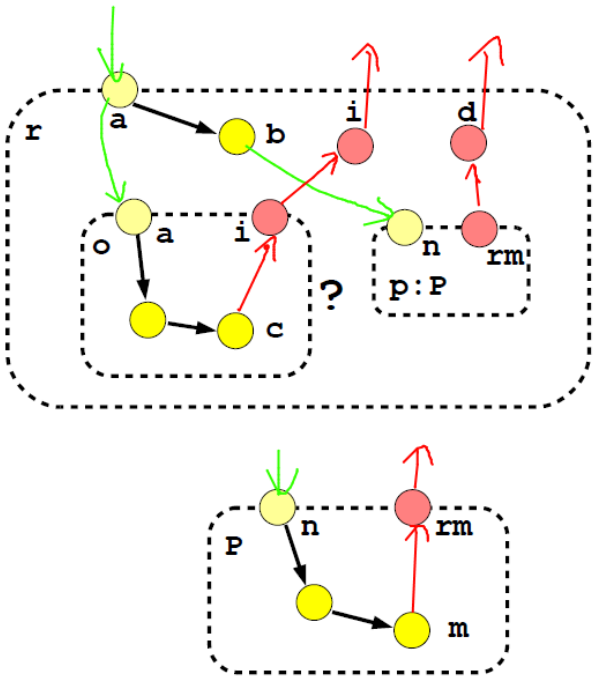


Figure 10.1: Parameter flow in matching

EXAMPLE (56)

```
1  rule r(a:Node) : (Node, int, Node, int) {
2    a --> b:Node;
3    optional o {
4      a --> . --> c:N;
5    ---
6      yield { yield i = c.i; }
7
8      modify {
9      ---
10       eval { yield j = i + u.i; }
11     }
12   }
13   p:P(b --- yield d);
14 ---
15   def var i:int;
16   def d:Node;
17
18   modify {
19     b --> u:N;
20     p(u --- yield e);
21   ---
22     def e:Node; def var j:int;
23     return(d, i, e, j);
24   }
25 }
26 pattern P(n:Node --- def rm:Node) modify(k:Node --- def x:Node) {
27   n --> . --> m:Node;
28 ---
29   yield {   yield rm = m; }
30
31   modify {
32     m --> k; m --> l:Node;
33   ---
34     eval { yield x = l; }
35   }
36 }
```

In Example 10.3, a rule `r` is matched, which contains a nested `optional` pattern `o` and uses a subpattern `P`. The rule matches from its input node `a` on a neighbouring node `b` and hands it in to the subpattern `p`. In the optional pattern it matches a two-step neighbouring node `c` from `a` on. The subpattern `P` matches from its input parameter `n` on a two-step neighbouring node `m`.

A `def` variable `i` is declared in the pattern, and `yield` assigned from the optional `o` an attribute of the `c` found there. Another `def` variable `d` is declared in the pattern, and assigned from the subpattern call `p` with a `yield` parameter passing. The two pattern `def` variables are then `return`ed out of rule `r`. The subpattern `P` `yield`s to its output `def` parameter `rm` the `m` found in its body.

In the extended Example 10.3, the rewrite part of the rule `r` creates a node `u`, links it to `b`, and hands it in to the subpattern rewrite employment of `p`. The rewrite part of subpattern `P` creates a node `l` and links it to `m`. Additionally, an edge is inserted in between `m` and the rewrite input parameter `k`.

A `def` variable `j` is declared in the rewrite part of `r`, and `yield` assigned from the `eval` inside the rewrite part of the optional `o`. Another `def` variable `e` is declared in the rewrite

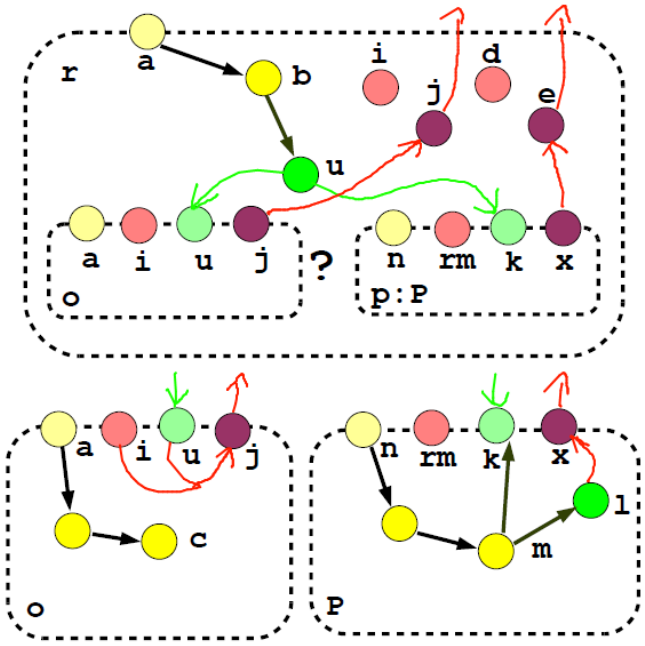


Figure 10.2: Parameter flow in rewriting

part, and assigned from the subpattern rewrite call `p`. The two rewrite-`def` variables are then `return`ed out of the rule, in addition to the two pattern-`def` variables. The subpattern `yield`s from its `eval` part to its output `def` rewrite parameter `x` the `l` just created.

Bottom line: you can flexibly combine patterns with nested and subpatterns, including input and output parameters. You can pass parameters in during matching alongside matching order. When matching completed, during match-tree building, you can yield elements found in contained patterns out. When the rewrite is applied, you can pass parameters in alongside the buildup of the patterns, and yield elements out once more. But not only are those two passes strictly distinct, but also are the parameter passing directions strictly distinct, first, all input parameter passing is carried out during descent, then all output parameter synthesizing is carried out during ascent.

10.4 Ordered Evaluation

Normally the rewrite order is as given in table 10.2:

So first the subpattern rewrites, then the iterated rewrites, then the alternative rewrites are executed, and finally the local eval statements (computations). This might be sufficient in some cases, but in other cases, e.g. when you want to compute an attributation over a tree/a graph, you want to have local computations influenced by attributes in nested/called children or their siblings, and attributes in nested/called children influenced by its parents or siblings. So we need a language device which allows us to specify the execution order of the alternative pattern rewrites versus the iterated pattern rewrites versus the subpattern rewrites versus the local attribute evaluations.

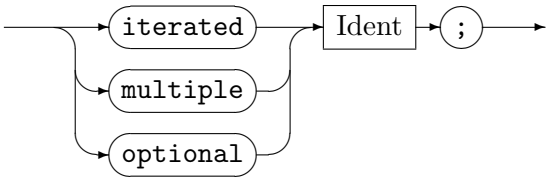
To achieve attribute evaluation or pattern rewriting in a defined order, we use order specifications. They get executed in the order in which they are given syntactically, in the form of *CardinalityRewriteOrderSpecification*s and *AlternativeRewriteOrderSpecification*s for nested patterns, in the form of *SubpatternRewriteOrderSpecification*s for subpatterns, and in the form of local attribute evaluations in `evalhere` blocks; a further statement executed in order is `emithere`, introduced in 13.2.

1.	`Extract elements needed from match`
2.	`Create new nodes`
3.	`Call rewrite code of used subpatterns` *and more...*
4.	`Call rewrite code of nested iterateds`
5.	`Call rewrite code of nested alternatives`
6.	`Redirect edges`
7.	`Retype (and merge) nodes`
8.	`Create new edges`
9.	`Retype edges`
10.	`Create subpatterns`
11.	`Attribute reevaluation (eval)`
12.	`Remove edges`
13.	`Remove nodes`
14.	`Remove subpatterns`
15.	`Emit / Exec`
16.	`Return`

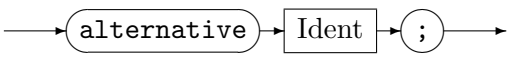
and more... at `3.` are `evalhere`, `emithere`, `emitheredebug`, `alternative` Name, `iterated` Name

Table 10.2: Execution order rewriting

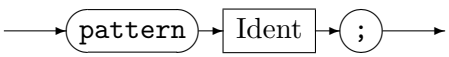
CardinalityRewriteOrderSpecification



AlternativeRewriteOrderSpecification



SubpatternRewriteOrderSpecification



The *CardinalityRewriteOrderSpecification* and *AlternativeRewriteOrderSpecification* reference the iterated or alternative by its name, so a name must be defined, cf. 7.5. So does the *SubpatternRewriteOrderSpecification*, which is not to be confused with the *SubpatternRewriteApplication* introduced in 9.2, the latter only specifies that the subpattern is to be rewritten (disregarding order), additionally giving the rewrite arguments.

What we've seen so far is applied in the following example. A `yield` prefix is needed whenever a `def` variable from a level above is written to (as assignment target as well as a subpattern output argument).

EXAMPLE (57)

```
rule R {
  iterated foo { .; modify { ..read i.. } }
  alternative bar { case { modify { ..read i.. } } }
  sub1:Subpattern1();
  sub2:Subpattern2();

  modify {
    sub1(i);
    sub2(i);
  ---
    def var i:int = 0; // initializes i to 0
    evalhere { yield i = i + 1; } // afterwards i==1
    pattern sub1; // input 1 to subpattern rewrite
    evalhere { yield i = i + 1; } // afterwards i==2
    iterated foo; // nested iterated reads i==2
    evalhere { yield i = i + 1; } // afterwards i==3
    alternative bar; // nested alternative reads i==3
    evalhere { yield i = i + 1; } // afterwards i==4
    pattern sub2; // input 4 to subpattern rewrite
    evalhere { yield i = i + 1; } // afterwards i==5
    eval { yield j = i + 1; } // assign 6 to j
  }
}
```

NOTE (28)

For rewriting, the execution order of the parts can be defined, to allow programming attribute evaluation orders of interest, defining when to descend into which part and defining glueing/local computations in between. (A depth-first run with a defined order in between the siblings, comparable to an LAG/RAG run in compiler construction, but with an explicitly defined sequence of children visits, instead of a temporal succession implicitly induced by the syntactical left-to-right ordering). In contrast to rewriting, the *matching* order of the pattern parts can *not* be defined, to allow the compiler/the runtime to use the evaluation order it estimates to be the best. So we can't access attributes from sibling elements, we can only compute attributes top down from local elements or elements handed in on matching, and later on bottom up from local elements or elements bubbling up at match object tree construction. Top down attribute evaluation operates on the already matched elements and attribute values or the ones received as inputs, which are handed down implicitly into nested patterns or explicitly via subpattern parameters into subpattern instances. (A depth-first run too, but without a defined order in between the siblings, comparable to an IAG run in compiler construction for computing inherited attributes during matching while descending). Bottom up attribute evaluation operates on the matched elements and attribute values locally available or the ones received into def elements yielded implicitly upwards from the nested patterns or explicitly accumulating iterated results or with assigning out parameters of subpatterns. (The same depth-first run, but with attributes computed while ascending, comparable to an SAG run in compiler construction for computing synthesized attributes.)

CHAPTER 11

RULE APPLICATION CONTROL LANGUAGE (SEQUENCES)

Graph rewrite sequences (GRS), better extended graph rewrite sequences XGRS, to distinguish them from the older graph rewrite sequences, are a domain-specific GrGen.NET language used for controlling the application of graph rewrite rules. They are available

- for controlled rule application within the GrShell.
- for controlled rule application on the API level out of user programs.
- as an imperative enhancement to the rule set language.

If they appear in rules, they get compiled, otherwise they get interpreted. Iff used within GrShell, they are amenable to debugging.

Graph rewrite sequences are built from a pure *rule control* language, written down in a syntax similar to boolean and regular expressions, with rule applications as atoms, and a *computations* sublanguage, noted down as a sequence of assignments, function calls, or procedure calls. A computation is given as an atom of the rule control language, nested in curly braces.

The graph rewrite sequences are a means of composing complex graph transformations out of single graph rewrite rules and further computations. The control flow in the rule control language is determined by the evaluation order of the operands. Graph rewrite sequences have a boolean return value; for a single rule, `true` means the rule was successfully applied to the host graph. A `false` return value means that the pattern was not found in the host graph.

In order to store and reuse return values of rewrite sequences and most importantly, for passing return values of rules to other rules, *variables* can be defined. A variable is an arbitrary identifier which can hold a graph element or a value of one of the attribute or value types GrGen.NET knows. There are two kinds of variables available in GrGen.NET, i) graph global variables and ii) sequence local variables. A variable is alive from its first declaration on: graph global variables are implicitly declared upon first usage of their name, sequence local variables are explicitly declared with a typed variable declaration of the form `name:type`. Graph global variables must be prefixed with a double colon `::`, local variables are referenced just with their name (i.e. *Ident*ifier). Graph global variables are untyped; their values are typed, though, so type errors cause an exception at runtime. They belong to and are stored in the graph processing environment – if you save the graph in GrShell then the variables are saved, too, and restored next time you load the saved graph. Further on, they are nulled if the graph element assigned to them gets deleted (even if this happens due to a transaction rollback), thus saving one from debugging problems due to zombie elements (you may use the `def()` operator to check during execution if this happened). Sequence local variables are typed, so type errors are caught at compile time (parsing time for the interpreted sequences); an assignment of an untyped variable to a typed variable is checked at runtime. They belong to the sequence they appear in, their life ends when the

sequence finishes execution (so there is no persistence available for them as for the graph global variables; neither do they get nulled on element deletion as the graph does not know about them).

If used in some rule, i.e. within an `exec`, named graph elements of the enclosing rule are available as read-only variables.

Note that we have two kinds of return values in graph rewrite sequences. Every rewrite sequence returns a boolean value, indicating whether the rewriting could be successfully processed, i.e. denoting success or failure. Additionally rules may return graph elements. These return values can be assigned to variables on the fly (see example 58).

EXAMPLE (58)

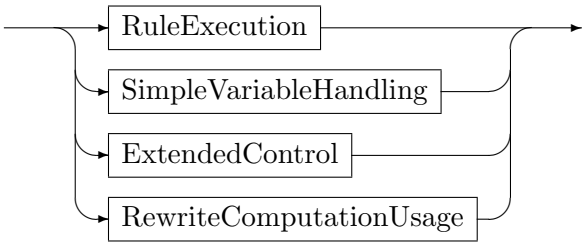
The graph rewrite sequences

```
1 (b,c)=R(x,y,z)=>a
2 a = ((b,c)=R(x,y,z))
```

assign the two returned graph elements from rule `R` to variables `b` and `c` and the information whether `R` matched or not to variable `a`. The first version is recommended.

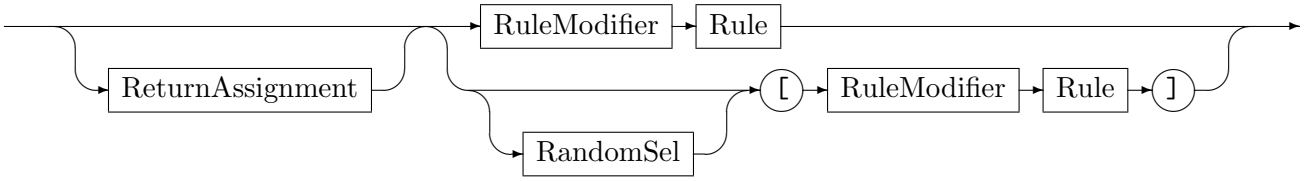
11.1 Rule Application

RewriteFactor

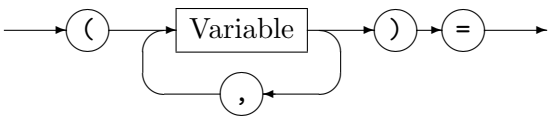


Rewrite factors are the building blocks of graph rewrite sequences. They are split into four major areas: rule (and sequence) application, simple variable handling, extended control, and sequence computation usages. Here we start with the most important one, applying rules. In section 11.3 we visit simple variable handling. In section 22.2 we have a look at advanced control, and in section 19.1 at the sequence computations.

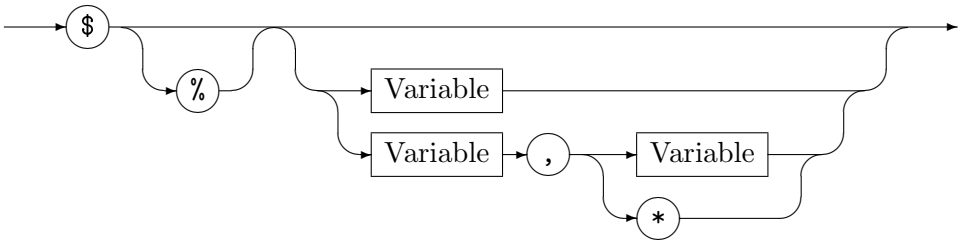
RuleExecution



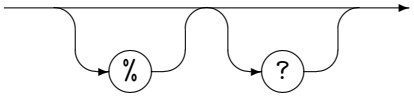
ReturnAssignment



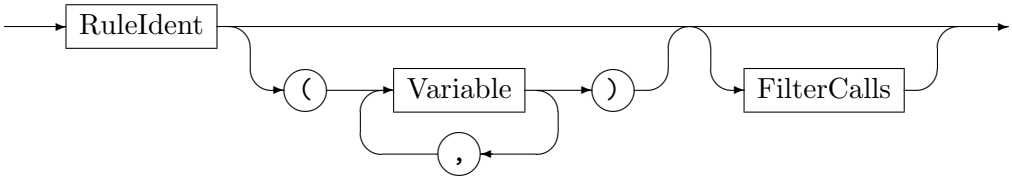
RandomSel



RuleModifier



Rule



The *RuleExecution* clause applies a single rule or test. In case of a rule, the first found pattern match will be rewritten. Application will fail in case no match was found and succeed otherwise. Variables and named graph elements can be passed into the rule. The returned graph elements can be assigned to variables again.

The rule modifier `?` switches the rule to a test, i.e. the rule application does not perform the rewrite part of the rule but only tests if a match exists. The rule modifier `%` is a multi-purpose flag. In the GrShell (see Chapter 25) it dumps the matched graph elements to `stdout`; in `debug`-mode (see Chapter 27) it acts as a break point (which is its main use in fact); you are also able to use this flag for your own purposes, when using GrGen.NET via its API interface (see Section 29.9). The filter calls (which allow e.g. to order the matches list and rewrite only the top ones, or to filter symmetric matches) are explained in 17.

The *RuleExecution* clause can be applied to a defined sequence (cf. 22.1), or an external sequence (cf. 34.5), too. Application will succeed or fail depending on the result of the body of the sequence definition called. In case of success, the output variables of the sequence definition are written to the destination variables of the assignment. In case of failure, no assignment takes place, so sequence calls behave the same as rule calls. The break point `%` can be applied to a sequence call, but neither the `?` operator nor all braces (`[]`).

The square braces (`[]`) introduce a special kind of multiple rule execution: every pattern match produced by the rule will be rewritten. If at least one was found, rule execution will succeed, otherwise it will fail. If *Rule* returns values, the value of each rule application will be returned. This happens in the form of an array, or better one array per return parameter, with type `array<T>` for return parameter type `T`. The return parameter arrays are filled with as many values as there were succeeding rule applications.

The random match selector `$[r]` searches for all matches and then randomly selects one of them to get rewritten. Rule application will fail in case no match was found and succeed otherwise (the return values are of array type, as with all-bracketed rule calls without random match selector). An `%` appended to the `$` denotes a choice point allowing the user to choose the match to be applied from the available ones in the debugger (see Chapter 27).

The generalized version `$u[r]` searches for all matches and then randomly selects `u` of them to be rewritten, but at most as much as are available (`$[r]` is equivalent to `anonymousTempVar=1 & $anonymousTempVar[r]`. So you specify an upper bound on the matches to be rewritten

this way. Here also, rule application will fail in case no match was found and succeed otherwise (and array-typed values are returned).

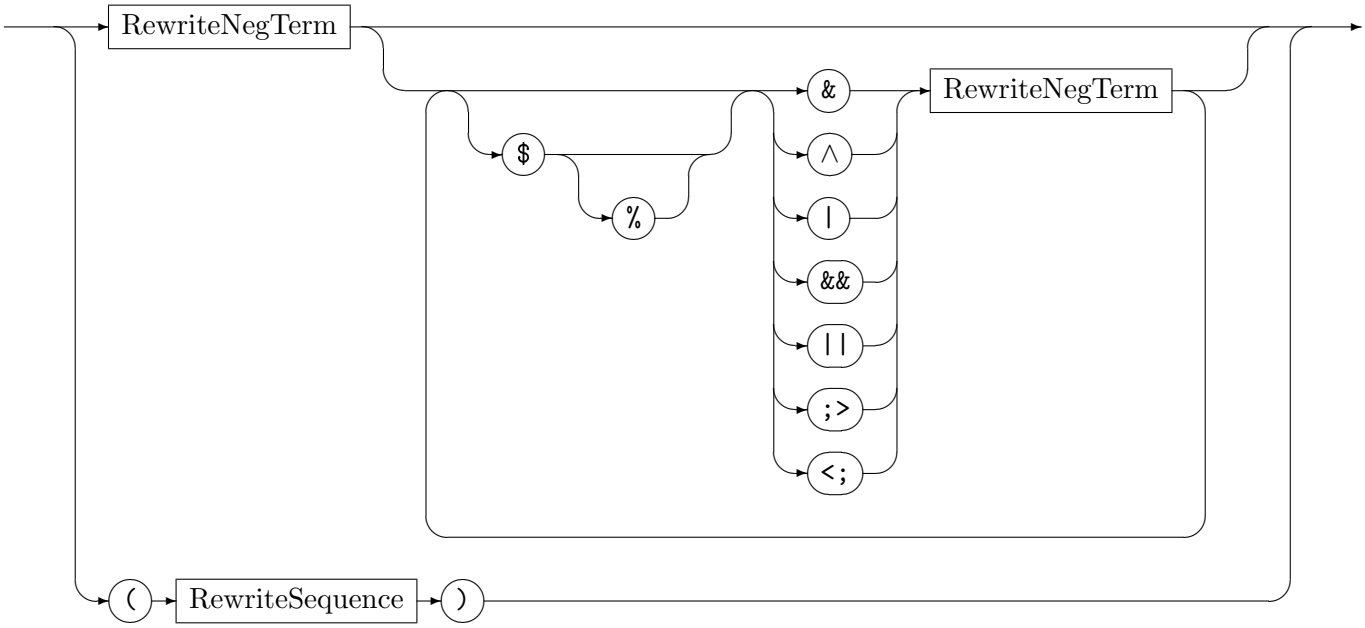
An even more generalized version `$l,u[r]` allows you to increase the lower bound for rule application success by giving a variable containing the required minimum amount of matches before the comma-separated upper bound variable. In order to be allowed to give a lower bound, you have to specify an upper bound, yet it may be set to unlimited with the `*` wildcard.

EXAMPLE (59)

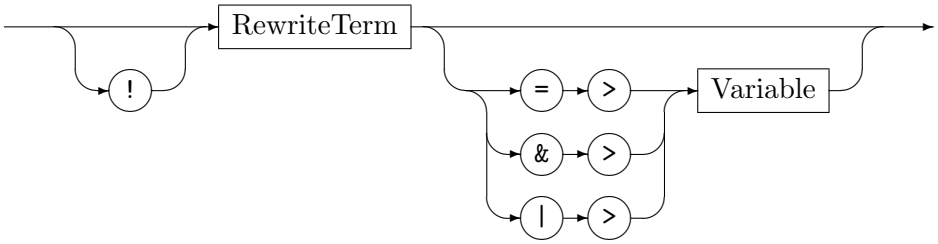
The sequence `(u,v)=r(x,y)` applies the rule `r` with input from the variables `x` and `y` on the host graph and assigns the return elements from the rule to the variables `u` and `v` (the parenthesis around the out variables are always needed, even if there's only one variable assigned to). The sequence `$[t]` determines all matches of the parameterless rule `t` on the host graph, then one of the matches is randomly chosen and executed. Rule `r` applied with all-bracketing may look like `(u:array<Node>,v:array<int>)=[r(x,y)]`, note the return value arrays, compared to a single application with `(u:Node,v:int)=r(x,y)`.

11.2 Logical and Sequential Connectives

RewriteSequence



RewriteNegTerm



A graph rewrite sequence consists of several rewrite terms linked by operators. Table 11.1 gives the priorities and semantics of the operators, priorities in ascending order. Forcing execution order against the priorities can be achieved by parentheses. The modifier `$` changes

the semantics of the following operator to randomly execute the left or the right operand first (i.e. flags the operator to act commutative); usually operands are executed / evaluated from left to right if not altered by bracketing. In contrast the sequences `s`, `t`, `u` in `s $<op> t $<op> u` are executed / evaluated in arbitrary order. The modifier `%` appended to the `$` overrides the random selection by a user selection (cf. see Chapter 27, choice points).

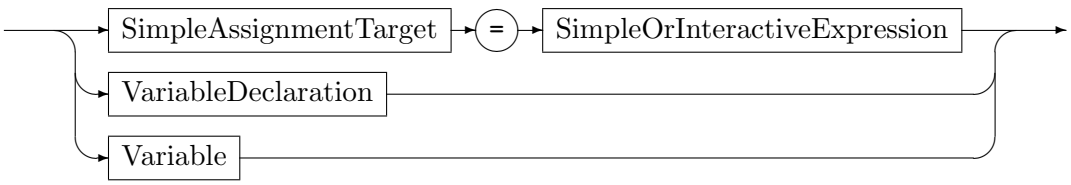
The assign-to operator `=>` optionally available at the end of the *RewriteNegTerm* assigns the (negated in case of `!`) result of the *RewriteTerm* execution to the given variable; the and-to `&>` operator assigns the conjunction and the or-to `|>` operator assigns the disjunction of the original value of the variable with the sequence result to the variable.

Operator	**Meaning**
`s1 <; s2`	Then-Left, evaluates `s1` then `s2` and returns(/projects out) the result of `s1`
`s1 ;> s2`	Then-Right, evaluates `s1` then `s2` and returns(/projects out) the result of `s2`
`s1 || s2`	Lazy Or, the result is the logical disjunction, evaluates `s1`, only if `s1` is false `s2` gets evaluated
`s1 && s2`	Lazy And, the result is the logical conjunction, evaluates `s1`, only if `s1` is true `s2` gets evaluated
`s1 | s2`	Strict Or, evaluates `s1` then `s2`, the result is the logical disjunction
`s1 ^ s2`	Strict Xor, evaluates `s1` then `s2`, the result is the logical antivalence
`s1 & s2`	Strict And, evaluates `s1` then `s2`, the result is the logical conjunction
`!s`	Negation, evaluates `s` and returns its logical negation

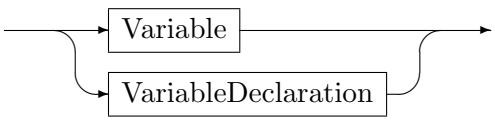
Table 11.1: Semantics and priorities of rewrite sequence operators

11.3 Simple Variable Handling

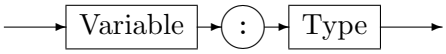
SimpleVariableHandling



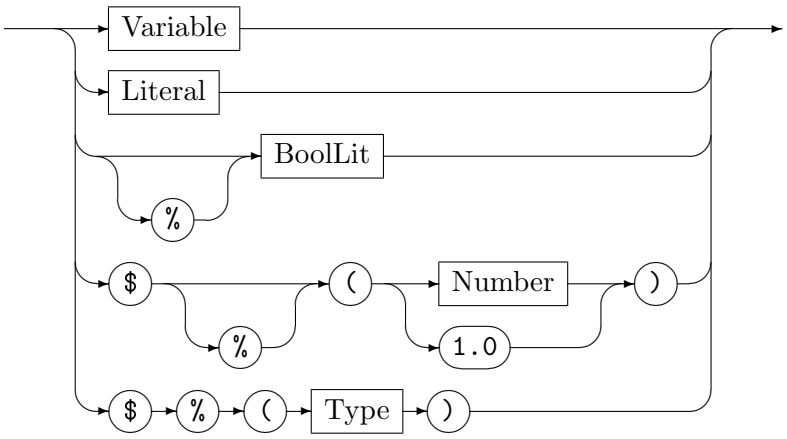
SimpleAssignmentTarget



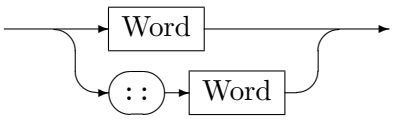
VariableDeclaration



SimpleOrInteractiveExpression



Variable



The simple variable handling in the sequences allows to assign a variable or a constant to a variable, to interactively query for an element of a given type or a number and assign it to a variable, or to declare a local variable; these constructs always result in true/success. In addition, a boolean variable may be used as a predicate; using such a variable predicate together with the sequence result assignment allows to directly transmit execution results from one part of the sequence to another one. Furthermore, a boolean constant may be used as a predicate. These sequence constants being one of the boolean literals `true` or `false` come in handy if a sequence is to be evaluated but its result must be a predefined value; furthermore a break point may be attached to them.

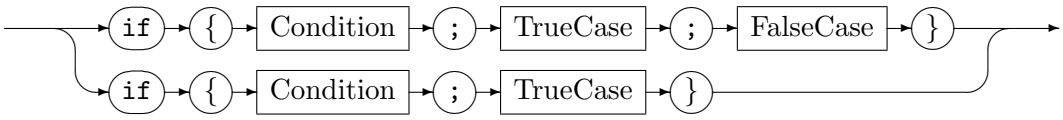
Variables can hold graph elements, or values of value/attribute types, including booleans. The typed explicit declaration (which may be given at an assignment, rendering that assignment into an initialization) introduces a sequence local variable, the name alone references a sequence local variable. A global variable is accessed with the double colon prefix, it gets implicitly declared if not existing yet (you can't declare a graph global variable). The random number assignment `v=$(42)` assigns an integer random number in between 0 and 41 (42 excluded) to the variable `v`. The random number assignment `v=$(1.0)` assigns a double random number in between 0.0 and 1.0 exclusive to the variable `v` (here you can't change the upper bound as with the integer assignment). Appending a `%` changes random selection to user selection (defining a choice point). The user input assignment `v=$%(string)` queries the user for a string value – this only works in the GrShell. The user input assignment `v=$%(Node)` queries the user for a node from the host graph – this only works in the GrShell in debug mode. The non simple variable handling is given in 19.1, even further variable handling constructs are given in 19.3.

EXAMPLE (60)

The sequence `(x)=s || (x)=t ;> [r(x)] & !u(::k,::l)` is executed in two halves, first `(x)=s || (x)=t`, then `[r(x)] & !u(::k,::l)`, as the then-right operator exercises the weakest binding of the used operators. The evaluation result of the first part is thrown away after it was computed, only the result of the second part defines the outcome of the sequence. From the first part, first `s` is executed, writing `x` in case of success. If `s` matches, execution of the left part is complete as the outcome of the lazy or is determined to be `true`, and `t` not needed any more. If it does not match, `t` is executed, defining `x` (in case of success, which we assume). Then the right part is executed, first applying `r` on all matches found for the previously written `x` argument, then `u` on the values stored in the global variables used as arguments. Here, `u` is executed due to the eager operator even if `r` was not found, thus forcing the result to be `false`. The result of the entire sequence is true iff `r` was found at least once, and `u` was *not* found (because of the negation). But to really give a valid sequence, we need to first declare the local variable `x` specifying its type, in contrast to the dynamically typed `::k` and `::l` global variables that we can use right away, so the complete sequence is `x:Node ;> (x)=s || (x)=t ;> [r(x)] & !u(::k,::l)`.

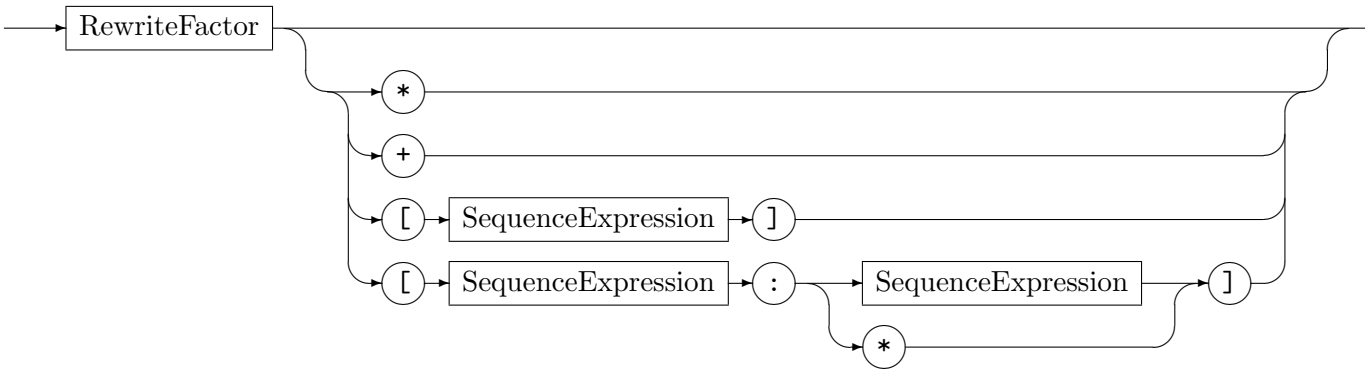
11.4 Decisions and Loops

ExtendedControl



The conditional sequences, or condition execution (/decision) statement `if` executes the condition sequence, and if it yielded true executes the true case sequence, otherwise the false case sequence. The sequence `if{Condition;TrueCase}` is equivalent to `if{Condition;TrueCase;true}`, thus giving a lazy implication.

RewriteTerm



A rewrite term consists of a rewrite factor which can be executed multiple times. The star (`*`) executes a sequence repeatedly as long as its execution does not fail. Such a sequence always returns `true`. A sequence `s+` is equivalent to `s && s*`. The brackets (`[m]`) execute a sequence repeatedly as long as its execution does not fail but m times at most; the min-max-brackets (`[n:m]`) additionally fail if the minimum amount n of iterations was not reached.

EXAMPLE (61)

The sequence `if{ (x:Node)=s; (::cnt)=r(x,::cnt)* ; count[q(x)]=>::cnt ;> true }` executes first `s`, writing the output value to the variable `x` that is declared on-the-fly. The result of `s` is used to decide which part to execute, in case of `true` it's `(::cnt)=r(x,::cnt)*`, in case of `false` it's `count[q(x)]=>::cnt ;> true`. The former executes the rule `r` as often as it matches, incrementing a variable `::cnt` used as counter each time (assuming that `r` contains a `return(incnt+1)` for an input parameter `incnt`). Each match is sought in the host graph at the state left behind by the execution of the rule in the previous iteration step. The latter sequence part executes the rule `q` on all matches in the current host graph at once, assigning the number of matches found (equalling the number of rewrites) to `::cnt`. The result of the overall sequence is always `true`, for one because the star operator used in the true-case always succeeds, and for the other because the false-case explicitly fixes its result to the constant `true`.

NOTE (29)

Consider all-bracketing introduced in the first section for rewriting all matches of a rule instead of iteration if they are independent. Attention: The all bracketing is **not** equal to `Rule*`. Instead this operator collects all the matches first before starting to rewrite. So if one rewrite destroys other matches or creates new match opportunities the semantics differ; in particular the semantics is unsafe, i.e. one needs to avoid deleting or retyping a graph element that is bound by another match (will be deleted/retyped there). On the other hand this version is more efficient and allows one to get along without marking already handled situations (to prevent a rule matching again and again because the match situation is still there after the rewrite; normally you would need some match preventing device like a negative or visited flags to handle such a situation).

11.5 Quick reference table

Table 11.2 lists the basic operations of the graph rewrite sequences at a glance.

`s ;> t`	Execute `s` then `t`. Success if `t` succeeded.
`s <; t`	Execute `s` then `t`. Success if `s` succeeded.
`s | t`	Execute `s` then `t`. Success if `s` or `t` succeeded.
`s || t`	The same as `s | t` but with lazy evaluation, i.e. if `s` is successful, `t` will not be executed.
`s & t`	Execute `s` then `t`. Success if `s` and `t` succeeded.
`s && t`	The same as `s & t` but with lazy evaluation, i.e. if `s` fails, `t` will not be executed.
`s ^ t`	Execute `s` then `t`. Success if `s` or `t` succeeded, but not both.
`if{r;s;t}`	Execute `r`. If `r` succeeded, execute `s` and return the result of `s`. Otherwise execute `t` and return the result of `t`.
`if{r;s}`	Same as `if{r;s;true}`
`!s`	Switch the result of `s` from successful to fail and vice versa.
`$<op>`	Use random instead of left-associative execution order for `<op>`.
`s*`	Execute `s` repeatedly as long as its execution does not fail.
`s+`	Same as `s && s*`.
`s[n]`	Execute `s` repeatedly as long as its execution does not fail but `n` times at most.
`s[m:n]`	Same as `s[n]` but fails if executed less than m times.
`s[m:*]`	Same as `s*` but fails if executed less than m times.
`?`*`Rule`*	Switches *Rule* to a test.
`%`*`Rule`*	This is the multi-purpose flag when accessed from LIBGR. Also used for graph dumping and break points.
`[`*`Rule`*`]`	Rewrite every pattern match produced by the action *Rule*.
`[`*`Rule`*`\f]`	Rewrite every pattern match produced by the action *Rule* that was left by the filter `f`, in the order left by the filter.
`(v:array<T>)=[`*`Rule`*`]`	Searches for all matches and rewrites them, for *Rule* returning `T`.
`true`	A constant acting as a successful match.
`false`	A constant acting as a failed match.
`v`	A boolean variable acting as a predicate.

Let `r`, `s`, `t` be sequences, `u`, `v`, `w` be variable identifiers, `f` be a filter, `<op>` be $\in \{|, \hat{}, \&, ||, \&\&\}$

Table 11.2: Sequences at a glance

CHAPTER 12

ADVANCED MATCHING AND REWRITING

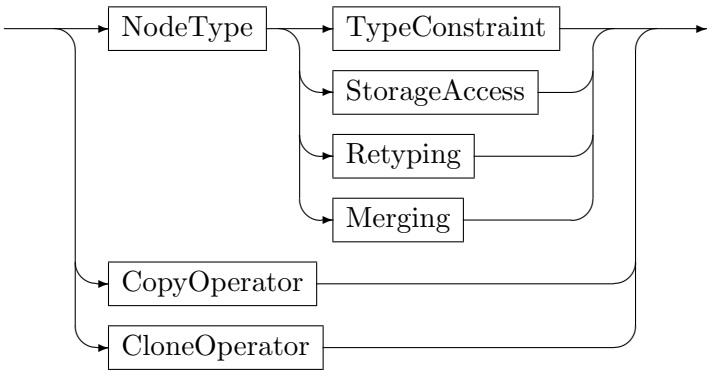
The following rewrite rule mentioned in Geiß et al. [GBG+06] shows some of the advanced constructs. In this chapter we explain the constructs of Example 62 that are exceeding the basics already introduced in Chapter 5, and the nested `negative`s already described in Chapter 7. Furthermore, we visit some additional advanced constructs.

EXAMPLE (62)

```
1  #using "SomeModel.gm"
2
3  rule SomeRule {
4    n1:NodeTypeA;
5    n2:NodeTypeA;
6    hom(n1, n2);
7    n1 --> n2;
8    n3:NodeTypeB;
9    negative {
10     n3 -e1:EdgeTypeA-> n1;
11     if {n3.a1 == 42*n2.a1;}
12   }
13   negative {
14     n4:Node\(NodeTypeB);
15     n3 -e1:EdgeTypeB-> n4;
16     if {typeof(e1) >= EdgeTypeA;}
17   }
18   replace {
19     n5:NodeTypeC<n1>;
20     n3 -e1:EdgeTypeB-> n5;
21     eval {n5.a3 = n3.a1*n1.a2;}
22   }
23 }
```

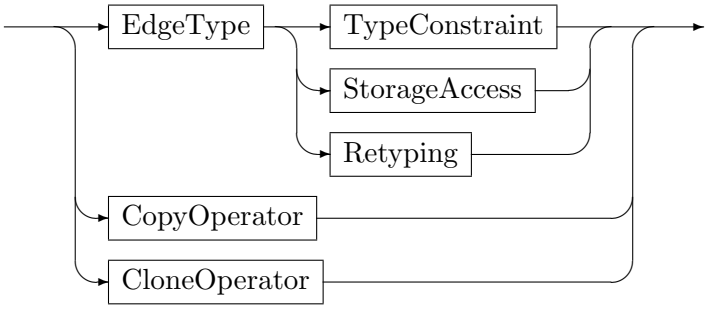
The advanced *modifiers* introduced in the following section allow annotating patterns or actions with keywords which restrict what graph patterns are accepted as matches (some of them independent of the rewrite part, some of them depending on the rewrite specification). But first the advanced *matching* constructs are introduced in this section, before they are elaborated on in a later section: they allow requesting more fine-grained or more dynamically what types to match, as well as specifying a match from a storage. Followed by the advanced *rewrite* constructs which are handled in the same way (introduction here, then elaboration in a later section); they enable the specification of retyping (relabeling) and copying, as well as node merging and edge redirection.

AdvancedNodeTypeConstructs



Specifies a node of type *NodeType*, constrained in type with a *TypeConstraint* (see Section 12.2, *TypeConstraint*), or bound by a storage access (see 15.6, *StorageAccess*), or retyped with a *Retyping* (see Section 12.4, *Retyping*), or merged with a *Merging* (see Section 12.6, *Merging*). Alternatively it may define a node having the same type and bearing the same attributes as another matched node (see Section 12.5, *CopyOperator* and *CloneOperator*). Type constraints are allowed in the pattern part only. The CopyOperator and the CloneOperator as well as the Merging clause are allowed in the replace/modify part only. The Retyping clause is a chimera which restricts the type of an already matched node when used on the LHS, and casts to the target type when used on the RHS, which is its primary use.

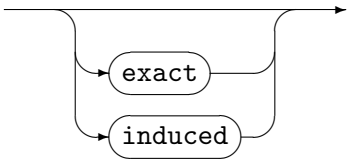
AdvancedEdgeTypeConstructs



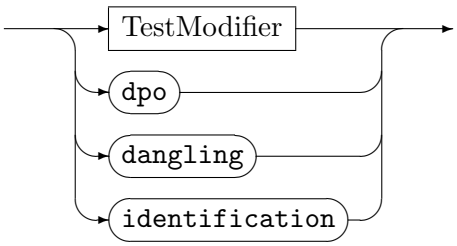
The *AdvancedEdgeTypeConstructs* specify an edge of type *EdgeType* or a copy of an edge. Type constraints are allowed in the pattern part only (see Section 12.2, *TypeConstraint*); the same holds for the storage access (see 15.6, *StorageAccess*). The CopyOperator and the CloneOperator as well as the Redirect clause are allowed in the replace/modify part only (see Section 12.5, *CopyOperator* and *CloneOperator*, see Section 12.7, *Redirect*). The Retyping clause is a chimera which restricts the type of an already matched edge when used on the LHS, and casts to the target type when used on the RHS (its primary use). Furthermore edges may be redirected, this is shown in Section 12.7, *Redirection*.

12.1 Rule and Pattern Modifiers

TestModifier



RuleModifier



By default GrGen.NET performs rewriting according to SPO semantics as explained in Section 5.4.1. This behaviour can be changed with *pattern modifiers* and *rule modifiers* (and the other advanced rewrite constructs introduced in the following sections which spoil the theoretical foundation but are highly useful in practice). Such modifiers add certain conditions to the applicability of a pattern. The idea is to match only parts of the host graph that look more or less exactly like the pattern. The level of "exactness" depends on the chosen modifier. A pattern modifier in front of the `rule`/`test`-keyword is equivalent to one modifier-statement inside the pattern containing all the specified nodes (including anonymous nodes). Table 12.1 lists the pattern modifiers with their semantics, table 12.2 lists the rule only modifiers with their semantics. Example 63 explains the modifiers by small toy-graphs.

Modifier	**Meaning**
`exact`	Switches to the most restrictive mode. An exactly-matched node is matched, if all its incident edges in the host graph are specified in the pattern.
`induced`	Switches to the induced-mode, where nodes contained in the same `induced` statement require their induced subgraph within the host graph to be specified, in order to be matched. In particular this means that in general `induced(a,b,c)` differs from `induced(a,b); induced(b,c)`.

Table 12.1: Semantics of pattern modifiers

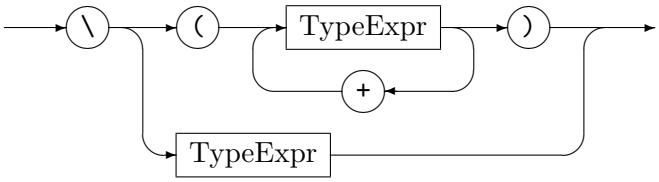
NOTE (30)

Internally all the modifier-annotated rules are resolved into equivalent rules in standard SPO semantics. The semantics of the modifiers is mostly implemented by NACs. In particular you might want to use such modifiers in order to avoid writing a bunch of NACs yourself. The number of internally created NACs is bounded by $\mathcal{O}(n)$ for `exact` and `dpo` and by $\mathcal{O}(n^2)$ for `induced` respectively, where n is the number of specified nodes.

12.2 Static Type Constraint

A static type constraint given at a node or edge declaration limits the types on which the pattern element will match (by excluding forbidden types).

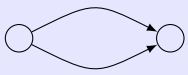
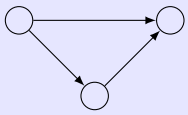
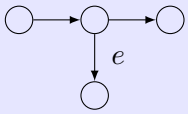
TypeConstraint



Modifier	Meaning
`dpo`	Switches to DPO semantics . This modifier affects only nodes that are to be deleted during the rewrite. DPO says—roughly spoken—that implicit deletions must not occur under any circumstances. To ensure this the dangling condition (see `dangling` below) and the identification condition (see `identification` below) get enforced (i.e. `dpo = dangling + identification`). In contrast to `exact` and `induced` this modifier applies neither to a pattern as such (can't be used with a `test`) nor to a single statement but only to an entire rule. See Corradini et al.[CMR+99] for a DPO reference.
`dangling`	Ensures the dangling condition . This modifier affects only nodes that are to be deleted during the rewrite. Nodes going to be deleted due to the rewrite part have to be specified exactly (with all their incident edges, `exact` semantics) in order to be matched. As with `dpo`, this modifier applies only to rules.
`identification`	Ensures the identification condition . This modifier affects only nodes that are to be deleted during the rewrite. If you specify two pattern graph elements to be homomorphically matched but only one of them is subject to deletion during rewrite, those pattern graph elements will never actually be matched to the same host graph element. As with `dpo`, this modifier applies only to rules.

Table 12.2: Semantics of rule only modifiers

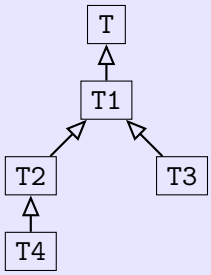
EXAMPLE (63)

Host Graph	Pattern / Rule	Effect
	`{ . --> .; }`	Produces no match for `exact` nor `induced`
	`{ x:node --> y:node; }`	Produces three matches for `induced(x,y)` but no match for `exact(x,y)`
	`{ x:node; induced(x); }`	Produces no match
	`{ --> x:node --> ; }` `modify{ delete(x); }`	Produces no match in DPO-mode because of edge `e`

A type constraint is used to exclude parts of the type hierarchy. The operator `+` is used to create a union of its operand types. So the following pattern statements are identical:

`x:T \ (T1 + ··· + Tn);`	`x:T;`
	`if {!(typeof(x) >= T1) && ···`
	`&& !(typeof(x) >= Tn)}`

EXAMPLE (64)

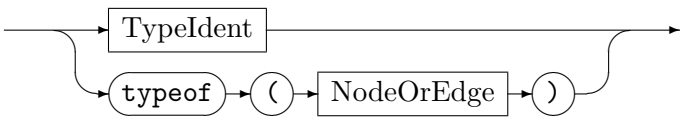


The type constraint `T\(T2+T3)` applied to the type hierarchy on the left side yields only the types `T` and `T1` as valid.

12.3 Exact Dynamic Type

In the following section we'll have a look at a language construct which allows to require an element to be typed the same as another element or to create an element with the same exact dynamic type as another element.

Type



The type of a graph element may be given by a type identifier, or a typeof denoting the exact dynamic type of a matched graph element. The element declaration `el:typeof(x)` introduces a graph element of the type the host graph element `x` is actually bound to. It may appear in the pattern or in the rewrite part. If it is given in the pattern, the element to match must be of the same exact dynamic type as the element referenced in the `typeof`, otherwise matching will fail. If is is given in the rewrite part, the element to create is created with the same exact dynamic type as the element referenced in the `typeof`; have a look at the next section for the big brothers of this language construct, the `copy` and `clone` operators, which are only applicable in the rewrite part.

EXAMPLE (65)

The following rule will add a reverse edge to a one-way street.

```
1  rule oneway {
2      a:Node -x:street-> y:Node;
3      negative {
4          y -:typeof(x)-> a;
5      }
6      replace {
7          a -x-> y;
8          y -:typeof(x)-> a;
9      }
10 }
```

Remember that we have several subtypes of `street`. By the aid of the `typeof` operator, the reverse edge will be automatically typed correctly (the same type as the one-way edge). This behavior is not possible without the `typeof` operator.

12.4 Retyping

In addition to graph rewriting, GrGen.NET allows graph relabeling[LMS99], too; we prefer to call it retyping. Nodes as well as edges may be retyped to a different type; attributes common to the initial and final type are kept. The target type does not need to be a subtype or supertype of the original type. Retyping is useful for rewriting a node but keeping its incident edges; without it you'd need to remember and restore those. Syntactically it is specified by giving the original node enclosed in left and right angles.

Retyping



Pattern (LHS)	**Rewrite (RHS)**	$r : L \longrightarrow R$	**Meaning**
`x:N1;`	`y:N2<x>;`	$r : \text{lhs}.x \mapsto \text{rhs}.x$	Match `x`, then retype `x` from `N1` to `N2` and bind name `y` to retyped version of `x`.
`e:E1;`	`f:E2<e>;`	$r : \text{lhs}.e \mapsto \text{rhs}.e$	Match `e`, then retype `e` from `E1` to `E2` and bind name `f` to the retyped version of `e`.

Table 12.3: Retyping of matched nodes and edges

Retyping enables us to keep all adjacent nodes and all attributes stemming from common super types of a graph element while changing its type (table 12.3 shows how retyping can be expressed both for nodes and edges). Retyping differs from a type cast: During rewriting both of the graph elements are alive. Specifically both of them are available for evaluation, a respective evaluation could, e.g., look like this:

```
eval {
  y.b = ( 2*x.i == 42 );
  f.a = e.a;
}
```

Furthermore the source and destination types need *not* to be on a path in the directed type hierarchy graph, rather their relation can be arbitrary. However, if source and destination

type have one or more common super types, then the respective attribute values are adopted by the retyped version of the respective node (or edge). The edge specification as well as *ReplaceNode* supports retyping. In Example 62 node `n5` is a retyped node stemming from node `n1`. Note, that—conceptually—the retyping is performed *after* the SPO conforming rewrite.

EXAMPLE (66)

The following rule will promote the matched city `x` from a `City` to a `Metropolis` keeping all its incident edges/streets, with exception of the matched street `y`, which will get promoted from `Street` to `Highway`, keeping all its adjacent nodes/cities.

```
rule oneway {
  x:City -y:Street->;

  replace {
    x_rt:Metropolis<x> -y_rt:Highway<y>->;
  }
}
```

The following rule will retype the matched city `x` to the exact dynamic type of `z` (which might be e.g. `Metropolis`).

```
rule retypeToTypeof {
  x:City; z:City;

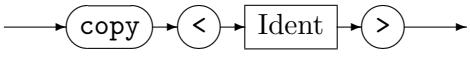
  modify {
    x_rt:typeof(z)<x>;
  }
}
```

The retyping clause `new:type<old>` can be used on the LHS, too. When it appears on the left hand side, the node(/edge) `old` is not changed in any way, instead a further node(/edge) `new` is made available for querying, being identical to `old` regarding the object reference, but additionally giving access to the attributes known to the `type` – if matching was successful. The construct tries to cast `old` to `new`, if successful `new` allows to access `old` as `type`, otherwise matching fails for this `old` (and binding another graph element to `old` is tried out). Please have a look at example 29 for more on this.

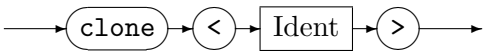
12.5 Copy and Clone

The copy and clone operators allow creating a node or edge of the type of another node/edge, bearing the same attributes like that other node/edge. They can be seen as an extended version of the `typeof` construct not only copying the exact dynamic type but also the attributes of the matched graph element. Together with the `iterated` construct they allow to simulate node replacement grammars or to copy entire structures, see 24 and 24.5. The difference in between the two operators lies in the handling of attributes of object class type, the copy operator copies them deeply (/transitively), including all objects referenced, the clone operator just assigns the attribute to the same reference; for more on them cf. 18.1.3.3.

CopyOperator



CloneOperator



EXAMPLE (67)

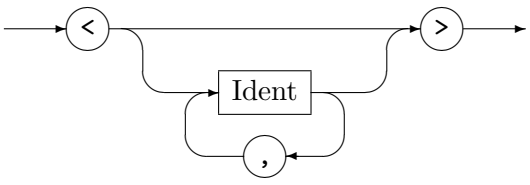
The following rule will add a reverse edge to a one-way street, of the exact dynamic subtype of `street`, bearing the same attribute values as the original street.

```
1  rule oneway {
2    a:Node -x:street-> y:Node;
3    negative {
4      y -:typeof(x)-> a;
5    }
6
7    replace {
8      a -x-> y;
9      y -:copy<x>-> a;
10   }
11 }
```

12.6 Node Merging

The retyping construct for nodes can be extended into a node merging construct, which internally redirects edges.

Merging



Merging enables us to fuse several nodes into one node. Syntactically it is given by a retyping clause which not only mentions the original node inside angle brackets, but several original nodes. Semantically the first node in the clause is retyped, then all edges of the other original nodes are redirected to the retyped node, and finally the other original nodes are deleted. As the type of the merging clause can be set to `typeof(first original node)`, a pure merging without retyping can be achieved.

EXAMPLE (68)

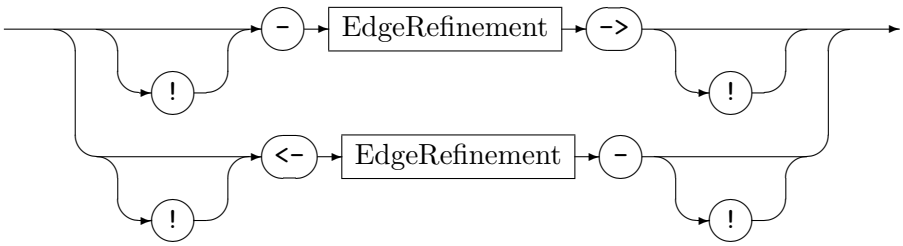
The following rule will match two `State`s and merge them. Every edge incident to `b` before and every edge incident to `w` before will be incident to the merged successor state `bw` afterwards; edges connecting the two `State`s become reflexive edges.

```
1  rule merge {
2    b:State;
3    w:State;
4    if { b.name == "Baden" && w.name == "Wuerttemberg"; }
5
6    modify {
7      bw:typeof(b)<b,w>;
8      eval { bw.name = b.name + w.name; }
9    }
10 }
```

12.7 Edge Redirection

The redirect statement allows to exchange the source or the target node of a directed edge (or both) with a different node; it can be seen as syntactic sugar for removing one edge and creating a new one with the source/target node being replaced by a different node, with the additional effect of keeping edge identity.

Redirect



Redirection is specified with an exclamation mark at the end to be redirected as seen from the edge center; the exclamation mark enforces the redirection which would normally be rejected by the compiler "This is different from what was matched, but that's intentional, make it happen!"

EXAMPLE (69)

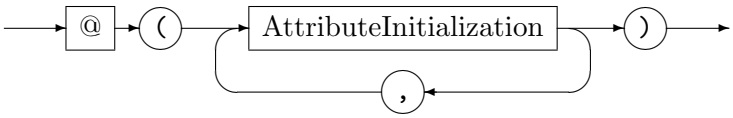
The following rule will reverse the one-way street in between `a` and `y` by rewriting the old source to the new target and the old target to the new source.

```
1  rule oneway {
2    a:Node -x:street-> y:Node;
3
4    modify {
5      a !<-x-! y;
6    }
7  }
```

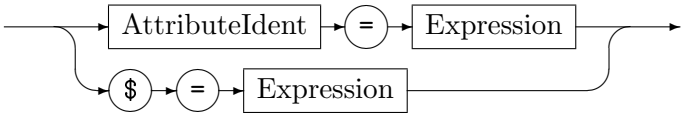
12.8 Attribute Initialization

At a node or edge declaration in the modify or replace part (causing creation of the element), you can give an attribute initialization list, with the name of the element (the persistent name of a named graph) seen as a special attribute.

AttributeInitializationList



AttributeInitialization



The initialization list is appended to the declaration with an at-Symbol, the dollar sign stands for the name of the element, whereas the other attributes of the type are simply given by their name (i.e. *Ident*ifier). The expression must be of the type of the attribute (and string in case of the name). The initialization list is more convenient than the otherwise needed `eval` block, containing an assignment per line – in fact it is syntactic sugar that is dissolved to `eval`-like assignments. Please take into account that the names must be unique if you intend to set them from the rules.

EXAMPLE (70)

The following example rule creates a new node `n` of type `N`, initializing the name to `"foo"`, the attribute `i` to 42, and the attribute `s` to `"Hallo Welt"`.

```
1 rule create {
2   modify {
3     n:N@($="foo", i=42, s="Hallo Welt");
4   }
5 }
```

CHAPTER 13

EMBEDDED SEQUENCES AND TEXTUAL OUTPUT/INPUT

In this chapter we'll have a look at language constructs which allow you to emit text from rules (and a light version of an inverse of it, the scan/tryscan functions), and at language constructs which allow you to execute a graph rewrite sequence at the end of a rule invocation (with direct access to the elements of the left and right patterns). Similar constructs are available for nested and subpatterns.

EXAMPLE (71)

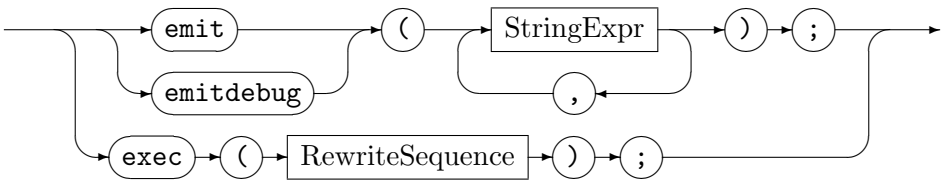
The example rule below works on a hypothetical network flow. We neither define all the rules nor the graph meta model, we just want to introduce you to the usage of the `exec` and `emit` statements, as well as their look and feel.

```
1  rule AddRedundancy
2  {
3    s: SourceNode;
4    t: DestinationNode;
5    modify {
6    ---
7      emit ("Source node is ", s.name, ". Destination node is ", t.name, ".");
8      exec ( (x:SourceNode) = DuplicateNode(s) & ConnectNeighbors(s, x)* );
9      exec ( [DuplicateCriticalEdge] );
10     exec ( MaxCapacityIncidentEdge(t)* );
11     emit ("Redundancy added");
12   }
13 }
```

13.1 Exec and Emit in Rules

The following syntax diagram gives an extension to the syntax diagrams of the Rule Set Language chapter 5:

ExecStatement



The statements `emit` and `emitdebug` as well as `exec` enhance the declarative rewrite part by imperative clauses. This means that these statements are executed in the same order as they appear within the rule, but particularly that they operate on the modified host graph, as they are executed *after* the rewrite part.

The rewrite part contains the `eval` statements, i.e. the execution statements work on the recalculated attributes. The rewrite part especially `delete`s pattern elements (explicitly in `modify` mode, implicitly in `replace` mode) – those consequently cannot be accessed in the execution statements anymore. *Attribute* values of deleted graph elements are still available for reading, though – the compiler temporarily stores them as needed, for your convenience.

Sequence execution

The `exec` statement executes a graph rewrite sequence, which is a composition of graph rewrite rules. See Chapter 11 for a description of graph rewrite sequences.

The potential input to the embedded sequence are the graph elements declared in the LHS and RHS patterns (or their attributes), the actual *input* are the ones really referenced by the sequence. The *output* from an embedded sequence is defined with `yield`s to `def` variables from the RHS pattern – to be used by following `exec`ution statements, or by the `return` statement that is executed last.

The `exec` statement is one of the means available in GrGen.NET to build complex rules and split work into several parts, see Chapter 24 for a discussion of this topic. The noteworthy point is that this happens rule-internally, without having to pass parameters out into an enclosing sequence and in again from the enclosing sequence to following rule applications (as it is required by external composition).

EXAMPLE (72)

This is an example for returning elements yielded from an `exec` statement. The results of the rule `bar` are written to the variables `a` and `b`; The `yield` is a prefix to an assignment showing that the target is from the outside.

```
rule foo : (A,B)
{
  modify {
  ---
    def u:A; def v:B;
    exec( (a,b)=bar ;> yield u=a ;> yield v=b) );
    return(u,v);
  }
}
```

Text output

The `emit` and the `emitdebug` statement prints a string or a comma-separated sequence of strings to the currently associated output stream or always the console. See Chapter 6 for a description of string expressions.

The argument(s) must be of string type, but any type is automatically cast into its string representation as needed. Prefer comma separated arguments over string concatenation, they are more efficient as no intermediate strings need to be computed, just to be garbage collected thereafter.

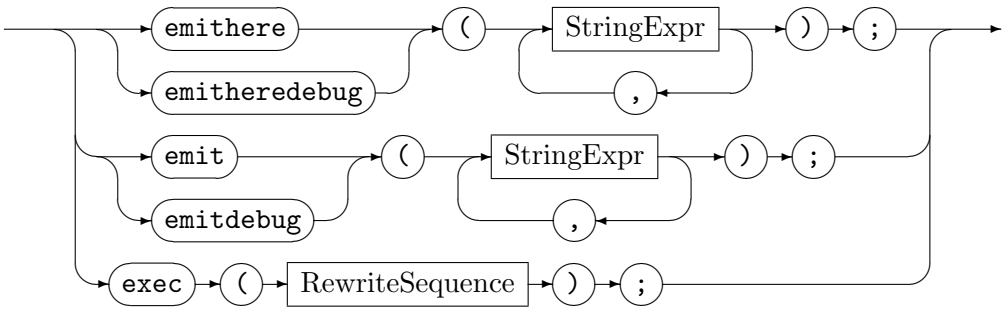
The currently associated output stream is by default `stdout`, you use the emit then typically to print debug or execution state information to the console. It may be a text file (the one output was redirected to, see the `redirect emit` command in Chapter 25), you

use the emit then commonly to write a file format of your choice (maybe a simple extract, maybe a full-blown graph export), building a model-to-text transformation. The `emithere` statement always prints out to the console, even if output was redirected.

13.2 Deferred Exec and Emithere in Nested and Subpatterns

The following syntax diagram gives an extension to the syntax diagrams of the Subpatterns chapter 8:

SubpatternExecEmit



The statements `emit` and `emitdebug`, `emithere` and `emitheredebug`, as well as `exec` enhance the declarative rewrite part of nested and subpatterns by imperative clauses.

The `emit` and `emithere` statements (as well as `emitdebug` and `emitheredebug`) get executed during rewriting before the `exec` statements; the `emithere`-statements get executed before the `emit` statements, in the order in between the subpattern rewrite applications they are specified syntactically (see 10.4 for more on this). The `debug`-suffixed versions always print to the console, the plain versions may get redirected to a file.

The *deferred* `exec` statements are executed in the order as they appear syntactically, but particularly after the top-level rule which employed the pattern they are contained in was executed. They are a variant of the rule-based embedded `exec`-statements from the *ExecStatement* (mentioned in 5.4.3 and explained above in 13.1), only available in the rewrite parts of subpatterns or nested alternatives/iterateds. They are executed after the original rule employing them was executed (which may be syntactically separated by a deep nesting of patterns or a deep chain of subpattern usages in between), so they can't get extended by `yields`, as the containing rule is not available anymore when they get executed (and in case they appear inside a subpattern maybe not even known statically).

EXAMPLE (73)

The exec from `Subpattern sub` gets executed after the exec from `rule caller` was executed.

```
rule caller
{
  n:Node;
  sub:Subpattern();

  modify {
    sub();
  ---
    exec(r(n));
  }
}
pattern Subpattern
{
  n:Node;
  modify {
  ---
    exec(s(n));
  }
}
```

EXAMPLE (74)

This is an example for emithere, showing how to linearize an expression tree in infix order.

```
pattern BinaryExpression(root:Expr)
{
  root --> l:Expr; le:Expression(l);
  root --> r:Expr; re:Expression(r);
  root <-- binOp:Operator;

  modify {
    le(); // rewrites and emits the left expression
    emithere(binOp.name); // emits the operator symbol in between the left tree and the
        right tree
    re(); // rewrites and emits the right expression
  }
}
```

NOTE (31)

The embedded sequences are executed after the top-level rule which contains them (in a nested pattern or in a used subpattern) was executed; they are *not* executed during subpattern rewriting. They allow you to put work you can't do while executing the rule proper (e.g. because an element was already matched and is now locked due to the isomorphy constraint) to a waiting queue which gets processed afterwards — with access to the elements of the rule and contained parts which are available when the rule gets executed. Or to just split the work into several parts, reusing already available functionality, see 24 for a discussion on this topic.

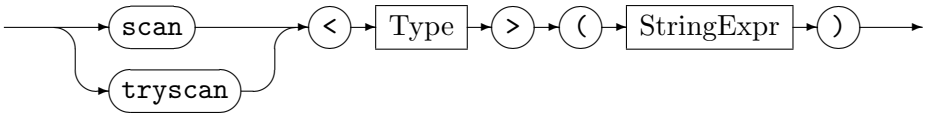
NOTE (32)

And again — the embedded sequences are executed *after* the rule containing them; thus rule execution is split into two parts, a declarative of parts a) and b), and an imperative. The declarative is split into two subparts: First the rule including all its nested and subpatterns is matched. Then the rule modifications are applied, including all nested and subpattern modification.
After this declarative step, containing only the changes of the rule and its nested and used subpatterns, the deferred execs which were spawned during the main rewriting are executed in a second, imperative step; during this, a rule called from the sequence to execute may do other nifty things, using further own sequences, even calling itself recursively with them. First all sequences from a called rule are executed, before the current sequence is continued or other sequences of its parent rule get executed (depth first).
Note: all changes from such dynamically nested sequences are rolled back if a transaction/a backtrack enclosing a parent rule is to be rolled back (but no pending sequences of a parent of this parent).

13.3 Scan and Tryscan

The `scan` and `tryscan` functions can be seen as a form of text-to-model transformation. They are a kind of inverse of emit (which is also available as a procedure, see 14.4), and are used to transform from (instead of to) strings that are intended for user consumption (this stands in contrast to machine consumption, and the full serialization/deserialization offered by the `export` procedure and the `import` function, cf. 16.4).

ScanTryscanCall



The `scan` function expects a target type that specifies the type of the object to be computed from the string, and an expression specifying the string that is to be transformed. It returns the obtained object (scalar values and containers are created, graph elements specified by their persistent name in contrast are fetched from the graph, object classes and match types are not supported), or diverges (i.e. crashes). The `tryscan` function expects the same parameters, but it returns whether the given string can be transformed into an object of the requested type.

EXAMPLE (75)

This is an example showing how tryscan and scan are used.

```
1  if(tryscan<int>("42") && tryscan<set<N>>("set<N>{$0,$1}")) {
2    def var i:int = scan<int>("42");
3    def ref s:set<N> = scan<set<N>>("set<N>{$0,$1}");
4  }
```

CHAPTER 14

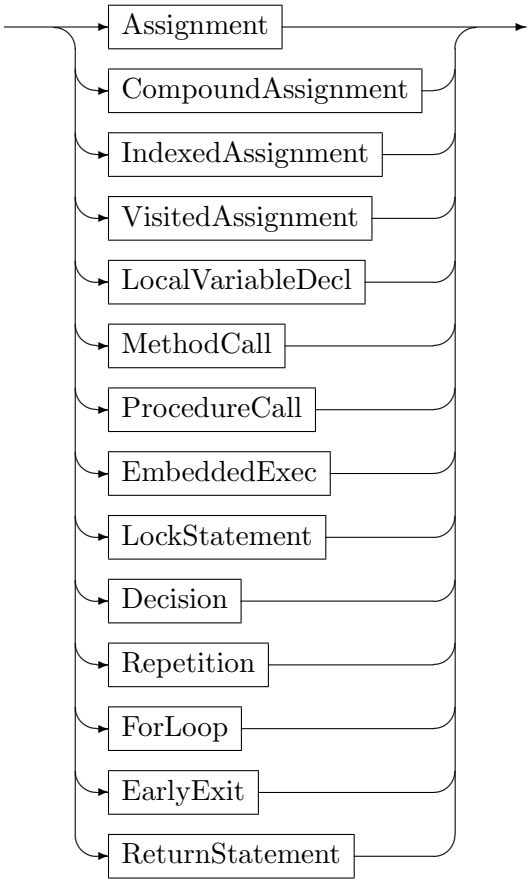
COMPUTATIONS (ATTRIBUTE EVALUATION STATEMENTS)

In this chapter we'll have a look at the *statements* of the *computations* available in the rule language. The *computations* are the superordinate concept, they consist of side-effect free *expressions* already introduced in 6, and side-effect causing *statements*. Their corresponding abstractions are the *functions* free of externally visible side-effects, defining an expression drop-in replacement, and the *procedures* that can only be employed from statement context, defining a statement drop-in replacement; we'll visit those abstractions at the end of this chapter.

The most important statement is the *attribute assignment*, which is used for assigning new values computed from expressions to graph element attributes. Besides this, further types of assignments are available, and several control flow constructs as known from imperative programming languages of the C-family. Furthermore, container methods may be called (more on this in the following chapter 15), and global graph functions may be called (more on this in the following chapter 16).

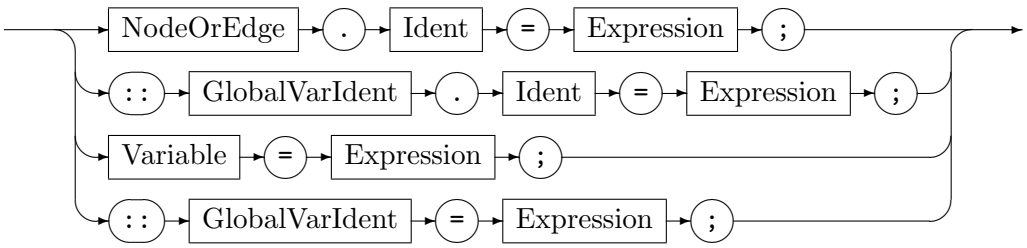
An example implementing depth-first search, breadth-first search, and iterative deepening as computations of the rule language can be found in `test/should_pass`, in `/DfsBfsSearch.grg`; you may have a look at the `/computations_*.grg` files in that folder for further (but fabricated) examples.

Statement



14.1 Assignments

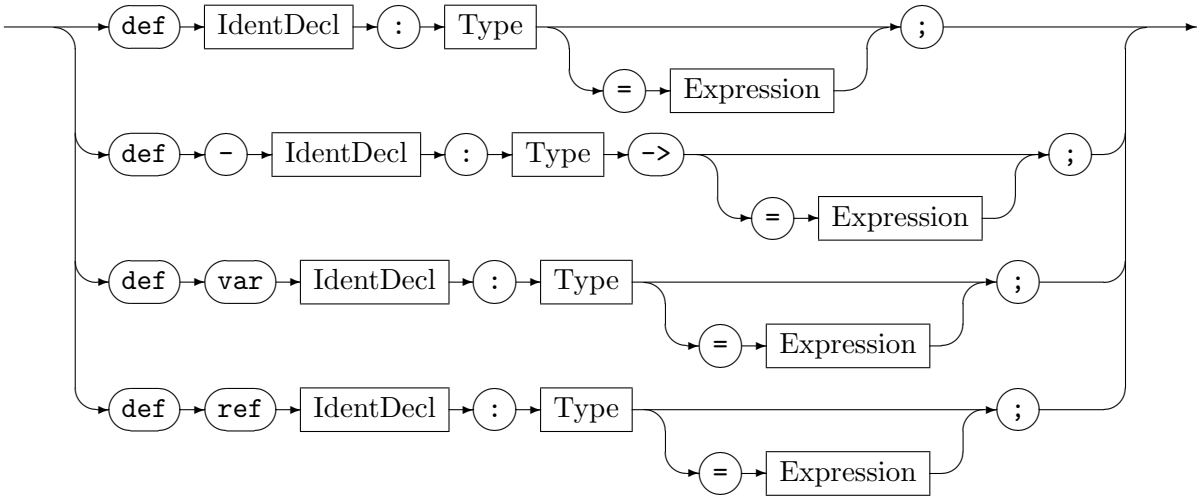
Assignment



Evaluation statements have imperative semantics. In particular, GrGen.NET does not care about data dependencies (You must care for them! The assignments are just executed one after the other.). Assignment is carried out using value semantics, even for entities of container or `string` type. The only exception is the type `object`, there reference semantics is used. You can find out about the available *Expressions* in chapter 6.

14.2 Local Variable Declarations

LocalVariableDecl



Local variables can be defined with the syntax known for local def pattern variables, as already introduced in 10.4, i.e. `def var name:type` for elementary variables, or `def ref name:type` for container variables or internal class object variables or internal transient class object variables, or `def name:type` for nodes, or `def -name:type->` for edges. At their definition, an initializing expression may be given.

EXAMPLE (76)

The following rule eval part gives some example statements regarding data flow.

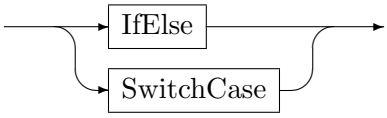
```
1  var ::v:int;
2
3  rule example
4  {
5    n:N; // we assume attributes a:int, arr:array<int>, narr:array<N>
6
7    modify {
8    ---
9       eval {
10         def var sum:int = 0; // declares a local variable of type int and initializes it
11         n.a = 42; // assigns 42 to the attribute a of the node n of type N
12         ::v = n.a - 1; // assigns the global variable the value 41
13         def ref arr:array<int> = array<int>[ sum,1,2,3 ];
14         arr[0] = arr[1] + n.arr[sum];
15         def ref narr:array<N> = array<N>[ n ];
16         //narr[0].a = 0xAFFE; -- currently not possible, you must work around it this way:
17         def tmp:N = narr[0];
18         tmp.a = 0xAFFE;
19         n.narr[0] = n; // this is possible in contrast
20      }
21   }
22 }
```

14.3 Control flow

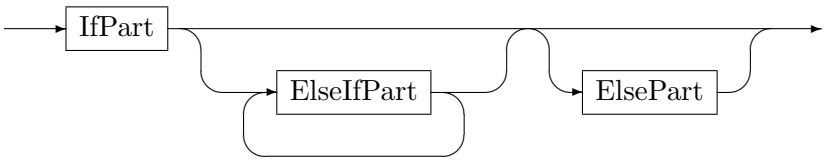
Based on the value of a boolean expression different computations can be executed conditionally with the `if` and `else` statements. Please note that the braces are mandatory. With

the `switch` and `case` statements, you can match constants against the value of an expression computed once. Every case-branch is enclosed in an block, at block end also the execution of the block ends, there is no implicit fall-through carried out, a fall-through is not even available. Only one else branch is allowed, it is hit in case no other branch was executed. The construct is only available for expressions with values of type `byte`, `short`, `int`, `long`, `string`, `boolean`, and `enum` types.

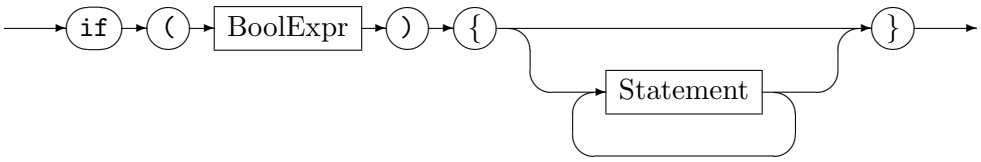
Decision



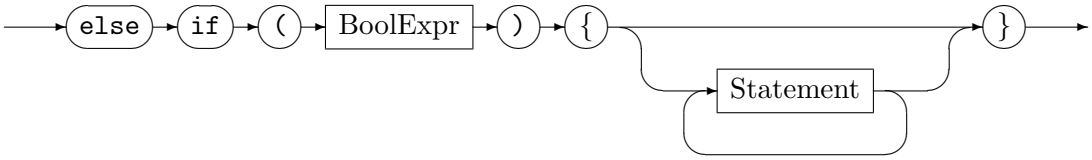
IfElse



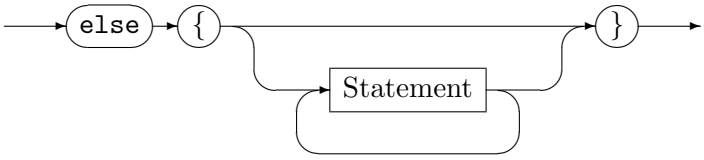
IfPart



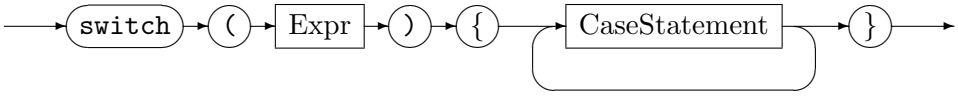
ElseIfPart



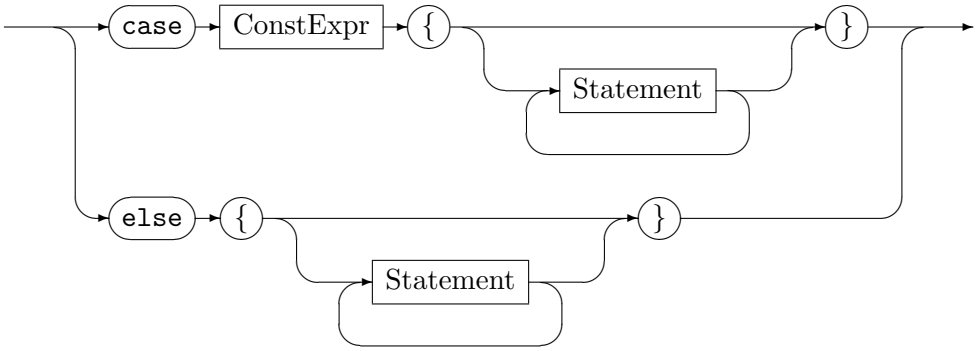
ElsePart



SwitchCase

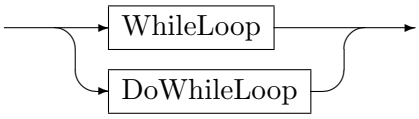


CaseStatement

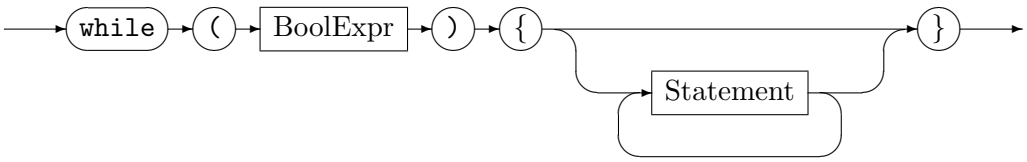


Computations may be executed repeatedly by the `while` and `do-while` statements. Please note that the braces are mandatory and that the `do-while` loop comes without a terminating semicolon.

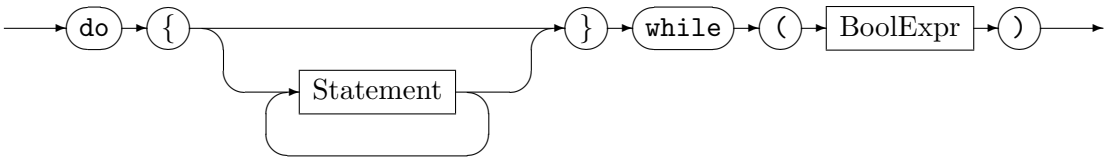
Repetition



WhileLoop

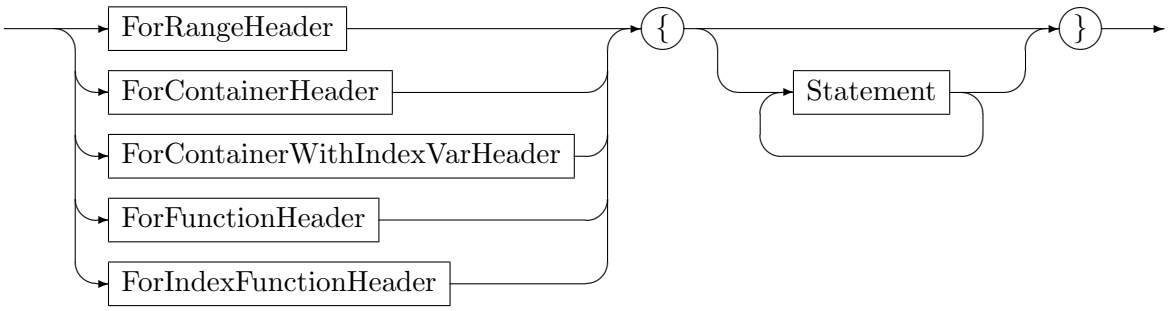


DoWhileLoop

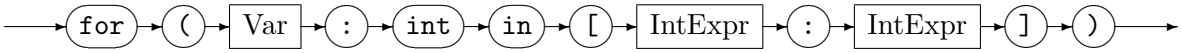


A for loop is available for iterating a range of integers, with the lower and the upper bounds specified. The containers introduced in chapter 15 may be iterated over with a for loop, one assigning the current value to a variable, the other assigning the current index and the current value to a variable. Note that you can only iterate directly over a container variable, not over a container typed attribute — as of now, you have to assign a container typed attribute to a container variable in case you want to iterate over it. Furthermore, a for loop is available for iterating the nodes or edges of the host graph with multiple graph access functions, as introduced in chapter 16. Moreover, a for loop is available for iterating the graph elements stored in an index, with multiple index access functions, see chapter 30 or more precisely 30.4 for more on this.

ForLoop

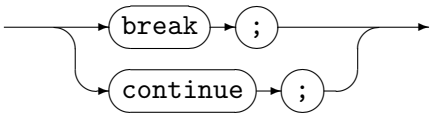


ForRangeHeader



Loop body execution may be exited early or continued with the next iteration with the `break` and `continue` statements. Such a statement must not occur outside of a loop.

EarlyExit



EXAMPLE (77)

The following rule eval part gives some example statements regarding control flow.

```
rule example
{
  n:N; // we (again) assume attributes a:int, arr:array<int>, narr:array<N>

   modify {
   ---
      eval {
         if(n.a < 42) {
            while(true) {
               n.a = n.a + 1;
               if(n.a == 42) {
                  break; // leaves the while loop, n.a==42 afterwards
               }
            }
         } else {
            def ref narr:array<N> = n.narr;
            def var i:int = 0;
            for(tn:N in narr) {
               i = i + 1;
               n.a = n.a + tn.a + n.arr[i];
            }
         }
      }
   }
}
```

14.4 Embedded Exec

As in the rules (cf. chapter 13), `exec` statements may be embedded. They are executing the graph rewrite sequence (see chapter 11) contained, with access to the variables defined outside. This means for one reading these variables, but for the other assigning to those variables with `yield`. The embedded `exec`s allow applying rules from the computations, more exactly: from the procedures (embedded execs are not available in functions), while the sequence computations (cf. 19.1) may call defined computations (functions in expression context, and procedures in statement context), mixing and merging declarative rule-based graph rewriting (pattern matching and dependent modifications) with traditional graph programming. This construct achieves a further step in the direction of blending rule-based with traditional programming; the main step is the extension of the attribute assignments, the only statement available in the original GrGen, into the full-fledged computations pictured in this and the following chapters.

`exec(.)`
: executes the graph rewrite sequence given.

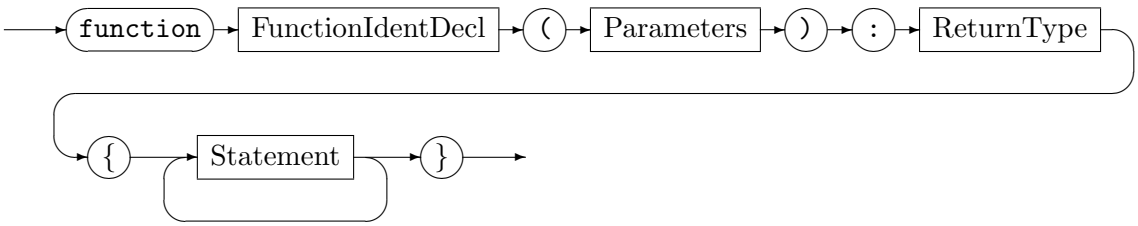
14.5 Computation Definition and Call

Computations that are occurring frequently may be factored out into a computation definition given in the header of a rule file. Such compound computations can be built and abstracted into reusable entities in two different forms, *functions* usable(/callable) from expression context (and statement context), and *procedures* usable(/callable) from statement context (only). Besides, several built-in functions and procedures are available, ready to be called and reused with the same syntax. (Furthermore, external functions and procedures may be declared, callable with the same syntax, too.)

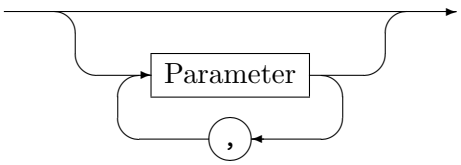
14.5.1 Function Definition and Call (and Quick Reference Tables)

Functions return exactly one output value, and can thus be used from the expressions with their compositional semantics. They are side-effect free, meaning they are not allowed to change the graph while being executed — so you are not allowed to call procedures (self-defined as well as built-in) from within a function definition. On the other hand do we consider pure functional programming not a help but a burden, so you are free to declare local variables and assign them as you like inside the function definitions; they only must behave as functions at the outside.

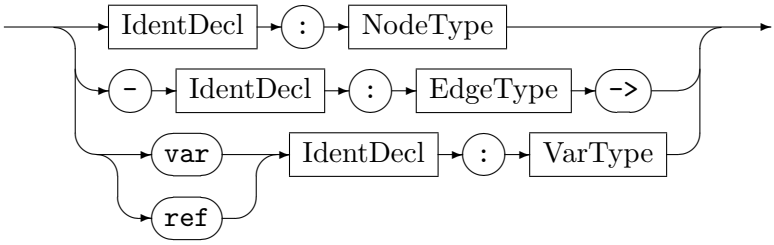
FunctionDefinition



Parameters



Parameter

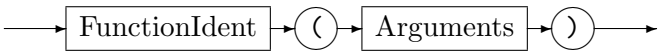


The function definition must return a value with a return statement (so there must be at least one return statement). The type of the one return value must be compatible to the return type specified in the computation definition. (A return statement must not occur outside of a computation definition.)

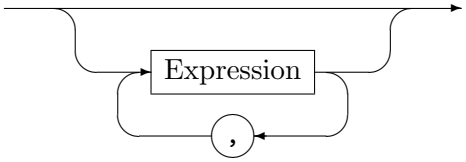
ReturnStatement



FunctionCall



Arguments



EXAMPLE (78)

An example showing how functions are declared and used.

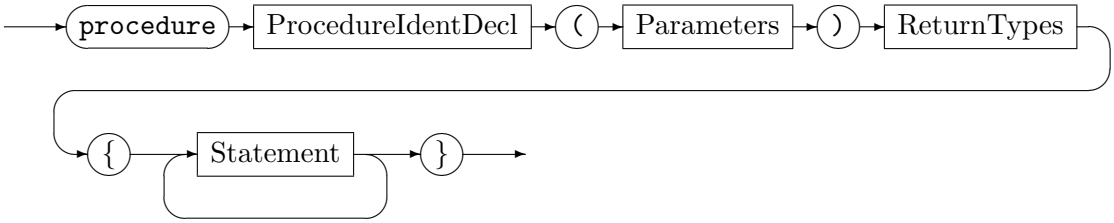
```
1  function fac(var x:int) : int
2  {
3     if(x>1) {
4        return( x * fac(x-1) );
5     } else {
6        return( 1 );
7     }
8  }
9  function foo(m:N) : boolean
10 {
11    def var tmp:int = fac(m.a);
12    tmp = tmp - 1;
13    return( m.a < tmp );
14 }
15 test example : (int)
16 {
17    n:N;
18    if{foo(n);}
19    return ( fac(n.a) );
20 }
```

Such a function may then be called as an expression atom from anywhere in the rule language file where an expression is required; or even from the sequence computations where an expression is required. In case you are missing them: these are functions from outside a class, you find more on function methods and their calls in Section 18.2.1. The built-in functions are called with the same syntax; table 14.1 and table 14.2 list the global built-in functions of the rule language at a glance, table 14.3 lists the built-in functions contained in built-in packages at a glance.

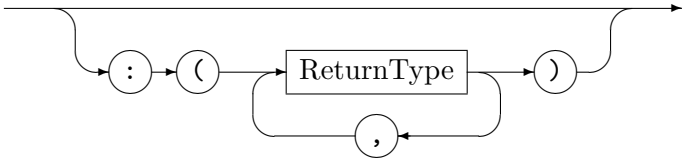
14.5.2 Procedure Definition And Call (and Quick Reference Tables)

Procedures may return between zero and k (arbitrary, but statically fixed) output values, and are thus only callable as a statement, with (or without) a multiple-value assignment. They are allowed to manipulate the graph as needed while being executed; so in a procedure definition you are free to call other procedures (self-defined as well as built-in).

ProcedureDefinition

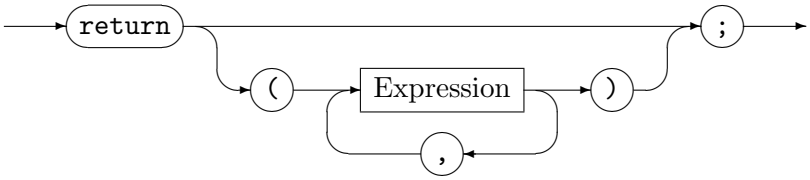


ReturnTypes

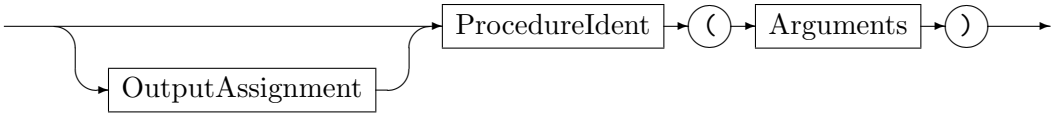


Please note the syntactic difference distinguishing the procedures from the functions (besides the different keyword): the return types are enclosed in parentheses (or omitted altogether). The computation definition may return between zero and k values. It must be always ended with a return statement; the type of the return values must be compatible to the return types specified in the computation definition. (A return statement must not occur outside of a computation definition.)

ReturnStatement



ProcedureCall



nodes([NodeType]):set<Node> edges([EdgeType]):set<Edge>
source(Edge):Node target(Edge):Node opposite(Edge,Node):Node
adjacent(Node[,EdgeType[,NodeType]]):set<Node> adjacentIncoming(Node[,EdgeType[,NodeType]]):set<Node> adjacentOutgoing(Node[,EdgeType[,NodeType]]):set<Node> incident(Node[,EdgeType[,NodeType]]):set<Edge> incoming(Node[,EdgeType[,NodeType]]):set<Edge> outgoing(Node[,EdgeType[,NodeType]]):set<Edge>
reachable(Node[,EdgeType[,NodeType]]):set<Node> reachableIncoming(Node[,EdgeType[,NodeType]]):set<Node> reachableOutgoing(Node[,EdgeType[,NodeType]]):set<Node> reachableEdges(Node[,EdgeType[,NodeType]]):set<Edge> reachableEdgesIncoming(Node[,EdgeType[,NodeType]]):set<Edge> reachableEdgesOutgoing(Node[,EdgeType[,NodeType]]):set<Edge>
boundedReachable(Node,int[,EdgeType[,NodeType]]):set<Node> boundedReachableIncoming(Node,int[,EdgeType[,NodeType]]):set<Node> boundedReachableOutgoing(Node,int[,EdgeType[,NodeType]]):set<Node> boundedReachableEdges(Node,int[,EdgeType[,NodeType]]):set<Edge> boundedReachableEdgesIncoming(Node,int[,EdgeType[,NodeType]]):set<Edge> boundedReachableEdgesOutgoing(Node,int[,EdgeType[,NodeType]]):set<Edge>
boundedReachableWithRemainingDepth(Node,int):map<Node, int> boundedReachableWithRemainingDepthIncoming(Node,int):map<Node, int> boundedReachableWithRemainingDepthOutgoing(Node,int):map<Node, int> The three functions above allow optional edge and node type constraints, too.
inducedSubgraph(set<Node>):graph definedSubgraph(set<Edge>):graph equalsAny(graph,set<graph>):boolean equalsAnyStructurally(graph,set<graph>):boolean getEquivalent(graph,set<graph>):graph getEquivalentStructurally(graph,set<graph>):graph copy(graph):graph clone(match<r>):match<r> clone(match<class mc>):match<class mc> clone(container):container copy(container):container clone(Object):Object copy(Object):Object clone(TransientObject):TransientObject copy(TransientObject):TransientObject
nameof(Node/Edge):string nameof(graph):string uniqueof(Node/Edge):int uniqueof(graph):int uniqueof(Object):long nodeByName(string[,NodeType]):Node edgeByName(string[,EdgeType]):Edge nodeByUnique(int[,NodeType]):Node edgeByUnique(int[,EdgeType]):Edge graphof(Node/Edge):graph

Table 14.1: Global functions at a glance, part I

scan<T>(string):T tryscan<T>(string):boolean
random():double random(int):int
countNodes([NodeType]):int countEdges([EdgeType]):int
countAdjacent(Node[,EdgeType[,NodeType]]):int countAdjacentIncoming(Node[,EdgeType[,NodeType]]):int countAdjacentOutgoing(Node[,EdgeType[,NodeType]]):int countIncident(Node[,EdgeType[,NodeType]]):int countIncoming(Node[,EdgeType[,NodeType]]):int countOutgoing(Node[,EdgeType[,NodeType]]):int
countReachable(Node[,EdgeType[,NodeType]]):int countReachableIncoming(Node[,EdgeType[,NodeType]]):int countReachableOutgoing(Node[,EdgeType[,NodeType]]):int countReachableEdges(Node[,EdgeType[,NodeType]]):int countReachableEdgesIncoming(Node[,EdgeType[,NodeType]]):int countReachableEdgesOutgoing(Node[,EdgeType[,NodeType]]):int
countBoundedReachable(Node,int[,EdgeType[,NodeType]]):int countBoundedReachableIncoming(Node,int[,EdgeType[,NodeType]]):int countBoundedReachableOutgoing(Node,int[,EdgeType[,NodeType]]):int countBoundedReachableEdges(Node,int[,EdgeType[,NodeType]]):int countBoundedReachableEdgesIncoming(Node,int[,EdgeType[,NodeType]]):int countBoundedReachableEdgesOutgoing(Node,int[,EdgeType[,NodeType]]):int
isAdjacent(Node,Node[,EdgeType[,NodeType]]):boolean isAdjacentIncoming(Node,Node[,EdgeType[,NodeType]]):boolean isAdjacentOutgoing(Node,Node[,EdgeType[,NodeType]]):boolean isIncident(Node,Edge[,EdgeType[,NodeType]]):boolean isIncoming(Node,Edge[,EdgeType[,NodeType]]):boolean isOutgoing(Node,Edge[,EdgeType[,NodeType]]):boolean
isReachable(Node,Node[,EdgeType[,NodeType]]):boolean isReachableIncoming(Node,Node[,EdgeType[,NodeType]]):boolean isReachableOutgoing(Node,Node[,EdgeType[,NodeType]]):boolean isReachableEdges(Node,Edge[,EdgeType[,NodeType]]):boolean isReachableEdgesIncoming(Node,Edge[,EdgeType[,NodeType]]):boolean isReachableEdgesOutgoing(Node,Edge[,EdgeType[,NodeType]]):boolean
isBoundedReachable(Node,Node,int[,EdgeType[,NodeType]]):boolean isBoundedReachableIncoming(Node,Node,int[,EdgeType[,NodeType]]):boolean isBoundedReachableOutgoing(Node,Node,int[,EdgeType[,NodeType]]):boolean isBoundedReachableEdges(Node,Edge,int[,EdgeType[,NodeType]]):boolean isBoundedReachableEdgesIncoming(Node,Edge,int[,EdgeType[,NodeType]]):boolean isBoundedReachableEdgesOutgoing(Node,Edge,int[,EdgeType[,NodeType]]):boolean

Table 14.2: Global functions at a glance, part II

`Math::min(Number,Number):Number`
`Math::max(Number,Number):Number`
`Math::abs(Number):Number`

`Math::ceil(double):double`
`Math::floor(double):double`
`Math::round(double):double`
`Math::truncate(double):double`

`Math::sqr(double):double`
`Math::sqrt(double):double`
`Math::pow([double,]double):double`
`Math::log(double[,double]):double`
`Math::sgn(double):double`

`Math::sin(double):double`
`Math::cos(double):double`
`Math::tan(double):double`

`Math::arcsin(double):double`
`Math::arccos(double):double`
`Math::arctan(double):double`

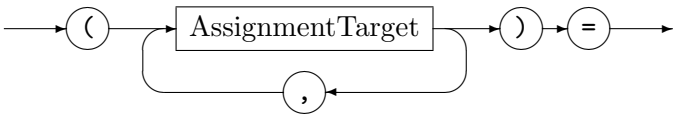
`Math::pi():double` `Math::e():double`

`Math::byteMin():byte`
`Math::byteMax():byte`
`Math::shortMin():short`
`Math::shortMax():short`
`Math::intMin():int`
`Math::intMax():int`
`Math::longMin():long`
`Math::longMax():long`
`Math::floatMin():float`
`Math::floatMax():float`
`Math::doubleMin():double`
`Math::doubleMax():double`

`File::import(string):graph` `File::exists(string):boolean`
`Time::now():long`

Table 14.3: Functions in packages at a glance

OutputAssignment



EXAMPLE (79)

An example showing how procedures are declared and used.

```
1  procedure foo(n:N) : (N,int)
2  {
3     (def m:N) = add(N); // create new node of type N and add it to the graph
4     return( m, n.a );
5  }
6
7  procedure bar(m:N, var i:int)
8  {
9     def var fac:int = 1;
10    while(i>0) {
11       fac = fac * i;
12       i = i - 1;
13    }
14    m.a = m.a - fac;
15    return;
16 }
17
18 rule example : (int,int)
19 {
20    n:N;
21    modify {
22    ---
23       def var i:int;
24       eval {
25          def m:N;
26          (m,yield i)=foo(n);
27          bar(m,i);
28       }
29       return(i,n.a);
30    }
31 }
```

Such a procedure may then be called as a statement atom from anywhere in the rule language file where an attribute evaluation (/computation) is required; or even from the sequence computations where a statement is required. In case you are missing them: these are procedures from outside a class, you find more on procedure methods and their calls in Section 18.2.2. The built-in procedures are called with the same syntax; table 14.4 lists the global built-in procedures of the rule language at a glance, table 14.5 lists the built-in procedures contained in built-in packages at a glance.

Please note that the *OutputAssignment* is optional, you may execute a procedure just for its side effects; this kind of call is even the most common as many built-in procedures don't return values at all but are just executed for their state changes. The *AssignmentTarget* may be any of the LHS expressions supported by the different *AssignmentStatements* (a def variable, a def variable yielded to, a global variable, a graph element attribute, an indexed

target, or a visited flag of a graph element), and it may be def variable that gets just declared (*LocalVariableDecl*), using the procedure output for initialization.

`add(NodeType):(Node)`
`addCopy(Node):(Node)`
`add(EdgeType,Node,Node):(Edge)`
`addCopy(Edge,Node,Node):(Edge)`
`rem(Node)`
`rem(Edge)`
`retype(Node,NodeType):(Node)`
`retype(Edge,EdgeType):(Edge)`
`clear()`

`merge(Node,Node)`
`redirectSource(Edge,Node)`
`redirectTarget(Edge,Node)`
`redirectSourceAndTarget(Edge,Node,Node)`

`insert(graph)`
`insertCopy(graph,Node):(Node)`
`insertInduced(set<Node>,Node):(Node)`
`insertDefined(set<Edge>,Edge):(Edge)`

`valloc():(int)`
`vfree(int)`
`vreset(int)`
`vfreenonreset(int)`

`emit(string[,string]*)`
`emitdebug(string[,string]*)`
`record(string)`

`getEquivalentOrAdd(graph,array<graph>):(graph)` `getEquivalentStructurallyOrAdd(graph,array<graph>):(graph)`
`assert(boolean[,string[,object]*])` `assertAlways(boolean[,string[,object]*])`

Table 14.4: Global procedures at a glance

`File::export(string)`
`File::export(graph,string)`
`File::delete(string)`

`Transaction::start():(int)`
`Transaction::pause(int)`
`Transaction::resume(int)`
`Transaction::commit(int)`
`Transaction::rollback(int)`

`Debug::add(string[,object]*)`
`Debug::rem(string[,object]*)`
`Debug::emit(string[,object]*)`
`Debug::halt(string[,object]*)`
`Debug::highlight(string[,object,string]*)`

`Synchronization::enter(object)`
`Synchronization::tryenter(object):(boolean)`
`Synchronization::exit(object)`

Table 14.5: Procedures in packages at a glance

14.6 The Big Picture

Figure 14.1 displays the different computation types of GrGen.NET, and how they may use and call each other. On top is the strategy language for rule control, to the left the declarative rules and subpatterns, and to the right the imperative parts for attribute computations or traditional graph programming. The statements include the expressions and non-graph changing statements.

The following tables give an overview over the rule and computation sublanguages of GrGen.NET; separated into pattern and rewrite part, and into expressions and statements.

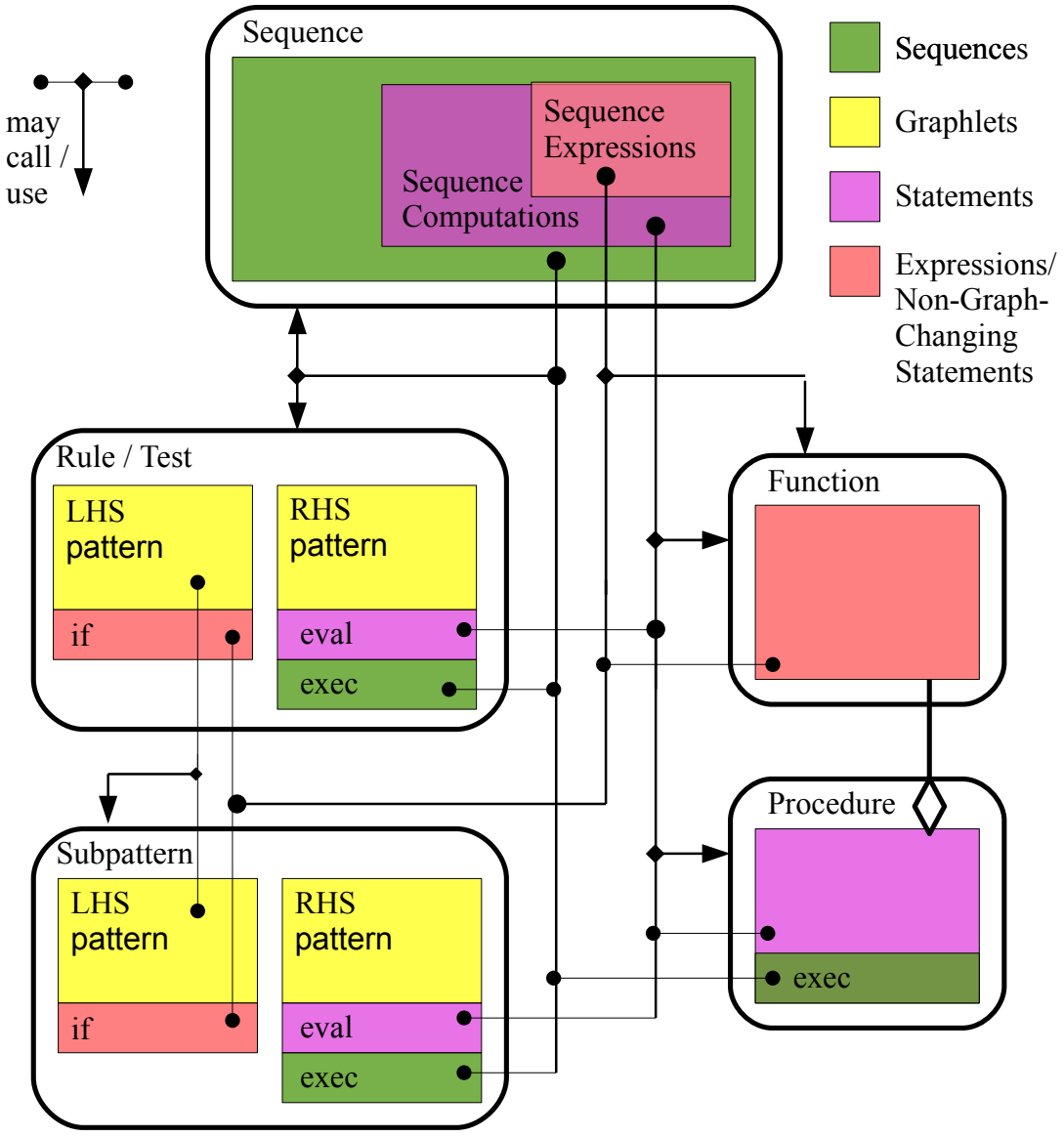


Figure 14.1: Computation Types and Possible Calls/Uses

`.`	Declaration of an anonymous node of type `Node`, to be matched.
`-->`	Declaration of an anonymous edge of type `Edge`, to be matched.
`<-->`	Declaration of an anonymous edge of type `Edge` that can get matched in either direction.
`--`	Declaration of an anonymous undirected edge of type `UEdge`, to be matched.
`n:N`	Declaration of a node `n` of type `N`, to be matched.
`-e:E->`	Declaration of an edge `e` of type `E`, to be matched.
`n`	Reference to node `n` within a graphlet.
`-e->`	Reference to edge `e` within a graphlet.
`def var v:B`	Defines a def-variable `v` of basic type `B`, to be yielded to.
`def ref c:C`	Defines a def-variable `c` of container type `C` or of internal (transient) class object type `C`, to be yielded to.
`x:typeof(y)`	The element `x` to be matched must be of the same or a subtype as `y`.
`x:T\S`	The element `x` must be of type `T` but not of subtype `S`.
`x:T<y>`	The element `y` casted to `T`, accessible as `x`.
`hom(x,y)`	The pattern elements `x` and `y` are allowed to match the same graph element.
`independent(x)`	The pattern element `x` may be matched homomorphically to all other pattern elements.
`exact(x,y)`	All edges incident to `x` and `y` in the host graph must have been specified in the pattern.
`induced(x,y)`	Induced edges in between `x` and `y` in the host graph must have been specified in the pattern.
`if{exp;}`	An attribute condition constraining the pattern.
`yield{stmt}`	A computation for yielding to def variables.
`nested{pattern}`	A nested pattern, with `nested` being one of the pattern construction modi listed in 8.1.
`p:P()`	A usage/call `p` of a (user-defined) subpattern of type `P`.

Let `exp` be an expression, `stmt` be a statement, and `pattern` be a LHS pattern.

Table 14.6: Pattern (LHS) elements at a glance

`modify{pattern}`	An RHS pattern nested in an arbitrary LHS pattern, specifying the changes.
`replace{pattern}`	An RHS pattern nested in an arbitrary LHS pattern, specifying the target pattern.
`.`	Declaration of an anonymous node of type `Node`, to be created.
`-->`	Declaration of an anonymous edge of type `Edge`, to be created.
`n:N`	Declaration of a node `n` of type `N`, to be created.
`-e:E->`	Declaration of an edge `e` of type `E`, to be created.
`n`	Reference to node `n` within a graphlet, ensures it is kept in `replace` mode.
`-e->`	Reference to edge `e` within a graphlet, ensures it is kept in `replace` mode.
`def var v:B`	Defines a def-variable `v` of basic type `B`, to be yielded to.
`def ref c:C`	Defines a def-variable `c` of container type `C` or of internal (transient) class object type `C`, to be yielded to.
`delete(x)`	Deletes element `x` in `modify` mode.
`x:typeof(y)`	The element `x` is created with the same type as `y`.
`x:T<y>`	The element `y` is casted to type `T`, accessible as `x` afterwards.
`x:copy<y>`	The element `x` is created as a copy of `y`.
`x:clone<y>`	The element `x` is created as a clone of `y`.
`x:typeof(y)<y,z>`	The nodes `y` and `z` are merged and casted to the original type of `y`, accessible as `x` afterwards.
`x !-e->!  y`	The edge `e` is redirected to source node `x` and target node `y`.
`eval{stmt}`	A computation assigning to attributes or def variables.
`exec(s ;> yield o=i)`	Executes the sequence `s`, and then yields the value of an inner variable `i` to an outer variable `o`. Yielding from an `exec` is only possible in the top-level `exec` of a rule, in nested patterns and subpatterns the `exec`ution is deferred.
`delete(p)`	Deletes the subpattern used as `p`.
`:P()`	Creates a simple subpattern of type `P`.
`:p()`	Uses or calls the rewrite part of a (user-defined) subpattern used as `p` in the containing LHS pattern.

Let `stmt` be a statement.

Table 14.7: Rewrite elements (RHS pattern) at a glance

`v`	Reads the variable `v`.
`::v`	Reads the global variable `v`.
`v.a`	Reads the attribute `a` of `v`.
`v[i]`	Reads the value at position `i` of container `v`.
`idx[n]`	Reads the incidence count of node `n` from incidence count index `idx`.
`x.a[i]`	Reads the value at position `i` of container attribute `a` of `x`.
`x.visited[i]`	Reads the visited flag `i` of graph element `x`.
`typeof(v)`	Returns the type of graph element `v`.
`(T)v`	Casts `v` to type `T`.
`nameof(v)`	Returns the name of graph element `v`.
`random()`	Returns a random value in between 0.0 and 1.0.
`cond ? exp1 : exp2`	Returns `exp1` if `cond`, otherwise `exp2`.
`e op f`	For `op` being one of the binary operators listed in 6.9.
`op f`	For `op` being one of the unary operators listed in 6.9.
`f(...)`	Calls one of the functions listed in 14.1. Or calls a user defined function `f`.
`v.fm(...)`	Calls one of the function methods listed in 18.2. Or calls a user defined function method `fm`.

Let `cond`, `exp1`, and `exp2` be expressions.

Table 14.8: Expressions at a glance

`def n:N`	Defines a variable `n` of node type `N`.
`def -e:E->`	Defines a variable `e` of edge type `E`.
`def var v:B`	Defines a variable `v` of basic type `B`.
`def ref c:C`	Defines a variable `c` of container type `C` or of internal (transient) class object type `C`.
`v = exp`	Assigns the value of `exp` to the variable `v`.
`::v = exp`	Assigns the value of `exp` to the global variable `v`.
`v.a = exp`	Assigns the value of `exp` to the attribute `a` of `v`.
`v[i] = exp`	Assigns the value of `exp` to the position `i` of container `v`.
`x.a[i] = exp`	Assigns the value of `exp` to the position `i` of container attribute `a` of `x`.
`x.visited[i] = exp`	Assigns the value of `exp` to the visited flag `i` of graph element `x`.
`if(cond) {S1} else {S2}`	Executes `S1` iff `cond` evaluates to `true`, or `S2` otherwise.
`switch(exp) { cases }`	Evaluates `exp`, then dispatches to one of the cases, see below.
`case c-exp {S1} else {S2}`	If the constant `exp` matches the value switched upon, `S1` is executed, if none of the cases matches, `else` is executed (if given).
`while(cond) {S1}`	Executes `S1` as long as `cond` evaluates to `true`.
`do {S1} while(cond)`	Executes `S1` until `cond` evaluates to `false`.
`for(v:T in c) {S1}`	Executes `S1` for each value `v` contained in container `c`.
`for(i:T->v:S in c) {S1}`	Executes `S1` for each key/position to value pair `(i,v)` contained in container `c`.
`for(i:int in [l:r]) {S1}`	Executes `S1` for each integer `i` in the range from `l` to `r`, in steps of 1, upwards if `l<=r`, otherwise downwards.
`for(n:N in func(...)) {S1}`	Execute `S1` for every `n` in the graph query function `func`
`for(n:N in indexFunc(...)) {S1}`	Execute `S1` for every `n` in the index query function `indexFunc`
`lock(exp) { S1 }`	Evaluates `exp` and locks it, then `S1` is executed, afterwards releasing the lock.
`break`	Aborts execution of the containing loop.
`continue`	Continues execution of the containing loop at its condition.
`return`	Returns from a containing function or procedure.
`return(v)`	Returns from a containing function or procedure with result value `v`.
`exec(s ;> yield o=i)`	Executes the sequence `s`, and then yields the value of an inner variable `i` to an outer variable `o`.
`(...)=p(...)`	Calls one of the procedures listed in 14.4. Or calls a user defined procedure `p`.
`(...)=v.pm(...)`	Calls one of the procedure methods listed in 18.4. Or calls a user defined procedure method `pm`.
`exec(s)`	Executes the sequence `s`.

Let `exp` and `cond` be expressions, and `S1` and `S2` be statements.

Table 14.9: Statements at a glance

CHAPTER 15

CONTAINER TYPES AND COMPUTATIONS

15.1 Built-In Types and Concept of Containers

Besides the types already introduced, GrGen.NET supports the built-in generic types in Table 15.1. The exact type format is backend specific. The LGSPBackend maps the generic types to generic C#-Dictionaries (i.e. hashmaps) or generic C#-Lists (misnamed dynamic arrays) or a GrGen.NET supplied generic C#-Deque of their corresponding primitive types, with `de.unika.ipd.grGen.libGr.SetValueType` as target type for sets, only used with the value `null`.

`set<T>`	A (mathematical) set of type `T`, where `T` may be an enumeration type or one of the primitive types from 6.1; it may even be a node or edge or graph type, then we speak of storages.
`map<S,T>`	A (mathematical) map from type `S` to type `T`, where `S` and `T` may be enumeration types or one of the primitive types from 6.1; it may even be a node or edge or graph type, then we speak of storages.
`array<T>`	An array of type `T`, where `T` may be an enumeration type or one of the primitive types from 6.1; it may even be a node or edge or graph type, then we speak of storages. Shares some similarities with `map<int,T>`.
`deque<T>`	A deque of type `T`, where `T` may be an enumeration type or one of the primitive types from 6.1; it may even be a node or edge or graph type, then we speak of storages. Shares some similarities with `array<T>`.

Table 15.1: GrGen.NET built-in generic types

The four container types supported by GrGen share a lot of conceptual similarities and can be accessed in a similar way. They support multiple methods to update them: `add` to add an element to the container, `rem` to remove an element from the container, and `clear` to remove all elements from the container. Furthermore, they support multiple methods to query them: `size` to return the count of elements in the container, `empty` to return whether the container is empty or not, and `peek` to return an element from the container.

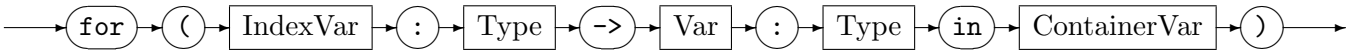
In addition to those common methods there is special syntax support available. Left associative binary expressions allow computing a new container from two input containers, or comparing two containers. The `in` operator allows to query for containment. Indexed access returns an element at an index position, and indexed assignment overwrites an element at a specified position with another one. Container typed variables as such may be assigned a container (with value semantics if the target is a graph element attribute, in all other cases with reference semantics). Compound assignments combine a binary expression with an assignment.

Constructor literals may be used to create and initialize containers; a copy constructor allows to get a new container, of duplicated content and potentially different type. And finally iteration over the elements in the container is supported with a `for` loop, cf. 14.3.

ForContainerHeader



ForContainerWithIndexVarHeader

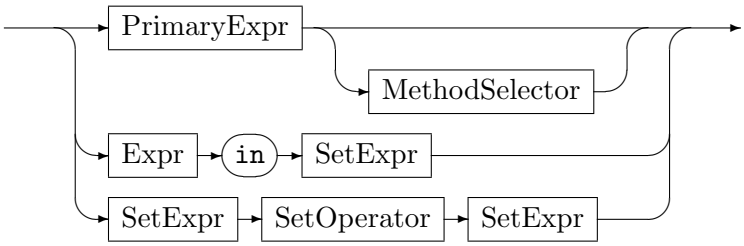


These operations are available in the rule language, as an extension of the expressions from 6 and the computation statements from 14; most of them are available in the sequence computations language, too (19.3 will tell about the differences compared to the rule language). In the `if` attribute evaluation clause only side-effect free container queries are allowed. In the following the container operations will be explained in detail for one type after another.

15.2 Set Operations

Set expressions consist of the known mathematical set operations, plus some operations in method call notation (on `set<T>`).

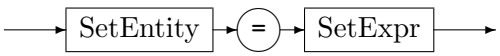
SetExpr



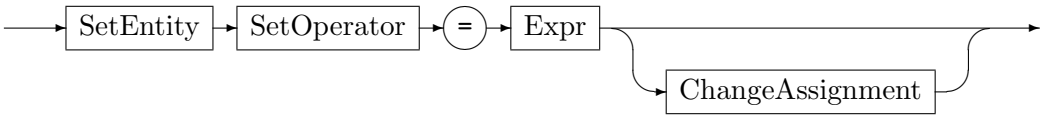
MethodCall



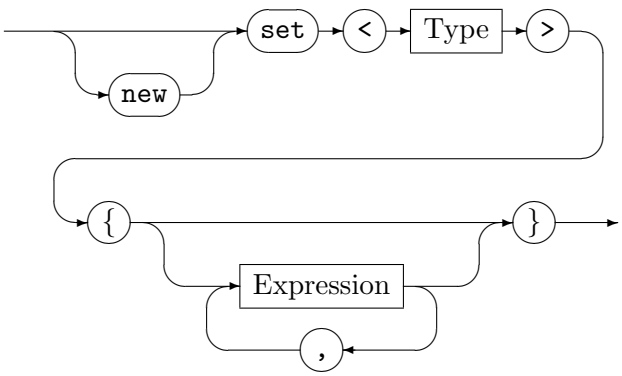
Assignment



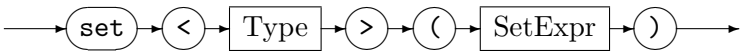
CompoundAssignment



SetConstructor



SetCopyConstructor



The query method calls on sets are:

.size()
: returns the number of elements in the set, as `int`

.empty()
: returns whether the set is empty, as `boolean`

.peek(num)
: returns the element which comes at position `num:int` in the sequence of enumeration, as `T` for `set<T>`; the higher the number, the longer retrieval takes

.min()
: returns the minimum of the elements in the set, as `T'` for `set<T>` (see 20.1 regarding `T'`)

.max()
: returns the maximum of the elements in the set, as `T'` for `set<T>` (see 20.1 regarding `T'`).

.asArray()
: returns an array of the set content, as `array<T>` for `set<T>`, in enumeration order

The update method calls on sets are:

Set addition:
: `s.add(v)` adds the value `v` to the set `s`.

Set removal:
: `s.rem(v)` removes the value `v` from the set `s`.

Set clearing:
: `s.clear()` removes all values from the set `s`.

Set union:
: `s.addAll(os)` adds all elements in set `os` to the set `s`. (Only available on targets that are set variables, not on attributes.)

The binary set operators are:

`|`	Set union (contained in resulting set as soon as contained in one of the sets)
`&`	Set intersection (contained in resulting set only if contained in both of the sets)
`\`	Set difference (contained in resulting set iff contained in left but not right set)

Table 15.2: Binary set operators, in ascending order of precedence

The binary set operators require the left and right operands to be of identical type `set<T>`. The operator `x in s` denotes set membership $x \in s$, returning whether the set contains the given element, as `boolean`. It is an O(1) operation for sets. Furthermore, the container may be iterated over with a `for` loop, as introduced in 14.3. The set only allows for non-indexed iteration.

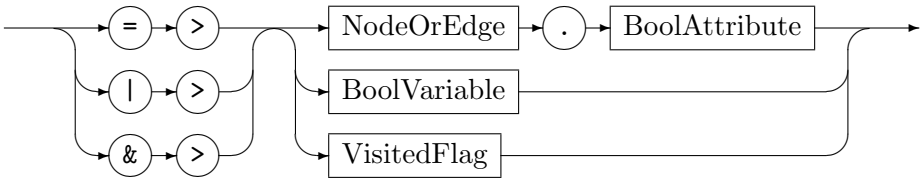
The relational expressions (already introduced in 6.4) used to compare entities of different kinds, mapping them to the type boolean, are extended to sets according to table 15.3: Some set `A` is a subset of `B` iff all elements in `A` are contained in `B`, too.

`A == B`	True, iff the content of A and B is identical.
`A != B`	True, iff the content of A and B is not identical.
`A ~~ B`	True, iff A and B are by-value equal (deeply).
`A <  B`	True, iff A is a subset of B, but A and B are not identical.
`A >  B`	True, iff A is a superset of B, but A and B are not identical.
`A <= B`	True, iff A is a subset of B or A and B are identical.
`A >= B`	True, iff A is a superset of B or A and B are identical.

Table 15.3: Binary set operators for comparison

The assignments implement the computation constructs introduced in 14. The pure assignment overwrites the target set with the source set, commonly with value semantics, creating a copy of the source set. Only if a local variable (i.e. not an attribute) is assigned to a local variable, is reference semantics used (i.e. both variables point afterwards to the same set). Compound assignments are assignments which use the first source as target, too, only adapting the target value instead of computing a new value and overwriting the target with it. For scalars this is not supported, but for container valued entities it is offered due to the potential for massive computational cost savings. The compound assignment statements on sets are a set union `|=`, intersection `&=` and difference `\=` assignment.

ChangeAssignment



The compound assignments on sets and maps may be enhanced with a change assignment declaration. The change value is `true` in case the target collection changes and `false` in case the target collection is not altered. The assign-to operator `=>` assigns the change value to the specified target, the or-to operator `|>` assigns the boolean disjunction of the change value target with the change value to the change value target, the and-to operator `&>` assigns the boolean conjunction of the change value target with the change value to the change value target. This addition allows for efficient data flow computations not needing to check for a change by set comparison, see 24.6.

The *SetConstructor* extends the *Literal* from 6.8 (as a refinement of the *ContainerConstructor* there). It is constant if only primitive type literals, enum literals, or constant expressions are used; this is required for container initializations in the model. It is non-constant if it contains nodes/edges/or member accesses, which may be the case if used in the rules. The elements given in the type constructor are cast to the specified member type if needed and possible. The *SetCopyConstructor* creates a new set of the specified type, filling it with the elements from the old set. In case of a node or edge type, the types of the original and the constructed set are allowed to differ, only the elements from the old set that match the type of the new set are taken over then (this is a remedy for set value type invariance).

EXAMPLE (80)

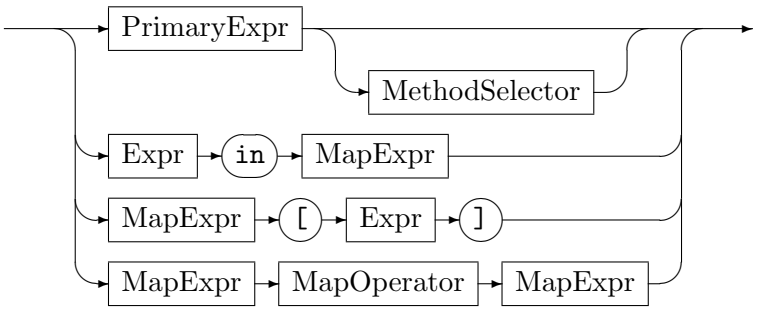
An example explaining some set operations.

```
function setExample(ref si:set<int>) : boolean
{
   def ref s:set<int> = set<int>{};
   s.add(1); // { 1 }
   s.add(2); // { 1, 2 }
   s.add(3); // { 1, 2, 3 }, s.size()==3
   s.rem(2); // { 1, 3 }
   for(v:int in s) {
      si.add(v);
   } // s == si == { 1, 3 }, assuming input was {}
   def var i:int = s.size(); // i==2
   if(!(i in s)) {
      s.add(i);
   } // { 1, 2, 3 }
   s.add(3); // still { 1, 2, 3 }
   return (2 in (s | set<int>{ 4,5 })); // true
}
```

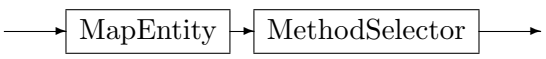
15.3 Map Operations

Map expressions consist of the known mathematical set operations extended to maps, and map value lookup, plus some operations in method call notation (on `map<S,T>`).

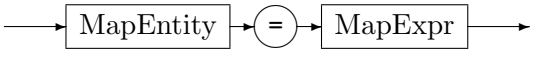
MapExpr



MethodCall



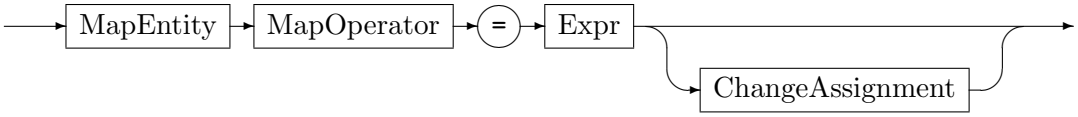
Assignment



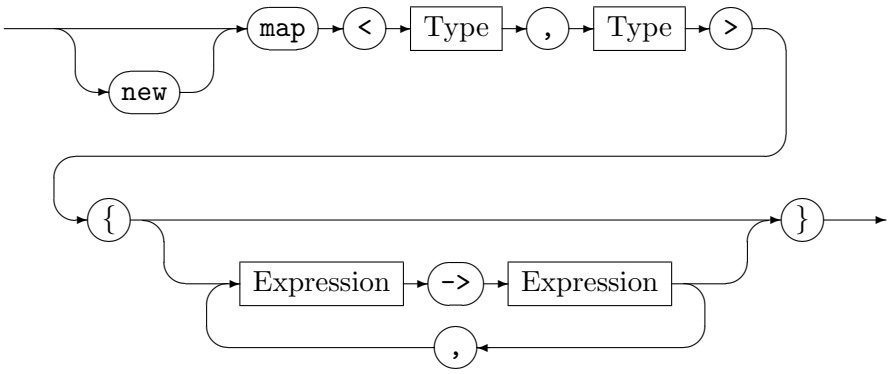
IndexedAssignment



CompoundAssignment



MapConstructor



MapCopyConstructor



The query method calls on maps are:

.size()
: returns the number of elements in the map, as `int`

.empty()
: returns whether the map is empty, as `boolean`

.peek(num)
: returns the key of the element which comes at position `num:int` in the sequence of enumeration, as `S` for `map<S,T>`; the higher the number, the longer retrieval takes

.domain()
: returns the set of elements in the domain of the map, as `set<S>` for `map<S,T>`

.range()
: returns the set of elements in the range of the map, as `set<T>` for `map<S,T>`

.asArray()
: returns an array of the map content, as `array<T>` for `map<int,T>` (only available for domain type `int`); gaps are filled with the default value, up to the maximum index available in the map

The update method calls on maps are:

Map addition:
: `m.add(k,v)` adds the pair (`k`,`v`) to the map `m`, overwrites the old value if a pair (`k`,unknown) was already existing.

Map removal:
: `m.rem(k)` removes the pair (`k`,unknown) from the map `m`.

Map clearing:
: `m.clear()` removes all values from the map `m`.

The binary map operators are:

The binary map operators require the left and right operands to be of identical type `map<S,T>`, with one exception for map difference, this operator accepts for a left operand of type `map<S,T>` a right operand of type `set<S>`, too. The operator `x in m` denotes map domain membership $x \in dom(m)$, returning whether the domain of the map contains the

`|`	Map union: returns new map with elements which are in at least one of the maps, with the value of map2 taking precedence
`&`	Map intersection: returns new map with elements which are in both maps, with the value of map1 taking precedence
`\`	Map difference: returns new map with elements from map1 without the elements with a key contained in map2

Table 15.4: Binary map operators, in ascending order of precedence

given element, as `boolean`. It is an O(1) operation for maps. The operator `m[x]` denotes map lookup, i.e. it returns the value `y` which is stored in the map `m` for the value `x` (domain value `x` is mapped by the mapping `m` to range value `y`). The value `x` *must* be in the map, i.e. `x in m` must hold. Furthermore, the container may be iterated over with a `for` loop, as introduced in 14.3. The map only allows for indexed iteration, with the key getting assigned to the index variable and the corresponding value getting assigned to the value variable.

The relational expressions (already introduced in 6.4) used to compare entities of different kinds, mapping them to the type boolean, are extended to sets according to table 15.5: A map `A` is a submap of `B` iff all key-value pairs of `A` are contained in `B`, too. If they have a key in common but map to a different value, they count as not identical.

`A == B`	True, iff the content of A and B is identical.
`A != B`	True, iff the content of A and B is not identical.
`A ~~ B`	True, iff A and B are by-value equal (deeply).
`A <  B`	True, iff A is a submap of B, but A and B are not identical.
`A >  B`	True, iff A is a supermap of B, but A and B are not identical.
`A <= B`	True, iff A is a submap of B or A and B are identical.
`A >= B`	True, iff A is a supermap of B or A and B are identical.

Table 15.5: Binary map operators for comparison

The assignments implement the computation constructs introduced in 14. The pure assignment overwrites the target map with the source map, commonly with value semantics, creating a copy of the source map. Only if a local variable (i.e. not an attribute) is assigned to a local variable, is reference semantics used (i.e. both variables point afterwards to the same map). The indexed assignment `m[i]=v` overwrites the old value at index `i` in the map `m` with the new value `v`. Compound assignments are assignments which use the first source as target, too, only adapting the target value instead of computing a new value and overwriting the target with it. For scalars this is not supported, but for container valued entities it is offered due to the potential for massive computational cost savings. The compound assignment statements on maps are a map union `|=`, intersection `&=` and difference `\=` assignment.

The *MapConstructor* extends the *Literal* from 6.8 (as a refinement of the *Container-Constructor* there). It is constant if only primitive type literals, enum literals, or constant expressions are used; this is required for container initializations in the model. It is non-constant if it contains nodes/edges/or member accesses, which may be the case if used in the rules. The elements given in the type constructor are cast to the specified member types if needed and possible. The *MapCopyConstructor* creates a new map of the specified types, filling it with the elements from the old map. In case of a node or edge type, the types of the original and the constructed map are allowed to differ, only the elements from the old map that match the type of the new map are taken over then (this is a remedy for map key/value type invariance).

EXAMPLE (81)

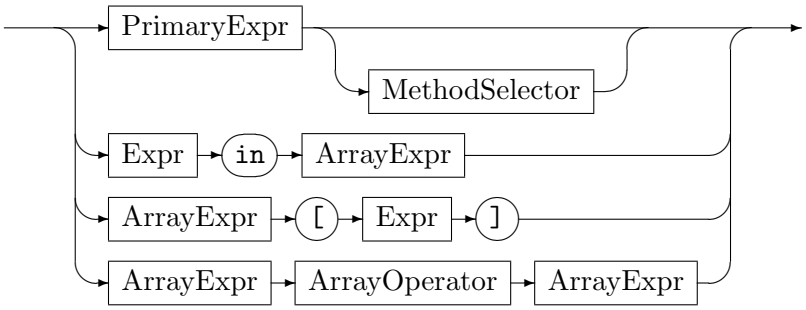
An example explaining some map operations.

```
function mapExample(ref mi:map<int,int>) : boolean
{
   def ref m:map<int,int> = map<int,int>{};
   m.add(1,1); // { 1->1 }
   m.add(2,2); // { 1->1, 2->2 }
   m.add(3,3); // { 1->1, 2->2, 3->3 }, m.size()==3
   m.rem(2); // { 1->1, 3->3 }
   for(k:int -> v:int in m) {
      mi.add(k,v); // mi[k] == m[k]
   } // m == mi == { 1->1, 3->3 }, assuming input was {->}
   m[3] = m[1]; // m = { 1->1, 3->1 }
   def var i:int = m.size(); // i==2
   if(!(i in m)) {
      m.add(i, 1);
   } // { 1->1, 2->1, 3->1 }
   m.add(3, 3); // { 1->1, 2->1, 3->3 }
   return (2 in (m | map<int,int>{ 4->4, 5->m[0] })); // true
}
```

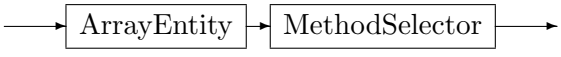
15.4 Array Operations

Array expressions consist of array membership checking, array value lookup, and array concatenation, plus some operations in method call notation (on `array<T>`).

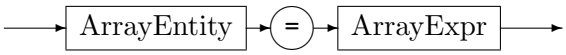
ArrayExpr



MethodCall



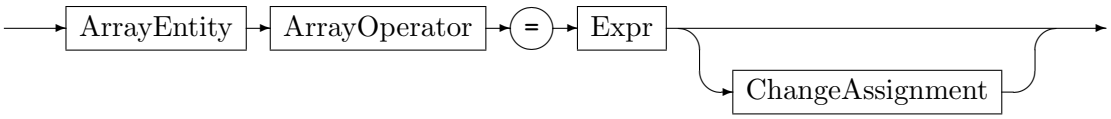
Assignment



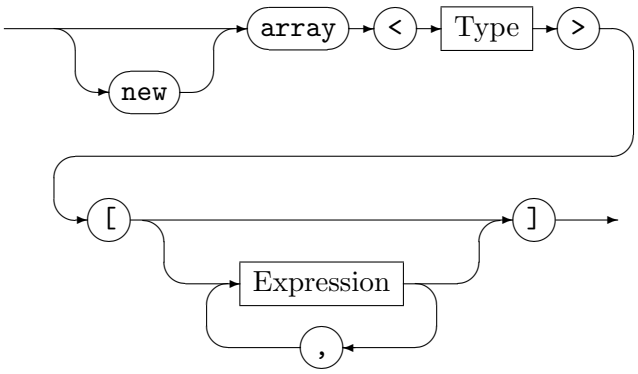
IndexedAssignment



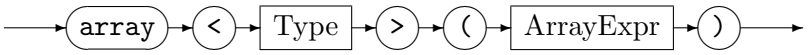
CompoundAssignment



ArrayConstructor



ArrayCopyConstructor



The query method calls on arrays are:

.size()
: returns the number of elements in the array, as `int`

.empty()
: returns whether the array is empty, as `boolean`

.peek(num)
: returns the value stored in the array at position `num:int` in the sequence of enumeration, is equivalent to (and implemented by) `a[num])`; retrieval occurs in constant time.

.peek()
: returns the last value stored in the array; retrieval occurs in constant time.

.indexOf(valueToSearchFor)
: returns the first position `valueToSearchFor:T` appears at, as `int`, or -1 if not found

.indexOf(valueToSearchFor, startIndex)
: returns the first position `valueToSearchFor:T` appears at (moving to the end), when we start the search for it at array position `startIndex:int`, as `int`, or -1 if not found

.lastIndexOf(valueToSearchFor)
: returns the last position `valueToSearchFor:T` appears at, as `int`, or -1 if not found

.lastIndexOf(valueToSearchFor, startIndex)
: returns the last position `valueToSearchFor:T` appears at (moving to the begin), when we start the search for it at array position `startIndex:int`, as `int`, or -1 if not found

.indexOfOrdered(valueToSearchFor)
: returns a position where `valueToSearchFor:T` appears at, as `int`, or -1 if not found. The array must be ordered, otherwise the results returned by the binary search employed will be wrong; in case of multiple occurrences, an arbitrary one is returned

.orderAscending()
: returns an array with the content of the input array ordered ascendingly, only available for basic types that are orderable

.orderDescending()
: returns an array with the content of the input array ordered descendingly, only available for basic types that are orderable

.group()
: returns an array with the content of the input array grouped, i.e. equal values being neighbours, only available for basic types or node and edge types

.keepOneForEach()
: returns an array with the content of the input array freed from duplicates, only available for basic types or node and edge types

.reverse()
: returns an array with the content of the input array reversed

.shuffle()
: returns an array with the content of the input array permuted

.subarray(startIndex, length)
: returns subarray of given `length:int` from `startIndex:int` on

.asDeque()
: returns a deque of the array content, as `deque<T>` for `array<T>`, in same order

.asSet()
: returns a set (without order) of the array content, as `set<T>` for `array<T>`

.asMap()
: returns a map of the array content, as `map<int,T>` for `array<T>`, the values are mapped-to by their index

.asString(separator)
: returns the array imploded to a string with the `separator:string` inserted in between the substrings from the array, only available for `array<string>` (a `string` contains an `asArray` method for reversal)

The ordering or access-alongside-order methods utilize the ordering relation, so they support only members or attributes of numeric type (`byte`, `short`, `int`, `long`, `float`, `double`,) or `string` or `enum` type – also `boolean` is supported, even if not of much use). The other query methods utilize equality comparisons, so they support additionally elements of node or edge type.

Furthermore, some array accumulation, by-member-or-attribute, and per-element methods are available, you find more on them in Section 20.1.
The update method calls on arrays are:

Array addition:
: `a.add(v)` adds the value `v` to the end of array `a`.

Array addition:
: `a.add(v,i)` inserts the value `v` at index `i` to array `a`.

Array removal:
: `a.rem()` removes the value at the end of the array `a`.

Array removal:
: `a.rem(i)` removes the value at index `i` from the array `a`.

Array clearing:
: `a.clear()` removes all values from the array `a`.

Array concatenation:

`a.addAll(oa)` adds all elements in array `oa` to the array `a`. (Only available on targets that are array variables, not on attributes.)

The binary array operators are:

`+`	Array concatenation: returns new array with the right appended to the left array the left and right operands must be of identical type `array<T>`

Table 15.6: Binary array operators, in ascending order of precedence

The operator `x in a` denotes array value membership, returning whether the array contains the given element, as `boolean`. It is an O(n) operation for arrays. The operator `a[x]` denotes array lookup, i.e. it returns the value `y` which is stored in the array `a` at the index `x`. The index `x` *must* be a valid array index.

NOTE (33)

GrGen.NET does not support multi-dimensional arrays, but note that you can easily emulate them, e.g. a two-dimensional array `a` with dimensions `n` and `m` by initializing it to `n*m` elements and accessing it by `a[i*n + j]` instead of `a[i,j]`. Jagged arrays require intermediate helper objects storing the nested array, i.e. `a:array<NestedArray>` with `class NestedArray { nestedArray : array<TargetType>; }`.

Furthermore, the container may be iterated over with a `for` loop, as introduced in 14.3. The array allows for non-indexed as well as indexed iteration; if non-indexed iteration is used the array values are iterated over, if indexed iteration is used the index is assigned to the index variable and the corresponding value is assigned to the value variable.

The relational expressions (already introduced in 6.4) used to compare entities of different kinds, mapping them to the type boolean, are extended to sets according to table 15.7: An array `A` is a subarray of `B` iff it is smaller or equal in size and the values at each common index are identical (lexicographic order as for strings).

`A == B`	True, iff the content of A and B is identical.
`A != B`	True, iff the content of A and B is not identical.
`A ~~ B`	True, iff A and B are by-value equal (deeply).
`A < B`	True, iff A is a subarray of B, but A and B are not identical.
`A > B`	True, iff A is a superarray of B, but A and B are not identical.
`A <= B`	True, iff A is a subarray of B or A and B are identical.
`A >= B`	True, iff A is a superarray of B or A and B are identical.

Table 15.7: Binary array operators for comparison

The assignments implement the computation constructs introduced in 14. The pure assignment overwrites the target array with the source array, commonly with value semantics, creating a copy of the source array. Only if a local variable (i.e. not an attribute) is assigned to a local variable, is reference semantics used (i.e. both variables point afterwards to the same array). The indexed assignment `a[i]=v` overwrites the old value at index `i` in the array `a` with the new value `v`. Compound assignments are assignments which use the first source as target, too, only adapting the target value instead of computing a new value and overwriting the target with it. For scalars this is not supported, but for container valued entities it is offered due to the potential for massive computational cost savings. The compound assignment statement on arrays is the concatenation assignment `+=`.

The *ArrayConstructor* extends the *Literal* from 6.8 (as a refinement of the *Container-Constructor* there). It is constant if only primitive type literals, enum literals, or constant expressions are used; this is required for container initializations in the model. It is non-constant if it contains nodes/edges/or member accesses, which may be the case if used in the rules. The elements given in the type constructor are cast to the specified member types if needed and possible. The *ArrayCopyConstructor* creates a new array of the specified type, filling it with the elements from the old array. In case of a node or edge type, the types of the original and the constructed array are allowed to differ, only the elements from the old array that match the type of the new array are taken over then (this is a remedy for array value type invariance).

EXAMPLE (82)

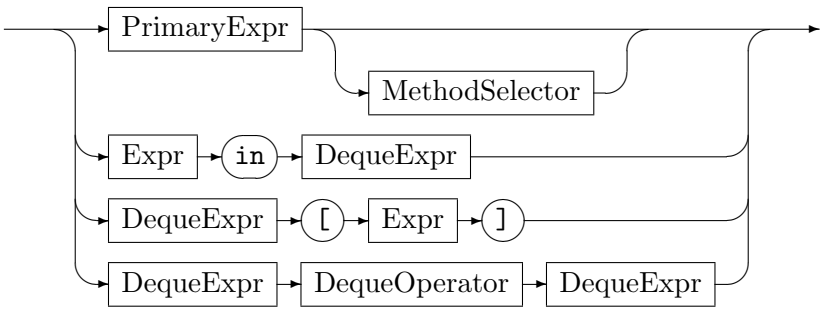
An example explaining some array operations.

```
function arrayExample(ref ai:array<int>) : boolean
{
   def ref a:array<int> = array<int>[];
   a.add(1); // [ 1 ]
   a.add(2); // [ 1, 2 ]
   a.add(3); // [ 1, 2, 3 ], a.size()==3
   def var tmp:int = a.peek(); // end of [ 1, 2, 3 ] is 3
   a.rem(); // [ 1, 2 ] LIFO
   for(k:int -> v:int in a) {
      ai.add(v); // ai[k] == a[k]
   } // a == ai == [ 1, 2 ], assuming input was []
   def var i:int = a.indexOf(2);
   if(i != -1) { // a[i]==2
      a[i] = 3;
   } // [ 1, 3 ]
   a.add(2, 1); // [ 1, 2, 3 ]
   return (2 in (a + array<int>[ 4, 5 ])); // true
}
```

15.5 Deque Operations

Deque expressions consist of deque membership checking, deque value lookup, and deque concatenation, plus some operations in method call notation (on `deque<T>`).

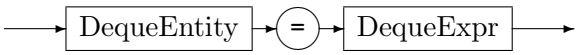
DequeExpr



MethodCall



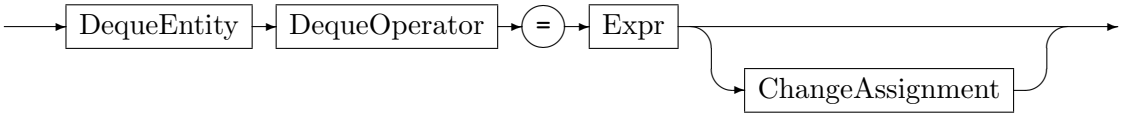
Assignment



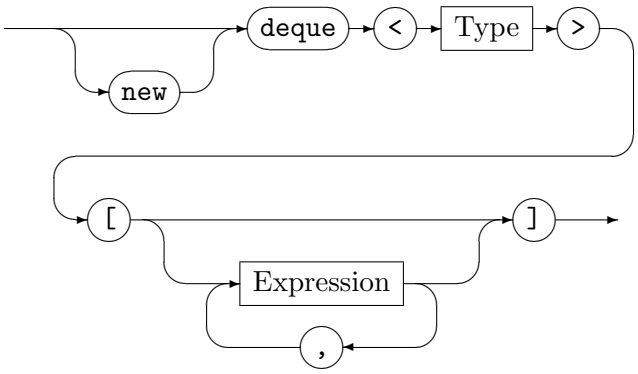
IndexedAssignment



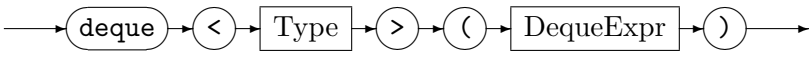
CompoundAssignment



DequeConstructor



DequeCopyConstructor



The query method calls on deques are:

.size()
: returns the number of elements in the deque, as `int`

.empty()
: returns whether the deque is empty, as `boolean`

.peek(num)
: returns the value stored in the deque at position `num:int` in the sequence of enumeration, is equivalent to (and implemented by) `a[num])`; retrieval occurs in constant time.

.peek()
: returns the first value stored in the deque; retrieval occurs in constant time.

.indexOf(valueToSearchFor)
: returns the first position `valueToSearchFor:T` appears at, as `int`, or -1 if not found

.indexOf(valueToSearchFor, startIndex)
: returns the first position `valueToSearchFor:T` appears at, when we start the search for it at deque position `startIndex:int`, as `int`, or -1 if not found

.lastIndexOf(valueToSearchFor)
: returns the last position `valueToSearchFor:T` appears at, as `int`, or -1 if not found

.lastIndexOf(valueToSearchFor, startIndex)
: returns the last position `valueToSearchFor:T` appears at (moving to the begin), when we start the search for it at deque position `startIndex:int`, as `int`, or -1 if not found

.subdeque(startIndex, length)
: returns subdeque of given `length:int` from `startIndex:int` on

.asArray()
: returns an array of the deque content, as `array<T>` for `deque<T>`, in same order

.asSet()
: returns a set of the deque content, as `set<T>` for `deque<T>`

The update method calls on deques are:

Deque addition:
: `d.add(v)` adds the value `v` to the end of deque `d`.

Deque addition:
: `d.add(v,i)` inserts the value `v` at index `i` to deque `d`.

Deque removal:
: `d.rem()` removes the value at the begin of the deque `d`.

Deque removal:
: `d.rem(i)` removes the value at index `i` from the deque `d`.

Deque clearing:
: `d.clear()` removes all values from the deque `d`.

The binary deque operators are:

`+`	Deque concatenation: returns new deque with the right appended to the left deque the left and right operands must be of identical type `deque<T>`

Table 15.8: Binary deque operators, in ascending order of precedence

The operator `x in d` denotes deque value membership, returning whether the deque contains the given element, as `boolean`. It is an O(n) operation for deques. The operator `d[x]` denotes deque lookup, i.e. it returns the value `y` which is stored in the deque `d` at the index `x`. The index `x` *must* be a valid deque index. Furthermore, the container may be iterated over with a `for` loop, as introduced in 14.3. The deque allows for non-indexed as well as indexed iteration; if non-indexed iteration is used the deque values are iterated over, if indexed iteration is used the index is assigned to the index variable and the corresponding value is assigned to the value variable.

The relational expressions (already introduced in 6.4) used to compare entities of different kinds, mapping them to the type boolean, are extended to sets according to table 15.9: A deque `A` is a subdeque of `B` iff it is smaller or equal in size and the values at each common index are identical (lexicographic order as for strings).

The assignments implement the computation constructs introduced in 14. The pure assignment overwrites the target deque with the source deque, commonly with value semantics, creating a copy of the source deque. Only if a local variable (i.e. not an attribute) is assigned to a local variable, is reference semantics used (i.e. both variables point afterwards to the same deque). The indexed assignment `d[i]=v` overwrites the old value at index `i` in the deque `d` with the new value `v`. Compound assignments are assignments which use the first source as target, too, only adapting the target value instead of computing a new value and

`A == B`	True, iff the content of A and B is identical.
`A != B`	True, iff the content of A and B is not identical.
`A ~~ B`	True, iff A and B are by-value equal (deeply).
`A <  B`	True, iff A is a subdeque of B, but A and B are not identical.
`A >  B`	True, iff A is a superdeque of B, but A and B are not identical.
`A <= B`	True, iff A is a subdeque of B or A and B are identical.
`A >= B`	True, iff A is a superdeque of B or A and B are identical.

Table 15.9: Binary deque operators for comparison

overwriting the target with it. For scalars this is not supported, but for container valued entities it is offered due to the potential for massive computational cost savings. The compound assignment statement on deques is the concatenation assignment `+=`.

The *DequeConstructor* extends the *Literal* from 6.8 (as a refinement of the *ContainerConstructor* there). It is constant if only primitive type literals, enum literals, or constant expressions are used; this is required for container initializations in the model. It is non-constant if it contains nodes/edges/or member accesses, which may be the case if used in the rules. The elements given in the type constructor are cast to the specified member types if needed and possible. The *DequeCopyConstructor* creates a new deque of the specified type, filling it with the elements from the old deque. In case of a node or edge type, the types of the original and the constructed deque are allowed to differ, only the elements from the old deque that match the type of the new deque are taken over then (this is a remedy for deque value type invariance).

NOTE (34)

The double ended queue allows for fast addition and removal both at the front and at the back, in contrast to arrays that only support this at the end. It is implemented by a ringbuffer that is grown as needed, a lookup is slightly more expensive than an array lookup. The primary usage of the deque is as a queue, as needed for breadth-first search, accessed FIFO (first-in first-out). This is in contrast to the array that is suited to be employed as a stack, e.g. in a depth-first search (unless that is programmed using the call stack), accessed LIFO (last-in first-out).

EXAMPLE (83)

An example explaining some deque operations.

```
function dequeExample(ref di:deque<int>) : boolean
{
   def ref d:deque<int> = deque<int>[];
   d.add(1); // ] 1 [
   d.add(2); // ] 1, 2 [
   d.add(3); // ] 1, 2, 3 [, d.size()==3
   def var tmp:int = d.peek(); // begin of ] 1, 2, 3 [ is 1
   d.rem(); // ] 2, 3 [ FIFO
   for(k:int -> v:int in d) {
      di.add(v); // di[k] == d[k]
   } // d == di == ] 2, 3 [, assuming input was ][
   def var i:int = d.indexOf(2);
   if(i != -1) {
      d[i] = 1;
   } // ] 1, 3 [
   d.add(2, 1); // ] 1, 2, 3 [
   return (2 in (d + deque<int>[ 4, 5 ])); // true
}
```

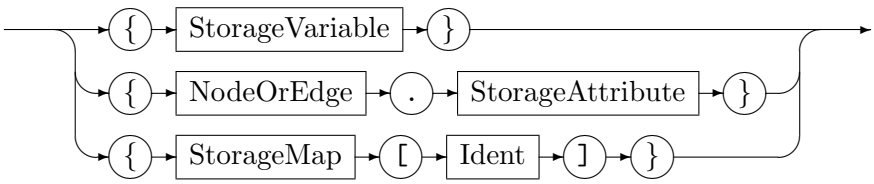
15.6 Storage Access in the Rules

Storages can be used in the rule application control language as introduced above 19.3, they can get filled or emptied in the rules as defined here 5.4.3, a discussion about their usage and examples are given here 24, here 24.5, and here 24.6. In the pattern part you may ask for an element to get bound to an element from a storage or a storage attribute; this is syntactically specified by giving the storage enclosed in left and right braces after the element declaration. You may ask for an element to get bound to the value element queried from a storagemap by a key graph element; this is syntactically specified by giving the storagemap indexed by the key graph element enclosed in left and right braces (this is not possible for storage map attributes due to internal limitations with the search planning). If the type of the element retrieved from the storage is not compatible to the type of the pattern element specified, or if the storage is empty, or if the key element is not contained in the storagemap, matching fails.

The advantage of this storage querying inside the rule over handing in a value from a for loop iterating the storage values outside the rule are: i) a more concise syntax, ii) the ability to access a storage attribute of an element just matched or to access a storage map with an element just matched in the same rule, which would require to break up the rule in two rules in the other case, and iii), a restriction of the iteration to the matching phase, so that at rewriting one can happily manipulate the storage without destroying the iterator/enumerator used in the loop which would be the case when using an outside loop.

The following syntax diagram gives an extension to the syntax diagrams of the Rule Set Language chapter 5, pattern part:

StorageAccess



EXAMPLE (84)

Queries the graph for the neighbouring cities to the cities contained in the storageset.

```
test neighbour(ref startCities:set<City>) : City
{
    :City{startCities} -:Street-> n:City;
    return(n);
}
```

EXAMPLE (85)

Queries for the neighbour of the neighbour of a city matched. With the first neighboring relation queried from the storagemap assumed to contain the neighbouring relation of some cities of interest, and the second neighbouring relation queried from the graph.

```
test neighbourneighbour(ref neighbours:map<City, City>) : City
{
    someCity:City;
    nc:City{neighbours[someCity]} -:Street-> nnc:City;
    return(nnc);
}
```

15.7 Hints on container usage

A container iterated over must stay unchanged during the iteration. But sometimes we want to change the iterated content, e.g. filter the container during iteration. As remedy the `clone` function may be used to clone the container before the iteration, so that the clone can be iterated over and the original container changed.

clone(set<T>):set<T>
: returns a clone of the original set given as argument.

clone(map<S,T>):map<S,T>
: returns a clone of the original map given as argument.

clone(array<T>):array<T>
: returns a clone of the original array given as argument.

clone(deque<T>):deque<T>
: returns a clone of the original deque given as argument.

The container copy constructors could be used instead, but they are syntactically a bit less convenient as the type must be specified – which is important for their intended use, to get a storage with a different node or edge type, containing only the elements matching that type.

EXAMPLE (86)

The container state change methods `add` and `rem` allow to add graph elements to storages or remove graph elements from storages, i.e. sets or maps or arrays or deques holding nodes and edges used for rewrite in the calling sequence (cf. 19.3). This way you can write transformations consisting of several passes with one pass operating on nodes/edges determined in a previous pass, without the need to mark the element in the graph by helper edges or visited flags.

```
1 rule foo(ref storage:set<Node>)
2 {
3   n:Node;
4   modify {
5     eval {
6       storage.add(n);
7     }
8   }
9 }
```

EXAMPLE (87)

The containers are passed as parameters by reference, so only a reference is copied, and changes inside are visible outside. The containers are assigned between variables in computations by reference, so only a reference is copied, and changes of one variable are visible in other variables referencing the same container. But when a container is assigned to a node or edge or internal class object attribute, it is copied by value. So this is an O(n) operation instead of an O(1) operation, yielding clearly worse performance – but the side effects of having references to a shared container would be mind-wrecking here. If you want to change a node or edge or internal class object attribute of container type, you should use for performance reasons `add`, `rem`, and indexed assignments (this esp. holds if you are using transactions).

```
rule foo(ref storage:set<Node>) // parameter passed by-reference
{
  n:N; // assuming an attribute a:set<Node>

  modify {
    eval {
      def ref s:set<Node> = storage; // by-reference
      s.add(n); // now s and storage are changed
      n.a = s; // by-value -- different containers
      n.a.rem(n); // only n.a changed
    }
  }
}
```

EXAMPLE (88)

Some examples of container literals:

```
set<string>{ "foo", "bar" } // a constant set<string> constructor
map<string,int>{ (n.strVal+m.strVal)->(m.intVal+n.intVal), intVal->strVal, "fool"->42 } // a
    non-constant map constructor
array<int>[ 1,2,3 ] // a constant array<int> constructor
deque<int>[ 1,2,3 ] // a constant deque<int> constructor
```

EXAMPLE (89)

A container copy constructor creates a full copy for storages of a more general or the same graph element type than the one stored in the original container. It creates a partial copy with only the elements matching if the type specified is more specific than the one of the original container.

```
1  procedure foo(ref storage:set<Node>)
2  {
3      def ref s:set<N> = set<N>(storage); // constructs new set containing the elements
4                                          // from storage that are of type N or a subtype
5      for(n:N in s) { // safe at runtime, only N contained
6        doSomething(n);
7      }
8      // for(n:N in storage) is allowed - implicitly casted - but could fail at runtime
9      return;
10 }
```

CHAPTER 16

GRAPH TYPE AND COMPUTATIONS

16.1 Built-In Types

Besides the types already introduced, GrGen.NET offers support for a `graph` type. If used explicitly, it denotes a subgraph of the host graph, which can be used for storing and comparing subgraphs of the current host graph. (Because of the "there is only a single host graph"-design of GrGen you must explicitly descend to the nested subgraph if you intend to process it, see 18.4).

But more important are the large number of global functions (with call syntax as already introduced in 6.8) and global procedures (with call syntax as already introduced in 14.5.2) that implicitly operate on the single host graph (and thus on the graph type). There are update functions available that allow to manipulate the graph available with `add`, `rem` and `retype`. The host graph may be queried for all `nodes` or `edges` of a given type. An edge may be queried for its `source` and `target` elements, while a node may be queried for its direct neighbourhood with all `incident` edges, or all `adjacent` nodes. Or even for its transitive neighbourhood with all `reachable` nodes or edges. In the form of functions returning `set`s of nodes or edges, or in the form of iterations with `for` loops, or in the form of boolean predicates to test the neighbourhood if two elements are given. The `induced` subgraph of a set of nodes or edges may be computed, or directly `insert`ed into the host graph.

Furthermore, `visited` flags may be used for marking already visited elements during graph walks or for partitioning a graph. These operations are available in the rule language, as function atoms of the expression sublanguage from 6, and as procedure atoms of the statement sublanguage from 14; most of them are available in the sequence computations language, too (19.1 will tell about the differences compared to the rule language).

The graph manipulation procedures, the transaction handling procedures, and the visited flag management and assignment procedures are not available in the function abstraction; they may only be called from the procedures.

16.2 Graph Functions And Procedures

16.2.1 Graph Updates / Basic Graph Manipulation

There are procedures available to update the graph by adding or removing or retyping, on nodes and edges:

`add(.)`
: adds a new node of the given node type to the host graph, returns the added node.

`addCopy(.)`
: adds a copy of the original node to the host graph, returns the added node.

`addClone(.)`
: adds a clone of the original node to the host graph, returns the added node.

add(.,.,.)
: adds a new edge of the given edge type to the host graph, in between the source node specified as second argument and the target node specified as third argument, returns the added edge.

addCopy(.,.,.)
: adds a copy of the original edge to the host graph, in between the source node specified as second argument and the target node specified as third argument, returns the added edge.

addClone(.,.,.)
: adds a clone of the original edge to the host graph, in between the source node specified as second argument and the target node specified as third argument, returns the added edge.

rem(.)
: removes the node or edge given from the host graph (no expression, does not return anything).

retype(.,.)
: retypes the node or edge given to the node type or edge type given as second argument, returns the retyped entity.

clear()
: clears the host graph.

Besides those basic graph manipulation functions, some advanced rewriting operations are available as procedures, too:

merge(.,.)
: merges the source node given as second argument into the target node given as first argument.

redirectSource(.,.)
: redirects the edge given as first argument to the new source node given as second argument.

redirectTarget(.,.)
: redirects the edge given as first argument to the new target node given as second argument.

redirectSourceAndTarget(.,.,.)
: redirects the edge given as first argument to the new source node given as second argument and the new target node given as third argument.

The versions introduced above are only available on named graphs, as they fetch the debug display name from the old element. If you want to use them on unnamed graphs you must supply an additional argument giving the name of the old element; in case of the `redirectSourceAndTarget` you must supply two additional arguments, first the string to use for the old source node name, then the string to use for the old target node name.

16.2.2 Graph Query by Types

There are functions to ask for all nodes or edges of a type available:

nodes(.)
: returns all nodes in the graph compatible to the given node type, as set.

nodes()
: returns all nodes in the graph, as set.

edges(.)
: returns all edges in the graph compatible to the given edge type, as set.

edges()
: returns all edges in the graph, as set.

The same functions can be used from for loops to iterate over the entities, omitting the filling of a set:

for(n:NodeType in nodes(NodeType)) {*Statements*}
: iterates over all nodes in the graph compatible to the given node type.

for(n:Node in nodes()) {*Statements*}
: iterates over all nodes in the graph.

for(e:EdgeType in edges(EdgeType)) {*Statements*}
: iterates over all edges in the graph compatible to the given edge type.

for(e:Edge in edges()) {*Statements*}
: iterates over all edges in the graph.

16.2.3 Graph Query by Neighbourhood

Multiple functions are available to query the neighbourhood of nodes and edges.

Edge Neighbourhood

The nodes incident to a given edge may be queried by the following functions:

source(.)
: returns the source node of the given edge.

target(.)
: returns the target node of the given edge.

opposite(.,.)
: returns the opposite node of the edge and the node (second argument) given.

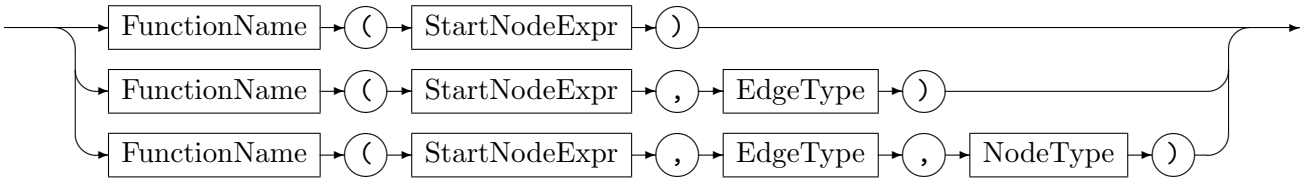
Node Neighbourhood Common Concepts

The edges `incident` or the nodes `adjacent` to a given node may be queried.

The neighbourhood query functions allow to additionally constrain the direction of the edges to `incoming` or `outgoing` edges, otherwise both directions are accepted.

The neighbourhood query functions furthermore allow to constrain the accepted situations by optional arguments. The type the incident edges must have to be accounted for can be specified. Or the type the incident edges must have to be accounted for and the type the adjacent nodes must have to be accounted for (cf. 16.2.3).

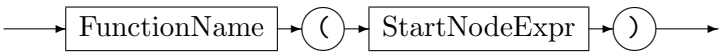
NeighbourhoodFunctionCall



The neighbourhood query function can be used in four possible ways:

Set functions:
The neighbourhood function returns a set of neighbouring entities of the start node. It builds a set that is likely thrown away thereafter, so esp. for large sets this function is less efficient than the other versions.

Expression



Iteration loops:
The neighbourhood function is employed from a for loop (cf. 14.3) that allows to iterate the neighbouring entities of the start node. No set needs to be built here. But if the source node has multiple edges to a target node, it might be iterated multiple times. And an edge may be iterated twice in case of the undirected functions.

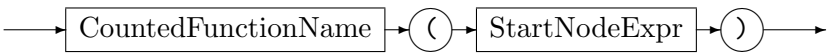
ForFunctionHeader



Such a for graph accessing function loop is also available for the nodes/edges functions introduced above, and many of the following (graph query by neighbourhood, including the transitive, only excluding the with-remaining-depth) functions, defining a general way of accessing the graph elements from the host graph.

Counted functions:
The counted neighbourhood function returns the count of neighbouring entities of the start node. This is at least as efficient as calling `size()` on the resulting set of the plain neighbourhood function, often it is more efficient.

Expression



The *CountedFunctionName* is built from the *FunctionName* by prepending `count` and switching the first character of *FunctionName* to upper case.

Boolean functions:
The neighbourhood function is employed in a boolean predicate version that allows to check whether a second target entity is in the queried neighbourhood of the start node. The computation has the smallest internal processing overhead of the three options and stops as soon as a positive result is obtained.

Expression



The *FunctionNamePredicate* is built from the *FunctionName* by prepending `is` and switching the first character of *FunctionName* to upper case.

Direct Node Neighbourhood

Available are queries for the neighbouring edges:

`incident(.)`
returns the set of the edges that are incident to the node given as argument value.

incoming(.)
same as the incident above, but restricted to incoming edges.

outgoing(.)
same as the incident above, but restricted to outgoing edges.

In the two argument version, only edges of the type given as second argument are contained. The three argument version behaves as the two argument version, but additionally only edges incident to an opposite node of the type given as third argument are contained.

EXAMPLE (90)

```
1  rule foo {
2      src:Node -e:Edge->;
3      if(!isIncoming(src, e));
4
5      modify {
6      ---
7          eval {
8              if(incident(src, NiftyEdge, NiftyNode).size()>2)
9              {
10                 for(ne:NiftyEdge in outgoing(src, NiftyEdge))
11                 {
12                     ne.attr = 42;
13                 }
14             }
15         }
16     }
17 }
```

Some fabricated example showing how to use the incoming, outgoing, and incident functions in their *boolean predicate*, *set function*, and *for iteration* versions, with and without constraining the edge and node types.

Available are queries for the neighbouring nodes:

adjacent(.)
returns the set of the nodes that are adjacent to the node given as argument value.

adjacentIncoming(.)
same as the adjacent above, but restricted to nodes reachable via incoming edges.

adjacentOutgoing(.)
same as the adjacent above, but restricted to nodes reachable via outgoing edges.

In the two argument version, nodes incident to an edge of the type given as second argument are contained. The three argument version behaves as the two argument version, but additionally only nodes of the node type given as third argument are contained.

EXAMPLE (91)

An example showing how to program a rule by hand with the graph query operations for matching and the graph update operations for rewriting. This is what GrGen does under the covers, and as you can see from the volume and style of code very helpful – use the pattern language, don't fall back to the computations language unless really needed.

```
rule example
{
   x:N -e:E-> y:N <--> z:M;
   if { e.a == 42; }

   modify {
      delete(y);
      xn:NN<x>;
      xn --> yn:N --> z;
   }
}
// this procedure behaves similarily to the rule above
procedure example
{
   // match LHS pattern
   def var leave:boolean = false;
   for(x:N in nodes(N)) // lookup n of type N in the graph
   {
      x.visited = true; // see the section on Visited Flags
      for(e:E in outgoing(x, E)) // from x on find outgoing edge e of type E
      {
         if(!(e.a == 42)) { // with e.a == 42
            continue;
         }
         def y:Node = target(e); // and target node y of type N
         if(!(typeof(y)<=N)) {
            continue;
         }
         if(y.visited) { // that is not the same as x
            continue;
         }
         for(z:Node in adjacent(y, Edge, M)) // from y on find adjacent node z of type M
         { // N and M are disjoint, can't match each other, otherwise visited would be needed

            // rewrite according to RHS pattern
            rem(y);
            (def xn:NN)=retype(x,NN);
            (def yn:N)=add(N);
            add(Edge, xn, yn);
            add(Edge, yn, z);

            leave = true; break;
         }
         if(leave) { break; }
      }
      x.visited = false;
      if(leave) { break; }
   }
   return;
}
```

Transitive Node Neighbourhood

Besides direct neighbourhood, transitive neighbourhood can be queried with the reachability functions.

Available are queries for the reachable edges:

reachableEdges(.)
: returns the set of the edges that are reachable from the node given as argument value.

reachableEdgesIncoming(.)
: same as the reachableEdges above, but restricted to incoming edges.

reachableEdgesOutgoing(.)
: same as the reachableEdges above, but restricted to outgoing edges.

In the two argument version, only edges of the type given as second argument are contained and followed. The three argument version behaves as the two argument version, but additionally only edges incident to an opposite node of the type given as third argument are contained and followed.

Available are queries for the reachable nodes:

reachable(.)
: returns the set of the nodes that are reachable from the node given as argument value.

reachableIncoming(.)
: same as any of the reachables above, but restricted to nodes reachable via incoming edges.

reachableOutgoing(.)
: same as any of the reachables above, but restricted to nodes reachable via outgoing edges.

In the two argument version, nodes incident to an edge of the type given as second argument are contained and followed. The three argument version behaves as the two argument version, but additionally only nodes of the node type given as third argument are contained and followed.

EXAMPLE (92)

```
rule bar {
    src:Node;
    tgt:Node;
    if(isReachableOutgoing(src, tgt, NiftyEdge));

    modify {
    ---
        eval {
            if(!(reachableIncoming(src) & reachableOutgoing(src)).empty())
            {
                for(ne:NiftyEdge in reachable(src))
                {
                    ne.attr = 42;
                }
            }
        }
    }
}
```

Some fabricated example showing how to use the `isReachableOutgoing` function to check for an iterated path between the `src` and `target` nodes, how to check for loops by intersecting the sets of nodes reachable by outgoing edges from a node and reachable by incoming edges to a node, and how to iterate with one loop over all edges reachable in either way from a node.
The isReachable functions give the most efficient and most convenient way to check for an iterated path in GrGen, if you need more elaborate checking than constraining the edge type to one type and the target node type to one type you need to program the iterated path with subpattern recursion.
The reachable iteration is the most concise way to note down a depth-first walk over a graph, visiting all elements reachable from a source node on.

Bounded Transitive Node Neighbourhood

Transitive neighbourhood can be queried also with a path of bounded length, with the bounded reachability functions (querying for a bounded iterated path).

Available are queries for the reachable-within-bounds edges:

boundedReachableEdges(.,.)
: returns the set of the edges that are reachable from the node given as first argument value, within at most as much steps as specified by the second argument.

boundedReachableEdgesIncoming(.,.)
: same as the reachableEdges above, but restricted to incoming edges.

boundedReachableEdgesOutgoing(.,.)
: same as the reachableEdges above, but restricted to outgoing edges.

In the three argument version, only edges of the type given as third argument are contained and followed. The four argument version behaves as the three argument version, but additionally only edges incident to an opposite node of the type given as fourth argument are contained and followed.

Available are queries for the reachable-within-bounds nodes:

boundedReachable(.,.)
: returns the set of the nodes that are reachable from the node given as first argument value, within at most as much steps as specified by the second argument.

boundedReachableIncoming(.,.)
: same as any of the reachables above, but restricted to nodes reachable via incoming edges.

boundedReachableOutgoing(.,.)
: same as any of the reachables above, but restricted to nodes reachable via outgoing edges.

In the three argument version, nodes incident to an edge of the type given as third argument are contained and followed. The four argument version behaves as the three argument version, but additionally only nodes of the node type given as fourth argument are contained and followed.

EXAMPLE (93)

```
1  // iterative deepening is trivial due to boundedReachable (encapsulating a depth-bounded
       depth first search)
2  procedure IterativeDeepening(root:Node, var maxdepth:int) : (boolean)
3  {
4      for(depth:int in [0:maxdepth])
5      {
6          for(n:Node in boundedReachableOutgoing(root, depth))
7          {
8              if(foundCondition()) {
9                  return(true);
10             }
11         }
12     }
13     return(false);
14 }
```

An example showing how to use `boundedReachable` in implementing iterative deepening. You find it and an example for depth-first search, as well as an example for breadth-first search implemented with a `deque` in `test/should_pass`, in `/DfsBfsSearch.grg`.

Available are queries for the reachable-within-bounds nodes that return the minimum distance to the start node in addition:

boundedReachableWithRemainingDepth(.,.)
: returns the map of the nodes that are reachable from the node given as first argument value, within at most as much steps as specified by the second argument, to the remaining depth.

boundedReachableWithRemainingDepthIncoming(.,.)
: same as any of the reachables above, but restricted to nodes reachable via incoming edges, to the remaining depth.

boundedReachableWithRemainingDepthOutgoing(.,.)
: same as any of the reachables above, but restricted to nodes reachable via outgoing edges, to the remaining depth.

In the three argument version, nodes incident to an edge of the type given as third argument are contained and followed. The four argument version behaves as the three argument version, but additionally only nodes of the node type given as fourth argument are contained and followed. The depth value for the minimum distance of the node to the root node in the result map is counted downwards from the depth requested originally, you must subtract it from the depth requested to get the real distance.

16.2.4 Subtle Points in the Semantics

Loops versus Containers

A graph query with a for loop iterates the type and/or incidence ringlists which define the graph. A graph query returning a set builds a set – employing the same iteration – and returns that set. That yields two differences:

- If you iterate the set, this happens in the iteration order of the container that was built – very likely you will visit the elements in an order that is different from the original iteration order.

- Duplicates during graph iteration are removed – this holds for reflexive edges that are visited twice during an incident-edges iteration, once in their role of incoming edge, and once in their role of outgoing edge.

Types

Graph functions without specified type return a node of type `Node` or an edge of type `AEdge`. If a type is specified, they return a node or edge of the specified type.

Graph queries which return a set type return for nodes a `set<Node>`, irrespective of a/the specified node type. You won't receive a `set<N>` with `N` being a node type from your model that you requested in the query.

Graph queries which return a set type return for edges

- a `set<Edge>` in case the requested edge type is a directed edge.

- a `set<UEdge>` in case the requested edge type is an undirected edge.

- a `set<AEdge>` in case the requested edge type is an arbitrary directed edge, or no edge type was specified.

You won't receive a `set<E>` with `E` being an edge type from your model that you requested in the query, but at least you get a set of the edge root type matching your requested type. This way, you can evade the unspecific `AEdge`, a recommended practice if you always only use directed edges.

If the behavior regarding set types is not what you want, you can employ a container copy constructor (cf. Chapter 15) to obtain a container of your desired type (containing only the elements that match the container element type), see the following two examples.

EXAMPLE (94)

```
def ref snode:set<Node> = nodes(N); // nodes returns set<Node> as static type,
                                    // even if only nodes of type N are returned at runtime
def ref sn:set<N> = set<N>(snode); // use a container copy constructor for getting
                                   // a set of intended type (here more concrete type)
for(n:N in snode) { // this works because the for loop applies an implicit type cast
                    // this will fail at runtime if snode contains a node
                    // that is not of type N or one of its subtypes
}
for(n:N in sn) { // this will not fail as the container copy constructor
                 // filters out nodes of not matching type (without runtime failure)
}
```

EXAMPLE (95)

```
def ref sedge:set<Edge> = edges(E); // edges returns set<Edge> as static type
                                    // (for E extends Edge), even if only edges
                                    // of type E are returned at runtime
def ref saedge:set<AEdge> = set<AEdge>(sedge); // use a container copy constructor
                                               // for getting a set of intended type
                                               // (here more general type)
saedge.add(uedge); // works for uedge:UEdge,
                   // a set of AEdge can also contain undirected edges
sedge.add(uedge); // fails for the same operation, an undirected edge
                  // cannot be added to a set of directed edges
```

16.3 Subgraph Operations

Several functions and procedures returning and accepting subgraphs are available; they are especially useful in state space enumeration, cf. 24.7, but also in graph-oriented programming with the structuring and information hiding supported by hierarchically nested graphs, cf. 18.4.

In addition to those computations explained below, you can access the currently processed graph via the `this` variable that is available in the expressions of the sequences and the rules. By default it is bound to the host graph, but if processing was relocated in the sequences to a subgraph, it is bound to the currently processed subgraph.

The global functions allow to compute (node-or-edge) induced subgraphs and clone subgraphs:

inducedSubgraph(.)
: returns the induced subgraph (type: `graph`) of the host graph for the set of nodes given as argument value.

definedSubgraph(.)
: returns the defined (edge-induced) subgraph (type: `graph`) of the host graph for the set of edges given as argument value.

copy(.)
: returns a copy of the original subgraph given as argument.

The global procedures allow to insert clones of subgraphs computed with the previously introduced function, or to insert clones of induced subgraphs directly.

insert(.)
: inserts a given subgraph to the current host graph (disjoint union of the nodes and edges); the original graph is destroyed by this (move semantics).

insertCopy(.,.)
: inserts a copy of the given subgraph to the current host graph (disjoint union of the nodes and edges); the original subgraph stays untouched. Returns the copy of the node given as second argument from the host graph.

insertInduced(.,.)
: adds a clone of the subgraph induced by the set of nodes given as first argument to the host graph, returns the clone of the anchor node given as second argument.

insertDefined(.,.)
: adds a clone of the subgraph defined (edge-induced) by the set of edges given as first argument to the host graph, returns the clone of the anchor edge given as second argument.

16.4 File Operations

The functions from the built-in package `File` allow to import subgraphs and check for file existance (for more on packages see 18.3.2).

File::import(.)
: returns the (sub)graph stored in the `.grs`-file given by its path (the main graph is *not* replaced, the graph must be compatible to the model of the current host graph).

File::exists(.)
: returns whether the file given by its path exists.

The `import` function is similar to the `import` command from the GrShell, see Section 26.2, applied to a `.grs`-file. It returns a graph object for further processing instead of replacing the host graph.

The procedures from the built-in package `File` allow to export subgraphs and delete files.

File::export(.)
: exports the current host graph to a file with the given path.

File::export(.,.)
: exports the subgraph given as first argument to a file with the given path.

File::delete(.)
: deletes the file with the given path.

The `export` procedure behaves like the `export` command from the GrShell, see Section 26.2, applied to a `.grs`-file. It allows for programmed exporting, and exporting of parts of the graph alongside subgraph containment.

EXAMPLE (96)

An example showing how to save a graph exploded into its connected components, and how to load it again from them.

```
1  procedure saveConnectedComponents()
2  {
3     def var i:int = 0;
4     while(!empty()) {
5        def var n:Node = fetchNode();
6        def ref connectedComponent:set<Node> = reachable(n) | set<Node>{n};
7        def var sub:graph = inducedSubgraph(connectedComponent);
8        File::export(sub, "cc"+i+".grs");
9        deleteSubgraph(connectedComponent);
10       i = i + 1;
11    }
12    return;
13 }
14
15 procedure loadConnectedComponents()
16 {
17    def var i:int = 0;
18    while(File::exists("cc"+i+".grs")) {
19       insert(File::import("cc"+i+".grs"));
20       i = i + 1;
21    }
22    return;
23 }
24
25 procedure removeSavedConnectedComponentsFiles()
26 {
27    def var i:int = 0;
28    while(File::exists("cc"+i+".grs")) {
29       File::delete("cc"+i+".grs");
30       i = i + 1;
31    }
32    return;
33 }
34
35 function fetchNode() : Node
36 {
37    for(n:Node in nodes()) {
38       return(n);
39    }
40    return(null);
41 }
42
43 procedure deleteSubgraph(ref sn:set<Node>)
44 {
45    for(n:Node in sn) {
46       rem(n);
47    }
48    return;
49 }
```

16.5 Graph comparison

Here we extend the relational expressions already introduced in 6.4 (and already extended in 15 to include container types) with the (sub)graph type.

`A == B`	True, iff A is isomorphic to B.
`A != B`	True, iff A is not isomorphic to B.
`A ~~ B`	True, iff A is structurally the same as B but maybe different regarding the attributes.

Table 16.1: Compare operators on graph expressions

The `graph` type supports the `==` and the `!=` operators, as well as the `~~` operator; on (sub)graph types they tell whether the (sub)graphs are isomorphic to each other (isomorphy checking/graph isomorphy checking) or not, including the attributes, or whether the (sub)graphs are isomorphic disregarding the attributes (i.e. structurally equal – note that on object and container types `~~` denotes deep by-value comparison).

These operators consist just of two characters, but don't underestimate their impact on performance: they do graph isomorphy checking, which is expensive. They are implemented in an early out style, i.e. the more different the graphs are, the earlier does the check return with the result they are not isomorphic. But if the graphs you are checking are isomorphic (which will happen easily if you use automorphic patterns), then you have to pay the full price for isomorphy checking; if this occurs often, your solution may become prohibitively costly – including an external graph canonization library or the `canonize` function may be of interest in that case. The function `canonize(g:graph):string` is intended to provide a canonical string representation for any graph, but currently does not work for all graphs, because it uses the SMILES[Wei88] method, which is producing an equitable partition of graph nodes, not a canonical order; while not offering full fledged graph canonization this algorithm is sufficient for many purposes. It allows to reduce graph comparisons to string comparisons, at the price of computing the Weininger algorithm for equitable partitions of nodes.

Some notes on the early out implementation: first the number of nodes and edges per type are checked, if they are different the graphs can't be isomorphic. The numbers are directly supplied by the `lgspBackend`, refuting isomorphy based on them is extremely cheap. Then the V-Structures (see 35.4) used in computing better matchers at runtime are first computed and then compared; if they are different the graphs can't be isomorphic. They are a good deal less expensive to compute than trying to match the one graph in the other; on well typed graphs the V-Structure counts are highly discriminating.

If these two pruning methods failed, a matcher is computed from one graph with search planning based on the V-Structure information just computed, and then applied on the other graph. The matchers are stored in the graphs from which they originated, so if you do repeated comparisons of a subgraph which does not change, take care to extract that subgraph only once, store it, e.g. as an attribute in the graph, and continue to compare against it. This will save you from the cost of repeated search planning; in addition, often-used isomorphy matchers get eventually compiled resulting in a further speed-up.

You may furthermore check for (sub)graph equality against a set of (sub)graphs at once:

equalsAny(.,.)
: returns whether the (sub)graph given as first argument (type: `graph`) equals any of the (sub)graphs available in the set of subgraphs given with the second argument (type: `set<graph>`).

equalsAnyStructurally(.,.)
: returns whether the (sub)graph given as first argument (type: `graph`) is structurally (neglecting attribute values) equal to any of the (sub)graphs available in the set of subgraphs given with the second argument (type: `set<graph>`).

The first function employs `==` in a loop until a decision is reached, the second function does so with `~~`. They are a bit more convenient in usage compared to writing the loop by hand, but above all are they amenable to parallelization. You must specify the (maximum) number of worker threads to enable parallel execution, see section 31.4.

More constructive versions of the `equalsAny` functions are available with the `getEquivalent` functions also returning an isomorphic witness graph (they are also amenable to parallelization with the same flag that applies to the `equalsAny` functions).

getEquivalent(.,.)
: returns the subgraph (type: `graph`) from the set of subgraphs given with the second argument (type: `set<graph>`) that is isomorphic to the (sub)graph given as first argument (type: `graph`).

getEquivalentStructurally(.,.)
: returns the subgraph (type: `graph`) from the set of subgraphs given with the second argument (type: `set<graph>`) that is isomorphic to the (sub)graph given as first argument (type: `graph`), neglecting attribute values (i.e. structural equality is sufficient).

Different versions (requiring an `array` of `graph`s instead of a `set` of `graph`s) return the isomorphic graph, or add the graph in case no matching one is found, you find more on `getEquivalentOrAdd(.,.)` and `getEquivalentStructurallyOrAdd(.,.)` in Section 23.3.

EXAMPLE (97)

An example showing some subgraph extraction, comparison, and insertion operations.

```
rule init
{
   modify { // creates the host graph our example rule and function are working on
      start1:SN -:contains-> s1x:Node;
      start1    -:contains-> s1y:Node;
      s1x --> s1y;

      start2:SN -:contains-> s2x:Node;
      start2    -:contains-> s2y:Node;
      start2    -:contains-> s2z:Node;
      s2x --> s2y --> s2z --> s2x;
   }
}

function unequalContainedSubgraphs(start1:SN, start2:SN) : boolean
{
   def ref adj:set<Node> = adjacentOutgoing(start1, contains); //adj=={s1x,s2x}
   def var sub1:graph = inducedSubgraph(adj); // sub1==graph(s1x' -->' s1y') -- does not
       contain s1x,s1y themselves!
   def var sub2:graph = inducedSubgraph(adjacentOutgoing(start2, contains));
   def var res:boolean = sub1 == sub2; // false as graph(s1x' -->' s1y') != graph(s2x' -->'
       s2y' -->' s2z' -->' s2x'), answered quickly because number of elements different
   def var sub3:graph = copy(sub1);
   def var res2:boolean = sub1 == sub3; // true as graph(s1x' -->' s1y') isomorphic
       graph(s1x'' -->'' s1y''), answered slowly because isomorphy matching
   return(res); // remark: the original graph is untouched
}

rule example
{
   start1:SN; start2:SN;
   if{ unequalContainedSubgraphs(start1, start2); }

   modify {
   ---
      eval {
         insert(inducedSubgraph(adjacentOutgoing(start2, contains) | set<Node>{start2}));
            // 1. computes the union of the set of the nodes outgoing from start2 with the
                set containing start2
            // 2. creates a structurally equal clone of the induced graph of the node set
                handed in (start2, s2x, s2y, s2z, and all edges in between)
            // 3. inserts the clone into the original graph (disjoint union)
            // the clone was destroyed by insert, can't be accessed further
         (def start2n:SN)=insertInduced(adjacentOutgoing(start2), start2);
            // does the same, just a bit simpler and more efficient,
            // with start2n you have a node that gives you access to the subgraph just
                inserted (by computing adjacentOutgoing(start2n, contains)),
            // so you can do further processing of that newly created piece,
            // e.g. link it to other nodes in the original graph
      }
   }
}
```

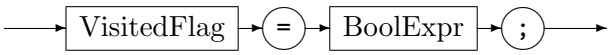
16.6 Visited Flags

The boolean `visited` flags are available for/in each graph element; they may be used for marking already visited graph elements during graph walks or for partitioning a graph. They can be queried with a function of the expressions, can be set with an assignment of the statements, and can be reset, allocated, and deallocated with procedures of the statements. The visited flags are stored in some excess bits of the graph elements, if this pool is exceeded they are stored in additional dictionaries, one per visited flag requested. This is why the flags must get allocated/deallocated, and all flag related operations require an integer number – the flag id – specifying the flag to operate on (with exception of the allocation operation returning this flag id). If you try to access a not previously allocated visited flag, an exception is thrown. The following syntax diagram gives an extension to the *Expression* clause of chapter 6 and an extension to the computation *Statements* of chapter 14:

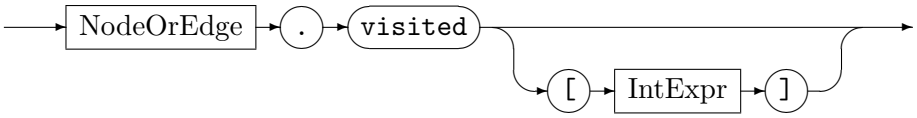
Expression



VisitedAssignment



VisitedFlag



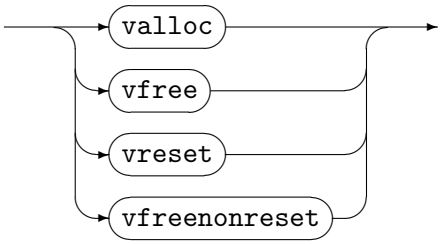
Flag reading:

By `e.visited[f]` – the function returns the `boolean` visited status of the flag given by the flag id variable `f` of the graph element `e` (visited flags are normally read by `if` conditions of the rule language). If no `int` flag number is given, the default number for the first visited flag of 0 is used; it still must have been allocated before.

Flag assignment:

By `e.visited[f] = b` – sets the visited status of the flag given by the flag id variable `f` of the graph element `e` to the given boolean value `b` as computed by an expression (visited flags are normally written by `eval` parts of the rule language). If no `int` flag number is given, the default number for the first visited flag of 0 is used; it still must have been allocated before.

ProcedureName



The signatures of the procedures `valloc`, `vfree`, `vreset`, `vfreenonreset` for managing the visited flags are defined in 14.4. Their semantics are:

Flag allocation:
By `valloc()` – allocates space for a visited flag in the elements of the graph and returns the id of the visited flag (integer number), starting at 0. Afterwards, the visited flag of the id can be read and written by the `visited`-expression and the `visited`-assignment. The first visited flags are stored in some excess bits of the graph elements and are thus essentially for free, but if this implementation defined space is used up completely, the information is stored in graph element external dictionaries.

Flag deallocation:
By `vfree` – frees the space previously allocated for the visited flag; afterwards you must not access it anymore. The value passed in `vfree(IntExpr)` must be of integer type, stemming from a previous allocation. This function internally calls a `vreset` to ensure that no corresponding visited flag is set in the graph.

Flag resetting:
By `vreset` – resets the visitor flag given by the flag id variable in *all* graph elements.

Flag deallocation without reset:
With `vfreenonreset` the space previously allocated for the visited flag is freed, too, but the implicit internal `vreset(id)` of `vfree` is not executed. It is your duty to ensure the flag is `false` in all graph elements – otherwise after a following allocation elements may start as being marked. This saves us an O(n) operation, but opens the door to nasty bugs if you can't design your algorithm in a way which renders unmarking trivial.

EXAMPLE (98)

An example showing how to compute the length of the longest path starting at some node.

```
1  procedure lengthOfLongestPath(start:Node, var lengthReached:int, var flag:int) : (int)
2  {
3     def var maxLength:int = lengthReached;
4     start.visited[flag] = true;
5     for(child:Node in adjacent(start)) {
6        if(!child.visited[flag]) {
7           def var lengthOfLongestPathStartingAtChild:int;
8           (lengthOfLongestPathStartingAtChild)=lengthOfLongestPath(child, lengthReached+1,
                flag);
9           maxLength = Math::max(maxLength, lengthOfLongestPathStartingAtChild);
10       }
11    }
12    start.visited[flag] = false;
13    return (maxLength);
14 }
15
16 rule getLolp(start:Node) : (int)
17 {
18    modify {
19    ---
20       def var lolp:int;
21       eval {
22          (def var flag:int) = valloc();
23          (yield lolp) = lengthOfLongestPath(start, 0, flag;)
24          vfree(flag);
25       }
26       return(lolp);
27    }
28 }
```

16.7 Graph Processing Environment Procedures

16.7.1 Transaction Handling

In addition to the procedures implemented directly in the graph, there are transaction manager procedures available, implemented in the graph processing environment. While a transaction is underway, an undo log is filled with commands to undo the changes that occurred in the graph in the meantime. Those transaction handling procedures from the built-in package `Transaction` (for more on packages see 18.3.2) are:

Transaction::start()
: starts a transaction and returns its transaction id (as number of type `int`).

Transaction::pause()
: pauses transaction handling so changes are not recorded and can't be undone.

Transaction::resume()
: resumes paused transaction handling.

Transaction::commit(.)
: keeps the changes of the transaction of the given id (of type `int`) in the graph, removing undo information.

Transaction::rollback(.)
: reverts the changes of the transaction of the given id (of type `int`) in the graph by executing the undo log.

Please note that transactions may be nested; those functions are used in implementing the transaction and backtracking constructs explained in 22.2.

16.7.2 Assertions

The procedures `assert` and `assertAlways` allow to ensure that a specific condition holds, which must be given as first parameter, a message to be printed in case the condition is false may be given as second parameter, further parameters are appended to the assertion message (after having been transformed to strings).

assert(.[,.])*
: fires when the condition yields false and assertions are enabled, cf. 25.12 (only in that case is the condition evaluated, the same holds for the other arguments).

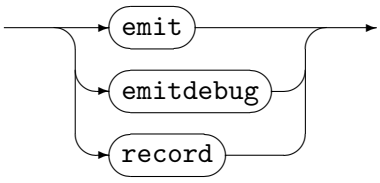
assertAlways(.[,.])*
: fires always when the condition yields false.

A firing aborts program execution by default, or allows you to choose in the debugger how to proceed (in between program abort, external debugging, and execution continuation). For the asserts in the rule language, a default message reporting the file/line/column (of the `.grg` file) is supplied by the compiler in case of a missing message (no such default message is available for asserts in the sequences language).

16.7.3 Misc. Global Procedures

Besides there are procedures to emit text or record graph changes available:

ProcedureName



emit(.[,.])*
: writes the argument value, or the comma separated list of argument values, to stdout, or to a file if output was redirected (with the shell command `redirect emit`). The argument(s) must be of string type, but any type is automatically cast into its string representation as needed. Prefer comma separated arguments over string concatenation, they are more efficient as no intermediate strings need to be computed, just to be garbage collected thereafter.

emitdebug(.[,.])*
: writes the argument value, or the comma separated list of argument values, always to stdout, even if file output was redirected. Besides that, the other explanations for `emit` apply.

record(.)
: writes the argument value, typically a text string, to the currently ongoing graph change recordings (see 26.3). This feature allows to mark states reached during the transformation process in order to replay only interesting parts of a recording. It is recommended to write only comment label lines, i.e. `"#"`, some label, and `"\n"`.

A remark on the graph global variables: they are in fact global to the graph processing environment. This difference becomes clear when you store graphs in attributes of graph type of another graph, see 18.4 for a discussion of this style of programming.

CHAPTER 17

FILTERING AND SORTING OF MATCHES

When a test or rule was matched, a match or an array of matches is available, with a mapping of the pattern elements to the bound host graph elements.

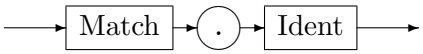
Filters may then be used to process the matches array of a rule all application (including the one-element matches array of a single rule application) – after all matches were found, but before they are rewritten. They allow you to follow only the *most promising matches* during a search task.

You may implement filters in the form of *filter functions* on your own. Alternatively, you may declare certain filters at their rules, they are then *auto-generated* for you. Some filters are *auto-supplied* and may just be used. All filters are used by *filter calls* from the sequences (normally together with a rule all application). A special kind of auto-supplied filter comes with a *lambda expression* at its call site.

17.1 Match types

The match of the pattern of a rule named `r` has the type `match<r>`. The match of the pattern of a first-level iterated `it` of a rule `r` has the type `match<r.it>`. A match class `mc` also defines a match type `match<class mc>`, see Section 21.4 for match classes.

MemberAccess



You can access the members of a match type value in form of the known *MemberAccess* (in the rule language as well as in the sequences), e.g. `m.n` returns from match `m` the value bound to pattern element/match class member `n`.

In Example 13, the match type `match<r>` contains the nodes `n:N` and `m:Node`. The edge `_edge0:Edge` only appears at API level, it stems from automatic naming of the anonymous edge in the pattern – if you want to access a pattern element, assign it a name.

Furthermore, you may execute a `clone` function to get a clone of a match; it is defined with the signature `clone(match<T>):match<T>`. Exceeding these already introduced match type operations, you can also apply a *MatchClassConstructor* `match<class mc>()` in order to create a value of a match class type; the values of the action match types in contrast are always created by pattern matching.

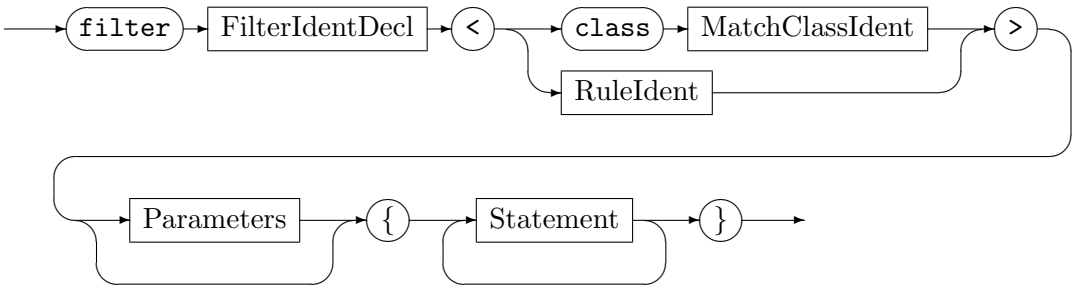
A rule query with test `t` specified by `[?t]` (cf. 20.2) yields an `array<match<t>>`. An iterated query specified by `[?it]` (cf. 6.8) yields an `array<match<t.it>>`. A multi rule query with test `t` implementing match class `mc` specified by `[?[t]\<class mc>]` (also cf. 20.2) yields an `array<match<class mc>>`. These arrays are to be processed by the array methods introduced in Section 15.4 and Section 20.1.

The other way to process arrays of matches is with filters. In a filter function, the matches array created by pattern matching is made available via the `this` variable, see below.

17.2 Filter Functions

Filter functions need to be *defined*, i.e. declared and implemented before they can be *used*.

FilterFunctionDefinition



A filter function definition is given in between the rules as a global construct. It specifies the name of the filter and the parameters supported, furthermore it specifies which rule (cf. 5) or match class (cf. 21) the filter applies to, and it supplies a list of statements in the body.

The restrictions of a function body apply here, i.e. you are not allowed to manipulate the graph. In contrast to the function body is a `this` variable predefined, which gives access to the matches array to filter, of type `array<match<r>>`, where `r` denotes the name of the rule the filter is defined for, or of type `array<match<class mc>>`, where `mc` denotes the name of the match class the filter is defined for. All the operations known for variables of array type (cf. Section 15.4 and Section 20.1) are available.

EXAMPLE (99)

The following filter `ff` for the rule `foo` clones the last match in the array of matches, and adds it to the array. So the last match would be rewritten twice (which is fine for `foo` that only creates a node, but take care when deleting or retyping nodes). But in the following the first entry and the last entry are deleted (again); the first in an efficient in-place way by assigning `null`.

```
1  rule foo
2  {
3     n:Node;
4     modify {
5        nn:Node;
6     }
7  }
8
9  filter ff<foo>
10 {
11    this.add(clone(this.peek())); // note the clone on match<foo>
12    this[0] = null; // removes first entry in matches array, efficiently
13    this.rem();
14 }
```

EXAMPLE (100)

The following rule `incidency` yields for each node `n` matched the number of incident edges to a def variable `i` and assigns it in the eval to an attribute `j`. The filter `filterMultiply` modifies the def variable `i` in the matches with the factor `f` handed in as filter parameter.

```
rule incidency
{
   n:N;
---
   def var i:int;
   yield { yield i = incident(n).size(); }

   modify {
      eval { n.j = i; }
   }
}
filter filterMultiply<incidency>(var f:int)
{
   for(m:match<incidency> in this)
   {
      m.i = m.i * f;
   }
}
```

The following call triples the count of incident edges that is finally written to the attribute `j` for each node `n` matched by `incidency`:

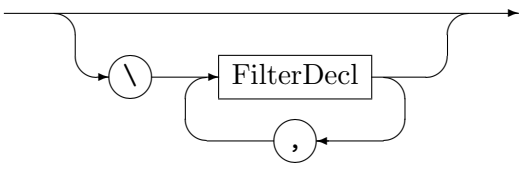
```
  exec [incidency \ filterMultiply(3)]
```

You may declare external filters that you have to implement then in a C# accompanying file, see 34.4 for more on that. With filter functions and external filter functions, you may implement matches-list modifications exceeding the available pre-implemented functionality.

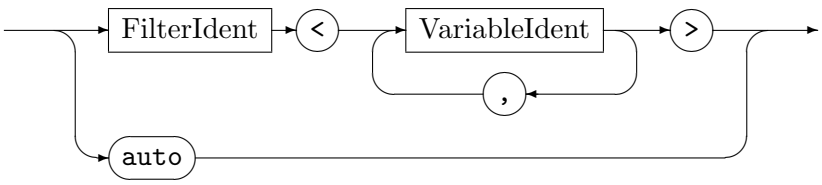
17.3 Auto-Generated Filters

Auto-generated filters need to be *declared* at their rule or iterated pattern or match class before they can be *used*, they are implemented by GrGen.NET.

FiltersDecl



FilterDecl



The auto-generated filters must be declared at the end of a rule definition (cf. 5.2), or iterated pattern definition (cf. 7.3), or match class definition (cf. 21.4). With exception of

the `auto` filter for removal of automorphic matches they have to specify the name of a `def` or non-`def` entity contained in the pattern (or in the iterated pattern or in the match class), which may be a variable of one of the basic types or a graph element type (not a container type), or a node or an edge. They filter based on certain conditions regarding that entity; filtering may mean to reorder a matches list alongside the chosen (def) variable.

The allowed types depend on the filter (and the accumulation method). The ordering filters utilize the ordering relation, so they support only (def) variables of numeric type (`byte`, `short`, `int`, `long`, `float`, `double`,) or `string` or `enum` type (also `boolean` is supported, even if not of much use). The other filters utilize the equality comparisons, so they support (def) variables of the types offering the ordering relation, plus node and edge types, or nodes or edges (i.e. entities). The accumulation variable of the `keepOneForEachAccumulateBy` filter must be of a numeric type, the exact type depends on the result type of the used array accumulation method (`double` is supported by all of them).

`orderAscendingBy<v>`
: orders the matches list ascendingly (from lowest to highest value according to the `<` operator), alongside the `v` contained in each match.

`orderDescendingBy<v>`
: orders the matches list descendingly, alongside the `v` contained in each match.

`keepOneForEach<v>`
: filters away all matches with duplicate `v` values, i.e. only one (prototypical) match is kept per `v` value.

`keepOneForEach<v>Accumulate<w>By<m>`
: filters away all matches with duplicate `v` values, i.e. only one (prototypical) match is kept per `v` value. The variable `w` receives the result of accumulating all `w` of equal `v` with the array accumulation method `m` (see Section 20.1 for the available array accumulation methods).

`groupBy<v>`
: ensures matches of equal `v` values are neighbours. Grouping can be applied on types that only support the `==` operator.

`keepSameAsFirst<v>`
: filters away all matches with `v` values that are not equal to the `v` value of the first match. May be used to ensure all matches of an indeterministically chosen value are rewritten, but typically you want to order the matches list before.

`keepSameAsLast<v>`
: filters away all matches with `v` values that are not equal to the `v` value of the last match.

The `orderAscendingBy<.[,.]*>` and `orderDescendingBy<.[,.]*>` allow to specify more than one variable, all others (with exception of `keepOneForEachAccumulateBy` and `auto`) support only one variable. The order-by filters order the matches in order of the (def) variables, e.g. `orderAscendingBy<v,w>` orders the matches primarily by `v` and secondarily by `w`, i.e in case the values of the first def variable are equal, the values of the second one decide.

Besides those variable-based filters you may use the automorphic matches filter to purge symmetric matches. In order to do so, specify the special filter named `auto` at the rule declaration, and use it with the same name at an application of that action (with the known backslash syntax). The filter is removing matches due to an automorphic pattern, matches which are covering the same spot in the host graph with a permutation of the nodes, the edges, or subpatterns of the same type at the same level. Other nested pattern constructs which are structurally identical are not recognized to be identical when they appear as commuted

subparts; but they are so when they are factored into a subpattern which is instantiated multiple times. It is highly recommended to use this symmetry reduction technique when building state spaces including isomorphic state collapsing, as purging the matches which lead to isomorphic states early saves expensive graph comparisons – and often it gives the semantics you are interested in anyway.

17.4 Auto-Supplied Filters

A few filters (that do not need information from a specific match) are auto-supplied, they can be used without implementation and even without declaration. They allow keeping or removing a certain number of matches from the beginning or the end of the matches list.

The following auto-supplied filters may be used directly:

keepFirst(count)
: keeps the first `count` matches from the begin of the matches list, `count` must be an integer number.

keepLast(count)
: keeps the first `count` matches from the end of the matches list, `count` must be an integer number.

keepFirstFraction(fraction)
: keeps the `fraction` of the matches from the begin of the matches list, `fraction` must be a floating point number in between `0.0` and `1.0`.

keepLastFraction(fraction)
: keeps the `fraction` of the matches from the end of the matches list, `fraction` must be a floating point number in between `0.0` and `1.0`.

removeFirst(count)
: removes the first `count` matches from the begin of the matches list, `count` must be an integer number.

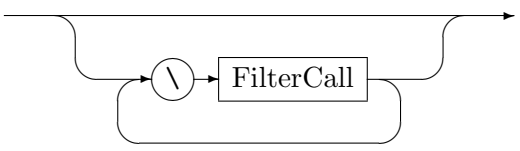
removeLast(count)
: removes the first `count` matches from the end of the matches list, `count` must be an integer number.

removeFirstFraction(fraction)
: removes the `fraction` of the matches from the begin of the matches list, `fraction` must be a floating point number in between `0.0` and `1.0`.

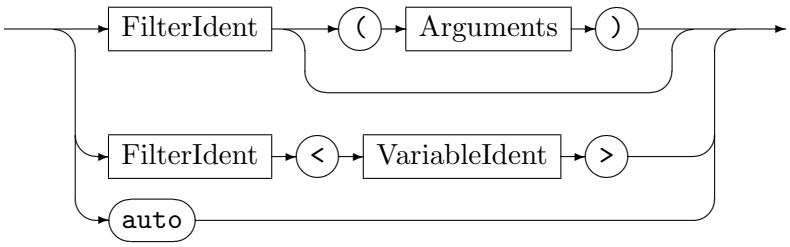
removeLastFraction(fraction)
: removes the `fraction` of the matches from the end of the matches list, `fraction` must be a floating point number in between `0.0` and `1.0`.

17.5 Filter Calls

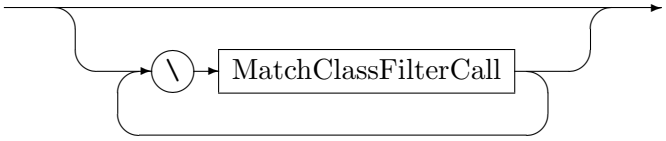
FilterCalls



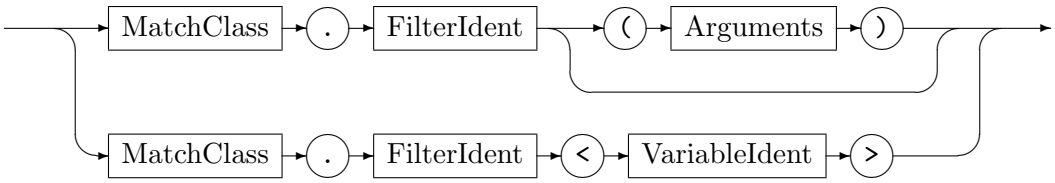
FilterCall



MatchClassFilterCalls



MatchClassFilterCall



Filters are employed with `r\f` notation at rule calls from the sequences language. This holds for user-implemented, auto-generated, and auto-supplied filters. Match class filters (defined on the match class) are employed with `[[r]\mc.f]` notation at multi rule calls from the sequences language. This holds for user-implemented, auto-generated, and auto-supplied filters, too.

The normal name resolving rules (regarding packages) apply in an extended version (point 2 added):

1. content from the package specified with the prefix, if that is "global" then global content
2. content from the package the rule or match class is contained in to which the filter applies (so the filter is a dependent entity)
3. content from package context (which thus overrides global content)
4. global content

EXAMPLE (101)

The following rule `deleteNode` yields to a def variable the amount of outgoing edges of a node, and deletes the node. Without filtering this behaviour would be pointless, but a filter ordering based on the def variable is already declared for it.

```
1  rule deleteNode
2  {
3     n:Node;
4  ---
5     def var i:int;
6     yield {
7        yield i = outgoing(n).size();
8     }
9
10    modify {
11       delete(n);
12    }
13 } \ orderAscendingBy<i>
```

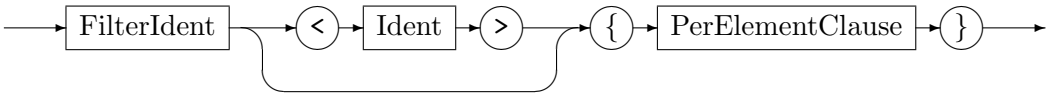
The rule may then be applied with a sequence like the following:

```
1  exec [deleteNode \ orderAscendingBy<i> \ keepFirstFraction(0.5)]
```

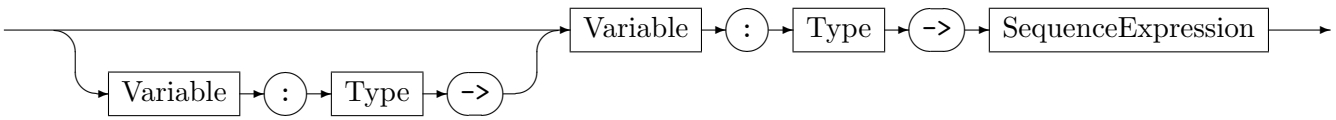
This way, the 50% of the nodes with the smallest number of outgoing edges are deleted from the host graph (because they don't get filtered away), or rephrased: the 50% of the nodes with the highest number of outgoing edges are kept (their matches at the end of the ordered matches list are filtered away).

17.6 Lambda Expression Filter Calls

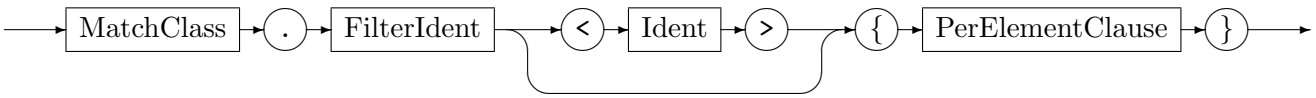
FilterCall



PerElementClause



MatchClassFilterCall



Some lambda expression filter calls are available with a special syntax giving a lambda expression (instead of arguments). The lambda expression is evaluated for each element of the matches array, with the current match being available in a to be declared element variable (of type `match<r>`), and optionally the array index in a to be declared index variable (of type `int`). A match member may be given enclosed in angles to denote an assignment target. No declaration of the filter is needed, the lambda expression is to be given with the filter call.

`\assign<member>{element:match<r> -> lambda-expr}`
: assigns to the `member:T` of a match of a rule `r` the result of the sequence lambda expression fed by the `element:match<r>` of the matches (the name can be chosen freely), for each element in the matches yielded by the rule.

`\removeIf{element:match<r> -> lambda-expr}`
: removes the matches for which the predicate given by the sequence lambda expression (which must be of type `boolean`) fed by the `element:match<r>` of the matches (the name can be chosen freely) yields `true`.

`\assign<member>StartWith{init-expr}AccumulateBy`
: `{accumulation-var:T, element:match<r> -> lambda-expr}` assigns to the `member:T` of a match of a rule `r` the result of the sequence lambda expression fed by the `element:match<r>` of the matches (the name can be chosen freely), and the previous result in `accumulation-var:T`, for each element in the matches yielded by the rule. For the first iteration, the previous result is initialized to `init-expr:T`.

According match class versions are available, to be called for a match class `mc` with
`\mc.assign<member>{element:match<class mc> -> lambda-expr}` and
`\mc.removeIf{element:match<class mc> -> lambda-expr}` and
`\mc.assign<member>StartWith{init-expr}AccumulateBy{accumulation-var:T, element:match<r> -> lambda-expr}`.
Also, according iterated versions are available, to be called for an iterated `it` of a rule `r` in the rule language (in that case, rule language expressions are employed) with
`\assign<member>{element:match<r.it> -> lambda-expr}` and
`\removeIf{element:match<r.it> -> lambda-expr}` and
`\assign<member>StartWith{init-expr}AccumulateBy{accumulation-var:T, element:match<r> -> lambda-expr}`.

They all allow for an optional index: the variable `index:int` (the name can be chosen freely) iterates the matches-array indices [0,a.size()-1] of the corresponding elements in the array `a:array<match<r>>` in the
`a.assign<member>{index:int -> element:match<r> -> lambda-expr}` and
`a.removeIf{index:int -> element:match<r> -> lambda-expr}` and
`a.assign<member>StartWith{init-expr}AccumulateBy{accumulation-var:T, index:int -> element:match<r> -> lambda-expr}` expressions.

Even more, they all allow for an optional array access variable: `this_:array<match<T>>` given as first variable declaration in the lambda expression (the name can be chosen freely), separated by a semicolon, references the original input array, allowing to access it in order to obtain further information, in the
`a.assign<member>{this_:array<match<r>>; element:match<r> -> lambda-expr}` and
`a.removeIf{this_:array<match<r>>; element:match<r> -> lambda-expr}` and
`a.assign<member>StartWith{this_:array<match<r>>; init-expr}AccumulateBy{this_:array<match<r>>; accumulation-var:T, index:int -> element:match<r> -> lambda-expr}` expressions.

This allows to e.g. decide based on a comparison against the average of an attribute:
`[?r\removeIf{this_:array<match<r>>; m:match<r> -> m.a < this_.extract<a>().avg()}]`

17.7 Quick Reference Table

orderAscendingBy<var [, var]*> orderDescendingBy<var [, var]*> groupBy<entity> keepSameAsFirst<entity> keepSameAsLast<entity> keepOneForEach<entity> keepOneForEach<entity>Accumulate<accvar>By<method> auto
keepFirst(int) keepLast(int) keepFirstFraction(double) keepLastFraction(double) removeFirst(int) removeLast(int) removeFirstFraction(double) removeLastFraction(double)
assign<entity>{[this_:array<match<T>>;][index:int->]elem:T->lambda-expr} removeIf{[this_:array<match<T>>;][index:int->]elem:T->lambda-expr} assign<entity>StartWith{[this_:array<match<T>>;]init-expr}AccumulateBy {[this_:array<match<T>>;]accumulation-var:T,[index:int->]elem:T->lambda-expr}

Let `var` be a (def-)variable of a type $\in$ {`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `string`, `enum`}. Let `entity` be a (def-)variable of one of the types admissible for `var` or `Node` or `Edge` type or a (def-)node or a (def-)edge. Let `accvar` be a def-variable of a numeric type, the exact type required depends on the result type of `method`

Table 17.1: Auto-generated and then auto-supplied filters at a glance

CHAPTER 18

ADVANCED MODELLING (OBJECT-ORIENTED AND GRAPH-ORIENTED PROGRAMMING)

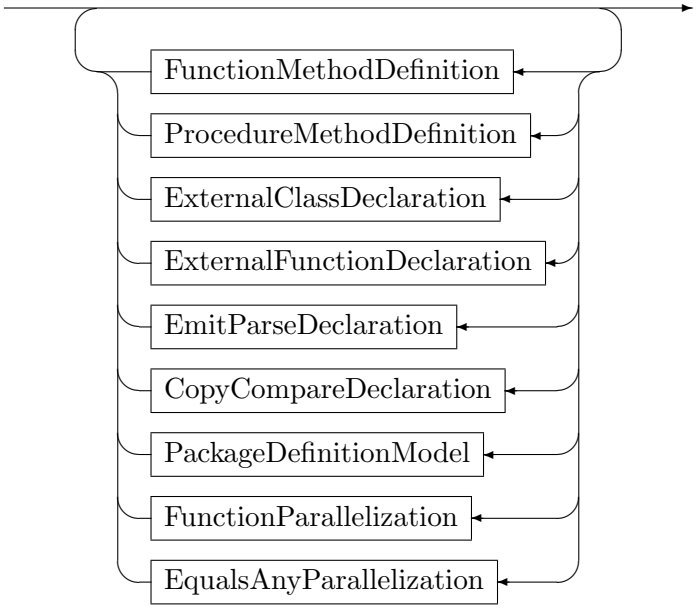
In addition to the node and edge classes (the key features of the *graph models* of GrGen.NET as already introduced in Chapter 4), you can define *object classes* and *transient object classes*. They allow you to group attributes that are commonly used together into an own entity. By means of them, list and tree-like (sub-)data structures can be used within a graph (in a storage efficient way, without the built-in graph indices of the nodes/edges, cf. 30.3).

The exact data layout can be even abstracted from with *methods* (i.e. you access by method instead of the attribute directly); or, in general, arbitrary object-specific behavior can be implemented by them. You may specify methods not only in object and transient object classes, but also in the node or edge classes. Methods are functions or procedures that are declared inside class scope. Methods support overwriting, in contrast to attributes (each attribute of a type is added to all subtypes, with multiple declarations being forbidden). In case a method is called on an object, the method that was defined nearest to the exact dynamic type of the object gets executed, in contrast to the statically known type of the variable storing the object.

This *dynamic dispatch* is the defining feature of object-oriented programming. For graph-oriented programming, the defining feature is *pattern-matching*.

Graph-oriented programming works based on a global host graph open for pattern-matching, in contrast to object-oriented programming with its heap of objects, which typically hide their innards, separating information into local parts. It is thus less scalable out of the box – it can be made scalable to larger tasks with hierarchically *nested graphs*, allowed-for by attributes of graph type in the model, implemented with switch-to-subgraph and return-from-subgraph operations.

AdvancedModelDeclarations

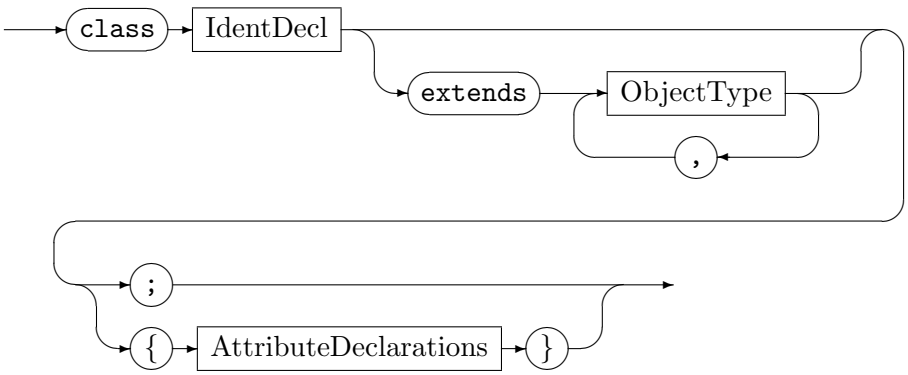


The *ExternalClassDeclaration* registers an external class with GrGen.NET– including its subtype hierarchy, excluding any attributes – which can be used subsequently in attribute computations in external functions. The *EmitParseDeclaration* declares that the user defines emit and parse functions for external or object type values, whereas the *CopyCompareDeclaration* declares that the user defines copy and compare functions for external or object type values. You find more on them in 34, here we will take a closer look on the *FunctionMethodDefinition* and *ProcedureMethodDefinition*, followed by the *PackageDefinitionModel*. The *FunctionParallelization* and *EqualsAnyParallelization* are explained in 31.4.

18.1 Object Types and Transient Object Types

18.1.1 Object Types

ObjectClass



An *ObjectClass* definition defines a new object type. Object types can inherit from other object types (defined within the same file). If the `extends` clause is omitted, *ObjectType* will inherit from the built-in type `Object`, the root type of all internal class objects. Note that `Object` is not the same as `object`, the root type of all types, which does not have much semantic meaning but acts as a mere placeholder, also including `external` class types. Optionally objects can possess attributes. The keywords `abstract` and `const` have the same meaning as for node and edge types, i.e. non-instantiable values and unmodifiable attributes, cf. 4.2.1.

UniqueObjectConstraintDecl



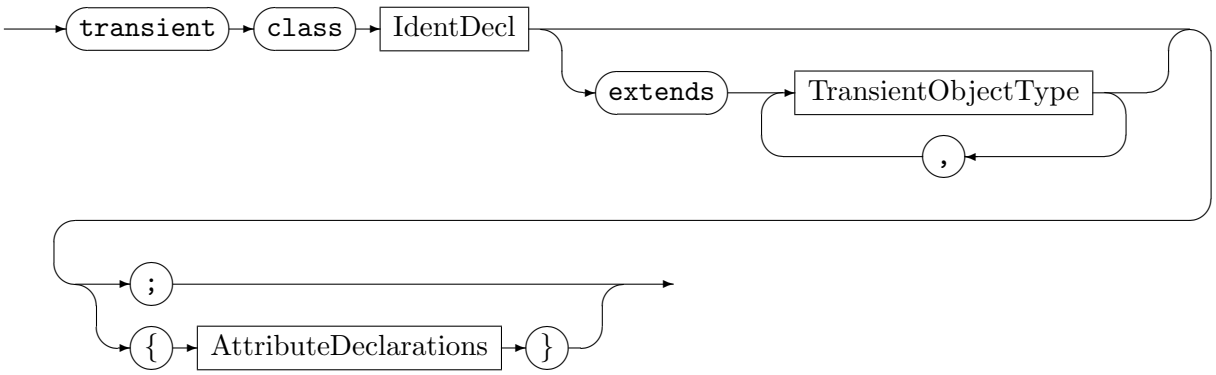
You may request the assignment of a unique id upon creation for an object of an object class with a declaration in the model (of type `long`). In case it is available, it is used in serialization and parsing of values of object types, it is displayed in the debugger, and it can be queried with the `uniqueof` function (which has the signature `uniqueof(.:ObjectType):long`, note the `long` compared to the `int` when applied to nodes, edges, and (sub)graphs). In case it is not available, an id is assigned on the fly as needed – but not stored, so another id may be assigned later on.

EXAMPLE (102)

```
1  object class unique;
2  class Bar;
3  class Foo extends Bar
4  {
5    i:int;
6    s:string;
7  }
8  node class N
9  {
10   foo:Foo;
11 }
```

18.1.2 Transient Object Types (preliminary)

TransientObjectClass



A *TransientObjectClass* definition defines a new transient object type. Transient object types can inherit from other transient object types (defined within the same file). If the `extends` clause is omitted, *TransientObjectType* will inherit from the built-in type `TransientObject`, the root type of all internal transient class objects. Optionally transient objects can possess attributes. The keywords `abstract` and `const` have the same meaning as for node and edge types, cf. 4.2.1.

In contrast to class objects, transient class objects cannot be stored in the graph, i.e. attributes of transient class type are forbidden. They are meant for processing purpose only (their semantics are geared towards that use case). Neither are transient classes assigned a lasting unique id (ids are assigned on the fly in the debugger or the shell as needed in order to support debugging). Transient objects can reference non-transient objects, though. Note that transient classes are a preliminary concept that may change in the future.

Objects and transient objects can be casted downwards and upwards in the type hierarchy (a cast to a more specific type only succeeds if the runtime type of the object is of that specific type or a subtype), the same holds for nodes and edges (this is not to be mistaken with the retype construct from Section 12.4 that also supports cross-casts and is always succeeding (potentially yielding a target type having nothing in common with the source type besides the name/unique-id))).

Figure 18.1 shows the type hierarchy of GrGen.NET.

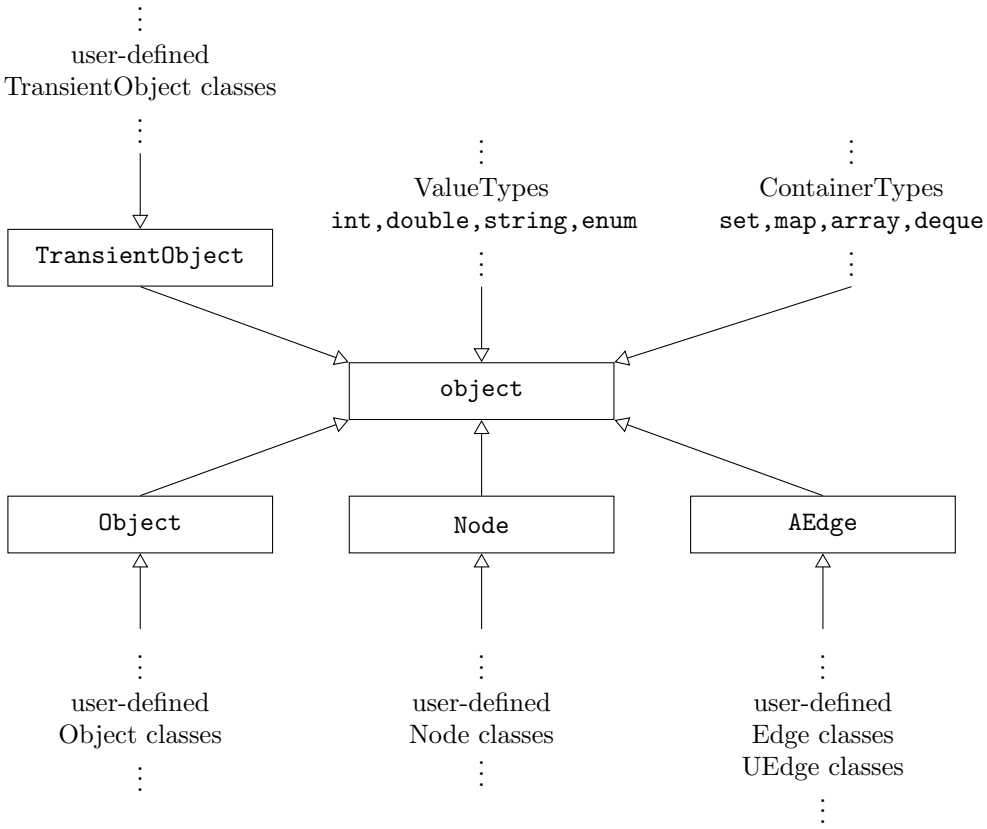


Figure 18.1: Type Hierarchy of GrGen.NET

EXAMPLE (103)

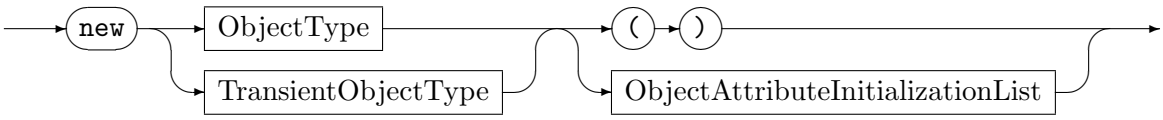
```
1 abstract transient class Baz;
2 transient class Qux extends Baz {
3   i:int;
4   foo:Foo;
5 }
```

18.1.3 Object Creation

18.1.3.1 new Operator

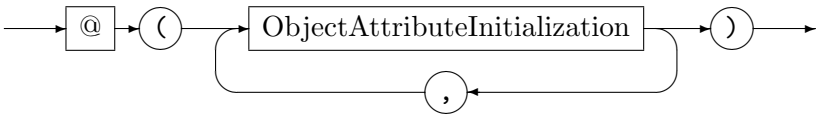
The `new` operator allows to instantiate *ObjectTypes* as well as *TransientObjectTypes*. It exists in two versions, a plain one just instantiating the (transient) object with default values for its attributes, and a version with an initializer list following an at character, which allows to explicitly initialize attributes.

NewOperator

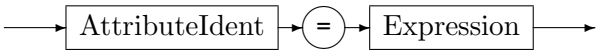


At a new operator application to a transient or non-transient object class (causing creation of the object), you can give an object attribute initialization list (similar to the attribute initialization list that can be applied to graph elements, cf. 12.8).

ObjectAttributeInitializationList



ObjectAttributeInitialization



The object initialization list is appended to the declaration following an at-Symbol, the attributes of the type are simply given by their name (i.e. *Ident*ifier). The expression must be of the type of the attribute.

Note that (transient) objects are automatically garbage collected when no reference to them exists anymore.

EXAMPLE (104)

The following example creates a new object of type `FooBar`, initializing the attribute `i` to `42` and the attribute `s` to `"Hallo Welt"`.

```
1 new FooBar@(i=42, s="Hallo␣Welt");
```

The new operator can be also used as a prefix to the container constructors, to have a syntax that is closer to the one known from the common programming languages (a container is created and a reference returned).

EXAMPLE (105)

The following example creates a new array and initializes it with the string `"Furchtbar"`.

```
1 function createArray : array<string>
2 {
3   return(new array<string>["Furchtbar"]);
4 }
```

18.1.3.2 Structure examples

Let us take a look at some more meaningful examples than the synthetic ones from the previous sections just introducing the new constructs. For one on an invasive linked list – it shows how to build linked object structures with recursively contained objects.

EXAMPLE (106)

```
class List {
  value:int;
  next:List;
}
function createList(var i:int, var j:int, var k:int) : List
{
  return(new List@(value = i, next = new List@(value = j, next = new List@(value = k, next =
      null))));
}
function sumList(ref list:List) : int
{
  if(list.next != null) {
    return(list.value + sumList(list.next));
  } else {
    return(list.value);
  }
}
```

And for the other one on a tree – it shows how to build linked object structures with objects recursively contained within a container.

EXAMPLE (107)

```
transient class Tree {
  value:string;
  children:array<Tree>;
}
function createTree() : Tree
{
  return(new Tree@(value = "FooBar", children = new array<Tree>[new Tree@(value =
      "Furchtbar", children = new array<Tree>[])]));
}
function serializePreorder(ref tree:Tree) : string
{
  def var res:string = tree.value;
  res = res + "(";
  def ref children:array<Tree> = tree.children;
  for(child:Tree in children)
  {
    res = res + serializePreorder(child);
  }
  res = res + ")";
  return(res);
}
```

You could introduce shortcuts or cycles by inserting the same object multiple times into the `Tree`, yielding a graph (without the pattern matching abilities offered by GrGen.NET). In case you intend to process such an object structure, you'd have to introduce a set storing already visited objects to the algorithm so that you don't run into an endless recursion upon visiting an already visited object again (meaning there is path from the object to itself). Or even a map of old, already visited objects to new ones, in case you are in need of a

copy algorithm that replicates such an object structure. But before you tackle the task of implementing it: such an algorithm is already offered natively and generically in the form of a pre-implemented copy function.

18.1.3.3 Copy and Clone functions

The copy and clone functions allow to create objects as deep copies or shallow clones of other objects. Both create a new object, but the copy function also creates copies of contained objects, i.e. of attributes of object class type, in contrast to the clone function which only assigns their references. So the copy function allows to copy transitively. Note that each copy has a different unique id. For the same functionality being applied to graph elements, see 12.5.

clone(Object):Object
: returns a shallow clone of the original internal class object given as argument.

copy(Object):Object
: returns a deep copy of the original internal class object given as argument.

The copy and clone functions are also available for objects of transient classes. The clone function on transient class objects differs in its handling of attributes of container types. For transient objects, only the container reference is taken over, while for non-transient objects (like for graph elements), always a clone of the container is created (in case of the copy function, always a clone of the container is created, even for transient objects, and the contained objects of transient or non-transient classes are copied).

clone(TransientObject):TransientObject
: returns a shallow clone of the original internal transient class object given as argument.

copy(TransientObject):TransientObject
: returns a deep copy of the original internal transient class object given as argument.

EXAMPLE (108)

The example shows a simplified form of the clone and copy functions, manually implemented on a List class. It highlights the differences in between the `copy` and the `clone` operator.

```
class List {
  value:int;
  next:List;
}
procedure cloneList(ref list:List) : (List)
{
  def ref listClone:List = new List(); // create new list...
  listClone.value = list.value; // ...and assign member attributes
  listClone.next = list.next; // assignment not possible in function to ensure graph is not
      changed (would be ok for transient object)
  return(listClone);
}
procedure copyList(ref list:List) : (List)
{
  (def ref listCopy:List) = cloneList(list); // clone list...
  if(list.next != null) {
    (listCopy.next) = copyList(list.next); // ...then copy object attributes deeply
  }
  return(listCopy);
}
```

Instead of implementing these procedures manually, you can simply write (and use in function context)

```
clone(list) // == cloneList(list);
copy(list) // == copyList(list);
```

18.1.3.4 Equality comparison

Objects and transient objects can be compared by the `==` and `!=` operators, as well as by the `~~` operator (preliminary). The `==` and `!=` operators check for reference identity.

The `~~` operator in contrast compares for attribute equality (ignoring identity), deeply along the nesting structure (attributes of scalar type are equal if their values are equal). It does not support objects appearing multiple times (joins in the object graph) or cycles, though (it crashes upon visiting those; it only supports tree-like structures). It also supports containers, it checks for an array or deque whether the objects at the same position are deeply equal, and for a set and map whether they are memberwise deeply equal, i.e. they contain the same amount of members, that can be mapped by a bijection to each other, with each pair being deeply equal (beware: this is an expensive operation, requiring $O(n*n)$ comparisons, with the constant being potentially very expensive because it may be a deep structure comparison, containing even further container comparisons).

`A == B`	True, iff A and B are the same object.
`A != B`	True, iff A is a different object than B.
`A ~~ B`	True, iff A is deeply equal (by-value) to B (preliminary).

Table 18.1: Compare operators on objects and transient objects

The same operators with the same semantics are supported on nodes and edges.

18.2 Methods

Classes do not only allow to structure data, but also to attach functionality. This occurs by means of method definitions. Methods define a reusable entity that comes in two different flavors, *function methods* usable(/callable) from expression context, and *procedure methods* usable(/callable) from statement context. They allow to factor out computations that are occurring frequently on attributes of classes (node, edge, object, transient object), bundling them with the class, saving code duplications.

Besides, you could hide the exact data members behind an interface defined by the methods, giving a means of data abstraction.

Furthermore, some built-in function methods and procedure methods on container types are available in rule language computations and sequence computations, cf. Chapter 15; in addition to the methods defined in Chapter 6.

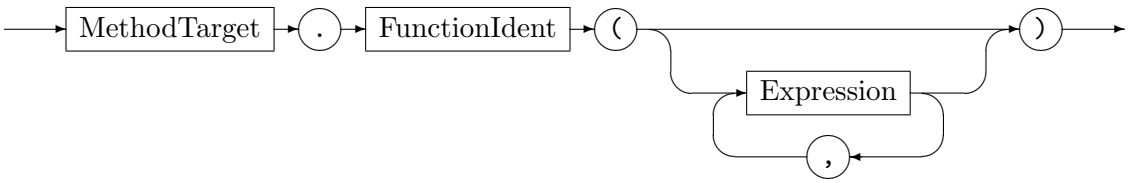
18.2.1 Function Method Definition and Call

Function methods are defined in exactly the same way as functions (cf. 14.5.1) are defined, just inside the hosting class. They have the same requirements, i.e. exactly one output value must be returned, and they must be side-effect free, which especially means that they are not allowed to change the attributes of their hosting type.

FunctionMethodDefinition



FunctionMethodCall



Such a function method may then be called as an expression atom from anywhere in the rule language file where an expression is required; or even from the sequence computations (where an expression is required). In contrast to a regular function call (cf. 14.5.1) with a method call target prefix added, separated by a dot (defining the value of the `this` reference; the call target prefix may be a variable or an attribute of an object/element, or a full expression). The built-in function methods listed in 18.2 are called with the same syntax. The syntax of function method calls was already previewed in 6.6, including the ability to chain function method calls.

Inside the function methods, the special entity `this` is available. It allows to access the attributes and methods of the type the method is contained in. In contrast to Java, C++, or C# where `this` may be used optionally to denote member or method access, the usage of `this` is mandatory in GrGen.NET in order to access the attributes of the type or in order to call the methods of the type.

18.2.2 Procedure Method Definition And Call

Procedure methods are defined in exactly the same way as procedures (cf. 14.5.2) are defined, just inside the hosting class. They may return an arbitrary number of return values, and are thus only callable as a statement. They are allowed to manipulate the graph and esp. the object/element they are contained in as needed, while being executed; so you are free to call other procedures – but in contrast to global procedures, you cannot include exec statements.

string.length():int
string.startsWith():boolean
string.endsWith():boolean
string.indexOf(string[,int]):int
string.lastIndexOf(string[,int]):int
string.substring(int[,int]):string
string.replace(int, int, string):string
string.toLower():string
string.toUpper():string
string.asArray(string):array<string>
set<T>.size():int
set<T>.empty():boolean
set<T>.peek(int):T
set<T>.asArray():array<T>
map<S,T>.size():int
map<S,T>.empty():boolean
map<S,T>.peek(int):S
map<S,T>.domain():set<S>
map<S,T>.range():set<T>
map<int,T>.asArray():array<T>
deque<T>.size():int
deque<T>.empty():boolean
deque<T>.peek([int]):T
deque<T>.indexOf(T[,int]):int
deque<T>.lastIndexOf(T[,int]):int
deque<T>.subdeque(int, int):deque<T>
deque<T>.asArray():array<T>
deque<T>.asSet():set<T>

Table 18.2: Non-array function methods at a glance

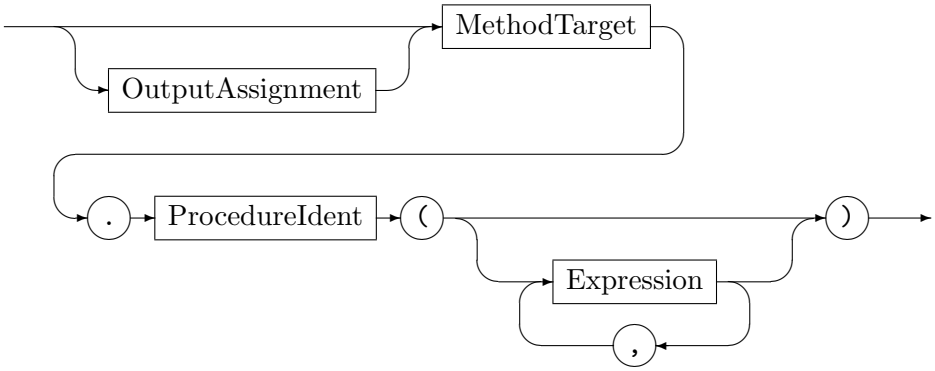
array<T>.size():int array<T>.empty():boolean array<T>.peek([int]):T array<T>.indexOf(T[,int]):int array<T>.lastIndexOf(T[,int]):int array<T>.indexOfOrdered(T):int array<T>.orderAscending():array<T> array<T>.orderDescending():array<T> array<T>.keepOneForEach():array<T> array<T>.reverse():array<T> array<T>.shuffle():array<T> array<T>.subarray(int, int):array<T> array<T>.asDeque():deque<T> array<T>.asSet():set<T> array<T>.asMap():map<int,T> array<string>.asString(string):string
array<T>.sum():T array<T>.prod():T array<T>.min():T array<T>.max():T array<T>.avg():double array<T>.med():double array<T>.medUnordered():double array<T>.var():double array<T>.dev():double array<boolean>.and():boolean array<boolean>.or():boolean
array<T>.extract<attr>():array<typeof(attr)> array<T>.orderAscendingBy<attr>():array<T> array<T>.orderDescendingBy<attr>():array<T> array<T>.keepOneForEach<attr>():array<T> array<T>.groupBy<attr>():array<T> array<T>.indexOfBy<attr>(typeof(attr)[,int]):int array<T>.lastIndexOfBy<attr>(typeof(attr)[,int]):int array<T>.indexOfOrderedBy<attr>(typeof(attr)):int
array<T>.map<entity>{[this_:array<T>;][index:int->]elem:T->lambda-expr} array<T>.removeIf{[this_:array<T>;][index:int->]elem:T->lambda-expr} array<T>.map<entity>StartWith{[this_:array<T>;]init-expr}AccumulateBy {[this_:array<T>;]accumulation-var:T,[index:int->]elem:T->lambda-expr}

Table 18.3: Array function methods at a glance

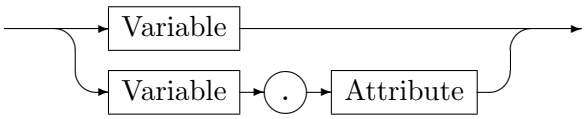
ProcedureMethodDefinition



ProcedureMethodCall



MethodTarget



Such a procedure may then be called as a statement atom from anywhere in the rule language file where an attribute evaluation statement (/computation) is required; or even from the sequence computations (where a statement is required). In contrast to a regular procedure call (cf. 14.5.2) with a method target prefix added, separated by a dot (defining the `this` value; the call target prefix may be a variable or an attribute of an object/element). The built-in procedure methods listed in 18.4 are called with the same syntax.

Inside the procedure method, the special entity `this` is available. It allows to access the attributes and methods of the type the method is contained in. In contrast to Java, C++, or C# where `this` may be used optionally to denote member or method access, the usage of `this` is mandatory in GrGen.NET in order to access the attributes of the type or in order to call the methods of the type.

`set<T>.add(T)`
`set<T>.rem(T)`
`set<T>.clear()`
`set<T>.addAll(set<T>)`
`map<S,T>.add(S,T)`
`map<S,T>.rem(S)`
`map<S,T>.clear()`
`array<T>.add(T[,int])`
`array<T>.rem([int])`
`array<T>.clear()`
`array<T>.addAll(array<T>)`
`deque<T>.add(T[,int])`
`deque<T>.rem([int])`
`deque<T>.clear()`

Table 18.4: Procedure methods at a glance

EXAMPLE (109)

In the following listing, we declare function and procedure methods inside a node class, and use them from a rule. Please note the mandatory `this` in the methods to denote members and call other methods.

```
node class N
{
   i:int;

   function get_i():int
   {
      return(this.i);
   }
   procedure set_i(var val:int)
   {
      this.i = val;
      return;
   }
   procedure inc_i(var val:int) : (int)
   {
      this.set_i( this.get_i() + val );
      return(this.i);
   }
}

rule foo : (int)
{
   n:N;
   modify {
   ---
      def var i:int;
      eval {
         n.set_i(41);
         n.inc_i(1);
         yield i = n.get_i();
      }
      return(i); // executing foo will return 42
   }
}
```

EXAMPLE (110)

The following listing highlights the effect of dynamic dispatch (the heart and soul of object-oriented programming): the method finally called depends not simply on the static type we know for sure the object is an instance of, but on the real dynamic type of the object, that it was created with. Please note that in reality you use this language device to achieve the *same behaviour* at the outside for objects of different types, but implemented *internally consistent* regarding the exact type, taking all of its members into account (including the ones unknown to parent types).

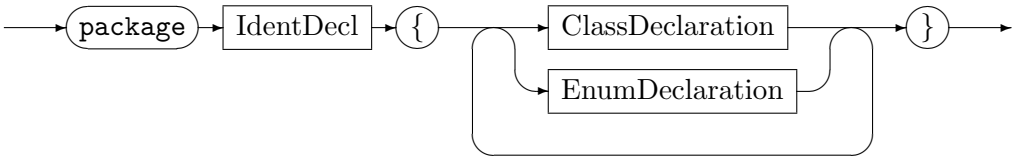
```
1  node class N
2  {
3     function get():int
4     {
5        return(0);
6     }
7  }
8
9  node class NN extends N
10 {
11    function get():int
12    {
13       return(1);
14    }
15 }
16
17 rule foo : (int)
18 {
19    n:N; // statically declared N, may match dynamically to node of type NN
20    modify {
21    ---
22       def var i:int;
23       eval {
24          yield i = n.get();
25       }
26       return(i); // executing foo will return 0 iff n was bound to node of type N, but 1 iff
             n was bound to node of type NN
27    }
28 }
```

18.3 Packages

Packages allow you to separate a project into namespaces that shield their content from name clashes when getting used (for models) or included (for actions) together. They incur some non-negligible notational overhead as every usage of a name from a package outside of that package *must* be prefixed with the package name (in the form `packagename::name`), so we recommend to use them only when needed. This holds esp. as we expect that a typical project will be limited to an algorithmic kernel written by a single programmer who has everything under control and can thus easily prevent name clashes.

18.3.1 Package Definition in the Model

PackageDefinitionModel



A package definition in the model is similar to a model definition as such, just reduced in its content. It consists of *ClassDeclaration*s, defining node or edge types, or *EnumDeclaration*s defining enum types. The *AdvancedModelDeclarations* allowed in the root model are *not* available inside a package, *neither* can other models be included inside a package with `using`. Package definitions *cannot* be nested.

PackageUsageModel



In every context where a node, edge, or enum type can be referenced, by noting down its declared *Ident*ifier, also a type from a package can be referenced, by noting down its package as a prefix. The package prefix may be omitted inside the package for other entities from the same package. But only for them, outside of that package you must always use the package prefixed form for referencing identifiers from the package.

EXAMPLE (111)

Types from a model defined like this:

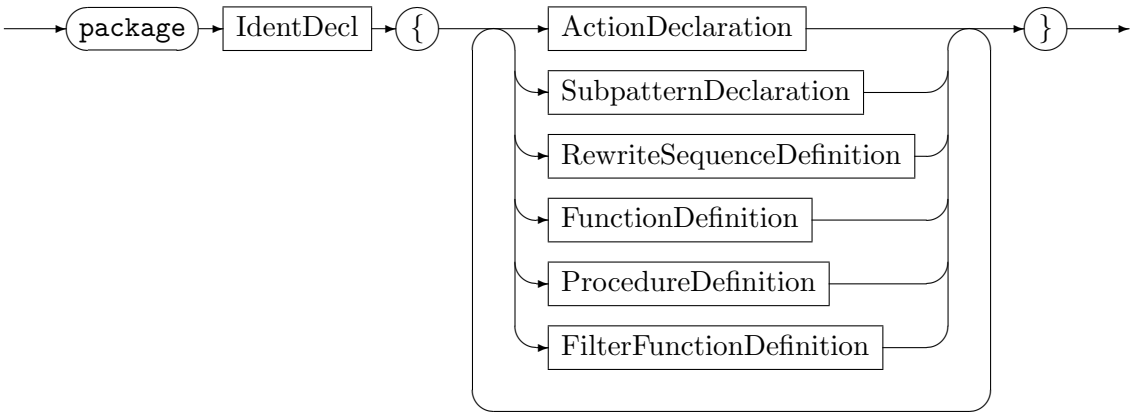
```
1  package Foo {
2     node class N {
3        attr : Lol;
4     }
5     enum Lol {
6        Bla,
7        Blub
8     }
9  }
10
11 package Bar {
12    node class N {
13       attr : Foo::Lol;
14    }
15 }
```

Can be used like this:

```
1  rule r(var inp : Foo::Lol) {
2     x : Foo::N --> y : Bar::N;
3     if{ x.attr==Foo::Lol::Bla; }
4
5     modify {
6     ---
7        eval{ add(Foo::N); }
8        eval{ y.attr = Foo::Lol::Blub; }
9     }
10 }
```

18.3.2 Package Definition in the Actions

PackageDefinitionAction



A package definition in the actions is similar to a rule set definition as such, just reduced in its content. It consists of *ActionDeclaration*s, defining rules and tests, or *SubpatternDeclaration*s to be used from the actions, or *FilterFunctionDefinition*s, for filtering the matches of the actions. It furthermore consists of *FunctionDefinition*s and *ProcedureDefinition*s, or *RewriteSequenceDefinition*s. The file header constructs for using models, including other actions, or declaring global variables are *not* available inside a package. *Neither* can package definitions be nested.

The packages `Math`, `File`, `Time`, `Transaction`, and `Debug` as well as `Synchronization` are built-in, their functions and procedures can be used directly (see 6.8, 16.3, 16.7.1 and 27.7 as well as 23.3 for their content).

PackageUsageActions



In every context where a rule, a test, a subpattern type, a filter function, a function, a procedure, or a sequence can be referenced, by noting down its declared *Ident*ifier, also an entity from a package can be referenced, by noting down its package as a prefix. The package prefix may be omitted inside the package for other entities from the same package – package content comes before global content. But only for them, outside that package you must always use the package prefixed form for referencing identifiers from the package. Inside a package you may use the `global` "package" in the form of `global::name` - it allows to access a global name from inside a package even if an entity with the same name is available locally.

Altogether, the name resolving rules are:

1. content from the package specified with the prefix, if that is "global" then global content
2. content from package context (which thus overrides global content)
3. global content

And as a refinement regarding point 2: there is no package context available for interpreted sequences (they only appear in the Shell/API outside of any packages of the rule language), so only rules 1 and 3 apply there.

The packages in the actions are declared with the same syntax as the packages in the model, but are otherwise separate. You cannot extend a model package with an actions package, or vice versa. A package with a given name can only be defined once, either in the model or the actions.

EXAMPLE (112)

The example highlights how action packages can be used.

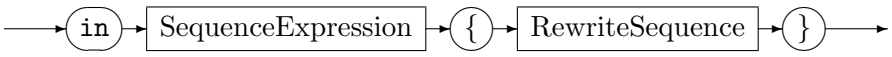
```
1  package Foo {
2     pattern P {
3        x:Node -e:Edge-> x;
4
5        modify {
6           delete(e);
7        }
8     }
9     rule r {
10       p:P();
11
12       modify {
13          p();
14       ---
15          eval { Bar::proc(true); }
16       }
17    }
18    filter f<r> {
19       return;
20    }
21 }
22
23 package Bar {
24    rule r {
25       x:Node -e:Edge-> x;
26       if{ func(); }
27
28       modify {
29          delete(e);
30       ---
31          exec(Foo::r\Foo::f);
32       }
33    } \ auto
34    function func : boolean {
35       return(true);
36    }
37    procedure proc(var b:boolean) {
38       return;
39    }
40 }
41
42 rule r2 {
43    if{ Bar::func(); }
44    p:Foo::P();
45
46    modify {
47       p();
48    }
49 }
50
51 sequence s2 {
52    Foo::r()\Foo::f ;> Bar::r\auto ;> { Bar::proc(Bar::func()); }
53 }
```

18.4 Graph References / Graph Nesting / Hierarchical Graphs

You can nest graphs by defining attributes of graph type (cf. 16) inside node or edge classes (cf. 4) at modelling time, filling them at runtime with assignments of values of graph type, created with the operations introduced in 16.3. Thus, in contrast to certain modellings identifying nested graphs with nodes, a host graph contains further graphs in the *attributes* of its nodes or edges.

Values of graph type are opaque and inaccessible to the computations that are available in – and automatically apply to – the host graph; from the outside, they can only be compared, i.e. isomorphy checked against other graphs (a feature needed for state space enumeration (cf. 24.7), in this case only immutable subgraphs are stored).

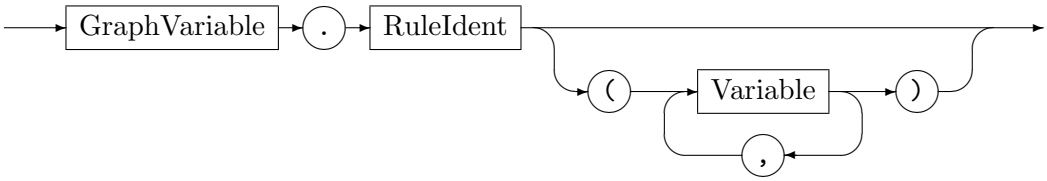
ExtendedControl



But they can be opened up by switching the location of processing with the `in g { seq }` sequence. Inside the braces, the host graph is switched to `g`, the sequence `seq` is executed in that new host graph, all queries and updates are carried out on it. After executing the construct, the old graph that was previously used is made the current host graph again.

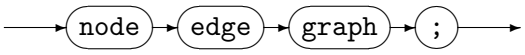
Switching the location of processing is a feature only available in the sequences, (re)defining the execution context for the rules; it is not available on lower levels, it is esp. not possible to access multiple graphs from within a single rule (a rule application is always carried out on a single by-then current host graph).

Rule



A simplified and more lightweight version of the full switch just introduced is available for single rule calls with syntax `g.r`; so a method call in the plain sequences (above the sequence computations) denotes in fact a temporary subgraph switch. This is syntactic sugar for the dedicated switch containing only a single rule application as sequence. The syntax of rule application is extended by the grammar rule above to allow for this.

ReferenceToContainingGraphDecl



If you work with multiple graphs or nested graphs, and esp. with nodes/edges being stored in variables that don't belong to the graph getting currently processed, or even being contained in member variables of a different graph, you are likely interested in fetching the (originally) containing graph of the node/edge so you can switch to it and continue processing inside it. The capability to do so can be requested with a `node edge graph;` declaration in the model. This incurs an additional cost of a 4 bytes reference to the containing graph per graph element in case of a 32bit CLR, and a cost of a 8 bytes reference to the containing graph per graph element in case of a 64bit CLR.

The reference to the containing graph may be queried from the rule language expressions or the sequence expressions with the `graphof` function:

`graphof(.)`
: returns the graph (type `graph`) containing the given node or edge.

The *information hiding* shown by the graph attributes is comparable to the information hiding shown by the objects in *object-oriented programming*, there the attributes but especially the neighbouring elements are only known to the containing object and accessible to the methods of the object. In *graph-oriented programming* the attributes but especially the neighboring elements are known to the containing graph, the connecting topology is open for *pattern matching*. This crucial difference also defines the main benefit compared to OO, removing it would mean to revert back to OO. But this openness might not be needed always for all parts. When building a *large system*, you typically only need a certain *layer* to be accessible at a time. You may use graph attributes and nested graph in this case, utilizing open graph-oriented programming for the parts you need to work globally with pattern matching at a time, and closed object-oriented programming for parts you only need to access locally, decoupled by explicit move-to-subgraph and return-to-subgraph steps.

You may model containment with edges denoting a containment type pointing to the contained parts instead of attributes of subgraph type when the pattern matcher needs overall access to the graph, but there are still some containment or nesting relationships in place. You can then still benefit from a hierarchical structure in debugging, utilizing the built-in nesting for visualization capabilities of GrGen.NET(cf. 27.1, the graphs nested in attributes are truly opaque and invisible, only when processing switches to them are they displayed instead of their containing host graph).

CHAPTER 19

SEQUENCE COMPUTATIONS AND CONTAINER TYPE

In this chapter we take a look at sequence computations, which are not concerned with directly controlling rules, but with computing values (sequence expressions) or causing side effects (sequence statements), that are then used to control rules.

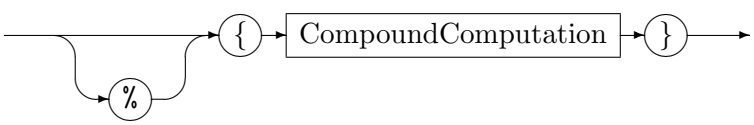
EXAMPLE (113)

Sequence computations are typically employed when storages have to be maintained: `now:set<Node>=set<Node>{};>next:set<Node>=set<Node>{};>initializeNow(now) ;> ( processNowFillNext(now, next) ;> { now.clear(); tmp:set<Node>=now; now=next; next=tmp; {!now.empty()} } )*` — that example sequence is used to implement a wavefront as rule control strategy.
A set `now` of current nodes is processed, filling a set of output nodes to be processed `next`, which are then used in the following iteration step as input nodes, until all parts of the graph reachable from the initial nodes have been passed, yielding an empty set. The sequence computation for switching in between the sets is given enclosed in braces in the sequence, the sequence expression to determine whether the wavefront came to a halt is given at the end of the computation, enclosed a level deeper in braces. As long as the set is still filled, the final expression, thus the computation, returns `true`, continuing with the loop – when it gets empty, the expression yields `false`, henceforth terminating the loop and the wavefront algorithm.

Sequences exist to control rules and not to carry out the bulk of the computational work (in spite of that, nearly all rule computations are available in the sequence computations). But the sequence computations are embedded directly in the controlling sequence, in contrast to the computations from the rule language that can only be used by calling their named abstractions.

19.1 Sequence Statements

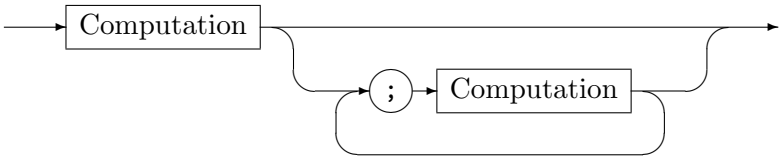
RewriteComputationUsage



The non-computational sequence constructs introduced before in 11 are used for executing rules, to determine which rule to execute next depending on success and failure of the previous rule applications, and where to apply it next by transmitting atomic variables of node or edge type in between the rules. Sequence computations in contrast are used for manipulating container variables, evaluating computational expressions, or for causing side effects like output or element markings. A computation always returns true, with exception

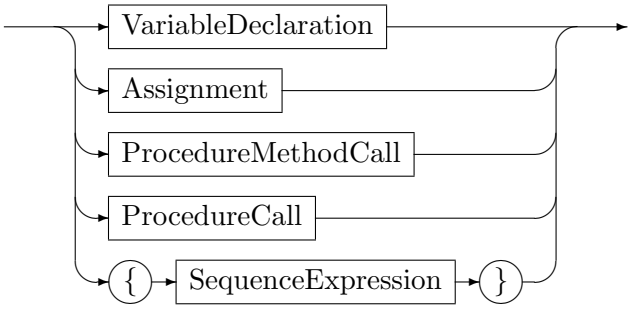
of an expression used as a computation (explained below). A prepended **%** attaches a break point to the computation.

CompoundComputation

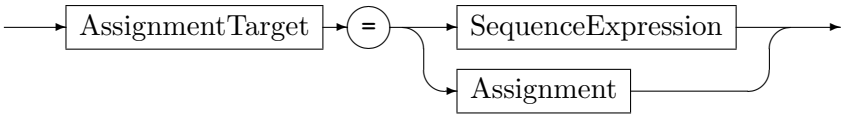


A compound computation consists of a computation followed by an optional list of computations separated by semicolons. The computations are executed from left to right; the value of the compound computation is the value of the last computation. So you must give an expression at that point in order to return a value, whereas it is pointless to specify an expression before. (But typically, when you are interested in a return value in a certain context, you only give a single computation, being an embraced sequence expression, which is then automatically the last computation.)

Computation



Assignment



A variable declaration declares a local variable in the same way as in the sequences. An assignment assigns the value of a sequence expression to an assignment target. It may be chained; such an assignment chain is executed from right to left, assigning the rightmost value to all the assignment targets given. The expression used as computation – enclosed in braces – will return a boolean value by comparing the return value of the expression to the default value of the corresponding type (e.g. 0 for an integer), returning false if equal, or true if not equal. So just using a boolean variable as expression returns the value of the variable. The form of expressions and assignment targets will be specified below.

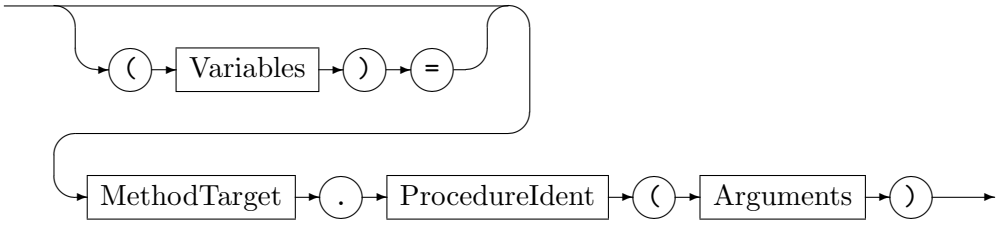
NOTE (35)

For returning an expression evaluation result value to a sequence you need two opening braces! One for entering the sequence computations, and the other for entering the sequence expressions.

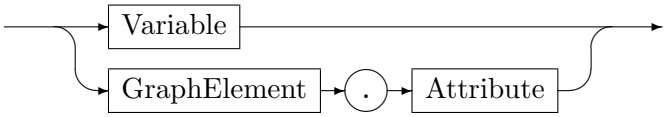
EXAMPLE (114)

The example sequence `b:boolean ;> {b=true} ;> {{b}}` shows a sequence with a variable declaration on sequence level, an assignment on sequence computation level (to initialize the variable), and a sequence expression defining the return value of the sequence to be the value of the variable.
The example sequence `if{ {{::i<10}}; {::i=::i+1}; false }*` increments the graph global variable `i` until it becomes greater than `10`.

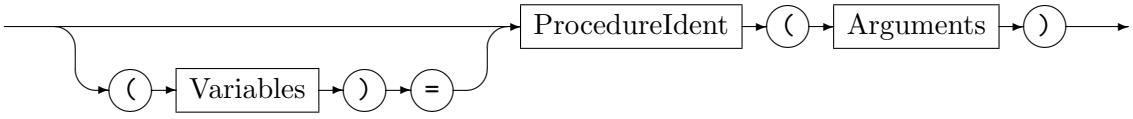
ProcedureMethodCall



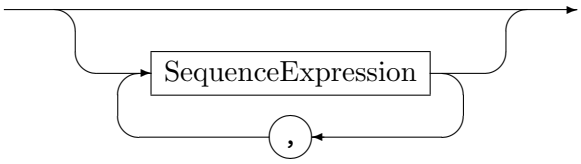
MethodTarget



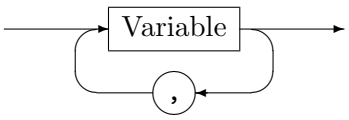
ProcedureCall



Arguments



Variables



A method call executes a method on a variable, passing further arguments. The method may be one of the predefined container methods, or a user-defined method (cf. 18.2).

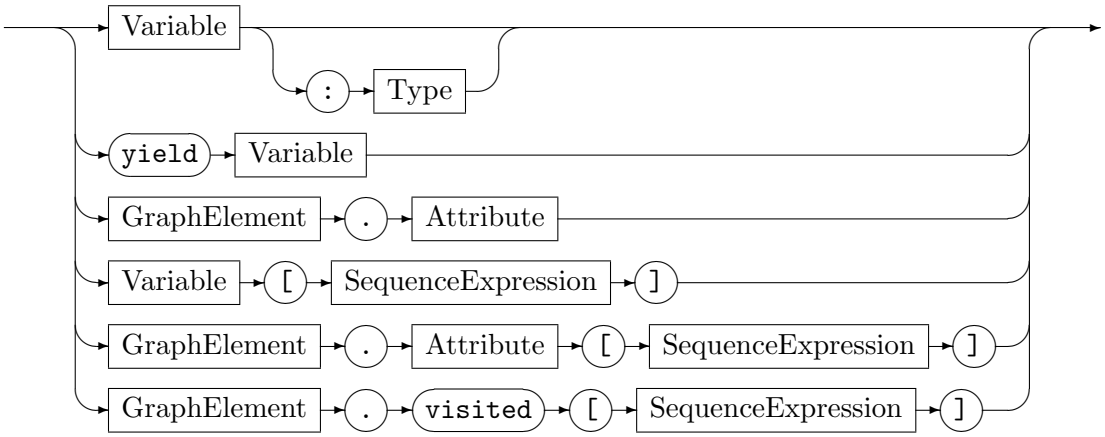
A procedure call executes a user defined procedure from the rules file, cf. 14.5, or a built-in procedure, passing further arguments. The built-in procedures you may call here are e.g. `emit`, `emitdebug`, `record`, or `canonize(g:graph):string`, you find more on them in Chapter 16.

Also procedures from packages may be called, the name *ProcedureIdent* is then prefixed by the *Ident* of the package, separated by a double colon `::`. A user-defined procedure from a user-defined package may be called, or one of the built-in procedures contained in the built-in package `File`, e.g. `export`, or in the built-in package `Transaction`, cf. Chapter 16, or in the built-in package `Debug` (for more on packages see 18.3.2).

The `Debug` package procedures, as explained in detail in 27.7, are of interest in case sequences contained in rule execs are used, these are not directly visible to the debugger. You call then `Debug::add` when entering and `Debug::rem` when leaving a subrule computation of interest (always pairwise!), `Debug::emit` to record some subrule computation milestones, `Debug::halt` to halt the debugger, and `Debug::highlight` to highlight some graph elements in the debugger (halting it).

Finally, an expression (without side effects) can be evaluated, this allows to return a (boolean) value from a computation.

AssignmentTarget



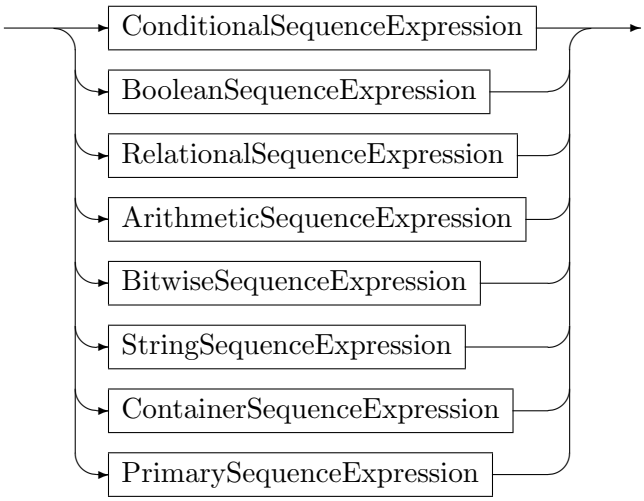
Possible targets of assignments are the variables and def-variables to be yielded to, as in the simple assignments of the sequences. A `yield` assignment writes the rhs variable value to the lhs variable which must be declared as a def-to-be-yielded-to variable (`def`-prefix) in the pattern containing the `exec` statement. Yielding is only possible from compiled sequences, it always succeeds. Furthermore, the attributes of graph elements or internal class objects or internal transient class objects may be written to, the values at given positions of array or deque or map variables may be written to, and the visited status of graph elements may be changed.

EXAMPLE (115)

The sequence computation `{ x:int=42; y:N; (y)=proc(x); y.meth(x); y.a=x }` shows a variable declaration including an initialization (which falls out of scope at the closing brace), a variable declaration of node type without initialization, a call of a procedure with one input argument, assigning an output value, a call of a method of the node type, and the assignment of a graph element attribute.
The example `{ x:array<int>=array<int>[]; x.add(42); x[0]=1; {x.size()>0} }` shows an array declaration and initialization, the adding of a value to the array, the indexed assignment to an array, and a terminal sequence expressions that makes the sequence computation succeed if the array is not empty (that's the case here) and fail otherwise. Sequence computations that don't end with a sequence expression always succeed.

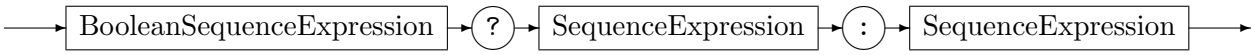
19.2 Sequence Expression

SequenceExpression



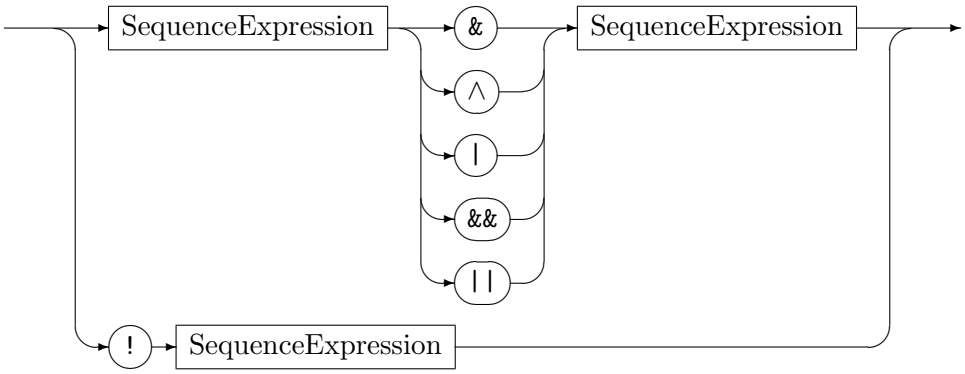
The sequence expressions support the same functionality as the expressions introduced in 6.2, containing the boolean expressions, the relational (comparison) expressions, the arithmetic expressions, the bitwise expressions, plus the string expressions. The container sequence expressions are the same as the container expressions introduced in 15, they will be explained in 19.3.

ConditionalSequenceExpression



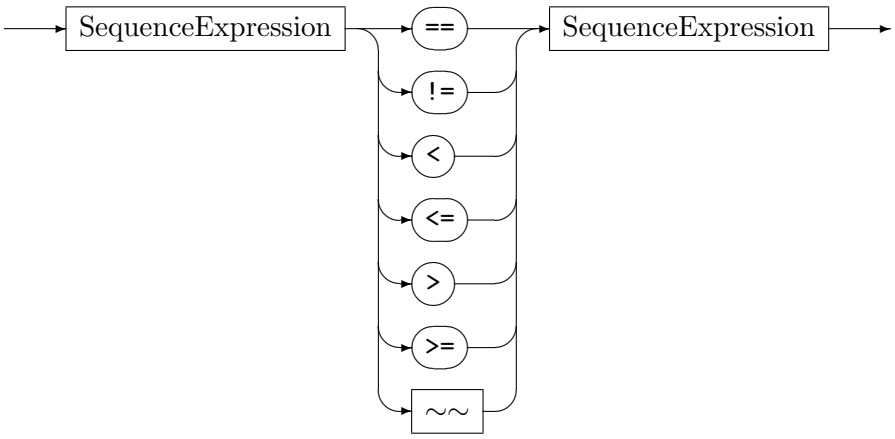
The conditional operator has lowest priority, if the condition evaluates to true the first expression is evaluated and returned, otherwise the second.

BooleanSequenceExpression



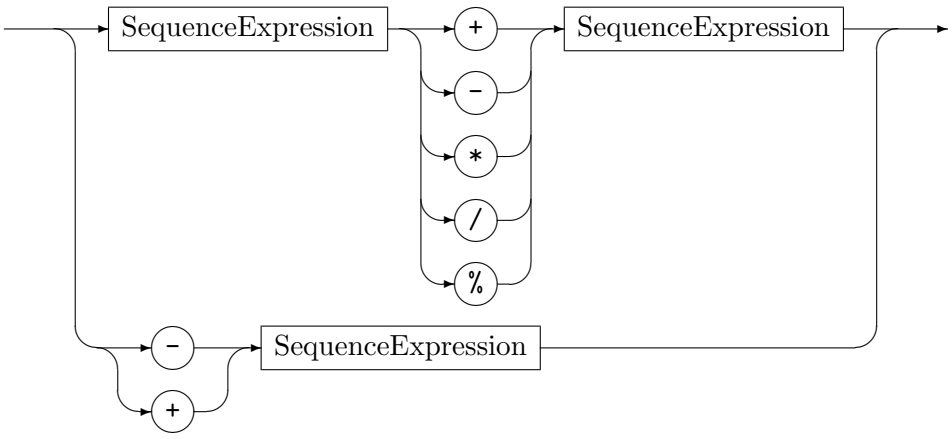
The boolean operators have the same semantics and same priority as in 6.2.

RelationalSequenceExpression



The equality operators work for every type and return whether the values to compare are equal or unequal. The relational operators work as specified in Section 6.2 for numerical types and string type, Chapter 18 for object types (as well as graph element types), and Chapter 16 for graph type.

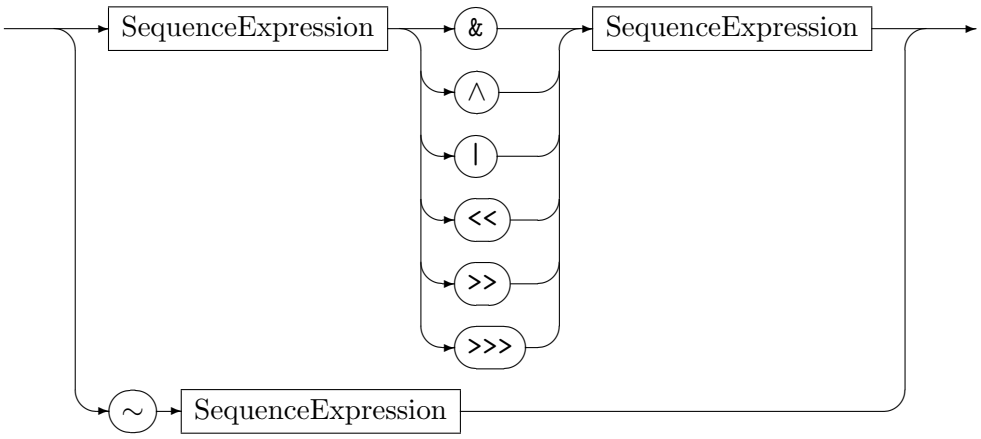
ArithmeticSequenceExpression



The arithmetic operator plus is used to denote addition of numerical values or string concatenation, the arithmetic operator minus is used to denote subtraction of numerical values. Furthermore, you can multiply and divide numbers, or compute the remainder of a division. Also, the arithmetic functions of package `Math` are available.

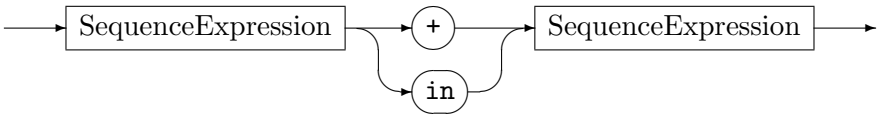
Even though everything is available in the sequence computations for carrying out computations, we recommend using the rule language for real computational work, you can employ the entities from the rule language easily, just call the functions from the sequence expressions, the procedures from the sequence statements, and the rules/tests from the sequences.

BitwiseSequenceExpression



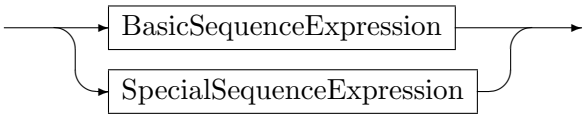
The bitwise operators have the same semantics and same priority as in 6.2.

StringSequenceExpression



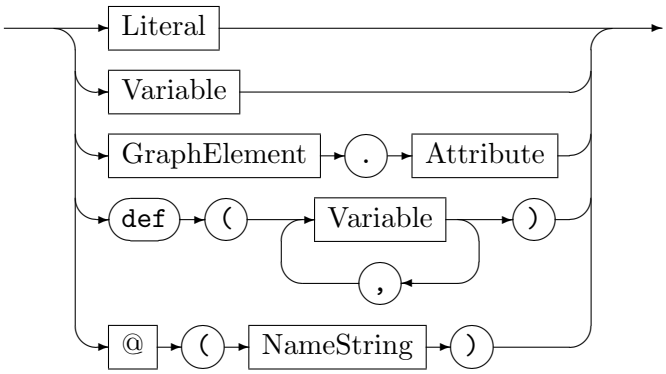
The string operators have the same semantics and same priority as in 6.2. In addition, the string methods are also available in the sequence expressions.

PrimarySequenceExpression



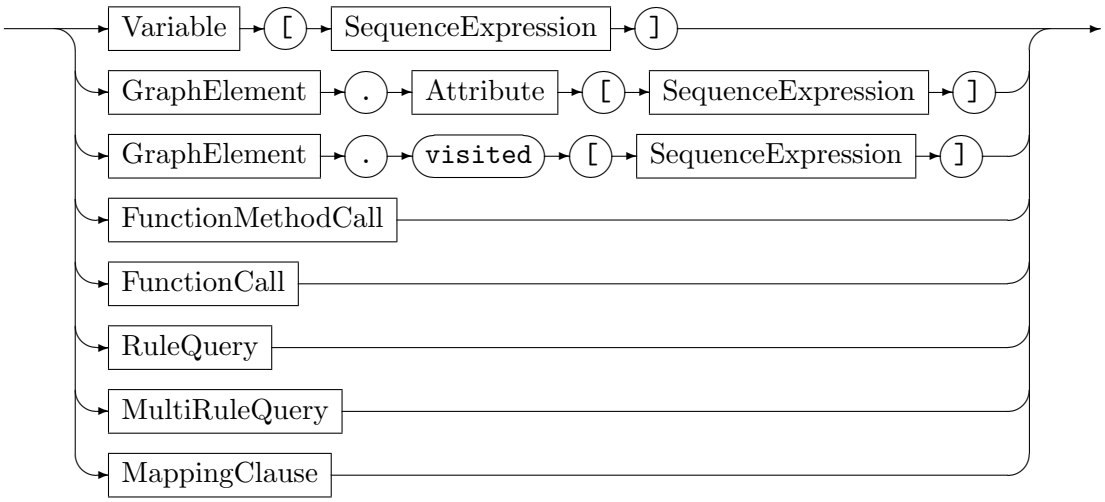
The atoms of the sequence expressions are the basic and the special sequence expressions.

BasicSequenceExpression



The basic sequence expressions are the foundational value sources. The literal and variable basic expressions are the same as in the *SimpleOrInteractiveExpression*, cf. 11.3, or as in the *PrimaryExpressions* of the rule language, cf. 6.8, this means esp. that a Variable may denote a graph global variable if prefixed with a double colon, here as well as in the *AssignmentTarget*. The attribute access clause returns the attribute value of the given graph element. A `def` term is successful iff all the variables are defined (not null). The at operator allows to access a graph element by its persistent name.

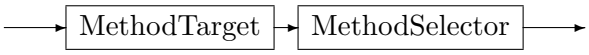
SpecialSequenceExpression



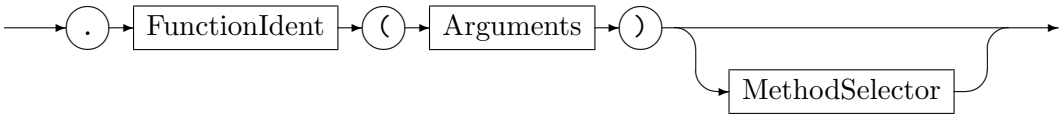
The special sequence expressions consist of indexed accesses, function method calls, function calls, and rule queries.

They are esp. used for storage handling, and for graph, subgraph and visited flag handling. The storage oriented ones are used to access a storage or to call a method on a storage. They were introduced in chapter 15, and are summarized below in 19.3. The graph and subgraph handling expressions allow to query the graph for its elements, the visited flags expressions allow to check whether a value is marked. They were introduced in chapter 16, and are summarized below in 19.4.

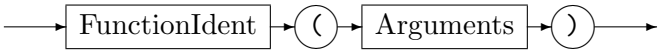
FunctionMethodCall



MethodSelector



FunctionCall



A method call executes a method on a variable, passing further arguments. The method may be one of the predefined container methods (or a string method), or a user-defined method (cf. 18.2). Method calls work as in the rule language, i.e. you can build chains of function method calls, like with the *MethodSelector* from the rule language, cf. Chapter 6 (in contrast to procedure method calls).

A function call executes a user defined function from the rules file, cf. 14.5, or a built-in function, passing further arguments.

An available built-in function you may call here is e.g. `random` that behaves like the random function from the expressions, see 6.8.

Further available are `nameof` to fetch the name of a node or edge or graph and `uniqueof` to fetch the unique id of a node or edge or graph. The function `nodeByName` allows to retrieve a node by its name, `edgeByName` does the same for an edge. The function `nodeByUnique` allows to retrieve a node by its unique id, `edgeByUnique` does the same for an edge. You find more on them in Chapter 31.

The `typeof` function returns the type *as a string*, for an arbitrary GrGen.NET-object fed as input. There is no type type supporting type comparisons as in the rule language, you are limited to string comparisons.

Also functions from packages may be called, the name *FunctionIdent* is then prefixed by the *Ident* of the package, separated by a double colon `::`. A user-defined function from a user-defined package may be called, or one of the built-in functions contained in the built-in package `File`, e.g. `import`, or `now` from the built-in package `Time`, cf. Chapter 16 (or one of the functions from package `Math` — for more on packages see 18.3.2).

Regarding the rule queries, take a look at Chapter 20, regarding the mapping clause, inspect Section 22.5.

EXAMPLE (116)

The sequence expression `{{ def(y) && y.meth(x)>=y.a || @("$1").a+1 != func(x) }}` checks whether the node typed variable `y` is defined, i.e. not null, and if so compares the return value of a method of that type with an attribute value. If the comparison succeeds, it defines the return value of the expression; in case the definedness check failed or the comparison yields false, is a graph element fetched from the graph by its persistent name `$1`, and its attribute `a` plus 1 compared against the result of a function call. Then the result of this latter comparison defines the outcome of the expression.

19.3 Storages – Container Type Handling in the Sequences

Storages are variables of container (set/map/array/deque) type (cf. 16.1) storing nodes or edges. They are primarily used in the sequences, from where they are handed in to the rules via `ref` parameters (but additionally container attributes in graph elements may be used as storages, esp. for carrying out data flow analyses, cf. 24.6). They allow to decouple processing phases: the first run collects all graph elements relevant for the second run which consists of a sequence executed for each graph element in the container. The splitting of transformations into passes mediated by container valued global variables allows for subgraph copying without model pollution, cf. 24.5; please have a look at 24, 24.5 and 24.6 regarding a discussion on when to use which transformation combinators and for storage examples. They were already defined and described in 15. Here we only give some refinements and explanations of the semantics.

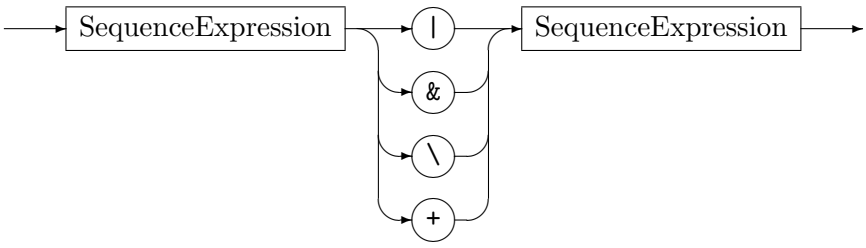
Common methods and operators

The update methods `add`, `rem`, and `clear` are available for all storages and allow to add elements to the container, remove elements from the container, or clear the container. Their return value is the changed container, thus they allow to chain method calls on the container.

The query methods `size`, `empty`, `peek` in contrast return the size of the container, whether the container is empty, or a certain element from the container, and thus can't be chained.

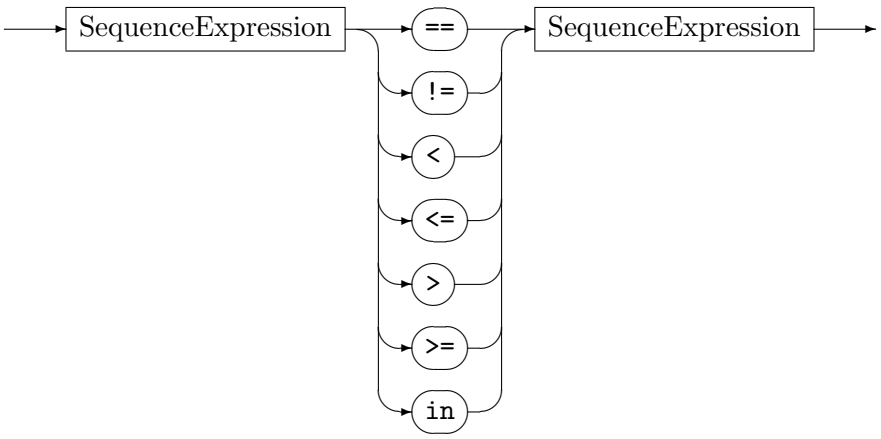
The indexed query `v=m[k]` is available on map and array and deque types and returns the element at the specified index, the indexed assignment `a[i]=v` overwrites the element at the specified index.

ContainerSequenceExpression



All infix container operators are available in the sequence expressions, they have the same semantics and same priority as in Chapter 15.

RelationalSequenceExpression



The relational (including equality) operators work as specified in Chapter 15 for container types. Further available is the sequence expression operator `in` for membership query.

All container constructors and copy constructors are supported, see Chapter 15 for more on them. The change assignments known from rule language in contrast are *not* available in the sequence computations (but this is the only exception).

Further support is discussed in the following separated by type (in a nutshell: everything that is available in the rule language is offered).

Array Operations

All array methods are supported (the array of match is the most important data structure in the sequence expressions, it is delivered by pattern-based graph queries), see Section 15.4 and Section 20.1 for a list of them. Also all array methods specifying and accessing an attribute/member of the objects stored in the array are supported (this esp. includes the ordering methods, note that further means of ordering are available as filter calls).

All infix operators are supported, which are array concatenation, as well as the relational operators.

Set Operations

All set methods are supported, this is in addition to the ones already mentioned in the section on the common parts of all container types the `asArray()` (cf. 15.2) method.

All infix operators are supported, which are set union, intersection, and difference, as well as the relational operators.

Map Operations

All map methods are supported, these are in addition to the ones already mentioned in the section on the common parts of all container types: `domain()`, `range()`, and `asArray()` (cf. 15.3).

All infix operators are supported, which are map union, intersection, and difference (including sets as right value), as well as the relational operators.

Deque Operations

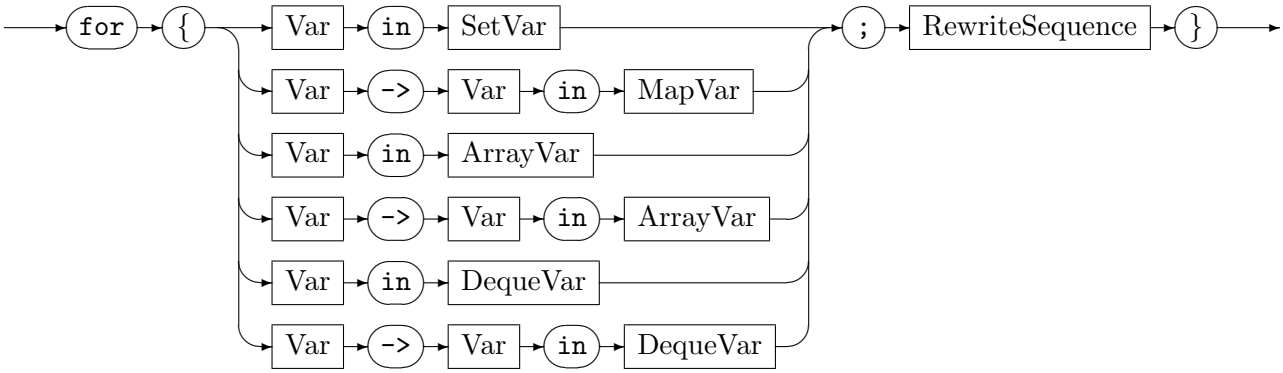
All deque methods are supported, these are in addition to the ones already mentioned in the section on the common parts of all container types: `subdeque()`, `asArray()`, `asSet()`, `indexOf(.)`, `indexOf(.,.)`, `lastIndexOf(.)`, `lastIndexOf(.,.)` (cf. 15.5).

All infix operators are supported, which are deque concatenation, as well as the relational operators.

Container For Loop

You may iterate in the sequences with a for loop over the elements contained in a storage.

RewriteFactor



The `for` command iterates over all elements in the set or array or deque, or all key-value pairs in the map or array or deque, and executes for each element / key-value pair the nested graph rewrite sequence; it completes successfully iff all sequences were executed successfully (an empty container causes immediate successful completion); the key in the key-value pair iteration of an array or deque is the integer typed index. (See 22.4 for another version of the `for` command.)

EXAMPLE (117)

The following XGRS is a typical storage usage. First an empty set `x` is created, which gets populated by a rule `t` executed iteratedly, returning a node which is written to the set. Then another rule is executed iterated for every member of the set doing the main work, and finally the set gets cleared to prevent memory leaks or later mistakes. If the graph should stay untouched during set filling you may need `visited` flags to prevent endless looping.

```
x=set<Node>{} ;> ( (v)=t() && {x.add(v)} )+ && for{v in x; r(v)} <; {x.clear()}
```

You could hand in the storage to the rule, and `add` there to the set, this would allow to shorten the sequence to:

```
x=set<Node>{} ;> ( t(x) )+ && for{v in x; r(v)} <; {x.clear()}
```

The for loop could be replaced by employing the storage access in the rule construct, cf. 15.6; this would be especially beneficial if the rule `r` inside the for loop would have to change the storage `x`, which would corrupt the iteration/enumeration variable.

NOTE (36)

The container over which the `for` loop iterates must stay untouched during iteration. Use the `clone(.:container):container` function to clone the container before the iteration if you need to iterate the container content *and* change its values.

19.4 Graph and Subgraph Based Queries and Updates

The graph and subgraph oriented parts of the sequence expressions are built from four groups:

1. the procedures for basic graph manipulation,
2. the functions for querying the graph structure,
3. the functions and procedures of the subgraph operations, and
4. the visited flag query, assignment, and procedures.

The first group is built from basic graph manipulation operators, as defined in 14.4 and described in 14.5.2. Elements may be added, removed, or retyped, and nodes may be merged or edges redirected. Not available are numerical functions, they are only offered by the computations of the rules.

EXAMPLE (118)

The sequence computation `{ (::x)=add(N); (::x)=retype(::x,M); rem(::x) }` adds a newly created node of type `N` to the graph (storing it in the global variable `::x`), retypes it to `M`, and finally removes it again from the graph.

The second group is built from the operators querying primarily the connectedness of graph elements, as defined in 14.5.1. You may ask for one for the nodes or edges of a type. You may query for the other for the source or target or opposite node of an edge. Furthermore, you may query for adjacent nodes and incident edges, maybe even transitively for the reachability. Furthermore you may ask with a predicate whether nodes or edges are adjacent or incident to other nodes or edges, maybe even transitively for reachability.

These global functions returning elementary data define the simple way of graph querying – the more powerful way is defined by pattern-based queries (employing rules/tests) – they return an array of pattern matches, see Chapter 20.

EXAMPLE (119)

```
for{x:N in nodes(N); for{::y in outgoing(x); { ::z=target(::y); ::z.a = 42 } } }
```

is a sequence that sets the attribute `a` to 42 for all nodes that are adjacent as targets to a source node `x` of type `N`. You will receive a runtime exception if the type of `::z` does not possess an attribute `a`.

A more realistic example is to check whether two nodes returned by some rule applications are reachable from each other, carrying out a change only in this case:

```
(::x)=r() ;> (::y)=s() ;> if{ {{isReachable(::x,::y)}} ; doSomething(::x, ::y) }
```

The third group is defined by functions and procedures that operate on (sub-)graphs, as defined in 14.5.1 and 14.5.2. They are especially useful in state space enumeration, cf. 24.7.

To this end, parallelized graph isomorphy checking with the `equalsAny` and `getEquivalent` functions is especially of interest. You may import, clone, or compute induced subgraphs. You may export a subgraph or insert a subgraph into the hostgraph.

EXAMPLE (120)

`( doSomething() ;> { File::export("graph"+i+".grs") } )*` is a sequence scheme for exporting a graph after each iteration step in a loop, gaining a series of snapshots on the hard drive. In a later step, you may then conditionally add exported graphs to the host graph:
`if{cond; { ::g=File::import("graph"+n.a+".grs"); insert(::g) } }`

EXAMPLE (121)

When you model a state space with `Graph` representative nodes (*not* `graph` standing for a real (sub)graph), which are pointing with `contains` edges to the nodes contained in their state (i.e. subgraph), and store additionally a replica of the subgraph in a `sub` attribute of the `Graph` node, so it is readily available for comparisons, then the step of a state space enumeration with isomorphic state pruning is controlled with code like this:
`<< modifyCurrent(gr) ;; {adj=adjacent(gr, contains); sub=inducedSubgraph(adj)}`
Inside the backtracking double angles, a new state is computed as first step by modifying the currently focused state received as input `gr:Graph` from the previous step. The modified subgraph is extracted for comparison by computing the `inducedSubgraph` from the nodes `adjacent` via `contains`-edges to the `gr`-node.
`;> for{others:Graph in nodes(Graph); {{sub!=others.sub}} } &&`
The extracted subgraph is compared with all already enumerated subgraphs that can be accessed by their `Graph` representative node. Only if none is isomorphic to it, do we continue with making the state persistent.
`/ {(ngr)=insertInduced(adj, gr)} && link(gr,ngr) && {ngr.sub=sub} /`
During a backtracking pause, the modified subgraph is cloned and inserted flatly into the host graph again with `insertInduced`. A link is added from the old representative to this new representative, to reflect ancestry. Then the subgraph attribute of the new representative `ngr` is filled with the previously computed subgraph `sub`. Remark: the first `inducedSubgraph` above does not contain the representative node and thus is missing all containment edges, too. This `insertInduced` includes the representative node and thus the containment edges. Syntactical remark: `inducedSubgraph` is used in an assignment with a function call as RHS, whereas `insertInduced` is employed from a procedure call which requires parenthesis around the output arguments.
`&& stateStep(ngr, level+1) >>`
Finally, we continue state space construction with the next step, modifying the just inserted subgraph. After this step returns (with `false` as result), do the backtracking double angles roll back the modification – keeping the changes written during the pause untouched – and execute `modifyCurrent` on the next match available in `gr`.

The fourth group are the visited flags related operations, as described in chapter 16.6. Available is an expression for reading a visited flag, an assignment for writing a visited flag, and procedures for managing the visited flags as defined in 14.4.

EXAMPLE (122)

Because of the need to allocate and deallocate them, the visited flags are typically used with code like this: `flag:int ;> {(flag)=valloc()} ;> r(flag) ;> {vfree(flag)}`
In addition, they may be read in the sequence expressions, and written in the sequence computations: `if{ {{!n.visited[flag]}} ; { n.visited[flag] = true } }`

In the sequences only the sequence expressions are available to compute the parameters for the functions and procedures, compared to the full-fledged expressions of the computations language.

19.5 Quick Reference Table

Table 19.1 lists most of the operations of the graph rewrite computations at a glance.

`c;d`	Computes c then d; the value of the computation is d
`{e}`	An unspecified sequence expression executed for its result, defining the success of the computation.
`t=e`	Simple assignment of an expression value to an assignment target
`e ? f : g`	Returns f if e evaluates to true, otherwise g
`e op f`	For `op` being one of the boolean operators `||,|,&,&&,^`
`e op f`	For `op` being one of comparison operators `==,!=,<,<=,>,>=,in`
`e op f`	For `op` being one of arithmetic operators `+,-,*,/,%`
`e op f`	For `op` being one of the bitwise operators `&,^,|,<<,>>,>>>`
`e op f`	For `op` being one of the set or map operators `|,&,\`
`e + f`	For array or deque or string concatenation.
`op e`	For `op` being boolean `!` or arithmetic `-,+` or bitwise `~`
`[?r\f]`	Rule query, returns every match produced by `r` that was left by the filter `f`, in the order left behind by the filter, as `array<match<r>>`.
`[?[q,r]\mc.f\<mc>]`	Multi rule query, returns every match produced by `q` and `r` that was left by the filter `f` of match class `mc`, in the order left behind by the match class filter, as `array<match<class mc>>`.
`v`	Variable. Assignment target or expression.
`v.name`	Attribute of graph element or internal (transient) class object. Assignment target or expression.
`@(name)`	Return graph element of given name.
`def(`*`Parameters`*`)`	Check if all the variables are defined.
`random(upperBound)`	Returns random number from [0;upper bound[, if upper bound is missing from [0.0;1.0[.
`typeof(v)`	Returns the name of the type of the entity handed in.
`u=set<Node>{}`	Example for container constructor, creates storage set and assigns to `u`.
`u[e]`	Target value of `e` in `u`. Fails if `!(e in u)`. Assignment target or expression.
`f(...)`	Calls one of the functions for graph querying defined in 14.1 and explained in 16. Or calls a user defined function. Also the mathematical functions may be called.
`(...)=p(...)`	Calls one of the procedures for graph manipulation defined in 14.4 and explained in 16. Or calls a user defined procedure.
`v.fm(...)`	Calls esp. one of the function methods `size,empty,peek` for container querying defined in 18.2 and explained in 15. Or calls a user defined function. Also the other container function methods as well as the string function methods are available.
`(...)=v.pm(...)`	Calls esp. one of the procedure methods `add,rem,clear` for container manipulation defined in 18.4 and explained in 15. Or calls a user defined procedure. The change assignments are *not* available.

Let `c` and `d` be computations, `t` be an assignment target, `e`, `f`, `g` be expressions, `u`, `v`, `w` be variable identifiers, `q` and `r` be rules, `mc` be a match class, `f` be a filter

Table 19.1: Sequence computations at a glance

CHAPTER 20

GRAPH QUERIES

In this chapter we take a look at pattern queries – the most powerful and flexible way of querying the graph. They are more powerful than the simple global functions returning elementary data (described in Section 19.4), and more flexible than the returning of preselected values from rules or tests (described in Chapter 5).

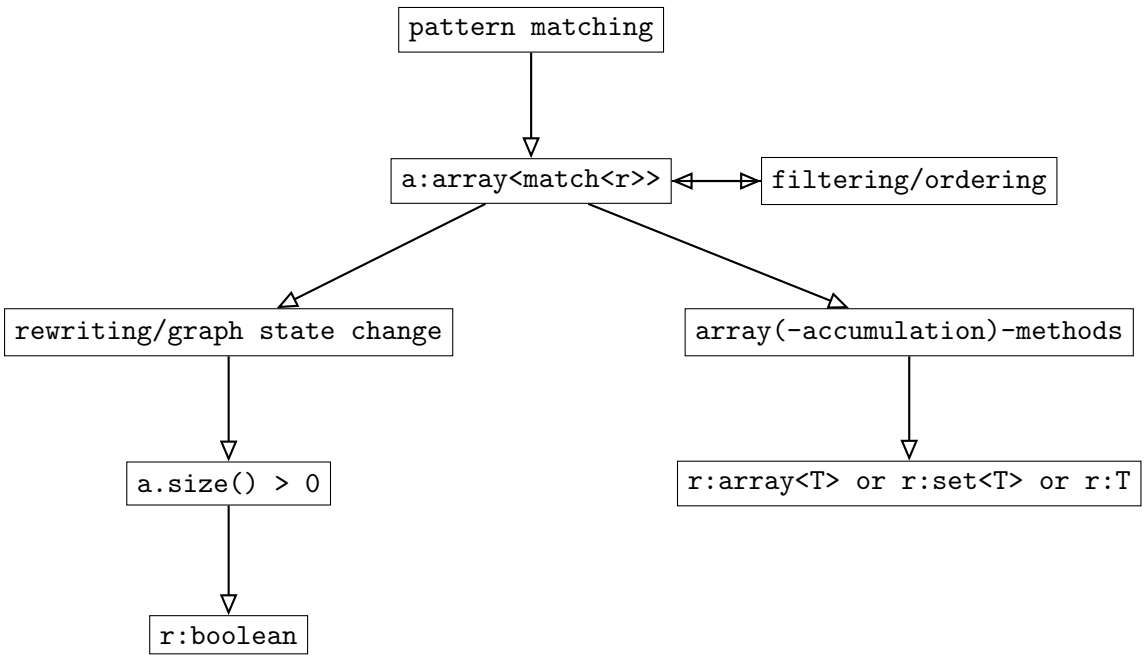
Pattern queries employing tests or rules as tests in order to obtain some result value define the second way of pattern usage besides the already known one employing rules to carry out host graph state changes.

Query Workflow Compared to Transformation Workflow

The heart and soul of GrGen.NET is pattern matching – it is utilized from two different workflows. Both start the same way, the pattern is matched, yielding an *array* of *match* objects, which gets further processed by *filters*.

rule application
: The resulting matches are used to apply the rewrite part of the rule on the resulting matches, causing a graph state change, ending in a boolean value telling about rule application success (this can be seen as a built-in automatically applied reduction)

querying
: The resulting matches are fed into post-matches-accumulation methods, that distill a scalar value out of them, or an array or set of interesting values; the matches may be also reported directly instead (no graph state change is carried out when working with queries).



In order to manipulate a graph you need rules, for querying you can use rules as well as tests, but the more natural partner for querying are tests (coming without the specification of a graph change). The advanced array operations employed in the queries allow to flexibly pick at the call site (outside of the rule/test) the values of interest as well as the accumulation method, instead of having to encode them in the rule, and allow to accumulate the data from multiple matches (typically obtained from different spots) directly, instead of having to work with some temporary accumulation variables (but only in case a method suiting your needs is available). The same holds for iterated queries inside a rule (allowing to accumulate the data from multiple matches of an iterated pattern, while some other yielding constructs allow to accumulate bottom-up in case no method is available or a nested iterated requires value accumulation, as of now only one level of iterated query filtering and post-matches-accumulation is supported; see Chapter 10 for more).

In the following section we introduce the post-matches-accumulation methods/advanced array operations (these are predominantly employed in the query workflow).

20.1 Advanced Array Operations (Querying Support)

The following array methods can be employed in the rule language and in the sequence expressions. We distinguish the

- array accumulation methods
- by-member-or-attribute access methods
- per-array-element with lambda expression access methods

20.1.1 Array Accumulation Methods

Some accumulation method calls are available on arrays of esp. numbers:

.sum()
: returns the sum of the elements in the array, as `T'` for `array<T>`

.prod()
: returns the product of the elements in the array, as `T'` for `array<T>`

.min()
: returns the minimum of the elements in the array, as `T'` for `array<T>`

.max()
: returns the maximum of the elements in the array, as `T'` for `array<T>`

.avg()
: returns the average (mean) of the elements in the array, as `double`

.med()
: returns the median of the elements in the array that must be ordered, as `double`

.medUnordered()
: returns the median of the elements in the array, as `double`

.var()
: returns the variance of the elements in the array, as `double`

.dev()
: returns the deviation of the elements in the array, as `double`

.and()
: returns the conjunction of the elements in the array (of `boolean` values), as `boolean`, `true` in case of an empty array

.or()
: returns the disjunction of the elements in the array (of `boolean` values), as `boolean`, `false` in case of an empty array

The input type `T` must be a numeric type, i.e. one of the integer types `byte` or `short` or `int` or `long` or one of the floating point types `float` or `double`. The result type `T'` is `int` for input types `byte` or `short` or `int`, it is `long` for input type `long`, it is `float` for input type `float`, and it is `double` for input type `double` (the same type is used for carrying out the accumulation, defining bounds and precision).

Note that also the array methods introduced in Section 15.4 can be used for accumulation (/towards query purpose), e.g. the `.asString(separator)` method (or the `.asSet()` method).

20.1.2 Array By-Member Methods

Several by-member access (or array query) methods are available on

- arrays of node or edge types bearing attributes
- arrays of match types having match member entities
- arrays of iterated match types having match member entities
- arrays of match class types having match class member entities

They inspect the value of the attribute `attr` or member of the graph element or internal class object or match stored in the array, instead of the value contained in the array directly.

.indexOfBy<attr>(valueToSearchFor)
: returns the first position `valueToSearchFor:T` appears at, as `int`, or -1 if not found.

.indexOfBy<attr>(valueToSearchFor, startIndex)
: returns the first position `valueToSearchFor:T` appears at (moving to the end), when we start the search for it at array position `startIndex:int`, as `int`, or -1 if not found.

.lastIndexOfBy<attr>(valueToSearchFor)
: returns the last position `valueToSearchFor:T` appears at, as `int`, or -1 if not found.

.lastIndexOfBy<attr>(valueToSearchFor, startIndex)
: returns the last position `valueToSearchFor:T` appears at (moving to the begin), when we start the search for it at array position `startIndex:int`, as `int`, or -1 if not found.

.indexOfOrderedBy<attr>(valueToSearchFor)
: returns a position where `valueToSearchFor:T` appears at, as `int`, or -1 if not found. The array must be ordered, otherwise the results returned by the binary search employed will be wrong; in case of multiple occurrences, an arbitrary one is returned.

The search methods just introduced return the index of a found element, the following ordering or grouping methods return an array.

.extract<attr>()
: returns for an `array<T>` an `array<S>`, where `S` is the type of the attribute or member `attr` to be extracted (or projected out).

.orderAscendingBy<attr>()
: returns an array with the content of the input array ordered ascendingly; available for an attribute or member `attr` of an orderable basic type.

.orderDescendingBy<attr>()
: returns an array with the content of the input array ordered descendingly; available for an attribute or member `attr` of an orderable basic type.

.groupBy<attr>()
: returns an array with the content of the input array grouped alongside the attribute (equal values being neighbours); available for an attribute or member `attr` of basic type or a node or edge type.

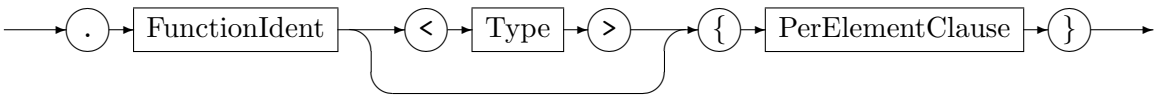
.keepOneForEach<attr>()
: returns an array with the content of the input array freed from duplicates; available for an attribute or member `attr` of basic type or a node or edge type.

The ordering or access-alongside-order methods utilize the ordering relation, so they support only members or attributes of numeric type (`byte`, `short`, `int`, `long`, `float`, `double`,) or `string` or `enum` type – also `boolean` is supported, even if not of much use. The other query methods utilize equality comparisons, so they support additionally members or attributes of node or edge type.

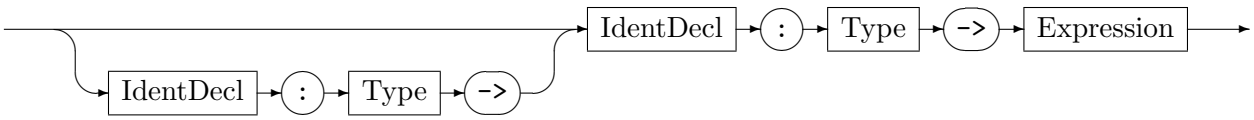
20.1.3 Array Per-Element Methods with Lambda Expressions

Some per-element array methods are available with a special syntax giving a lambda expression (instead of arguments). The lambda expression is evaluated for each element of the array, with the current array element being available in a to be declared element variable (of type `T` for an array of type `array<T>`), and optionally the array index in a to be declared index variable (of type `int`). A type may be given enclosed in angles to denote a mapping result type.

MethodSelector



PerElementClause



.map<S>{element:T -> lambda-expr}
: returns a new `array<S>`, created by mapping each `element:T` (the name can be chosen freely) of the original `array<T>` with the lambda expression (of type `S`) to its resulting counterpart.

.removeIf{element:T -> lambda-expr}
: returns a new `array<T>`, created by removing each `element:T` (the name can be chosen freely) of the original `array<T>` for which the predicate given by the lambda expression (which must be of type `boolean`) yielded true.

.map<S>StartWith{init-expr}AccumulateBy
: `{accumulation-var:T, element:T -> lambda-expr}` returns a new `array<S>`, created by mapping each `element:T` (the name can be chosen freely) of the original `array<T>` with the lambda expression (of type `S`) to its resulting counterpart. The lambda expression is also fed the previous mapping result, for the first iteration initialized by `init-expr`.

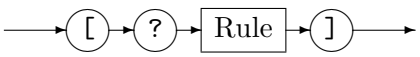
All methods allow for an optional index: the variable `index:int` (the name can be chosen freely) iterates the array indices [0,a.size()-1] of the corresponding elements in the array `a:array<T>` in the
`a.map<S>{index:int -> element:T -> lambda-expr}` and
`a.removeIf{index:int -> element:T -> lambda-expr}` and
`a.map<S>StartWith{init-expr}AccumulateBy`
`{accumulation-var:T, index:int -> element:T -> lambda-expr}` expressions.

Even more, they all allow for an optional array access variable: `this_:array<T>` given as first variable declaration in the lambda expression (the name can be chosen freely), separated by a semicolon, references the original input array, allowing to access it in order to obtain further information, in the
`a.map<S>{this_:array<T>; element:T -> lambda-expr}` and
`a.removeIf{this_:array<T>; element:T -> lambda-expr}` and
`a.map<S>StartWith{this_:array<T>; init-expr}AccumulateBy`
`{this_:array<T>; accumulation-var:T, index:int -> element:T -> lambda-expr}` expressions.

This allows to e.g. decide based on a comparison against the average of an attribute:
`[a.removeIf{this_:array<T>; element:T -> element.a < this_.extract<a>().avg()}]`

20.2 Pattern Matching Based Queries

RuleQuery



The *RuleQuery* clause applies a single rule or test. Note that this is a *Rule* as specified in Section 11.1, that may receive input parameters, and may have filters applied to. The filter calls (which allow e.g. to order the matches list and rewrite only the top ones, or to filter symmetric matches) are explained in Chapter 17.

The square braces (`[]`) and the `?` are part of the syntax, but act like the corresponding rule modifiers specified in Section 11.1, switching for one a rule to a test, i.e. the rule application does not perform the rewrite part of the rule, and searching for the other for all pattern matches.

The execution semantics is: all matches of the rule pattern in the host graph are searched for, and returned as an `array` of `match<r>`. For query purpose, a `test` is to be preferred over a `rule`, as the rewrite part won't be applied anyway. No values are returned in either case, and no return assignments are specified. The data of interest is to be extracted from the array of matches, or the match (type) respectively. Towards this goal, the filters are to be applied, cf. Chapter 17, but esp. the array accumulation and query methods introduced earlier in this chapter, as well as in Section 15.4, notably the `extract` method. No debugging modifier `%` may be applied, rule queries are evaluated silently.

Note that the rule query is a sequence expression – it cannot be applied in a sequence in the place of a rule.

EXAMPLE (123)

```
node class namedObject
{
   name:string;
}

edge class WORKS_FOR;

edge class INTERESTED_IN
{
   weight : double;
}
```

EXAMPLE (124)

```
test interests(subject:namedObject<null>)
{
   subject -:INTERESTED_IN-> interest:namedObject;
---
   def var interestString:string = interest.name;
} \ keepOneForEach<interestString>

test weightedInterests(subject:namedObject<null>)
{
   subject -i:INTERESTED_IN-> interest:namedObject;
---
   def var weight:double = i.weight;
} \ orderDescendingBy<weight>

test sameCompanySharedInterest(subject:namedObject)
{
   subject -:WORKS_FOR-> company:namedObject <-:WORKS_FOR- person:namedObject;
   iterated it {
      subject -:INTERESTED_IN-> interest:namedObject <-:INTERESTED_IN- person;
   }
---
   def var score:double = yield([?it].extract<interest>().size());
   def ref interests:array<string> = yield([?it].extract<interest>().extract<name>());
} \ orderDescendingBy<score>

test sameCompanySharedWeightedInterestClipped(subject:namedObject, var clipAmount:int)
{
   subject -:WORKS_FOR-> company:namedObject <-:WORKS_FOR- person:namedObject;
   iterated it {
      subject -is:INTERESTED_IN-> interest:namedObject <-ip:INTERESTED_IN- person;
   ---
      def var sharedWeightedInterest:double = Math::min(is.weight, ip.weight);
   } \ orderDescendingBy<sharedWeightedInterest>
---
   iterated it\orderDescendingBy<sharedWeightedInterest>\keepFirst(clipAmount);
   def var score:double = yield([?it].extract<sharedWeightedInterest>().sum());
   def var interestsString:string =
       yield([?it].extract<interest>().extract<name>().asString(","));
} \ orderDescendingBy<score>

test fetchByName(var name:string) : (namedObject)
{
   namedObj:namedObject;
   if{ namedObj.name == name; }
   return(namedObj);
}
```

EXAMPLE (125)

Let us take a look at some example queries, based on Example 124, which in turn is based on a test that you can find under `tests/queries` – also including some example data.
In the model we introduce a "generic" namedObject used for subjects, i.e. persons, and objects like companies or hobbies, and edges showing employment and interest for subjects (the model shows only a few general types, which is untypical for GrGen, but common for modelling with Neo4j/Cypher where the idea for this example stems from).
We start with some simple queries, asking for the interests of a subject, and the intensity of the interest. In the rules file, they show up as tests containing the patterns to query for.
Then we continue with some more complex queries, asking for the shared interests of persons working in the same company, and the intensity of the shared interest. In the rules file, they again show up as tests containing the patterns to query for.

In order to obtain all interests (`namedObject`s) all persons are interested in, we drop the following query in GrShell:

```
1 eval [?interests(null)\keepOneForEach<interestString>]
```

For an unspecified subject (`null`, thus for all subjects, due to the `null` specifier at the parameter in the test, which causes searching), it fetches all interests, in the form of `array<match<interests>>`. The array is freed from duplicates within the `interestString` member, so only matches with distinct `interestString`s remain (the first distinct value from the member in the array is kept).
In order to obtain the overall intensity of interests of a single person, we use the following query in GrShell:

```
1 eval [?weightedInterests(::subject)].extract<weight>().sum()
```

From the array of matches, an array of the match members `weight` of type `array<double>` is extracted (resulting from the equivalently named pattern element), and then reduced to a single scalar value with the array accumulation method `sum`.
In order to obtain the interests of two persons, we use the following query in GrShell:

```
1 eval [?interests(::subject)].extract<interestString>().asSet() |
      [?interests(::anotherSubject)].extract<interestString>().asSet()
```

It fetches the interests of `::subject` and `::anotherSubject`, in the form of `array<match<interests>>`, extracts the `interestString`s, resulting in two `array<string>`, transforms them to sets (of type `set<string>`), and finally joins them with set union `|`.

EXAMPLE (126)

Let us continue Example 125. In order to obtain the persons working in the same company with the highest amount of shared interests, we drop the following query in GrShell:

```
1 eval [?sameCompanySharedInterest(::subject)\orderDescendingBy<score>\keepFirst(3)]
```

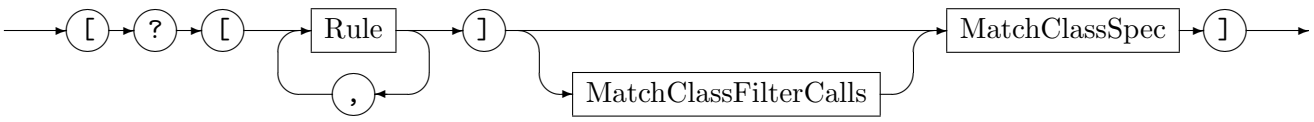
It searches for all matches of pattern `sameCompanySharedInterest` for the given subject, in the form of `array<match<sameCompanySharedInterest>>`. The array is then ordered descendingly alongside `score` (which just gives the amount of shared interests), and clipped to the first 3 matches, thus persons. Besides the order-by that shows up directly in the query expression, a kind of group-by is employed in the used test, with a common pattern giving the shared part, and the differences being encapsulated in the iterated (we don't obtain a cartesian product of all matches and then group afterwards to factor out common parts, but match common parts and extend them by the differences to give the complete match). The interesting information from the iterated is extracted and accumulated with an iterated query (cf. 6.8), which returns an array of matches of the iterated pattern (`array<match<sameCompanySharedInterest.it>>`). (The iterated query can be seen as a get-all-matches rule query inside a rule, and the iterated as a nested all-bracketed rule).
We improve upon the last query by obtaining the persons working in the same company with the highest intensity of interest regarding their shared interests:

```
1 eval [?sameCompanySharedWeightedInterestClipped([?fetchByName("hans")][0].namedObj,
      ::clipAmount)]
```

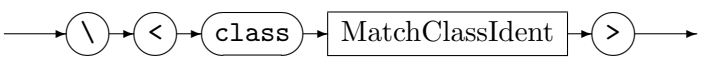
It employs a subquery `fetchByName` to obtain a dedicated person, i.e. `namedObject` by name, reducing the resulting array of matches first to a single array element (the first one) and then to a single member of the match (using the name from the named graph would be preferable in a model fully fitting to the GrGen.NET-technological space); utilizing a query to compute an argument is also of interest in regular rule applications. Furthermore, it computes the shared interest in an object as the smaller of the interests of both subjects in the object, accumulating over all objects the subjects are interested in, with a cut-off applied by a `clipAmount`.

Multi Rule Query

MultiRuleQuery



MatchClassSpec



The *MultiRuleQuery* clause applies multiple (distinct) rules or tests, that all implement the same match class, cf. Section 21.4. It defines a query version of the *MultiRuleExecution*, cf. Section 21.2. It does not allow return assignments in contrast to the latter one, as no output values are available; only the pattern elements – but all of them – are accessible (this holds for all pattern-based query constructs). In contrast to the *MultiRuleExecution*, a *MatchClassSpec* is required. It defines the type of the result (it cannot be derived from the rules, as these may implement multiple (shared) match classes).

Note that the multi rule query is a sequence expression – it cannot be applied in a sequence in the place of a multi rule.

EXAMPLE (127)

The rules from Example 131 can be applied as a query with e.g.
`eval [?[r,s]\Score.orderDescendingBy<score>\Score.keepFirst(3)\<class Score>].`

20.3 Query Result Combination (Auto-generated functions) (preliminary)

You can combine the results of multiple tests or rules with functions that receive arrays of match types as inputs and return an array of match class type. You may do so for one by manually implemented functions, and for the other by auto-generated functions.

Example 128 shows how to combine the matches of the querying tests `sameCompany` and `sharedInterest` in the manner of a SQL natural join statement with the `naturalJoin` function into a `SameCompanySharedInterest` match class (cf. Section 21.4 for match classes; the natural join is a cartesian product with a selection condition requiring value equivalence for common names (to be seen as columns) in the matches (each one to be seen as a row) of the left array (table) and the right array (table)).

EXAMPLE (128)

```
test sameCompany(subject:namedObject)
{
   subject -:WORKS_FOR-> company:namedObject <-:WORKS_FOR- person:namedObject;
}

test sharedInterest(subject:namedObject)
{
   subject -:INTERESTED_IN-> interest:namedObject <-:INTERESTED_IN- person:namedObject;
}

match class SameCompanySharedInterest
{
   subject:namedObject;
   company:namedObject;
   person:namedObject;
   interest:namedObject;
}

function naturalJoin(ref matchesSameCompany:array<match<sameCompany>>, ref
    matchesSharedInterest:array<match<sharedInterest>>) : array<match<class
    SameCompanySharedInterest>>
{
   def ref res:array<match<class SameCompanySharedInterest>> = array<match<class
       SameCompanySharedInterest>>[];
   for(matchSameCompany:match<sameCompany> in matchesSameCompany)
   {
      for(matchSharedInterest:match<sharedInterest> in matchesSharedInterest)
      {
         if(matchSameCompany.subject == matchSharedInterest.subject &&
             matchSameCompany.person == matchSharedInterest.person) {
            def ref m:match<class SameCompanySharedInterest> = match<class
                SameCompanySharedInterest>();
            m.subject = matchSameCompany.subject;
            m.person = matchSameCompany.person;
            m.company = matchSameCompany.company;
            m.interest = matchSharedInterest.interest;
            res.add(m);
         }
      }
   }
   return(res);
}
```

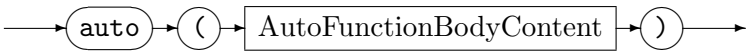
```
match class SameCompanySharedInterest
{
   auto(match<sameCompany> | match<sharedInterest>)
}

function naturalJoin(ref matchesSameCompany:array<match<sameCompany>>, ref
    matchesSharedInterest:array<match<sharedInterest>>) : array<match<class
    SameCompanySharedInterest>>
{
   auto(join<natural>(matchesSameCompany, matchesSharedInterest))
}
```

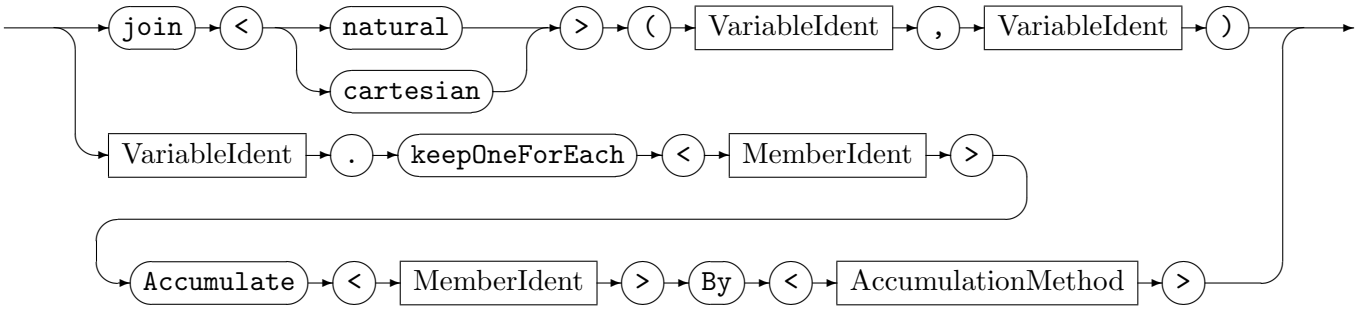
The match of the pattern of the test `sameCompany` contains the named members `subject`, `person`, and `company`, while the match of the pattern of the test `sharedInterest` contains the named members `subject`, `person`, and `interest`. They are combined into a `SameCompanySharedInterest` class, with single `subject` and `person` entries (a name can only appear once, you'd have to add a disambiguating prefix in order to keep two entries stemming from fields with same name, but as they'd contain the same entries due to the join anyway, we can just use a single unprefixed entry without losing information), and `company` as well as `interest` members. The combination is computed from the matches that agree in their `subject` and `person` fields (i.e. have equal values), so you get an array of matches of the common companies and shared interests of two persons – i.e. the persons that work in the same (have one common) company and have a shared (the same) interest, with subject being fixed this amounts to his/her colleagues with shared interests.

The upper part gives a manually implemented version, the lower part shows how to compute the natural join with an auto-generated function body – with the resulting code being the same as in the manually implemented version. It also shows how to compute the match class with an auto-generated match class body, combining the specified match types under name merging – the resulting code is also the same as in the manually implemented version. The auto-generated match class body is to be used with care, it is more concise and writing it is more convenient than listing the members explicitly, but readability is severely hampered, you have to mentally compute the cross product modulo name merging. Note that only named pattern entities are contained in the result; the join also ignores (projects away) unnamed entities.

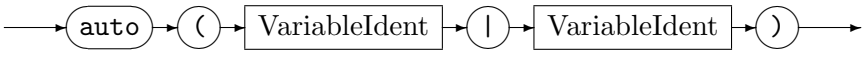
AutoFunctionBody



AutoFunctionBodyContent



AutoMatchClassBody



Another auto-generated function body is available for the cartesian product, noted down with `auto(join<cartesian>(leftArray, rightArray))`. The implementation is the same as the one for the natural join, with the exception of the missing `if` condition (so no real cartesian join is available in case of common names thus shared fields, you have to prepend/append name suffixes to obtain a result containing the values from both inputs). The auto generated functions also support arrays of match classes as input.

Auto-generated array method

The equivalent of an SQL GROUP BY statement with an accumulation function can be generated with the `keepOneForEach<X>Accumulate<Y>By<Z>` filter, cf. Section 17.3. It removes the entries in the matches array that are duplicates in/of member X, the member Y of the

remaining entry receives the result of accumulating the array of all values of Y of the same X by the array accumulation method Z (the available ones are listed in 20.1.1). This *filter* allows to "fuse" results, the quarantine test (to be found under `tests\quarantine`) shows how to use it for summing scores of entries that fall together.

EXAMPLE (129)

```
match class QuarantineScore
{
   person:Person;
   def var score:double;
} \keepOneForEach<person>Accumulate<score>By<sum>, orderDescendingBy<score>

rule quarantineByRiskLodging implements QuarantineScore
{
   person:Person -:livesIn-> l:Lodging;
   if { l.type == LodgingType::HomeForTheElderly; }
---
   def var score:double = 100.0;

   modify {
      person -:underQuarantine-> person;
   }
}
```

```
   eval [?[quarantineByRiskLodging, quarantineByRiskHabit, quarantineByRiskIllness]
      \QuarantineScore.keepOneForEach<person>Accumulate<score>By<sum>
      \QuarantineScore.orderDescendingBy<score>\<class QuarantineScore>]
```

An equivalent array-of-matches processing *function* can be generated with the function body `auto(target.keepOneForEach<X>Accumulate<Y>By<Z>)`. (The auto-generation is only supported for arrays of matches but not for arrays of node/edge types, in contrast to the auto-supplying of the array methods listed in 20.1.2 that support node/edge types (and attributes), because of the side effect of accumulation on the accumulation attribute of the graph element, thus the graph.) Being a function, it can be applied also after matches array return, in contrast to filters that must be applied directly after matching before the matches array is returned for further processing. Applying a function is esp. of interest for the use case of combining multiple matches arrays.

EXAMPLE (130)

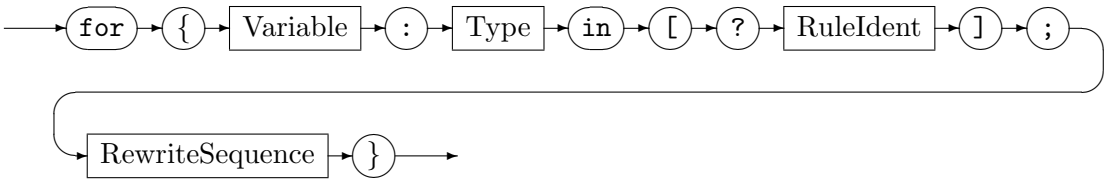
```
function keepOneAccumulate(ref matchesSameCompany:array<match<class QuarantineScore>>) :
    array<match<class QuarantineScore>>
{
   auto(matchesSameCompany.keepOneForEach<person>Accumulate<score>By<sum>)
}
```

```
   eval keepOneAccumulate([?[quarantineByRiskLodging, quarantineByRiskHabit,
       quarantineByRiskIllness]\<class QuarantineScore>]).orderDescendingBy<score>()
```

20.4 Query Looping and Counting Rule Execution

Instead of querying the graph with a sequence expression, and continuing processing with filters and array accumulation methods, is it possible to query it with a rule in a for loop of the sequences, processing the single matches one-by-one.

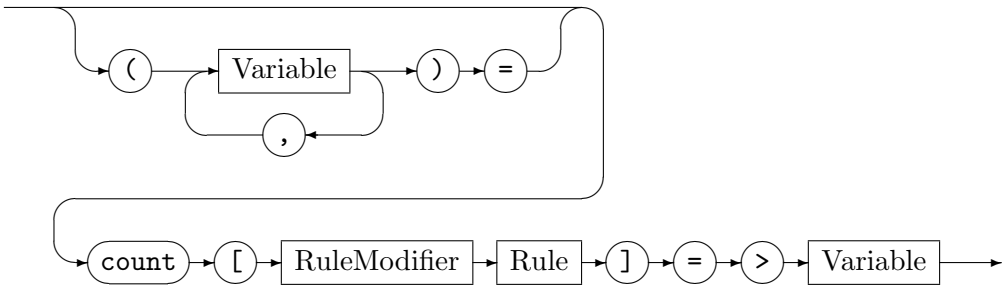
ExtendedControl



The `for` matches loop allows to iterate through the matches found for an all-bracketed rule reduced to a test (or a test); i.e. the rule is not applied, we only iterate its matches. The loop variable must be of type `match<r>` with `r` being the name of the rule matched (the type must be statically known). The elements (esp. the nodes and edges) of the pattern of the matched rule can then be accessed by applying the `.`-operator on the loop variable, giving the name of the element of interest after the dot. Note: the elements must be assigned to a variable so their attributes can be accessed, a direct attribute access after the match access is not possible. Note: the match object allows only to access the top level nodes, edges, or variables. If you use subpatterns or nested patterns and want to access elements found by them, you have to `yield`(10.2.2) them out to the top-level pattern.

In contrast to a rule query that only allows to fetch data from the graph without modifying it, is it possible for a counted rule application to apply a rule (including its rewrite part), and to assign the size of the matches array as rudimentary query result to a variable.

RuleExecution



The `count`ed all bracketing `count[r]=>c` assigns the count of matches of rule `r` to the variable `c`, and applies `r` on all the matches. With `count[?r]=>c` the matches are only counted, no rewrites are carried out.

CHAPTER 21

MULTI RULE APPLICATION AND MATCH CLASSES

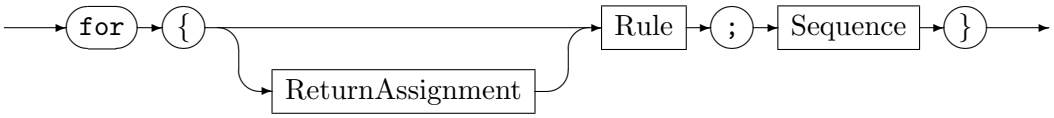
In this chapter we'll take a look at different ways of applying multiple rules with a single construct, and at match classes, which allow applying the same filter on the matches of different rules.

21.1 Rule Prefixed Sequence

RulePrefixedSequence



RulePrefixedSequenceAtom



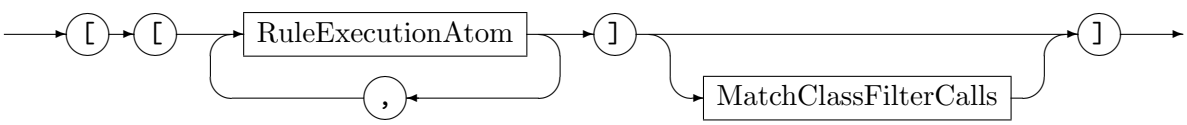
The *RulePrefixedSequence* searches for all matches of its contained rule, and then, for each match (one-by-one): rewrites it, and then executes the contained sequence. The result of the rule prefixed sequence is `true` when at least one of the executions of the contained sequence yielded `true` (which is only possible if at least one match was found for the rule, otherwise the contained sequence won't be executed).

Note that this is a *Rule* as specified in Section 11.1, that may receive input parameters, and may have filters applied to. Furthermore, its return values may get assigned. This happens like for a regular rule call, i.e. *not* in the form of an array per return parameter like for an all-bracketed rule, but in the form of a scalar – for every rewrite performed, the return parameters resulting from exactly that rewrite are bound.

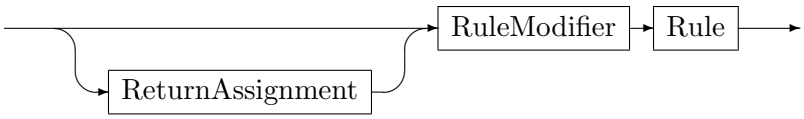
21.2 Multi Rule Application

The *MultiRuleExecution* clause defines a version of all-bracketing (`[]`, cf. 11.1) extended to multiple rules.

MultiRuleExecution



RuleExecutionAtom

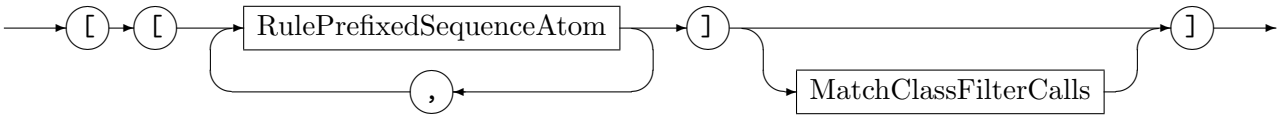


The *MultiRuleExecution* searches for all matches of all of its contained rules, and then rewrites all of them. If at least one was found, rule execution will succeed, otherwise it will fail. Note that this is a *Rule* as specified in Section 11.1, that may receive input parameters, and may have filters applied to. Furthermore, its return values may get assigned. This happens in the form of one array per return parameter per rule, with type `array<T>` for return parameter type `T`. The return parameter arrays are filled with as many values as there were succeeding rule applications.

In addition, match class filters may be executed (on all of the matches of all of the rules), which defines one of the primary reasons why this construct exists. It is your responsibility to ensure that applying the different matches of the different rules does not lead to conflicts.

The *MultiRulePrefixedSequence* clause defines a version of the rule prefixed sequence (cf. 21.1) extended to multiple rules.

MultiRulePrefixedSequence



The *MultiRulePrefixedSequence* searches for all matches of all of its contained rules, and then, rule-by-rule, for each match (one-by-one): rewrites it, and then executes the corresponding sequence. The result of the multi rule prefixed sequence is `true` when at least one of the executions of one of the contained sequences yielded `true` (which is only possible if at least one match was found for one of the rules, otherwise no contained sequence would be executed).

Note that these are *Rule*s as specified in Section 11.1, that may receive input parameters, and may have filters applied to. Furthermore, their return values may get assigned. This happens like for a regular rule call, i.e. *not* in the form of an array per return parameter like for an all-bracketed rule, but in the form of a scalar – for every rewrite performed, the return parameters resulting from that rewrite are bound.

In addition, match class filters may be executed (on all of the matches of all of the rules), which defines one of the primary reasons why this construct exists. It is your responsibility to ensure that applying the different matches of the different rules does not lead to conflicts.

The match class filters may remove matches, or reorder them, the rewrites are performed in order of the globally filtered matches list, in contrast to the syntactic sequence of rule prefixed sequences within the construct.

21.3 Indeterministic Choice

A graph rewrite system shows two points of indeterminism, the rule to apply next, and the place where to apply it.

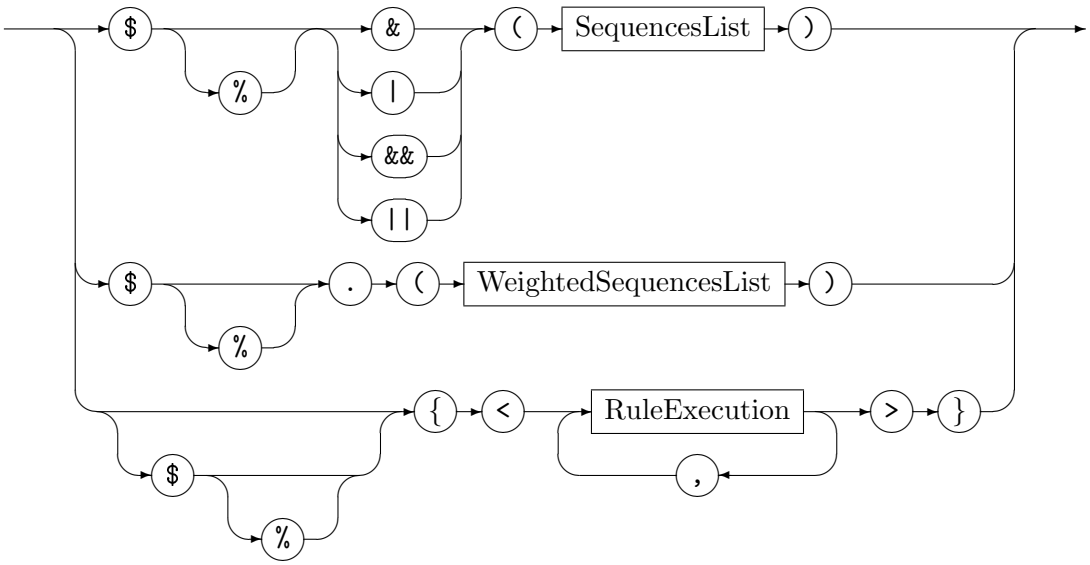
The match found by GrGen for a rule application is indeterministic (the where). It is not random, though, but implementation-specific – it esp. depends on the order of elements in the ringlists (cf. Section 35.3), which is opaque to the outside. Note that the system is implemented to be as deterministic as possible (that's easier to develop with), so when you are debugging the same unaltered project, you get the same results. You can switch to a more random behavior with the random selection modifier that allows you to choose randomly from the matches found for a rule (searching for all matches, then randomly choosing one, compared to aborting search after the first one was found); it is denoted by a dollar prefix for the all-matches-brackets, and was introduced in Section 11.1.

The rule to apply next is normally defined by the sequence it is called from, which is executed deterministically by default, following the execution order programmed with the logical and sequential connectives. The evaluation order of the operands is from left to right,

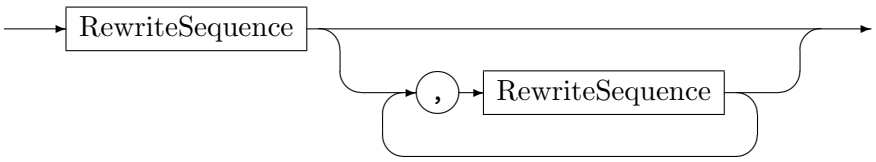
but it can be changed with the random execution order modifier (denoted by a dollar prefix for the logical connective), which was introduced in Section 11.2.

Further evaluation order randomizations, e.g. for simulation purpose, are possible with the indeterministic choice operators, which execute chosen elements from a set of rules or sequences.

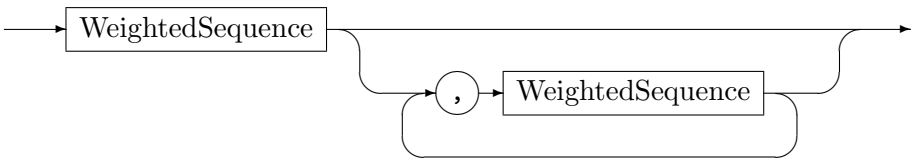
ExtendedControl



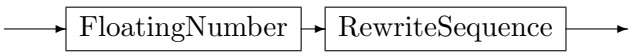
SequencesList



WeightedSequencesList



WeightedSequence



The random-all-of operators given in function call notation with the dollar sign plus operator symbol as name have the following semantics: The strict operators `|` and `&` evaluate all their subsequences in random order, returning the disjunction resp. conjunction of their truth values. The lazy operators `||` and `&&` evaluate the subsequences in random order as long as the outcome is not fixed or every subsequence was executed (which holds for the disjunction as long as there was no succeeding rule and for the conjunction as long as there was no failing rule). A choice point may be used to define the subsequence to be executed next.

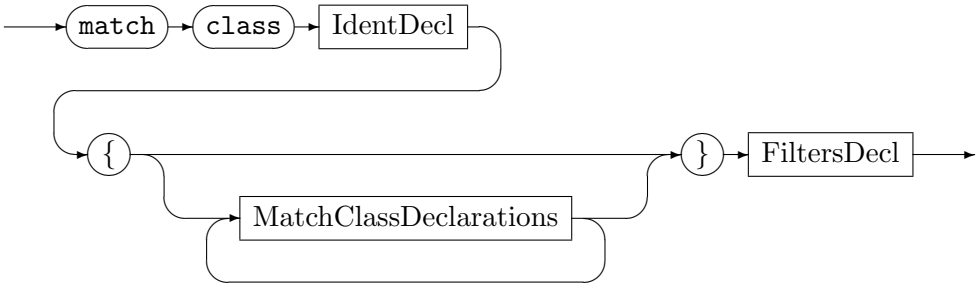
The some-of-set braces `{<r,[s],$[t]>}` matches all contained rules and then executes the ones which matched. The one-of-set braces `${<r,[s],$[t]>}` (some-of-set with random choice applied) matches all contained rules and then executes at random one of the rules

which matched (i.e. the one match of a rule, all matches of an all bracketed rule, or one randomly chosen match of an all bracketed rule with random choice). The one/some-of-set is true if at least one rule matched and false if no contained rule matched. A choice point may be used on the one-of-set; it allows you to inspect the matches available graphically before deciding on the one to apply.

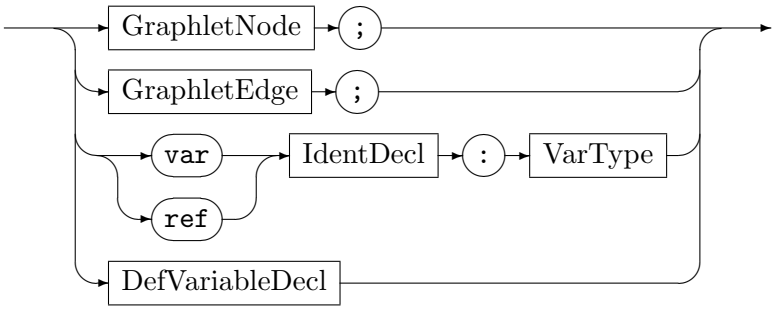
The weighted one operator `$.(w1 s1, ..., wn sn)` is executed like this: the weights `w1-wn` (numbers of type double) are added into a series of intervals, then a random number (uniform distribution) is drawn in between `0.0` and `w1+...+wn`, the subsequence of the interval the number falls into is executed, the result of the sequence is the result of the chosen subsequence.

21.4 Match Classes

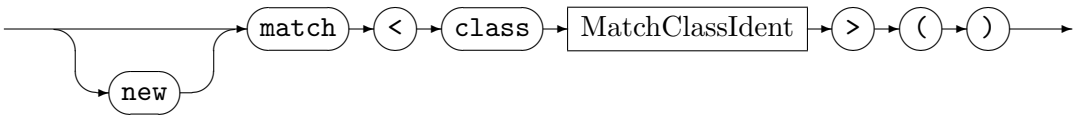
MatchClass



MatchClassDeclaration



MatchClassConstructor



A match class allows to define a pattern interface that can be implemented by several actions (acting as a kind of view). It consists of def node, def edge, and def variable declarations (the primary target of filters), but also normal node, edge, and variable declarations may be given. In contrast to a real pattern, only unlinked entities may be given in this "virtual" pattern. Normally, a match class instance is created by a test or rule that `implements` it. Alternatively, you may create a match of a match class with a match class constructor call `match<class T>()` (the members are initialized with the default value of their type).

Filter functions can then be defined for a match class (instead of a rule as normal), with syntax `filter f<class mc>`, see Chapter 17. Alternatively, auto-generated filters may be declared at a match class — the same that can be declared for a rule/test (with the exception of `auto`, also cf. Chapter 17). Together with the auto-supplied filters, they allow to filter the matches of a multi rule all call or multi rule backtracking construct, i.e. one filter can be applied on the matches of different rules.

EXAMPLE (131)

```
node class N;

edge class weighted
{
   weight:double;
}

edge class unweighted;
```

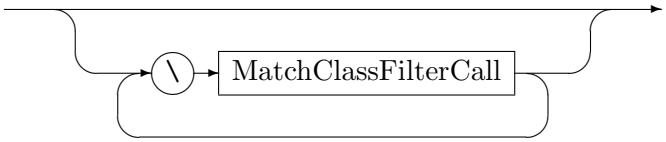
```
match class Score
{
   def var score:double;
} \ orderDescendingBy<score>

test t implements Score
{
   n:N;
   iterated it {
      n -:unweighted-> .;
   }
---
   def var score:double = yield([?it].size());
} \ orderDescendingBy<score>

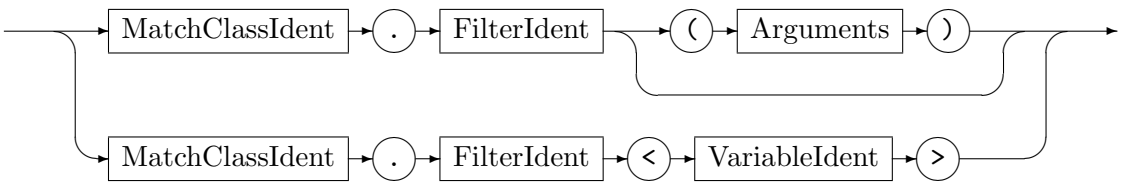
test s implements Score
{
   n:N;
   iterated it {
      n -w:weighted-> .;
   }
---
   def var score:double = yield([?it].extract<w>().extract<weight>().sum());
} \ orderDescendingBy<score>
```

21.5 Match Class Filter Calls

MatchClassFilterCalls



MatchClassFilterCall



Match class filters (defined on the match class) are employed with `\mc.f` notation at multi rule calls from the sequences language, cf. Section 21.2 and Section 22.3, i.e. the match

class identifier must be given as a prefix before the regular filter call. This holds for user-implemented, auto-generated, and auto-supplied filters; the name resolving rules for filter calls apply, cf. Section 17.5.

EXAMPLE (132)

The rules from Example 131 can be applied with e.g.
`eval [[r,s]\Score.orderDescendingBy<score>\Score.keepFirst(3)]`.

To define the match class type of the result array of a multi rule query, a match class specifier must be given. The example below is a reduced version of Example 130 – a multi rule query may be used with a single test/rule and a match class specifier to obtain an array of targeted type.

EXAMPLE (133)

```
1  eval keepOneAccumulate([?[quarantineByRiskLodging]\<class
       QuarantineScore>]).orderDescendingBy<score>()
```

21.6 Quick Reference Table

Table 21.1 lists the multi rule applications and some advanced single rule applications at a glance.

`[for{r;s}]`	For every match of `r`: Executes `r` then `s`. One `s` failing pins the execution result to false.
`count[r]=>v`	Rewrites every pattern match produced by `r`, and writes the count of the matches found to `v`.
`for{v:match<r> in [?r]; s}`	Execute `s` for every match `v` from `r`. One `s` failing pins the execution result to false.
`[[r1,r2]\mc.f\<mc>]`	Rewrites every match produced by `r1` and `r2` that was left by the filter `f` of match class `mc`, in the order left behind by the match class filter.
`[[for{r1;s},for{r2;t}]\mc.f]`	For every match of `r1`: Executes `r1` then `s`. For every match of `r2`: Executes `r2` then `s`. For the matches left by the filter `f` of match class `mc`, in the order left behind by the match class filter. One `s` or `t` failing pins the execution result to false.
`{<r1,[r2],$[r3]>}`	Tries to match all contained rules, then rewrites all of the rules that matched. True if at least one matched.
`${<r1,[r2],$[r3]>}`	Tries to match all contained rules, then rewrites indeterministically one of the rules that matched. True if at least one matched.
`$.(0.5 s, 0.5 t)`	A random number is drawn, and the sequence into whose interval it falls is executed.
`$op(s,t)`	`s` and `t` are executed in random order as long as the outcome is not fixed, returning the disjunction resp. conjunction of their truth values. For `op` being one of the boolean operators `||`,`&&`.
`$op(s,t)`	`s` and `t` are executed in random order, returning the disjunction resp. conjunction of their truth values. For `op` being one of the boolean operators `|`,`&`.

Let `r`, `r1`, `r2`, `r3` be rules or tests, `s`, `t` be sequences, `v` be a variable identifier, `mc` be a match class, `f` be a filter

Table 21.1: Multi rule applications at a glance

CHAPTER 22

ADVANCED CONTROL WITH BACKTRACKING

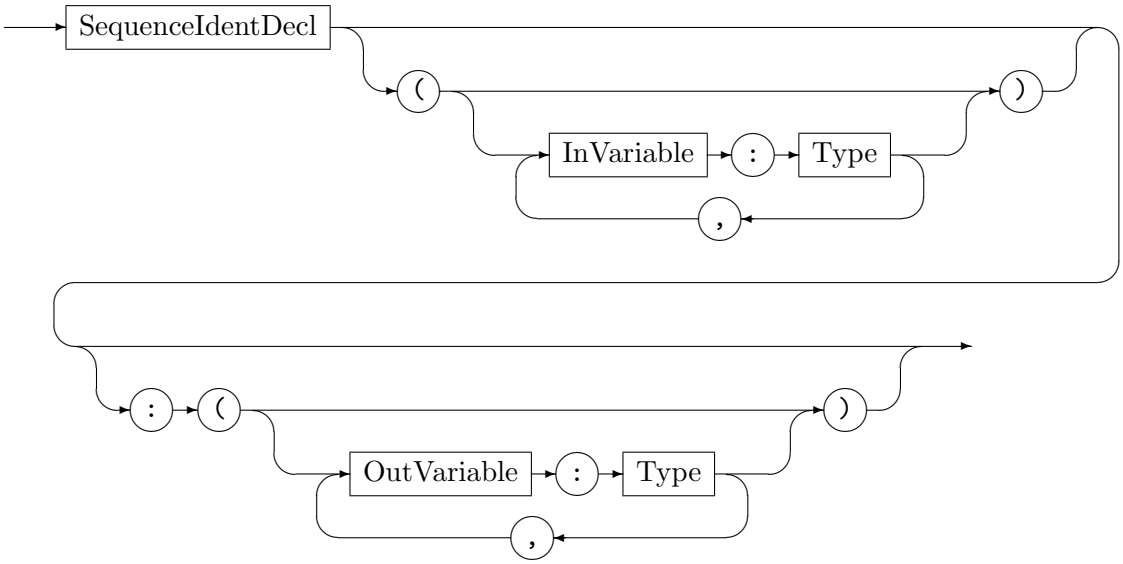
In this chapter we'll have a look at advanced graph rewrite sequence constructs, with the subsequences and the backtracking double angles as the central statements.

22.1 Sequence Definitions (Procedural Abstraction)

RewriteSequenceDefinition



RewriteSequenceSignature



If you want to use a sequence or sequence part at several locations, just factor it out into a sequence definition and reuse with its name as if it were a rule. A sequence definition declares input and output variables; when the sequence gets called the input variables are bound to the values it was called with. If and only if the sequences succeeds, the values from the output variables get assigned to the assignment target of the sequence call. Thus a sequence call behaves as a rule call, cf. 11.1.

A sequence definition may call itself recursively, as can be seen in example 134.

The compiled sequences must start with the `sequence` keyword in the rule file. The interpreted sequences in the shell must start with the `def` keyword; a shell sequence can be overwritten with another shell sequence in case the signature is identical. (Overwriting is needed in the shell to define directly or mutually recursive sequences, as a sequence must be defined before it can get used; apart from that it allows for a more rapid-prototyping like style of development in the shell.)

EXAMPLE (134)

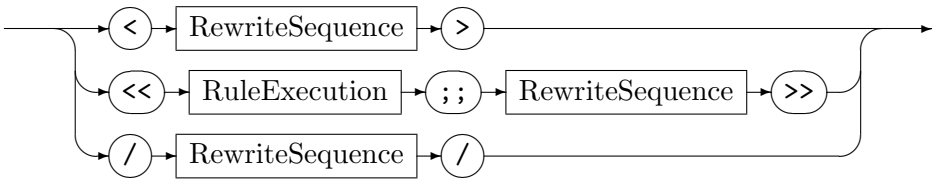
```
def rec(depth:int) {\
  if{ {{depth<::MAXDEPTH}}; foo() ;> rec(depth+1); bar() }\
}
```

This example shows a sequence defined in the shell which is executing itself recursively. The host graph is transformed by applying `MAXDEPTH` times the rule `foo`, until finally the rule `bar` is executed. The result of the sequence is the result of `bar`, returned back from recursion step to recursion step while unwinding the sequence call stack.

22.2 Transactions, Backtracking, and Pause Insertions

The extended control constructs offer further rule application control in the form of transactions, backtracking, and pause insertions.

ExtendedControl



Graph rewrite sequences can be processed transactionally by using angle brackets (`<>`), i.e. if the return value of the nested sequence is `false`, all the changes carried out on the host graph will be rolled back. Nested transactions are supported, i.e. a transaction which was committed is rolled back again if an enclosing transaction fails.

EXAMPLE (135)

We want to execute a sequence `rs` and if it fails a rule `t` – on the original graph. But `rs` is composed of two rules, it denotes the sequence `r&&s`. So first `r` must succeed, and then `s`, too. Unfortunately, it is possible that `r` succeeds and carries out its effects on the graph, and then `s` fails. So we'd need an explicit `undo-r` to reverse the effect of `r` after `s` failed. Luckily, we can omit writing such reversal code (which may require a horrendous amount of bookkeeping) by utilizing transactions, to just try out and roll back on failure: `if{!< r && s >; t}`. After `r` succeeded – with the graph in state post-`r` – `s` is executed. If `s` succeeds, the sequence as such succeeds. But if `s` fails we just roll back the effects of `r` and execute `t` instead.

Transactions as such are only helpful in a limited number of cases, but they are a key ingredient for backtracking, which is syntactically specified by double angle brackets (`<<r;;s>>`. The semantics of the construct are: First compute all matches for rule `r`, then start a transaction. For each match (one-by-one): rewrite the match, then execute `s`. If `s` failed then roll the effects on the graph back and continue with the matches processing loop. If `s` succeeded then commit the changes and break from the matches processing loop. If one `s` succeeded, the overall sequence succeeds, otherwise it fails. On first sight this may not look very impressive, but this construct in combination with recursive sequences is the key operation for crawling through search spaces or for the unfolding of state spaces.

The backtracking double angles separate matching from rewriting: first all matches are found, but then only one after the other is applied, without interference of the other matches. The "without interference of other matches" statement is ensured by rolling back the changes

of the application of the previous match, and much more, of the entire sequence which followed the rewriting of the previous match.

EXAMPLE (136)

The situation we find ourselves in now is to find an application of rule `r` that makes test `t` succeed. Unfortunately, the test would be very complicated to write based on the state before-`r`. So we want to execute `r` and try `t` afterwards. But remembering the other matches of `r` we found in case the just chosen match makes `t` fail, so we can exhaustively enumerate and try them all. Only in case none of them works do we accept failure. The sequence `<<r;;t>>` allows us directly to do so, it fails iff none of the possible applications of `r` can make a following `t` succeed, it succeeds with the first application of `r` for which `t` succeeds.

If you are just interested in the first goal state stumbled upon which satisfies your requirements during a search, then you only need to give a condition as last statement of the sequence which returns true if the goal was reached; the iteration stops exactly in the target state (see example above). But when you are interested in finding all states which satisfy your requirements, or even in enumerating each and every state, just force backtracking by noting down the constant false as last element of the sequence (see example below).

EXAMPLE (137)

The sequence `<< r ;; t ;> false >>` allows us to visit all applications of `r` that make `t` succeed. But everything is rolled back after sequence end. So this is only helpful if you remember something about the states that saw `r` and `t` succeed. Either in some variables, or in the graph itself, during a transaction pause.

The backtracking construct encodes a single decision point of a search, splitting into breadth along the different choices available at that point, and further continuing the search in the sequence. When this point of splitting into breadth is contained in a sequence, and this sequence calls itself again recursively on each branch of the decision taken, you get a recursion which is able to search into depth, continuing decision making on the resulting graph of the previous decision.

EXAMPLE (138)

The example shows the scheme for a backtracking search.

```
1 sequence rec(level:int) {
2    if{ {{level < ::MAXDEPTH}};
3       << r ;; t && changes-to-be-remembered && rec(level+1) >>;
4       false } // end of recursion, will make all the rec return false, causing full rollback
            to initial state, with exception of changes that occured during a pause or are
            outside the graph and thus out of reach of transaction control
5 }
```

With each sequence call advancing one step into depth and each backtracking angle advancing into breadth, you receive a depth-first enumeration of an entire search space (as sketched in 22.1). Each state is visited in *temporal succession*, with only the most recent state being available in the graph. But maybe you want to keep each state visited, because you are interested in viewing all results at once, or because you want to compare the different

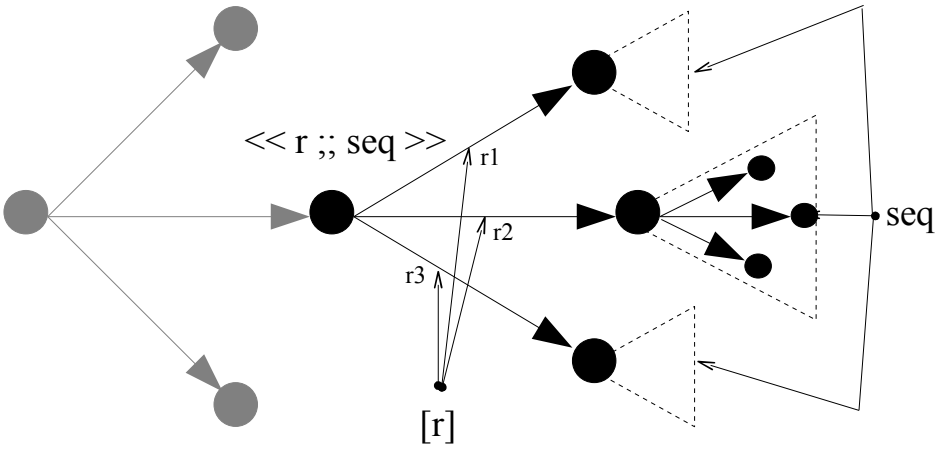


Figure 22.1: Search space illustration, a bullet stands for a graph

states. As there is only one host graph in GrGen.NET, keeping each visited state requires a partition of the host graph into separate subgraphs, each denoting a state.

After you changed the modeling from a host graph to a state space graph consisting of multiple subgraphs, each representing one of the graphs you normally work with, you can materialize the search space visited in temporal succession into a state space graph, by copying the subgraphs (which are normally only existing at one point in time) during pause insertions out into space. When subgraphs would be copied without pause insertions, they would be rolled back during backtracking; but effects applied on the graph from `/ in between here /` are bypassing the recording of the transaction undo log and thus stay in the graph, even if the transaction fails and is rolled back.

When you have switched from a depth-first search over one single current graph to the unfolding of a state space graph containing all the subgraphs reached, you may compare each subgraph which gets enumerated with all the already available subgraphs, and if the new subgraph already exists (i.e. is isomorph to another already generated subgraph), you may refrain from inserting it. This symmetry reduction allows to save the space and time needed for storing and computing equivalent branches otherwise generated from the equivalent states. But please note that the `==` operator on graphs is optimized for returning early when the graphs are different; when the graphs are isomorphic you have to pay the full price of graph isomorphy checking. This will happen steadily with automorphic patterns and then degrade performance. To counter this filter the matches which cover the same spot in different ways, see 34.4 on how to do this. Merging states with already computed ones yields a DAG-formed state space, instead of the always tree like search space. Have a look at the transformation techniques chapter for more on state space enumeration 24.7 and copying 24.5. One caveat of the transactions and backtracking must be mentioned: rollback might lead to an incorrect graph visualization when employed from the debugger. This holds especially when using grouping nodes to visualize subgraph containment (27.1). You must be aware that you can't rely on the online display as much as you can normally, and that you maybe need to fall back

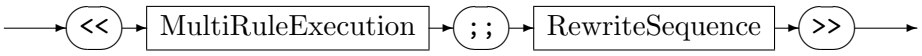
to an offline display by opening a `.vcg`-dump of the graph written in a situation when the online graph looked suspicious; a dump can be written easily in a situation of doubt from the debugger pressing the `p` key.

NOTE (37)

While a transaction or a backtrack is pending, all changes to the graph are recorded into some kind of undo log, which is used to reverse the effects on the graph in the case of rollback (and is thrown away when the nesting root gets committed). So these constructs are not horribly inefficient, but they do have their price — if you need them, use them, but evaluate first if you really do.

22.3 Multi-Rule Backtracking

MultiBacktracking



The semantics of the *MultiBacktracking* construct are: First compute all matches for all rules, then start a transaction. For each match (one-by-one): rewrite the match, then execute the contained sequence. If the contained sequence failed then roll the effects on the graph back and continue with the matches processing loop. If the contained sequence succeeded then commit the changes and break from the matches processing loop. If one sequence succeeded, the overall sequence succeeds, otherwise it fails.

Filters may be applied to the rules in the construct (e.g. `r\f`), but especially: match class filters may be applied to the multi rule execution part `([[...]\f])` – they allow to bring the matches of the different rules into a global order, so only the most promising ones are followed, while the other ones are purged – the default order follows syntactical appearance, first all matches of the first rule are processed, then all matches of the second rule, and so on.

MultiRulePrefixedSequenceBacktracking

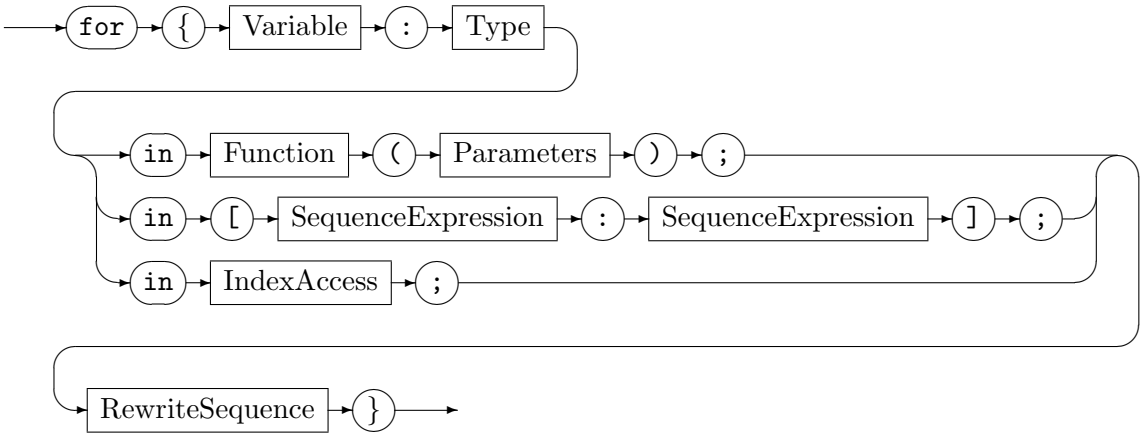


The semantics of the *MultiRulePrefixedSequenceBacktracking* construct are: First compute all matches for all rules, then start a transaction. For each match (one-by-one): rewrite the match, then execute the corresponding sequence. If the corresponding sequence failed then roll the effects on the graph back and continue with the matches processing loop. If the corresponding sequence succeeded then commit the changes and break from the matches processing loop. If one sequence succeeded, the overall sequence succeeds, otherwise it fails.

Filters may be applied to the rules in the construct (e.g. `r\f`), but especially: match class filters may be applied to the multi rule prefixed sequence part `([[...]\f])` – they allow to bring the matches of the different rules into a global order, so only the most promising ones are followed, while the other ones are purged – the default order follows syntactical appearance, first all matches of the first rule are processed, then all matches of the second rule, and so on.

22.4 For Loops

ExtendedControl



The `for` loop over the *Functions* `nodes` or `edges` are iterating over all the elements in the current host graph which are compatible to the type given. The iteration variable is bound to the currently enumerated graph element, then the sequence in the body is executed.

If you iterate a node type from a graph, you may be interested in iterating its incident edges or its adjacent nodes. This can be achieved with a for neighbouring elements loop, which binds the iteration variable to an edge in case the *Function* is one of `incoming`, `outgoing`, or `incident`. Or which binds the iteration variable to a node in case the *Function* is one of `adjacentIncoming`, `adjacentOutgoing`, or `adjacent`.

Moreover, you may iterate with `reachableIncoming`, `reachableOutgoing`, and `reachable` the nodes reachable from a starting node, or with `reachableEdgesIncoming`, `reachableEdgesOutgoing`, and `reachableEdges` the edges reachable from a starting node. Or you may use one of the bounded reachability functions `boundedReachable`, `boundedReachableIncoming`, `boundedReachableOutgoing`, `boundedReachableEdges`, `boundedReachableEdgesIncoming`, `boundedReachableEdgesOutgoing` here.

The admissible *Parameters* for the *Functions* are the source node, or the source node plus the incident edge type, or the source node plus the incident edge type, plus the adjacent node type — that's the same as for the sequence expression functions explained in 16.2/Connectedness queries. In contrast to these set returning functions, this loop contained functions enumerate nodes/edges multiple times in case of reflexive or multi edges.

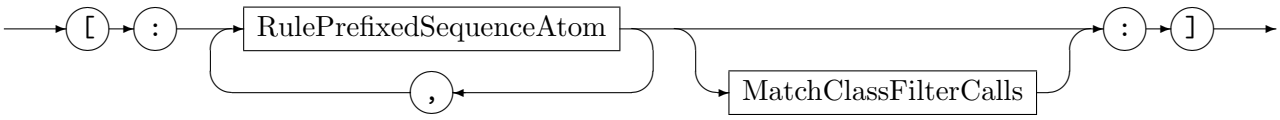
The third `for` loop from the diagram above, the for integer range loop, allows to cycle through an ascending or descending series of integers; the loop variable must be of type `int`. First the left and right sequence expressions are evaluated, if the left is lower than or equal to the right the variable is incremented from left on in steps of one until right is reached (inclusive), otherwise the variable is decremented from left on in steps of one until right is reached (inclusive).

The fourth `for` loop listed above allows to iterate the contents of an index, see 30.4 for more on this and the *IndexAccess*.

The most important `for` loop, the one iterating a container, for enumerating the elements contained in storages, was already introduced here: 19.3. All `for` loops fail if one of the sequence executions from the body failed (all are carried out, though, even after a failure), and succeed otherwise.

22.5 Mapping Clause

MappingClause



The *MappingClause* yields an array of new graphs derived from the input host graph. Note that the host graph is *not* modified by this sequence expression. It is changed during sequence execution though, `this` points to the freshly cloned and modified graph.

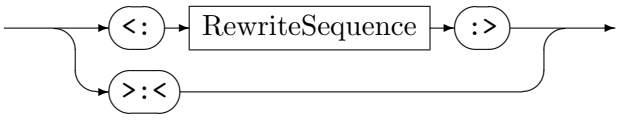
The example construct `[:for{r;seq}:]` returns the graphs for which executing `seq` yielded true, after applying a match of `r`, on a clone of the current host graph, for all matches found. It can be extended to multiple rules and sequences `[:for{r1;seq1},for{r2;seq2}:]`, you receive then the array concatenation of the single constructs.

The *MappingClause* should be handled with care, as for each match, a clone of the current host graph is created, which is expensive. So it should only be used if this is not of major importance, or what is expected anyway, as e.g. in a state space enumeration. An example implementation of such one can be found in the `tests/statespaceMapping` folder (in fact multiple example implementations, simple ones as well as full implementations, utilizing e.g. a tree of object class instances (cf. Section 18.1), storing the graphs reached during unfolding (or even a DAG of object class instances, with isomorphic graph pruning being carried out).

22.6 Persistence Provider Transactions

Further rule application control is available in the form of persistence provider transactions (aka database transactions of the persistent graph, see 26.1 for more on the persistent graph).

ExtendedControl



The sequence within the persistence provider transaction angles is executed within a database transaction. This defines the granularity (of the atomicity) of durability, which has major impact on performance. Without explicit database transaction angles, each and every single graph change will be processed within its own database transaction, giving a fine granularity of change persisting, ensuring even the slightest changes are durable after execution, at the price of wreaking havoc on performance. Note that rollback only occurs when the sequence crashes, otherwise at the closing angle, the database transaction will be committed.

Also note that the persistence provider transactions aka database transactions are unrelated to the application logic layer transactions that are processed in-memory (an undo log is recorded, that is executed upon rollback). Nested transactions are *not* supported.

With the commit-and-restart command denoted by `>:<` you can commit the database transaction before the closing angle, a new one is started immediately thereafter - this way you can control the granularity of durability even in a nested construct.

Of course it makes sense to couple the persistence provider transactions / commit-and-restart command with the application logic transactions introduced above at commit/rollback (or backtracking of the backtracking double angles) – you have to do so explicitly. Transactional behavior and change rollback in GrGen.NET is an application layer construct, with persistence provider transactions you only define the granularity of durability/performance of the changes from the layer above.

EXAMPLE (139)

The examples `persistentgraph_transaction_performance.grs` and `persistentgraph_performance.grs` that can be found in the `tests/persistentgraph` directory allow to compare the performance of using database transaction and working without them.
The sequence `<: ( create[25] ;> ( assignAttributesOverwrite[10] ;> ...)[3] ;> ... ;> delBiggerPartAddSmallerPart[8] )[5] ;> [cleanReferencesToZombies] :>` is executed about 300 times faster than the same sequence without persistence provider transaction angles.

22.7 Quick Reference Table

Table 22.1 lists most of the operations of the advanced graph rewrite sequence constructs at a glance, followed by the language constructs supporting graph nesting.

`<s>`	Execute `s` transactionally (rollback on failure).
`<<r;;s>>`	Backtracking: try the matches of `r` until `s` succeeds.
`/ s /`	Pause insertion: execute `s` outside of the enclosing transactions and sequences, i.e. the changes of `s` are not rolled back.
`<<[[r1,r2]];;s>>`	Multi-Rule-Backtracking: try the matches of `r1` and `r2` until `s` succeeds. You may apply a match class filter `\mc.f`, the matches are then tried in the order left behind by the match class filter.
`<<[[for{r1;s},for{r2;t}]]>>`	Multi-Rule-Prefixed-Sequence-Backtracking: try the matches of `r1` and `r2` until `s` or `t` succeeds. You may apply a match class filter `\mc.f`, the matches are then tried in the order left behind by the match class filter.
`[:for{r;s}:]`	Mapping clause: returns the graphs for which executing `s` yielded true, after applying a match of `r`, on a clone of the current host graph, for all matches found.
`for{v in u; s}`	Execute `s` for every `v` in storage set `u`. One `s` failing pins the execution result to failure.
`for{v->w in u; s}`	Execute `s` for every pair (`v`,`w` in storage map `u`. One `s` failing pins the execution result to failure.
`for{v:int in [l:u]; s}`	Execute `s` for every `v` in the integer range starting at `l` and ending at `u`, upwards by one if `l<=r`, otherwise downwards by one. One `s` failing pins the execution result to failure.
`for{v in func(...); s}`	Execute `s` for every node/edge in the graph query function `func`, which may be nodes/edges, or incident/adjacent, or reachable/boundedReachable. One `s` failing pins the execution result to failure.
`for{v:N in indexFunc(...); s}`	Execute `s` for every `v` in the index query function `indexFunc`. One `s` failing pins the execution result to failure.
`<:  s :>`	Execute `s` within a database transaction.
`>:<`	Commit and restart the currently active database transaction.
`{comp}`	An unspecified sequence computation (see table 19.1).
`(w)=s(w)`	Calls a sequence `s` handing in `w` as input and writing its output to `w`; defined e.g. with `sequence s(u:Node):(v:Node) { v=u }`.
`[var] g:graph`	Declares a variable `g` of type graph, as attribute inside a node or edge class, or as local variable or parameter.
`in g {s}`	Executes `s` in the graph `g`.
`(...)=g.r(...)`	Executes the rule `r` in the graph `g`
`this`	Returns the current graph, can be used like a constant variable in an expression, of the rule language, or the sequence computations. Inside a method, `this` denotes the object of the containing node or edge!

Let `r`, `r1`, `r2` be rules or tests, `s`, `t` be sequences, `u`, `v`, `w` be variable identifiers, `g` be a variable or attribute of graph type, `<op>` be $\in \{|, \hat{}, \&, ||, \&\&\}$

Table 22.1: Advanced sequences and graph nesting support at a glance

CHAPTER 23

PARALLELIZATION

GrGen.NET offers 3 ways of parallelization (hopefully performance-boosting):

- you may endow rules with a parallelization annotation, then a parallelized version is generated,
- you may employ parallelized graph isomorphy checking,
- you may parallelize explicitly at sequence level (in contrast to the former internal implementation measures).

23.1 Internal parallelization

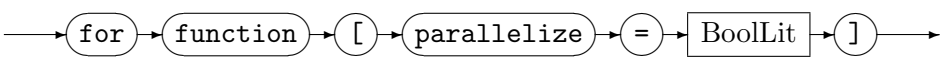
Action parallelization

A rule or test annotated with `[parallelize=k]` will be matched with `k` worker threads from a thread pool. More exactly: at most `k` worker threads, the number is clipped by the number of really available processors, on a single core the normal sequential matcher will be used. (The current implementation defined maximum is 64.)

Parallelization distributes work along the first loop that is binding a pattern element to candidate graph elements. If that loop only iterates once nothing is gained. If each loop iteration only executes very few search work following candidate assignment, things become *slower* because of threading and locking overhead. Don't just append the parallelize annotation to each and every rule in the hope things will become faster! Only search-intensive tasks benefit, only for them does the overhead of parallelization pay off. But for those search-intensive tasks, you can achieve high speedups, reaping benefits from our-days multicore machines.

Remark: Only the pattern matcher of an action is parallelized, so only the search process is parallelized. This offers the biggest bang for the bucks invested, as search is *the* expensive task, and it allows you to stick to the much simpler sequential programming model. A parallelization of the kind presented in [Sch08] offers potentially even higher speedups, but at the price of dealing with a parallel programming model, and at the price of graph partitioning that is very hard to get right in a general-purpose tool.

FunctionParallelization



In order to apply the `parallelize` annotation to an action, you need to first declare parallelization in the model with a `for function[parallelize=true];` clause. It causes the creation of (external) (method) functions that can be called from parallelized matchers (and implies "node edge unique;"). Actions with `parallelize` annotations causing the creation of parallelized matchers expect a "parallelized" model (functions callable from parallelized matchers). An error is reported in case a parallelized rule/test is used with a model without a parallelize (function) declaration.

EXAMPLE (140)

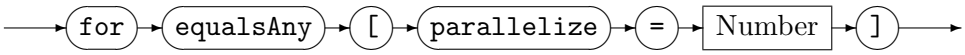
You need a

```
for function[parallelize=true];
```

in the model in order to use parallelized matchers in the rules:

```
test testPar[parallelize=16]
{
   n1:Node --> n2:Node --> n3:Node --> n1;
}
```

EqualsAnyParallelization



equalsAny parallelization

There's a second part that can be parallelized, the `equalsAny` and the `getEquivalent` functions checking for graph isomorphy of a new candidate graph against a set of known graphs. The *EqualsAnyParallelization* clause in the model must specify the number of worker threads used in parallel. A `for equalsAny[parallelize=8];` requests 8 worker threads for checking whether a graph already exists that is isomorphic to the candidate. Graph isomorphy checking is expensive, you may gain considerable speed-ups in state space enumeration (at the price of having to maintain a set of already known graphs).

23.2 Sequence parallelization (preliminary)

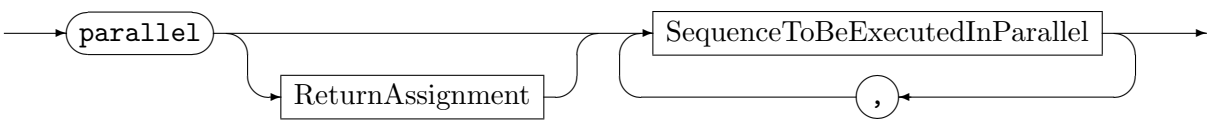
NOTE (38)

The features from this section come without debugger support. They are available in the language proper and can be executed, but we recommend you only use them for experiments. At least as of now, they cannot be used together with internal parallelization. They are also not supported by the persistent graph. Note that global variables access is not synchronized. (The feature in its current state should be considered experimental.)

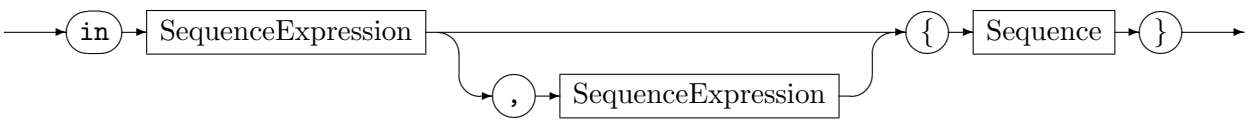
Sequence parallelization is only available in the interpreted sequences, not in the compiled sequences (because it must be executed at top-level, outside of a rule, so outside of an embedded exec).

Parallel sequence

ExtendedControl



SequenceToBeExecutedInParallel



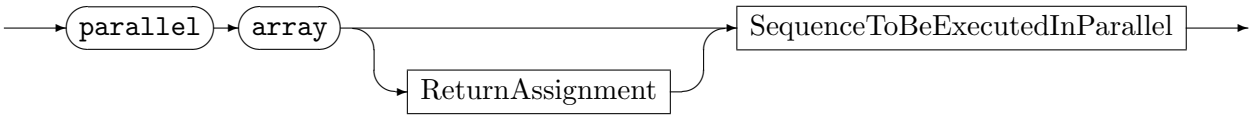
The parallel sequence construct executes the sequences in parallel (given sufficient execution resources), on the specified subgraphs, and evaluates the optional value expressions, making their result available as a local variable named `value` (in the sequences that are executed in parallel) optionally assigning the results of sequence execution.

EXAMPLE (141)

The example stemming from `tests/parallel/parallelWithValue.grs` shows how to generate Sierpinsky triangles in parallel in different (sub-)graphs, with an increasing number of generations (yielding different return values from the single sequences to be executed in parallel, assigning them after execution of the overall sequence).

```
exec parallel (::res1, ::res2, ::res3) =
in ::g1.subgraph, 1 { init && (gen0* & gen1*)[value] ;> true },
in ::g2.subgraph, 2 { init && (gen0* & gen1*)[value] ;> false },
in ::g3.subgraph, 3 { init && (gen0* & gen1*)[value] ;> true }
```

ExtendedControl



Its sibling, the parallel array sequence construct executes the sequence in parallel (`subgraph-array.size()` times, given sufficient execution resources), on the specified subgraphs, and optionally values, optionally assigning the results of parallel sequence execution. The amount of subgraphs in the subgraphs array and values in the values array must be the same, the `array<boolean>` in the return-var will be of the same size.

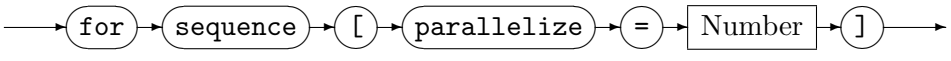
EXAMPLE (142)

The example stemming from `tests/statespaceMapping/statespaceMappingStatespace-NodesTreeParallel.grs` shows how to unfold a (tree-structured) state space (breadth-first) in parallel (in batches).

```
exec ::stopLevel=1 ;> for{step:int in [0 : ::stopLevel];
parallel array in ::tobeprocessed { stateStep() } ;>
{ ::tobeprocessed=::newlycreated.asArray(); ::newlycreated.clear() } }
```

Both constructs are implemented with a thread pool, also using the parent(/main) thread for execution in case the amount of threads from the thread pool is exceeded; the single tasks in the parallel execution construct are executed in sequence (by the parent thread) (at least one - the syntactically last - task is executed by the parent thread).

SequenceParallelization



The *SequenceParallelization* clause in the model allows you to specify the number of worker threads in the sequences thread pool. A `for sequence[parallelize=8];` requests 8 worker threads for parallel sequence execution.

23.3 Locking (preliminary)

NOTE (39)

The locking from this section makes only sense for the parallel sequence execution constructs from the previous section, which we recommend to use only for experiments.

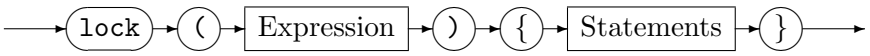
While you can execute in parallel on independent graphs/resources, you have to synchronize access to shared graphs/resources. Towards this purpose, some synchronization statements and methods are available in the sequences and rule language statements; they allow to implement critical sections.

ExtendedControl



With the sequence lock, the specified synchronization object is entered/acquired when the sequence is entered and exited/released when the sequence is left (yielding the result of the contained sequences).

LockStatement



A construct with the same semantics is available in the rule language statements (acting as a pure statement without an implicit result value).

The procedures from the built-in package `Synchronization` allow to enter and exit lock objects.

`Synchronization::enter(.)`
: enters the specified synchronization object (waiting for entry if it is already occupied by another thread (re-entering it in case it was already entered by the same thread)).

`Synchronization::tryenter(.) : (boolean)`
: tries to enter the specified synchronization object, returning false if it was already occupied, or enters it, returning true.

`Synchronization::exit(.)`
: exits the specified synchronization object (crashing if it was not already taken by the same thread, otherwise decreasing the entry counter, releasing other threads if the counter reaches 0).

getEquivalentOrAdd Locking

The following two procedures can be employed in graph isomorphy comparison from parallel worker threads in a thread-safe (yet efficient) manner.

`getEquivalentOrAdd(.,.)`
: returns the graph from the array of graphs that is isomorphic to the graph given as first argument, or null if no isomorphic graph exists, in that case the graph is added to the array of graphs (to the rule language as well as the sequence computations).

getEquivalentStructurallyOrAdd(.,.)
: returns the graph from the array of graphs that is isomorphic neglecting attributes (so only structurally) to the graph given as first argument, or null if no isomorphic graph exists (even when neglecting attributes), in that case the graph is added to the array of graphs.

Both procedures can be used concurrently from sequences executed in parallel, the array is processed atomically and consistently, i.e. only one thread will add a graph that is (not isomorphic to the existing ones yet) isomorphic to another graph concurrently processed by another thread (the other thread will then return this one); it is (/they are) implemented in a way so that the majority of graph comparison work (and array reading) can occur in parallel, outside of array locking.

EXAMPLE (143)

The following example stems from `tests/statespaceMapping/statespaceMapping-StatespaceNodesGraphIsoCheckParallel.grs`, which unfolds a (graph-structured) state space (breadth-first) in parallel (in batches). It uses the same driver as example 23.2, but carries out a graph isomorphy comparison in the parallel worker threads, utilizing the `getEquivalentOrAdd` procedure in a `processState` subsequence, called from the `stateStep` main subsequence (backslashes separating lines removed for better readability).

```
def processState(state:graph, parent:StatespaceNode, red:set<graph>,
                 green:set<graph>, blue:set<graph>)
{
 equivalent:graph ;> { (equivalent) = getEquivalentOrAdd(state, ::statespace) } ;>
 if{ {{equivalent==null}} ;
  child:StatespaceNode ;> lock(::newlycreated) { { ::newlycreated.add(state) } } ;>
  lock(::root) {
   (child)=::root.createAndLinkStatespaceNodeColored(parent, state, red, green, blue)
  } ;>
  lock(::graphToStatespaceNode) { { ::graphToStatespaceNode.add(state, child) } }
 ;
  equivalentNode:StatespaceNode ;>
  ( if{ lock(::graphToStatespaceNode) { { { equivalent in ::graphToStatespaceNode } } };
        lock(::graphToStatespaceNode) {
            { equivalentNode=::graphToStatespaceNode[equivalent] }
        } ;> false; true }
  )* ;>
  lock(::root) { ::root.linkStatespaceNodeColored(parent, equivalentNode,
                 state in red, state in green, state in blue) }
 }
}
```

23.4 Quick Reference Table

Table 23.1 lists the parallelization related operations of the advanced graph rewrite sequence constructs at a glance.

`parallel in g1,v1 {s}, in g2,v2 {t}`	Executes `s` and `t` in parallel, on the graphs `g1` and `g2`, correspondingly, with `value` being bound to `v1` in `s` and `v2` in `t`.
`parallel array in ga,va {s}`	Executes `s` in parallel on the graphs from `ga`, with `value` being bound to the according value from `va`.
`lock(e){s}`	Locks the object resulting from evaluation of `e` before entering `s`, and releases it again after exiting `s`.

Let `s`, `t` be sequences, `g1`, `g2` be variables of graph type, `v1`, `v2` be variables, `ga` be an array of graphs variable, `va` be an array variable, and `e` be a sequence expression

Table 23.1: Parallelization support of the advanced sequences at a glance

CHAPTER 24

TRANSFORMATION TECHNIQUES

In this chapter we'll have a look at transformation techniques to solve common graph rewriting tasks utilizing the constructs introduced so far. They could be offered directly by some dedicated operators, but these would need so much customization to be useful in the different situations one needs them, that we decided against dedicated operators; instead it is on you to program the version you need yourself by *combining* language constructs and rules.

The primary means to build transformations are top-level *sequences*, controlling rule applications (on one or all matches). They may call other sequences, even so recursively. The sequences allow for *external composition* of rules; a rule may be *(re-)used* multiple times. They follow the *imperative* programming paradigm, one rule is applied after the other, on the *then-changed* state (of the graph, changed after each step).

With *embedded sequences* you are able to call rules from within a rule application, after the rule proper was executed (deferred execution), with direct access to the elements of that rule. Seen from the outside, they allow you to build a complex rule and thus allow for rule *internal composition*, again in an *imperative* way. They allow to do work later on you can't do while executing the rule proper (e.g. because an element was already matched and is now locked due to the isomorphy constraint), to follow a breadth-splitting structure, or simply to split work into several parts, modularizing it, *reusing* already available functionality.

The other means for *internal composition* are the *nested and subpatterns*. They allow to match and rewrite complex patterns built in a structured way piece by piece. With different pieces connected together, pieces to decide in between, and pieces which appear repeatedly. They follow the *functional* programming paradigm, a complex state change is built compositionally without graph changes in between, only after applying the rule in a big step is the graph changed (in fact after the rewriting half-step, following the matching half-step). They shift processing below the rules-combined-by-sequences level.

The sequences typically use graph-element-valued variables to communicate a *single spot* in the host graph that is currently getting processed in between the rules. This can be generalized with *storages (container variables)* capable of holding *multiple spots*; they allow to store elements collected in one run and reuse them as input in another run (this is esp. useful if elements are reached on different paths). You can break up a transformation into self-contained *passes* mediated by some *intermediate state* with those multi-valued variables.

Besides, you can employ elementary procedures directly manipulating the graph and elementary functions directly querying the graph, combining them *programmatically* with expressions and statements, abstracting them into own procedures and functions. Those functions and procedures can then be *used* from the `if` and `eval` parts of the rules, and from the sequences.

NOTE (40)

Prefer declarative composition, it helps in keeping the code readable and easily adaptable. Prefer internal embedding over external state passing, it is typically a good deal more concise. But don't go to extremes regarding this, use the mode of composition that is suited to the task and feels best and most natural.
Avoid rules carrying out a tiny amount of work with a lot of input and output and heavy external orchestration (graph-rewriting assembler-level style, to be considered an anti-pattern), do as much as you can per rule call (but note that when you press too hard regarding this you can get over the top regarding understandability, so maybe a medium amount of work per rule call is the right solution).

NOTE (41)

An alternative to storage containment are visited flags, they allow to mark interesting elements directly. The storage sets are more efficient compared to the visited flags in case the count of elements of interest is a good deal smaller than the number of elements in the graph; they are looked up in the set, in contrast to the visited flags, which are used by enumerating all available graph elements, filtering according to the visited state. The visited flags on the other hand are extremely memory efficient (for free as long as you use only a few of them at the same time).

NOTE (42)

Don't forget the last resort if you must solve a task so complex that the above means of composition (regarding control-flow or data-flow) are not sufficient: adding helper nodes, edges, or attributes to the graph model and the graph itself, holding some intermediate state of processing.

In the following we'll employ those constructs to merge and split nodes, emulate node replacement grammars, execute transformations in the narrow sense, aka mappings, copy substructures, compute flow equations over the graph structure with sets in the nodes, and enumerate state spaces. Further examples can be found in [BJ11c] employing two storagesets switched in between for computing a wavefront running over a compiler graph and [JB11] using the nested and subpatterns for elegantly extracting a state machine model from a program graph.

24.1 Merge and Split Nodes

Merging a node `m` into a node `n` means transferring all edges from node `m` to node `n`, then deleting node `m`. Splitting a node `m` off from a node `n` means creating a node `m` and transferring some edges from node `n` to `m`.

In both cases there are a lot of different ways how to handle the operation exactly: Maybe only incoming or only outgoing edges, or only edges of a certain type `T` or only edges not of type `T`; maybe the node `n` is to be retyped, maybe the edges are to be retyped. But common is the transferring of edges; this can be handled succinctly by an `iterated` statement and the `copy` operator. In case the node opposite to an edge may be incident to several such edges, one must use an `exec` instead (or the `independent` operator 5.3.1), as every iteration locks the matched entities, so they can't get matched twice. Not needing the opposite node one

could simply leave it unmentioned in the pattern, only referencing node `n` or `m` and the edge, but unfortunately we need the opposite node so we can connect the edge copy to it.

NOTE (43)

In case a simple node merging without edge retyping is sufficient the retype'n'merge clause introduced in 12.6 offers a much simpler alternative for merging.

Now we'll have a look at an example for node merging: T1-T2 analysis from compiler construction is used to find out whether a control flow graph of a subroutine is reducible, i.e. all loops are natural loops. All loops being natural loops is a very useful property for many analyses and optimizations. The analysis is split into two steps, T1 removes reflexive edges, T2 merges a control flow successor into its predecessor iff there is only one predecessor available. These two steps are iterated until the entire graph is collapsed into one node which means the control flow is reducible, or execution gets stuck before, in which case the control flow graph is irreducible.

The analysis is defined on simple graphs, i.e. if two control flow edges between two basic block nodes appear because of merging they are seen as one, i.e. they are automatically fused into one. As GrGen.NET is built on multigraphs we have to explicitly do the edge fusion in a further step T3.

First let us have a look at T1 and T3, which are rather boring ... ehm, straight forward:

EXAMPLE (144)

```
1  rule T1 {
2    n:BB -:cf-> n;
3
4    replace {
5      n; // delete relexive edges
6    }
7  }
8  rule T3 {
9    pred:BB -first:cf-> succ:BB;
10   pred    -other:cf-> succ;
11
12   modify { // kill multiedges
13     delete(other);
14   }
15 }
```

The interesting part is T2, this is the first version using an iterated statement:

EXAMPLE (145)

```
1  rule T2 {
2    pred:BB -e:cf-> succ:BB;
3    negative {
4      -e->;
5      -:cf-> succ; // if succ has only one predecessor
6    }
7    iterated {
8      succ -ee:cf-> n:BB;
9
10     modify { // then merge succ into this predecessor
11       pred -:copy<ee>-> n; // copying the succ edges to pred
12     }
13   }
14
15   modify { // then merge succ into this predecessor
16     delete(succ);
17   }
18 }
```

In case a control flow graph would be a multi-graph, with several control flow edges between two nodes, one would have to use an `exec` with an all-bracketed rule instead of the `iterated`, to be able to match a multi-`cf`-edge target of `succ` multiple times (which is prevented in the `iterated` version by the isomorphy constraint locking the target after the first match).

This is the second version using exec instead, capable of handling multi edges:

EXAMPLE (146)

```
rule T2exec
{
  pred:BB -e:cf-> succ:BB;
  negative {
    -e->;
    -:cf-> succ; // if succ has only one predecessor
  }

  modify { // then merge succ into this predecessor
  ---
    exec([copyToPred(pred, succ)] ;> delSucc(succ));
  }
}

rule copyToPred(pred:BB, succ:BB)
{
  succ -e:cf-> n:BB;

  modify {
    pred -:copy<e>-> n;
  }
}

rule delSucc(succ:BB)
{
  modify {
    delete(succ);
  }
}
```

Natural loops are so advantageous that one transforms irreducible graphs (which only occur by using wild gotos) into reducible ones, instead of bothering with them in the analyses and optimizations. An irreducible graph can be made reducible by node splitting, which amounts to code duplication (in the program behind the control flow graph). In a stuck situation after T1-T2 analysis, a `BB` node with multiple control flow predecessors is split into as many nodes as there are control flow predecessors, every one having the same control flow successors as the original node. (Choosing the `cf` edges and `BB` nodes which yield the smallest amount of code duplication is another problem which we happily ignore here.)

EXAMPLE (147)

We do the splitting by keeping the indeterministically chosen first cf edge, splitting off only further cf edges, replicating their common target.

```
rule split(succ:BB)
{
  pred:BB -first:cf-> succ;
  multiple {
    otherpred:BB -other:cf-> succ;

    modify {
      otherpred -newe:cf-> newsucc:copy<succ>;
      delete(other);
    ---
      exec(copyCfSuccFromTo(succ, newsucc));
    }
  }

  modify {
  }
}

rule copyCfSuccFromTo(pred:BB, newpred:BB)
{
  iterated {
    pred -e:cf-> succ:BB;

    modify {
      newpred -:copy<e>-> succ;
    }
  }

  modify {
  }
}
```

The examples given can be found in the `tests/mergeSplit/` directory including the control scripts and test graphs; you may add `debug` prefixes to the `exec` or `xgrs` statements in the graph rewrite script files and call GrShell with e.g. `mergeSplit/split.grs` as argument from the `tests` directory to watch execution.

24.2 Node Replacement Grammars

With node replacement grammars we mean edNCE grammars [ER97], which stands for edge label directed node controlled embedding. In this context free graph grammar formalism, every rule describes how a node with a nonterminal type is replaced by a subgraph containing terminal and nonterminal nodes and terminal edges. The nodes in the instantiated graph get connected to the nodes that were adjacent to the initial nonterminal node, by connection instructions which tell which edges of what direction and what type are to be created for which original edges of what direction and what type, going to a node of what type.

This kind of grammars can be encoded in GrGen.NET by rules with a left hand side consisting of a node with a type denoting a nonterminal and iterateds matching the edges and opposite nodes it is connected to of interest; "of interest" amounts to the type and direction of the edges and the type of the opposite node. The right hand side deletes the original

node (thus implicitly the incident edges), creates the replacement subgraph, and tells in the rewrite part of the iterateds what new edges of what directedness and type are to be created, from the newly created nodes to the nodes adjacent to the original node. (Multiple edges between two nodes are not allowed in the node replacement formalism, in case you want to handle them you've to use `exec` as shown in the merge/split example above.)

The following example directly follows this encoding:

EXAMPLE (148)

This is an example rule replacing a nonterminal node `n:NT` by a 3-clique. For the outgoing `E1` edges of the original node, the new node `x` receives incoming `E2` edges. And for incoming `E2` edges of the original node, the new nodes `y` and `z` receive edges of the same type, `y` with reversed direction and `z` of the exact dynamic subtype bearing the same values as the original edges.

```
rule example
{
  n:NT;

  iterated {
    n -:E1-> m:T;

    modify {
      x <-:E2- m;
    }
  }

  iterated {
    n <-e2:E2- m:T;

    modify {
      y -:E2-> m;
      z <-:copy<e2>- m;
    }
  }

  modify {
    delete(n);
    x:T -- y:T -- z:T -- x;
  }
}
```

As another example for node replacement grammars we encode the two rules needed for the generation of the completely connected graphs (cliques) in two GrGen.NET rules. The first replaces the nonterminal node by a new nonterminal node linked to a new terminal node, connecting both new nodes to all the nodes the original nonterminal node was adjacent to. The second replaces the nonterminal node by a terminal node, connecting the new terminal node to all the nodes the original nonterminal node was adjacent to. This "we want to preserve the original edges" can be handled more succinctly and efficiently by retyping which we gladly use instead of the iteration.

EXAMPLE (149)

```
rule cliqueStep
{
  nt:NT;

  iterated {
    nt -- neighbour:T;

    modify {
      t -- neighbour;
      nnt -- neighbour;
    }
  }

  modify {
    delete(nt);
    t:T -- nnt:NT;
  }
}

rule cliqueTerminal
{
  nt:NT;

  modify {
    :T<nt>;
  }
}
```

The examples can be found in the `tests/nodeReplacementGrammar` directory.

24.3 Mapping in a Rewriting Tool

GrGen.NET is a *rewrite* tool, i.e. you modify *one* graph. This is in contrast to *transformation* or better *mapping* tools, which are used for specifying mappings in between *two* (or more) graphs.

A step towards mapping is now available with the mapping clause, cf. Section 22.5, which applies a rule (a single one, in a single step, in contrast to typical mapping tool usages, which apply multiple rules to reach a complete mapping to the result graph). It yields several result graphs, as many as there were matches, created by cloning the original host graph, then applying the rewrite part of the rule.

Graphs are typically called models in mapping/transformation tools. Don't confuse this with the notion of graph model in GrGen.NET as introduced in chapter 4. In model transformation parlance our graph model denotes a meta model (with a model (i.e. graph) adhering to a meta model (i.e. graph model), which in turn adheres to a meta meta model (climbing a ladder of bullshit abstraction leading right into the land of the hot-air merchants, eh..., the truthfully real software engineers. XML. Model. Meta. Meta-meta. Business-logic. Cloud. Bingo!)) The graph model might be a union of multiple graph models, and the graph may consist of multiple unconnected components, so you don't lose anything regarding expressiveness here.

What you lose is the automatic construction of the traceability links which allow to retrieve the source node from a mapped node or the mapped node from a source node, which is normally carried out by a mapping tool in the background. And there are no built-in

annotations by which you specify the model in which to match the elements. But again neither means a loss in expressiveness.

You can construct the traceability links on your own. Either by using simple edges between the nodes (this is not possible in a transformation tool as edges can only exist within a graph/model). Or by using maps of node type to node type (or from edge type to edge type) which you must fill manually: when you map a source node to a target node (i.e. for a matched source node you create a target node), you fill a global map from source to target nodes with this pair, and/or a global map from target to source nodes. When you need this information, you look it up easily with `target=map[source]` or `source=map[target]`. The highlight command 27.1 of the debugger can visualize this storage map neatly by marking the contained nodes and adding directed edges in between the source and domain elements. In some situations you are even better off: if the retype operator is sufficient, you can easily transform the graph in-place. A given, named graph element then denotes the source and the target, just at different points in time.

The lack of built-in model annotations can be overcome by several ways of partitioning the graph or its types. Often the types of the source and the target model are disjoint. This is the most convenient case, as you save any extra notational effort, the types alone define the source and target. If you experience name-clashes, because while being basically disjoint they use same names, you may use packages, to force disjointness for type names. This comes at the price of always having to specify the meta-model, similar to how you specify the model in a mapping tool.

In cases they are not disjoint but share certain parts/types, you can tell the elements from different graphs apart by either using visited flags — then each "graph" is marked to be visited with its own flag. Or you may use anchor nodes representing subgraphs, with an edge from such an anchor node to all nodes contained in its subgraph. Thus all nodes of a subgraph are adjacent to an anchor node, and the subgraph consists exactly of the induced subgraph of these adjacent nodes (see 24.5.1 for more on this kind of modelling).

So while you can write mapping (*out-of-place*) transformations leaving the source untouched, we recommend rewriting (*in-place*) transformations (unless you have a strong real need for the former). This of course and esp. holds for tasks that don't consist of mapping an entire representation into another one, completely changing its structure, but modifying an existing one (e.g. enriching it with additional information, or partial rewriting of constructs amenable to some kind of normalization, while keeping the rest simply intact). Here you can simply leave the unchanged parts out, saving you from specification effort, and from execution time, because you don't need to copy the elements untouched during the transformation pass. For simple mapping tasks you can employ relabeling as its concise rewriting alternative, for complex tasks that require a series of changes are you better off with rewriting due to the argument above, as only parts of each step/pass require a full change in structure.

Mapping *tools* and languages are typically *input-driven*, iterating the source model elements, to decide what target elements to produce from them (focusing one source element at a time, checking context conditions, to create a target pattern). In rewrite tools you typically use a *control program* that defines what is to be carried out, and commonly you work there *output-driven*, calling rules that produce one target element, decided upon by querying the source model for the patterns that produce it (calling rules that rewrite into a target pattern from a complex source pattern matched). This commonly results in a better understandable specification, showing an improved separation of concern. With explicit control by the control program, you are free to choose how to drive the computation, adapting it to whatever suits the problem best.

24.4 Comparing Structures

Structures are compared in three steps, the first is to collect all nodes of interest in a storage set, the second is to compute an induced subgraph from that set, and the third is to compare the subgraphs with the built-in graph comparison operators.

The first step typically consists of covering the nodes of the structure one wants to compare with iterated subpatterns, i.e. subpatterns which match from a root node on with iterateds along the incident edges into breadth, employing a subpattern again on the node adjacent to the root node to match into depth (see Fig. 24.1 for an illustration of how example 150 matches the DAG reachable from the root node `$1` on via outgoing edges). In case the structure to compare consists of multiple connected components or is difficult to capture with iterated and subpatterns, you can collect it step by step with a sequence of rule applications building up the node set.

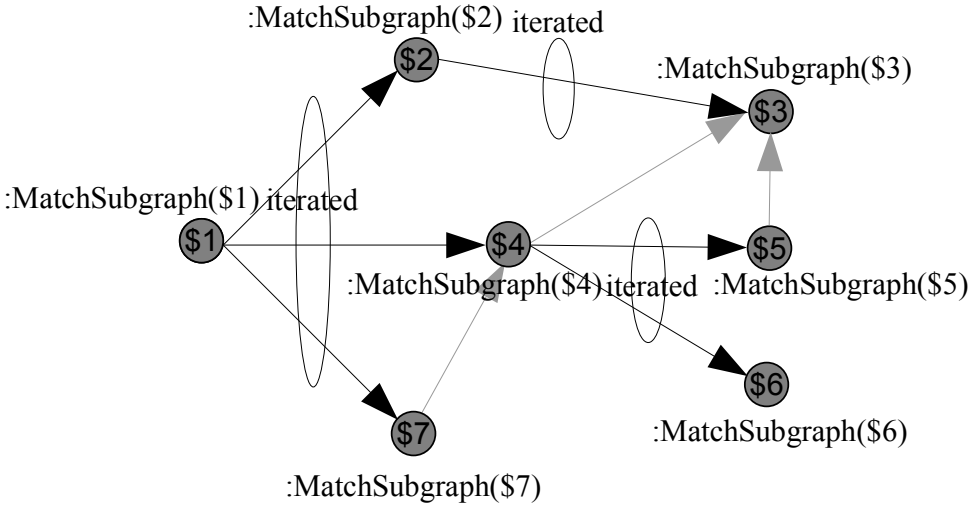


Figure 24.1: Matching a spanning tree in a graph

EXAMPLE (150)

```
pattern MatchSubgraph(root:Node, ref nodes:set<Node>)
{
  iterated { // match spanning tree of graph from root on
    root --> ch:Node;
    ms:MatchSubgraph(ch, nodes);

    modify {
      ms();
    }
  }

  modify {
    eval { nodes.add(root); } // build node set inducing graph
  }
}
```

The second step is to compute an induced subgraph from the node set collected, that can be done easily with the `inducedSubgraph` subgraph operation introduced in 16.3:
`exec sub:graph=inducedSubgraph(nodes)`.

The third step consists of comparing the subgraphs we filleted out of the common host graph with the graph comparison operators introduced in table 16.1:
`exec if{sub1==sub2; doIso; doNonIso}`.

In case each such subgraph needs to be compared more than once it is recommended to add an attribute of type `graph` (see chapter 15) to the subgraph starting anchor nodes, to assign the induced subgraph to this attribute, and to compare the subgraphs stored in these attribute. This saves us the cost of computing the attributes and allows for further internal optimizations which may result in huge speedups.

NOTE (44)

The subgraph is not adapted automatically when the host graph changes, if this happens you must compute the stored subgraph anew manually.

The comparison is then typically done with an idiom as such:
`exec if{ for{other:AnchorNodeType; {curSub!=other.sub}} ; doNonIso(curSub)}` or such:
`exec if{ for{other:graph in setOfCandidates; {curSub!=other}} ; doNonIso(curSub)}`.

Isomorphy comparison is an expensive operation, you can gain considerable speedups by using the parallelizable `equalsAny` and `getEquivalent` functions (cf. 16.3) (and by specifying the number of worker threads (cf. 31.4), otherwise only a single one is used, saving you only the effort of manually coding the loop). In order to use this function you have to supply and maintain a set of the known subgraphs to check against in addition.

24.5 Copying Structures

Structures are copied in two passes, the first consists of copying and collecting all nodes of interest, the second of copying all edges of interest in between the nodes.

The first pass consists of covering the nodes of the structure one wants to copy with iterated subpatterns, in the same way as already introduced in 24.4, i.e. subpatterns which

match from a root node on with iterateds along the incident edges into breadth, employing a subpattern again on the node opposite to the root node to match into depth. In the example we match the entire subgraph from a root node on, if one wants to copy a more constrained subgraph one can simply constrain the types, directions, and structures in the iterated subpattern covering the nodes. The nodes are copied with the `copy` operators and a storagemap is filled, storing for every node copied its copy.

EXAMPLE (151)

The example shows very generally how a subgraph reachable from a root node by incident edges can get copied, collecting and copying the nodes along a spanning tree from the root node on, then copying the edges in between the nodes in a second run afterwards. The edges get connected to the correct node copies via a mapping from the old to the new nodes remembered in a storage-map (traceability map).

```
1  pattern CopySubgraph(root:Node, ref oldToNew:map<Node, Node>)
2  {
3    iterated { // match spanning tree of graph from root on
4      root <--> ch:Node;
5      cs:CopySubgraph(ch, oldToNew);
6
7      modify {
8        cs();
9      }
10   }
11
12   modify {
13     newroot:copy<root>; // copy nodes
14     eval { oldToNew.add(root, newroot); }
15   ---
16     exec( [CopyOutgoingEdge(root, oldToNew)] ); // deferred copy edges
17   }
18 }
19
20 rule CopyOutgoingEdge(n:Node, ref oldToNew:map<Node, Node>)
21 {
22   n -e:Edge-> m:Node;
23   hom(n,m); // reflexive edges
24   nn:Node{oldToNew[n]}; nm:Node{oldToNew[m]};
25   hom(nn,nm); // reflexive edges
26
27   modify {
28     nn -ee:copy<e>-> nm;
29   }
30 }
```

The second pass is started after the structure matching ended by executing the deferred execs which were issued for every node handled. Each `exec` copies all outgoing edges (one could process all incoming edges instead) of a node: for each edge leaving the original node towards another original node a copy is created in between the copy of the original node and the copy of the other node. The copies are looked up with the original nodes from the storage map (which fails for target nodes outside of the subgraph of interest). Here too, one could constrain the subgraph copied by filtering certain edges. In case of undirected edges one would have to prevent that edges get copied twice (once for every incident node). This would require a visited flag for marking the already copied edges or a storage receiving them, queried in the edge copying pattern and set/filled in the edge copying rewrite part.

The example can be found in the `tests/copyStructure` directory. Without storagemaps one would have to pollute the graph model with helper edges linking the original to the copied nodes.

24.5.1 Built-In Graph Copying

In contrast to the just introduced general copying of an entire graph, you may choose a limited form of copying subgraphs. This is enabled by the `insertInduced` and `insertDefined` operations introduced in 16.3, which add a clone of the subgraph induced by the set of nodes or set of edges given as first argument to the host graph. The clone of the second argument node or edge which was inserted into the host graph is returned as anchor element for further operations.

Before you can apply these operations you must collect the nodes or edges which are used to compute the induced subgraph. Besides building the sets element-by-element with `add`-methods you may use set returning functions, e.g. `adjacent` introduced in 16.2, which returns the set of all the nodes adjacent to the node given as (first) argument.

Employing `insertInduced(adjacent())` is a common idiom for copying a subgraph when the host graph is partitioned into multiple subgraphs in the following way: a special node type `Graph` and a special edge type `contains` are introduced into the graph model. Every subgraph is represented by a `Graph` anchor node. From these anchor nodes on `contains` edges lead to all the non-subgraph nodes contained in the corresponding subgraph. With this modelling, all nodes in a subgraph are easily reachable from one anchor node via `adjacent`, and the entire subgraph can be easily retrieved by using `inducedSubgraph` or copied by `insertInduced`. Besides this simplified manipulation, the graph can be easily visualized as being partitioned into multiple subgraphs with a `dump add node Graph group by contains` graph visualization command (see 27.1 for more on this).

24.6 Data Flow Analysis for Computing Reachability

In compiler construction, given a program graph, one wants to compute non-local properties in order to transform the program graph. This is normally handled within the framework of data flow analysis, which employs flow equations telling how property values of a node are influenced by property values of the predecessor or successor nodes in addition to the node's local share on the overall information, with the predecessor or successor nodes being again influenced by their predecessor or successor nodes. Property values are modeled as sets; the information is propagated around the graph until a fix point is reached (the operations must be monotone in order for a fix point to exist on the finite domain of discourse). You might be interested in the transparencies under `http://www.imm.dtu.dk/~hrni/PPA/slides2.pdf` for some reading on this topic, but as of 2026, they seem to have gone for good, you should find the same knowledge in [NNH99]; especially as this is a general method to compute non-local information over graphs not limited to compiler construction.

We'll apply a backward may analysis (with only *gen* but no *kill* information) to compute for each node the nodes which can be reached from this node. Reachability is an interesting property if you have to do a lot of iterated path checks: instead of computing the iterated path with a recursive pattern each and every time you must check for it, compute it once and just look it up from then on. If you need to check several paths which must be disjoint you won't get around employing recursive subpatterns with one locking the elements for the other; but even in this case the precomputed information should be valuable (unless the graph is heavily connected), constraining the search to source and target nodes between which a path does exist, eliminating nodes which are not connected.

The reachability information will be stored in a storage set per node of the graph (indeed, we trade memory space for execution speed):

EXAMPLE (152)

```
node class N
{
   reachable:set<N>;
}
```

The analysis begins with initializing all the storage sets with the local information about the direct successors by employing the following rule on all possible matches:

EXAMPLE (153)

```
rule directReachability
{
  hom(n,m);
  n:N --> m:N;

  modify {
    eval { n.reachable.add(m); }
  }
}
```

The analysis works by keeping a global todo-set `todo` containing all the nodes which need to be (re-)visited, because the information in one of their successors changed; this set is initialized with all nodes available using the sequence `[addAllNodesToWorkset(todo)]`.

From then on in each iteration step a node `n` is removed with `(n)=pickAndRemove(todo)`, until the todo-set becomes empty, signaling the termination of the analysis; the node is processed by determining its successors `[successors(n, succs)]`, adding the reachability information available in each successor to `n` controlled by the sequence
`for{s in succs; (changed)=propagateBackwards(n,s,changed)}`.
If the information in `n` changed due to this, the predecessors of the node are added to the workset via `if{changed;[addPredecessors(n,todo)]}`.

EXAMPLE (154)

This is the core rule of the dataflow analysis: the reachability information from the successor node `s` in its `reachable` storage attribute is added to the storage attribute of the node `n` of interest; if the storage set changes this is written to the returned variable.

```
rule propagateBackwards(n:N, s:N, var changed:boolean) : (boolean)
{
  modify {
    eval { n.reachable |= s.reachable |> changed; }
    return(changed);
  }
}
```

The example can be found in the `tests/dataFlowAnalysis` directory, just add `debug` before the `exec` (or `xgrs`) in `dataFlowAnalysisForReachability.grs` and watch it run. A sample situation showing a propagation step is given in 24.2. The subgraph at the top-left is already handled as you can see by the reachable set displayed in each node.

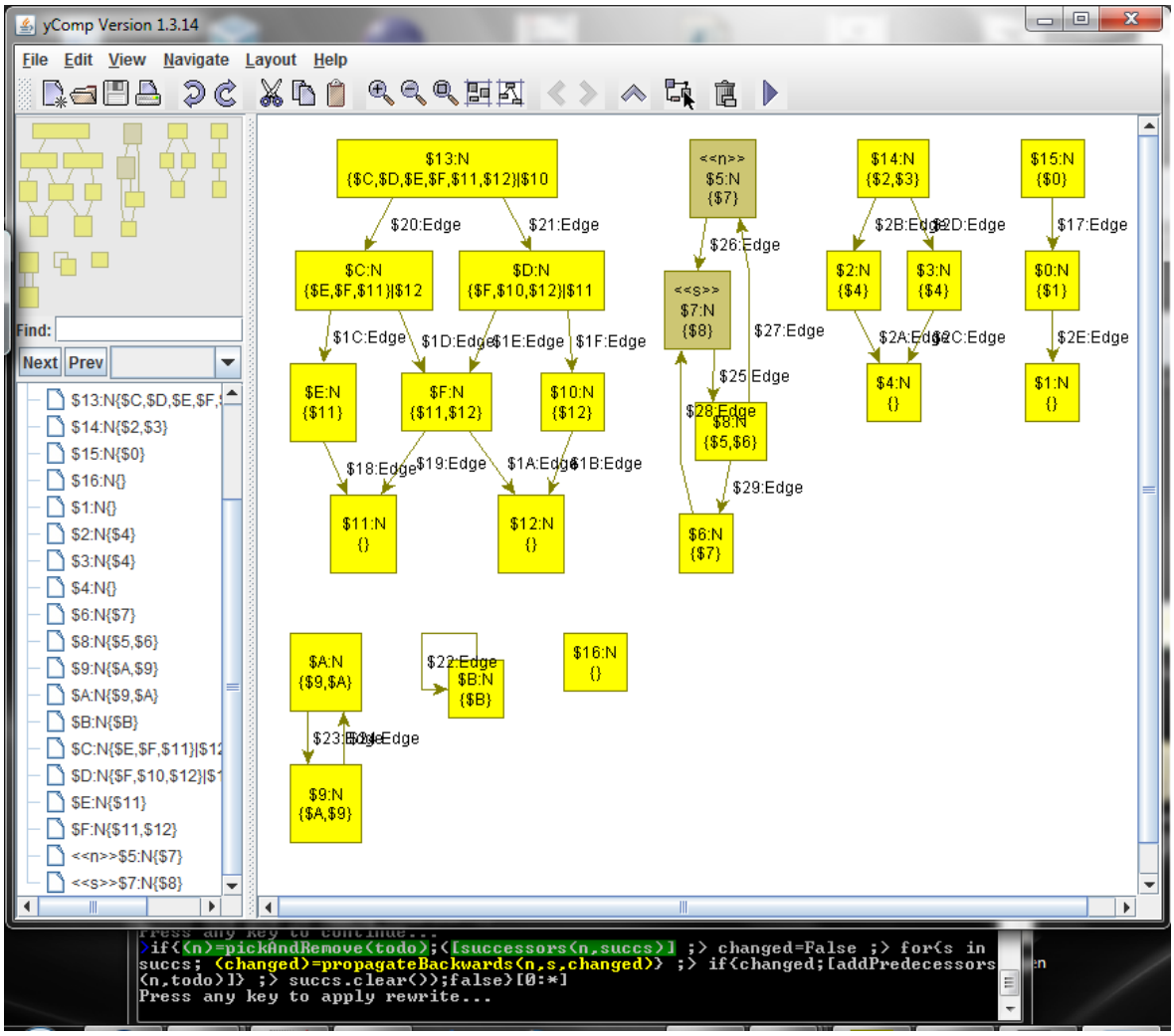


Figure 24.2: Situation from dataflow analysis

Worklist Based Data Flow Analysis

The approach introduced above implements the basics but will not scale well to large graphs – even medium sized graphs – due to the random order the nodes are visited. What is used in practice instead is a version employing a worklist built in postorder, so that a node is only visited after all its successor nodes have been processed. For graphs without backedges, i.e. loops for program graphs, this gives an analysis which visits every node exactly once in the propagation phase. For graphs with loops some nodes will be visited multiple times, but due to the ordering the analysis still terminates very fast.

The worklist is implemented directly in the graph by additional edges of the special type `then` between the nodes, and a special node for the list start; the `todo` set is kept, to allow for a fast "is the node already contained in the worklist"-check, used to save us from adding nodes again which are already contained (thus will be visited in the future anyway); i.e. the abstract worklist concept is implemented by the todo-set and the list added invasively to the graph.

EXAMPLE (155)

```
1 edge class then; // for building worklist of nodes to be handled
```

The initial todo-set population of the simple approach is replaced by worklist constructing, successively advancing the last node of the worklist given by the `last` variable; it starts with all nodes having no successor:
`(last)=addFinalNodesToWorklist(last, todo)*`
Then iteratively all nodes which lead to them get added:
`( (last)=addFurther(pos, last, todo)* ;> (pos)=switchToNextWorklistPosition(pos) )*`
In case of loops without terminal nodes we pick an arbitrary node from them:
`(last)=addNotYetVisitedNodeToWorklist(last, todo)`
and add everything what leads to them, until every node was added to the worklist.

Now we can start the analysis, which works like the simple one does, utilizing the very same propagation rule, but follows the worklist instead of randomly picking from a todo-set, shrinking and growing the worklist along the way.

EXAMPLE (156)

An example rule for worklist handling, adding a not yet contained node to the worklist; please note the quick check for containment via the set membership query.

```
1  rule addToWorklist(p:N, ref todo:set<N>, last:N) : (N)
2  {
3    if{ !(p in todo); }
4
5    modify {
6      last -:then-> p;
7      eval { todo.add(p); }
8      return(p);
9    }
10 }
```

EXAMPLE (157)

An example rule for worklist handling, removing the by-then processed node `pos` from the worklist.

```
1  rule nextWorklistPosition(pos:N, ref todo:set<N>) : (N)
2  {
3    pos -t:then-> next:N;
4
5    modify {
6      delete(t);
7      eval { todo.rem(pos); }
8      return(next);
9    }
10 }
```

The example can be found in the `tests/dataFlowAnalysis` directory, just add `debug` before the `exec` (or `xgrs`) in `dataFlowAnalysisForReachabilityWorklist.grs` and watch it run. A sample situation showing a worklist building step is given in 24.3. The subgraph at the top-left is already handled as you can see by the reachable set displayed in each node.

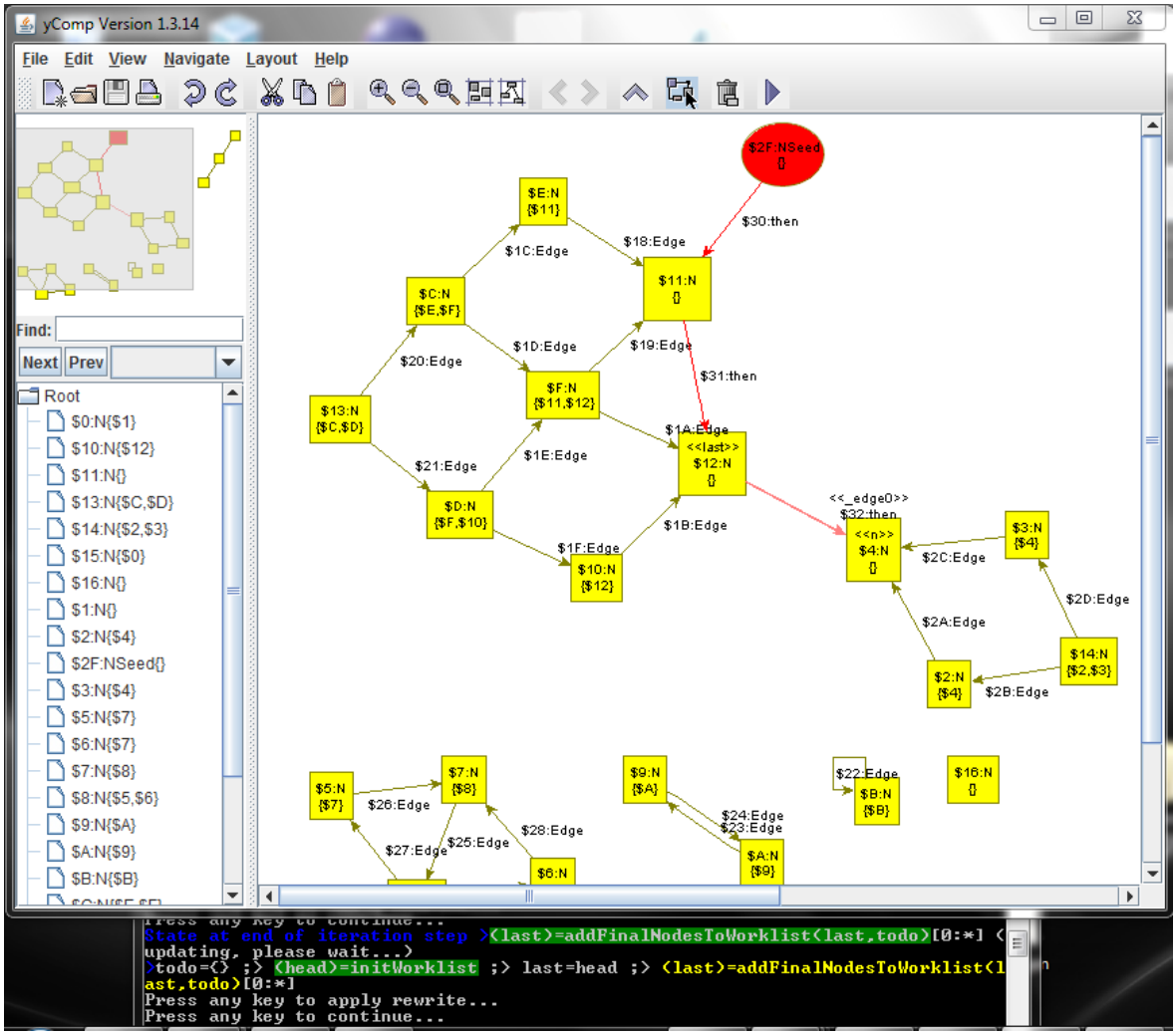


Figure 24.3: Situation from worklist building

24.7 State Space Enumeration

State space enumeration can be programmed in GrGen.NET utilizing the sequence constructs introduced up so far. GrGen.NET always operates on one host graph, over one combined model, applying actions from one combined rule set — so everything is always within reach. That is by far the most simple approach to graph transformation, and for a tool which is not specially geared towards enumerating state spaces what makes most sense. Thus state space enumeration (or transformation between different graphs/models) must be realized within the one host graph. This can be achieved by virtually partitioning the host graph into smaller graphs or subgraphs, following a convention like this one: There are nodes of type `Graph` which are representing a state of the statespace; each such representative is an anchor node for a subgraph. The containment in the subgraph is denoted by a `contains` edge from a node of type `Graph` to all the nodes contained in this (sub)graph. The edges between the `Graph` nodes give the relationship between the subgraphs, i.e. successorship between the states of the statespace. The edges between the non-`Graph` nodes are the "normal" graph edges inside the subgraphs.

The key ingredients for state space enumeration then are

- The aforementioned modelling with anchor nodes linking to the subgraphs, i.e. states
- Backtracking angles — they allow to recursively enumerate and apply the matches of a rule found, exhaustively stepping through the state space, rolling back the effects on the graph of the previous match tried before carrying out the next step (see section 22.2)
- Pause insertions — they allow to write the interesting subgraphs found during state space search out into the host graph while effects recording of backtracking supervision is paused, so that they are not rolled back during backtracking (see section 22.2)
- Recursive sequences — they allow to nest backtracking angles dynamically, so it becomes possible to enumerate an entire state space with a sequence just implementing one backtracking step (see section 22.1)
- The capability to create a copy of a subgraph and insert it into the host graph (see section 24.5, esp. subsection 24.5.1)
- The ability to compare subgraphs (see section 24.4). Implemented in an optimized way with a subgraph attribute in the anchor nodes which contains a copy of the subgraph in the host graph, ready to be compared to the current graph, in order to decide whether that one should be inserted into the state space or purged for being an isomorphic copy of an already available instance. Potentially even further improved with an additional set of subgraphs, compared against with the parallelizable `equalsAny` and `getEquivalent` functions.
- Adjacency and induced functions to compute the subgraphs from the host graph following the `contains` edges from the anchor nodes (see section 16.2, Connectedness Queries and section 16.3, Subgraph Operations)
- The `auto` filters from 34.4 finally allow to optimize performance by filtering symmetric matches stemming from automorphic patterns. This is a lot more efficient than creating states which are for sure isomorphic and filtering them later on by comparison with all the other states.

The modelling given above allows to employ the ability of GrGen.NET/yComp to visualize nested graphs from a flat graph by interpreting node containment along edges of a special type (cf. 27.1), ballooning the anchor node up to a subgraph.

An example implementation of this approach with a variant utilizing isomorphy checking and a variant without isomorphy checking can be found in the `tests/statespace` folder. Feel free to drop in some `debug` prefixes before the `exec` (/`xgrs`) used to watch the assembling of the state space graph. Another example implementation can be found in the `tests/statespaceChemistry` folder. It shows how to enumerate all possible reaction results derivable from a set of start molecules according to some reaction rules. The molecules are just accumulated in the host graph, it is taken care by storage sets that each step only processes the ones available at the start of the step. At the end, each resulting molecule (i.e. connected component) is exported as a `.grsi` file.

State Space Enumeration by Mapping

Since the introduction of the mapping clause (cf. Section 22.5, and 24.3 above), another approach is possible. You don't partition the host graph into subgraphs anymore, and don't carry out changes in order to roll them back, but map an original graph to result graphs (with the state graphs being contained in some data structure, for better visualization still in a graph, the statespace graph). You find an implementation of that approach in the `tests/statespaceMapping` folder, the parallelized versions offer a *breadth-first* statespace enumeration and a statespace graph (see Section 23.2 for more on sequence parallelization), the non-parallelized versions offer a *depth-first* statespace enumeration, and state graph storage in object structures (utilizing the object types from Section 18.1). Both versions (sequential and parallelized enumeration) are available without and with isomorphic graphs pruning (the former yielding a tree, the latter a graph).

CHAPTER 25

GRSHELL LANGUAGE

The console application GrShell and its WindowsForms counterpart, the GGrShell, are shell applications built on top of libGr (for the backend-independent interface) and *libGrShell* (containing most of the implementation) that offer an execution environment for your generated graph transformations. They belong to GrGen.NET's standard equipment, and implement nearly everything in common, so in the following we use only the term GrShell. GrShell is capable of creating, manipulating, and dumping graphs as well as performing and esp. debugging graph rewriting. The GrShell provides a line oriented scripting language. GrShell scripts are structured by simple statements separated by line breaks. The GrShell implements a command-line interpreter in the form of a read-eval-print loop, reading from the console or a file, evaluating (or better executing) the command, and printing the output (as applies), line by line (abstract semantics).

The elements of a graph (nodes and edges) can be accessed both by their (graph global) variable identifier and by their *persistent name* specified through a constructor (see Section 25.3) — the GrShell always uses named graphs, so a persistent name is always existing, if a name is not explicitly assigned, a pseudo-name will be automatically assigned.

EXAMPLE (158)

We insert a node, anonymously and with a constructor (see also Section 25.3):

```
1  > new graph "../lib/lgsp-TuringModel.dll" G
2  New graph "G" of model "Turing" created.
3
4  # insert an anonymous node...
5  # it will get a persistent pseudo name
6  > new :State
7  New node "$0" of type "State" has been created.
8  > delete node @("$0")
9
10 # and now with constructor
11 > new v:State($=start)
12 new node "start" of type "State" has been created.
13 # Now we have a node named "start" (via $ assignment) and a variable v assigned to "start"
```

NOTE (45)

Persistent names will be saved (`save graph...`, see Section 26.2) and exported, and, if you visualize a graph (`dump graph...`, see Section 26.2), graph elements will be labeled with their persistent names. Persistent names have to be unique for a graph (the graph they belong to).

Besides the GrShell-commands defined in this chapter, you find in the following Chapter 26 commands for persistent storage, in Chapter 27 commands for graph visualization and action debugging, and in Chapter 28 commands for model/graph inspection and validation/testing.

25.1 Building Blocks

Here we describe some common building blocks (alongside their syntax) of the commands introduced in the upcoming and the following sections. You may skip it and return to it when you are interested in learning more about the parts of the commands as you visit them.

GrShell is case sensitive. A line may be empty, may contain a shell command, or may contain a comment. A comment starts with a `#` and is terminated by end-of-line or end-of-file or `§`.

The following items are required for representing text, numbers, and rule parameters.

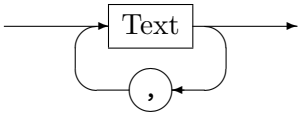
Text
May be one of the following:

- A non-empty character sequence consisting of letters, digits, and underscores. The first character must not be a digit.
- Arbitrary text enclosed by double quotes (`""`).
- Arbitrary text enclosed by single quotes (`''`).

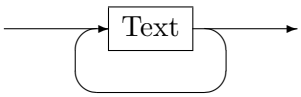
Shell keywords are not allowed for type names, attribute values and other entities (even if they are legal in the rule language, this is a constraint of the chosen parser generator). If this hits you, you can enclose the identifier by single or double quotes, i.e. Text can be used everywhere an identifier is required.

Number
Is an `int` or `float` constant in decimal notation (see also Section 6.1).

Parameters



SpacedParameters



In order to describe the commands more precisely, the following (semantic) specializations of *Text* are defined:

Filename
: A fully qualified file name without spaces (e.g. `/Users/Bob/amazing_file.txt`) or a single quoted or double quoted fully qualified file name that may contain spaces (`"/Users/Bob/amazing file.txt"`).

Variable
: Identifier of a (graph global) variable that contains an entity or a value. A double colon prefix as required in the sequences may be given, but as the shell only knows graph global variables, it is optional (in plain shell commands, not in sequences or sequence expressions appearing within shell commands).

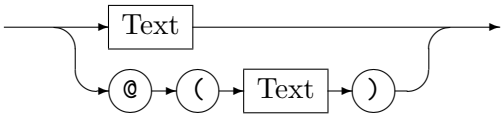
NodeType, EdgeType, ObjectType, TransientObjectType
Identifier of an entity type defined in the model of the current graph.

AttributeName
Identifier of an attribute.

Graph
Identifies a graph by its name.

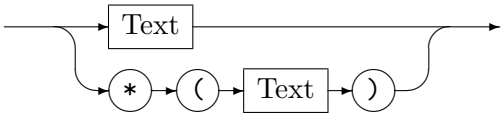
Action
Identifies a rule by its name.

GraphElement



Node, Edge
The specializations *Node* and *Edge* of *GraphElement* require the corresponding graph element to be a node or an edge respectively. The at-notation references by persistent name, otherwise a variable name is used.

Object



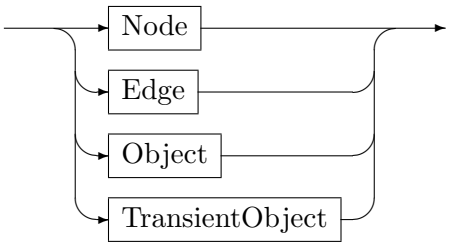
The *Object* requires the corresponding object to be an internal class object. The star-notation references by unique id, otherwise a variable name is used.

TransientObject



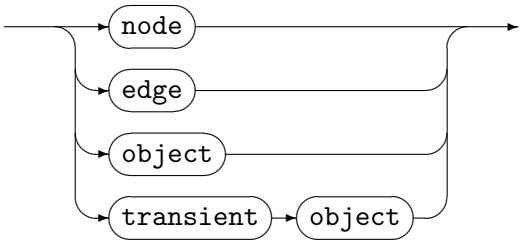
The *TransientObject* requires the corresponding object to be an internal transient class object, the *Text* denotes a variable name. Note that transient object support is preliminary.

Entity



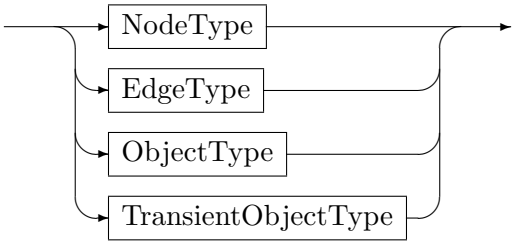
The *Entity* encompasses *Node*s, *Edge*s, *Object*s, and *TransientObject*s.

EntityKind



The *EntityKind* `node` denotes node classes, `edge` denotes edge classes, `object` denotes internal class objects, and `transient object` denotes transient internal class objects – it is used in conjunction with the respective types or elements. Please note that transient class support is preliminary. Also note that `object` is here used to denote internal class objects from the model with the base type `Object` (cf. 18.1), not general objects/external types.

EntityType

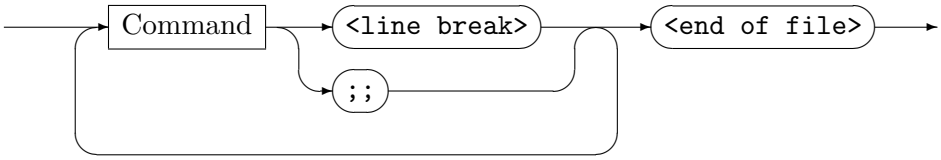


The *EntityType* encompasses *NodeTypes*, *EdgeTypes*, *ObjectTypes*, and *TransientObjectTypes*.

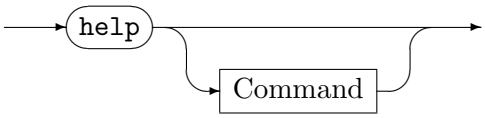
25.2 Common Commands

Here and in the following sections we describe the GrShell commands.

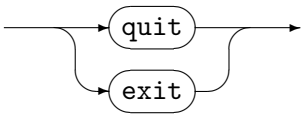
Script



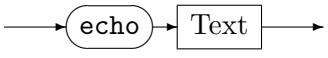
Commands are assembled from basic elements (see esp. Section 25.1). As stated before, commands are terminated by line breaks. Alternatively commands can be terminated by the `;;` symbol. Like an operating system shell, the GrShell allows you to span a single command over n lines by terminating the first $n-1$ lines with a backslash.



Displays an information message describing all the supported commands. A command `Command` displayed with `...` has further help available, which can be displayed with `help Command`.



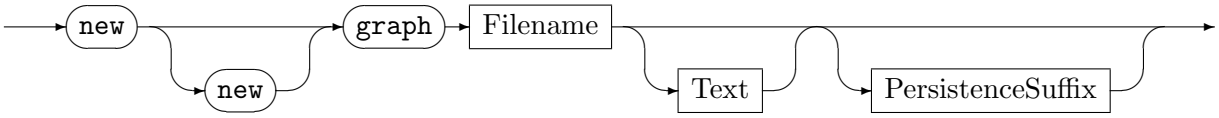
Quits GrShell. If GrShell is opened in debug mode, a currently active graph viewer (such as yComp) will be closed as well.



Prints *Text* onto the GrShell command prompt.

25.3 Graph and Graph Element Creation

The command most shell scripts start with is graph creation.



Creates a new graph with the model specified in *Filename*. Its name is set to *Text* (optional). The model file can be either source code (e.g. `turing_machineModel.cs`) or a .NET assembly (e.g. `lgsp-turing_machineModel.dll`). It's also possible to specify a rule set file as *Filename* (this is the most common usage). In this case the necessary assemblies will be created on the fly (as needed).

In case of a double `new`, the actions and model are created anew for sure, and not as needed according to the file change dates. Use this if you are working with one of the `new set` or `new add` commands from Section 25.10.

When a *PersistenceSuffix* is given, a new persistent graph is created, you find more on it in Section 26.1.

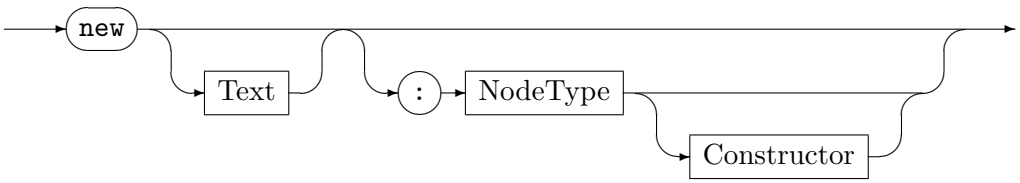
NOTE (46)

You may run into unexpected results because some `new set` or `new add` options that you apply and see in the shell file are not the ones actually compiled into the generated code. This happens easily when you just edit those options, but the actions are not regenerated because the sources did not change. Use `new new` in case you are working with the options from below to ensure the actions are regenerated irrespective of the change dates of their sources (at the cost of steady recompilations).

The following two commands create graph elements, initializing their attributes. On shell level they are available and mainly used as elementary instructions in creating an initial graph, in exporting and importing a graph, as well as in change recording and replaying. These are the commands you may find in the GRS export/import files.

NOTE (47)

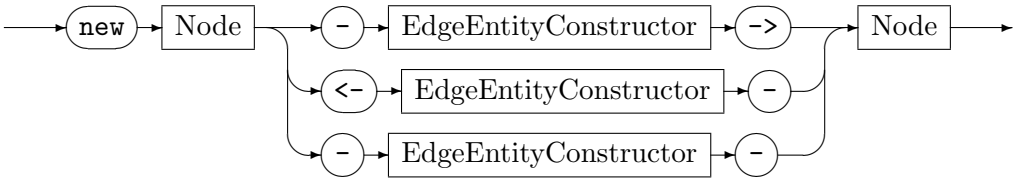
If you need to import data that comes in a format that is not directly supported by GrGen.NET, we recommend to serialize it into `GRS` format, as it can be written easily. It consists of the few `new` commands and the attribute initialization lists explained here in 25.3.



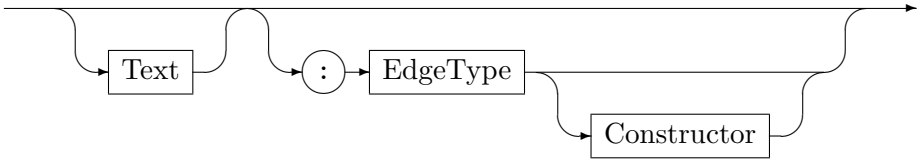
Creates a new node within the current graph. Optionally, the new node is assigned to a variable *Text*. If *NodeType* is supplied, the new node will be of type *NodeType* and attributes can be initialized by a constructor. Otherwise the node will be of the base node class type *Node*.

NOTE (48)

The GrShell can reassign variables. This is in contrast to the rule language (Chapter 5), where we mainly use *names* (bound once – with exception of var and ref input variables and def entities).

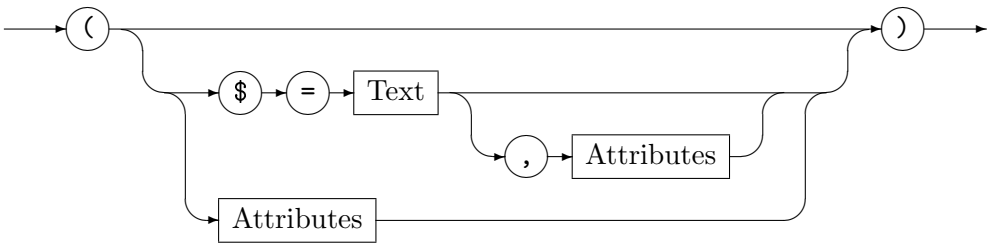


EdgeEntityConstructor

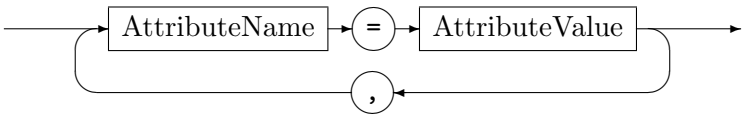


Creates a new edge within the current graph between the specified nodes, in direction of the first to the second *Node* in case of `-->`, in direction of the second to the first *Node* in case of `<--`, or undirected in case of `--`. Optionally, the new edge is assigned to a variable *Text*. If *EdgeType* is supplied, the new edge will be of type *EdgeType* and attributes can be initialized by a constructor. Otherwise the edge will be of the base edge class type `Edge` for `-->` or `UEdge` for `--`.

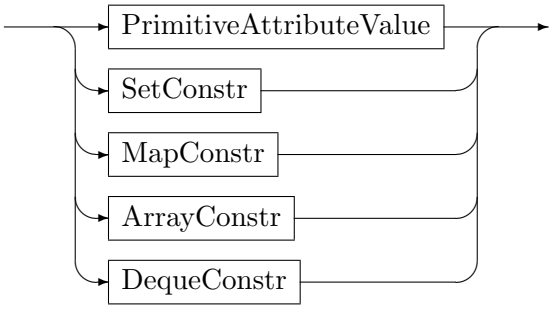
Constructor



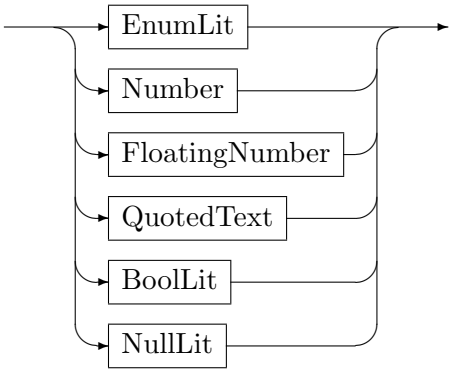
Attributes



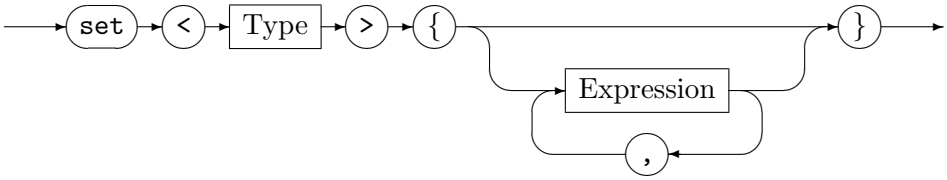
AttributeValue



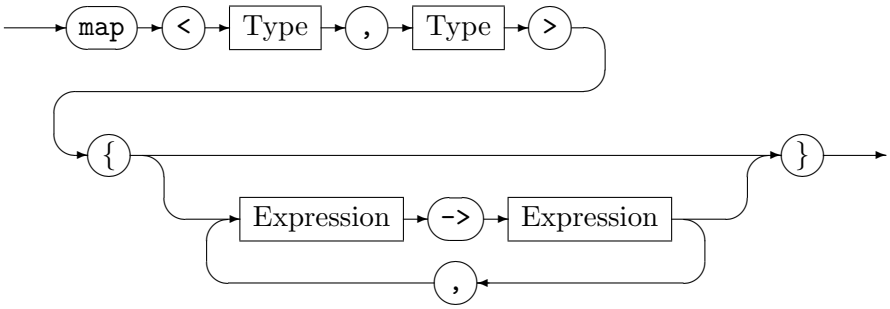
PrimitiveAttributeValue



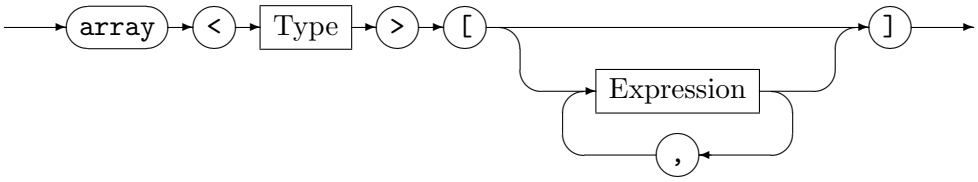
SetConstr



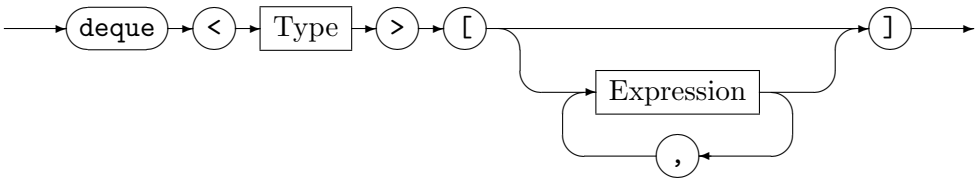
MapConstr



ArrayConstr

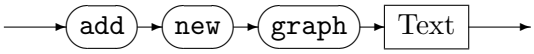


DequeConstr

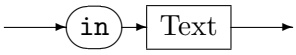


A constructor is used to initialize a new graph element (see `new ...` below). A comma separated list of attribute initializations is supplied to the constructor. Available attribute names are specified by the graph model of the current working graph. All the undeclared attributes will be initialized with default values, depending on their type (`int` ← `0`; `long` ← `0L`; `byte` ← `0Y`; `short` ← `0S`; `boolean` ← `false`; `float` ← `0.0f`; `double` ← `0.0`; `string` ← `""`; `set<T>` ← `set<T>{}`; `map<S,T>` ← `map<S,T>{}`; `array<T>` ← `array<T>[]`; `deque<T>` ← `deque<T>[]`; `enum` ← unspecified;).
The `$` is a special attribute name: a unique identifier of the new graph element. This identifier

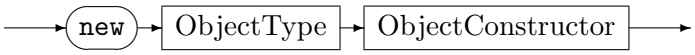
is also called *persistent name* (see Example 158). This name can be specified by a constructor only.



Creates a new subgraph of the same model as the current graph. Its name is set to *Text*; its unique name as used in exporting or recording may be a variant of this name in case another subgraph of the same name already exists. After execution this graph is the current subgraph.

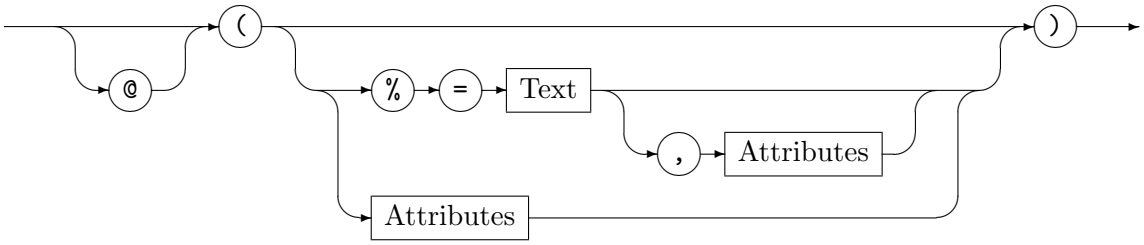


Switches graph processing to the given subgraph (denoted by its unique name). This command as well as the command above are supplied for importing grs files containing subgraph attributes resulting from a grs export or recording. Using them directly is discouraged; you would have to ensure unique names in order to use them, a task that is carried out by the exporter remembering the subgraphs already seen.



Creates a new internal class object (non-transient). The new object will be of type *ObjectType* and attributes can be initialized by an object constructor.

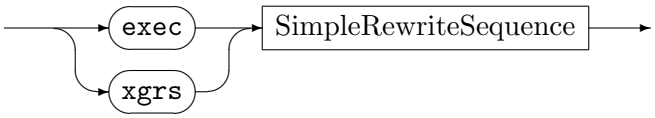
ObjectConstructor



An object constructor is used to initialize a new internal class object. A comma separated list of attribute initializations is supplied to the constructor. The `%` is a special attribute name: a unique identifier of the new object. This name can be specified by a constructor only (and is normally automatically assigned upon object construction – this also occurs when it is left out here).

25.4 Sequence Execution and Profiles

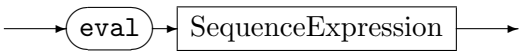
GraphRewriteSequence



This executes the graph rewrite sequence *SimpleRewriteSequence*. See Chapter 11 for graph rewrite sequences.

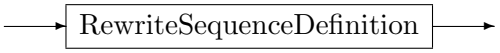
Additionally to the variable assignment in rule-embedded graph rewrite sequences, you are also able to assign *persistent names* to parameters via `Variable = @(Text)`. Graph elements returned by rules can be assigned to variables using `(Parameters) = Action`. The desired variable identifiers have to be listed in *Parameters*. Graph elements required by rules must be provided using `Action(Parameters)`, where *Parameters* is a list of variable identifiers. For undefined variables see Section 5.2, *Parameters*.

GraphRewriteSequenceExpression



This evaluates the graph rewrite sequence expression *SequenceExpression*. See Chapter 19 for graph rewrite sequences expressions (this command is esp. of interest for rule queries, cf. Chapter 20).

GraphRewriteSequence

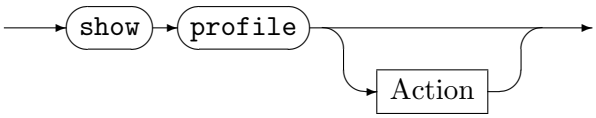


This command allows to define a named sequence at runtime, for *RewriteSequenceDefinition* have a look here 22.1 in the rule application control language chapter. Especially it allows to replace an old sequence definition, but only if the signature is identical. Compiled sequences defined in rule files can't be replaced. The defined sequence can then be used from following graph rewrite sequences (or following sequence definitions) in the shell.

EXAMPLE (159)

```
# a sequence definition (of an interpreted sequence) is only available
# after it was registered the first time
# but it can get overwritten with a sequence of the same signature
# -> (self or mutually) recursive sequences must be constructed with empty body first
def chain(first:A):(last:A){ true }
def chain(first:A):(last:A){ if{(next:A)=chainPiece(first); (last)=chain(next); last=first} }
```

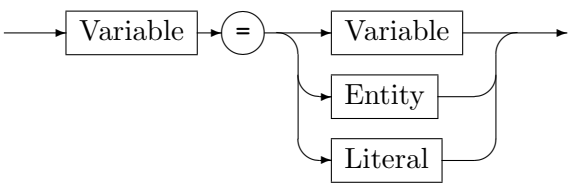
An *action* denotes a graph rewrite rule (or test).



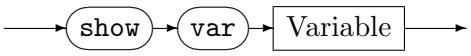
Shows the profile for the action specified by its name, or for all rules and tests in the rule set.

25.5 Variables

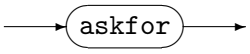
A variable is an identifier that can be bound to a value (scalars, but also containers and entities), which may be changed over time by assignments. A graph rewrite system offers graph global variables s in the form of a global flat mapping of names to values. These are identical with the GrShell variables. They can be accessed in the GrShell directly by their name, whereas in the sequences (or the rule language), a double colon prefix is required (using them outside of the shell is discouraged; see Chapter 11 for the distinction between graph global aka shell and sequence local variables).

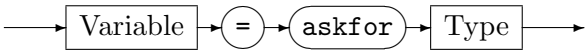


Assigns to the target *Variable* the source *Variable* (i.e. its value), or the *Entity* denoted by its persistent name or unique id, or the *Literal* (textual representation of a constant value). If *Variable* has not been defined yet, it will be defined implicitly. As usual for scripting languages, variables have neither static types nor declarations (the values are typed, though, you run into an exception when you try to execute an operation on a mismatching object).



Prints the content of the specified variable (for references only the identifier of the referenced entity).





The `askfor` command just waits until the user presses enter. The `askfor` assignment interactively asks the user for a value of the specified type. The entered value is type checked against the expected type, and assigned to the given variable in case it matches. If the type is a value type, the user is prompted to enter a value literal with the keyboard. If the type is a graph element type, the user is prompted to enter the graph element by double clicking in yComp. Note that in this case the debug mode must have been enabled before. (The command is equivalent to `debug exec Variable=$%(Type)`.)

EXAMPLE (160)

```
x = askfor int
```

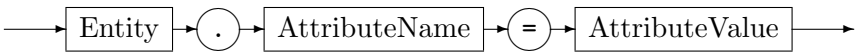
asks the user to enter an integer value; pressing 4 then 2 then enter will do fine.

```
x = askfor Node
```

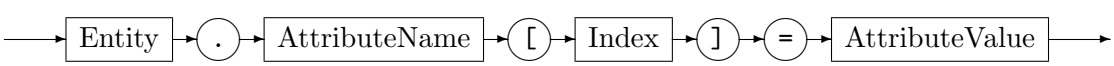
asks the user to select a graph element in yComp; double clicking any node will do fine.

25.6 Attribute Assignment and Graph Manipulation

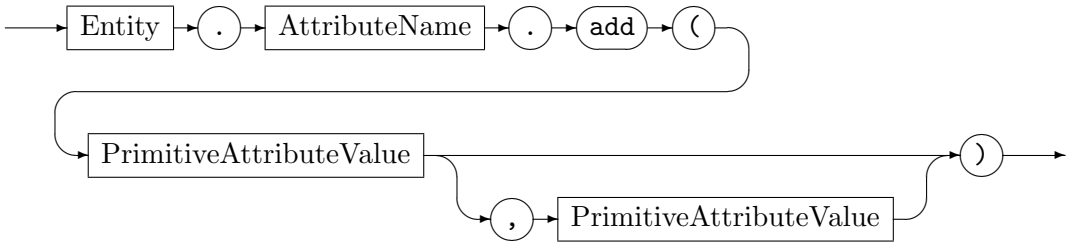
Graph manipulation commands alter existing graphs; they allow to retype and delete graph elements and change attributes. Creating elements was already introduced in the previous section. These are tasks which are or at least should be carried out by the rules of the rule language in the first place. On shell level they are available and mainly used as elementary instructions in creating an initial graph, in exporting and importing a graph, as well as in change recording and replaying.



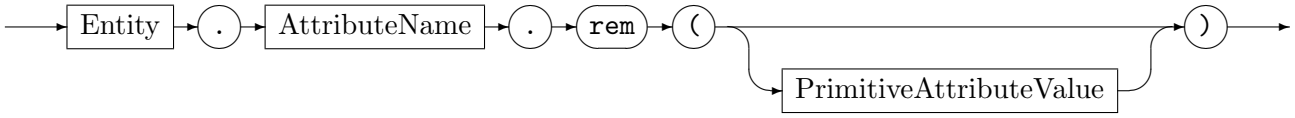
Set the attribute *AttributeName* of the graph element or internal class object *Entity* to the value *AttributeValue* (for the different possible attribute values see above).



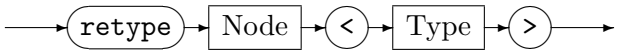
Overwrite the value in the array or deque or map attribute *AttributeName* of the graph element or internal class object *Entity* at the integer position or key value *Index* with the value *AttributeValue*.



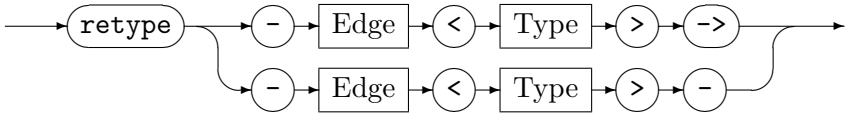
Add the value *PrimitiveAttributeValue* (for the different possible primitive attribute values see above) to the set valued attribute *AttributeName* of the graph element or internal class object *Entity* or add the key-value pair consisting of the two *PrimitiveAttributeValue*s to the map valued attribute *AttributeName* of the graph element or internal class object *Entity*. Or add the value *PrimitiveAttributeValue* to the end of the array/deque valued attribute *AttributeName* of the graph element or internal class object *Entity* in the one parameter case or insert the *PrimitiveAttributeValue* into the array/deque valued attribute *AttributeName* of the graph element or internal class object *Entity* at the index given by the second parameter in the two parameter case.



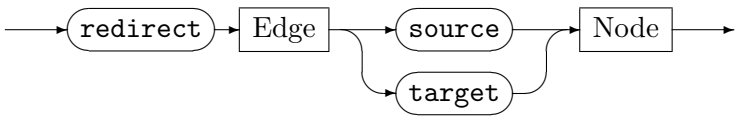
Remove the value *PrimitiveAttributeValue* from the set valued attribute *AttributeName* of the graph element or internal class object *Entity* or remove the key *PrimitiveAttributeValue* from the map valued attribute *AttributeName* of the graph element or internal class object *Entity* or remove the index *PrimitiveAttributeValue* from the array valued attribute *AttributeName* of the graph element or internal class object *Entity* or remove the index *PrimitiveAttribute-Value* from the deque valued attribute *AttributeName* of the graph element or internal class object *Entity* or remove the end element from the array valued attribute *AttributeName* of the graph element or internal class object *Entity* in the zero parameter case or remove the first element from the deque valued attribute *AttributeName* of the graph element or internal class object*Entity* in the zero parameter case.



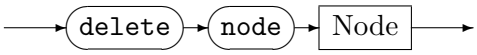
Retypes the node *Node* from its current type to the new type *Type*. Attributes common to initial and final type are kept. Incident edges are kept as well.



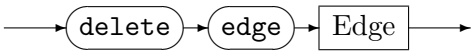
Retypes the edge *Edge* from its current type to the new type *Type*. Attributes common to initial and final type are kept. Incident nodes are kept as well.



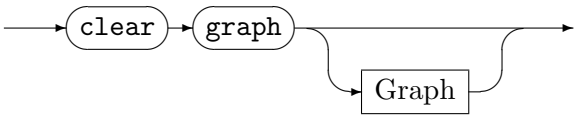
Redirects the edge *Edge* from the old source or target node to the new source or target *Node* given.



Deletes the node *Node* from the current graph. Incident edges will be deleted as well.



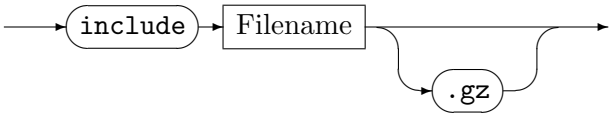
Deletes the edge *Edge* from the current graph.



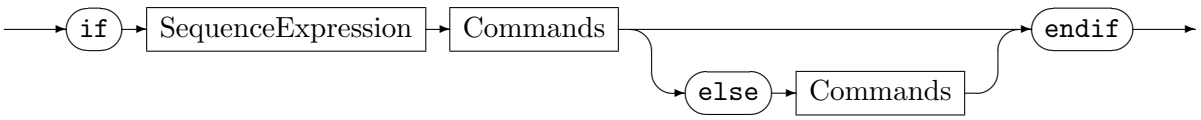
Deletes all graph elements of the current working graph resp. the graph *Graph*.

25.7 Inclusion and Conditional Execution

You may include further shell scripts, or execute parts of a shell script conditionally.



Executes the GrShell script *Filename* (which might be zipped). A GrShell script is just a plain text file containing GrShell commands. They are treated as they would be entered interactively, except for parser errors. If a parser error occurs, execution of the script will stop immediately.



If the sequence expression evaluates to true, the following command lines are executed, until the corresponding `else` or `endif` is reached. If the sequence expression evaluates to false, the following command lines are skipped, until the corresponding `else` or `endif` is reached. In case an `else` is given, the command lines following the else until the endif are executed in the first case, or skipped in the second case. The nature of shell execution is command-line based, without syntactic nesting. The `if` command is an exception, a corresponding amount of `endif` commands is required, and applies to its syntactically preceding if command, as does the `else` command.

25.8 File System Commands and External Shell Execution

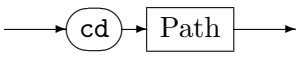
Here we describe commands for inspecting and changing the working directory, and for executing an arbitrary command line in the shell of the hosting OS.



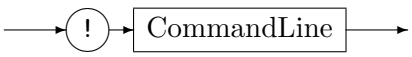
Prints the path to the current working directory.



Lists the directories and files in the current working directory, files relevant to GrGen are printed highlighted.



Changes the current working directory to the path given.



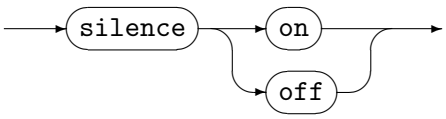
CommandLine is an arbitrary text, the operating system attempts to execute.

EXAMPLE (161)

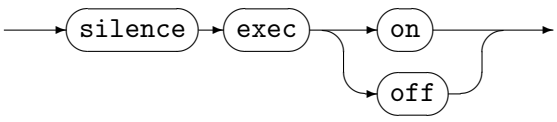
On a Linux machine you might execute

```
1 !sh -c "ls␣|␣grep␣stuff"
```

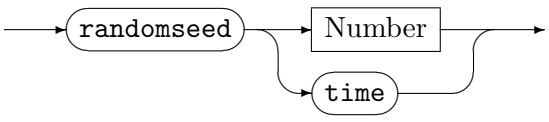
25.9 Shell and Environment Configuration



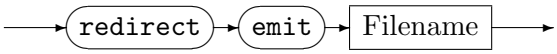
Switches the new node/edge created/deleted messages on(default) or off. Switching them off allows for much faster execution of scripts containing a lot of creation commands.



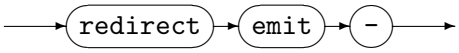
During non-debug sequence execution every second, match statistics are printed to the console; they allow you to assess the progress of long-running transformations (this functionality is *not* available in the GGrShell). With this command they can be disabled (or enabled again). Switching them off may be of interest if your own debug messages printed via emit from the sequences (or rules) should not be disturbed.



Sets the random seed to the given number for reproducible results when using the $-operator-prefix or the random-match-selector, whereas time sets the random seed to the current time in ms.

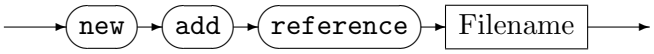


Redirects the output of the emit-statements (but not the emitdebug-statements) in the rules from stdout to the given file.

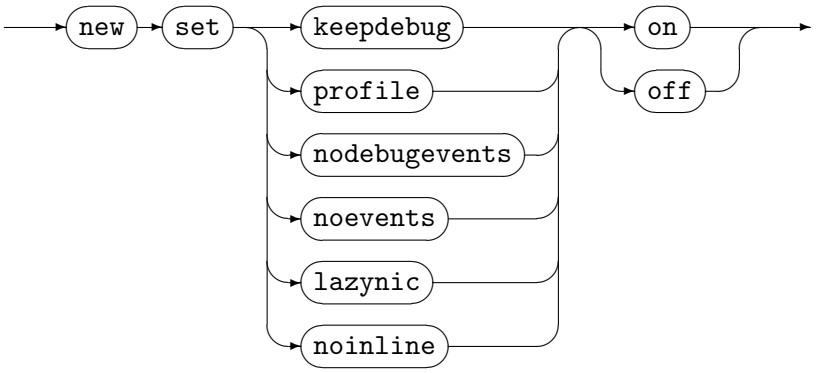


Redirects the output of the emit-statements in the rules to stdout (again).

25.10 Compilation Configuration



Configures a reference to an external assembly *Filename* to be linked into the generated assemblies, maps to the `-r` option of `grgen.exe` (cf. 29.1).

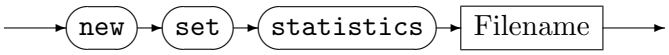


Configures the compilation of the generated assemblies to keep the generated files and to add debug symbols (includes emitting of some validity checking code), or configures the generation of the matchers. The latter in order to include profiling information, or not to emit debug event firing (mostly action events), or not to emit event firing (attribute change events), or to either execute negatives, independents, and conditions only lazily at the end of matching (normally as soon as possible), or to never inline subpatterns. Those flags are mapped to the `-keep` and the `-debug` options, or to the `-profile`, or to the `-nodebugevents` or the `-noevents`, or to the `-lazynic`, or to the `-noinline` options of `grgen.exe` (cf. 29.1).

In addition to setting the compilation flags, the debug events firing flag also changes the debug event firing of the interpreted sequences at runtime (thus also this part of event firing).

When profiling is turned on, the number of search steps carried out is printed to the console after each sequence execution. A search step is a binding of a graph element to a pattern entity in case there are at least potentially several choices available. Fetching an element by type chooses from all elements of that type in the graph, following an incident edge chooses from all incident edges of the corresponding node, and matching by storage access chooses from all elements in that storage, whereas getting the source or target node from an edge or mapping with a storage map just grabs the target element from the source element. (Each binding from a choice counts as one step, the non-choice-bindings are not counted.) Profiling the search steps allows you to assess the work needed for a transformation, to find the hot spots worth optimizing. The less search steps are needed to find a match the better (you typically want to efficiently find patterns and minimize the amount of search needed in order to do so).

The flags are applied when the actions are generated anew because the model or rule files changed. They are not when you just switch them in the shell script. It's your responsibility to delete the old generated dlls when you change those options (by switching them, or by introducing them differently from the compiler options used to generate the dlls)!

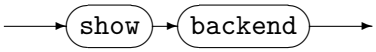


Configures the compilation of the generated assemblies to use the statistics file specified, yielding pattern matchers adapted to the class of graphs described in that file. Maps to the `-statistics` option of `grgen.exe` (cf. 29.1, and see 25.12 on how to get such statistics).

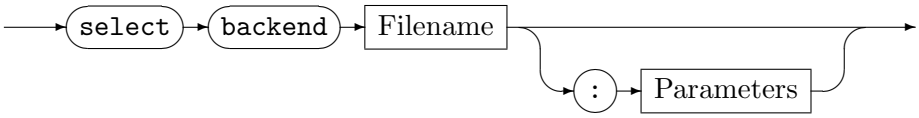
25.11 Backend, Graph, and Actions Selection

Backend

GrGen.NET is built to support multiple backends implementing the model and action interfaces of libGr. This is roughly comparable to the different storage engines MySQL offers. Currently only one backend is available, the libGr search plan backend, or short LGSPBackend.

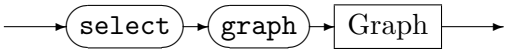


List all the parameters supported by the currently selected backend. The parameters can be provided to the `select backend` command.

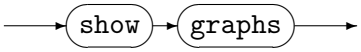


Selects a backend that handles graph and rule representation. *Filename* has to be a .NET assembly (e.g. `lgspBackend.dll`). Comma-separated parameters can be supplied optionally; if so, the backend must support these parameters. By default the LGSPBackend is used.

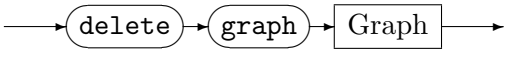
Graph



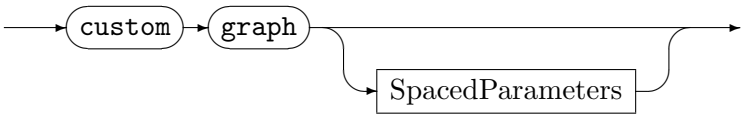
Selects the current working graph. This graph acts as *host graph* for graph rewrite sequences (see also Sections 1.5 and 25.4). Though you can define multiple graphs, only one graph can be the active "working graph".



Displays a list of currently available graphs.

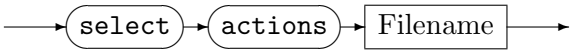


Deletes the graph *Graph* from the backend storage.

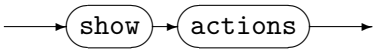


Executes a command specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSP backend see Sections 25.12).

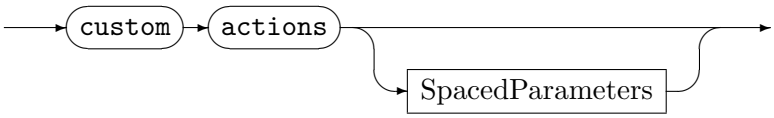
Actions



Selects a rule set. *Filename* can either be a .NET assembly (e.g. "rules.dll") or a source file ("rules.cs"). Only one rule set can be loaded simultaneously.



Lists all the rules of the loaded rule set, their parameters, and their return values. Rules can return a set of graph elements.



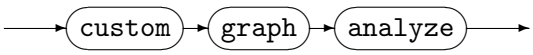
Executes an action specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSPBackend see Section 25.12).

25.12 LGSPBackend Custom Commands

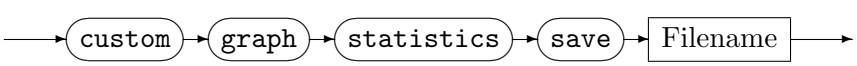
Don't be shy to use custom commands just because they are custom. The search plan generation and explanation offered by them are of outstanding importance for achieving high performance solutions. Leading directly to high-performance solutions by adapting the pattern matchers to the specifics of the host graph. Or indirectly by explaining the chosen search plan so you can inspect it for the spots where the planner was not able to circumvent massively splitting passages; you may have to rethink your solutions regarding those, with maybe a change in the modelling, a change in the pattern, or even imperative code with hash set intersections/joins.

The LGSPBackend supports the following custom commands, separated into the realms of the graph and the actions.

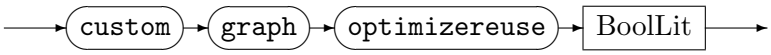
Graph related custom commands



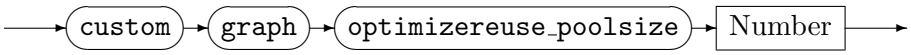
Analyzes the current working graph. The analysis data provides vital information for efficient search plans. Analysis data is available as long as GrShell is running, i.e. when the host graph is manipulated, the analysis data is still available but outdated (which does not pose an issue unless the graph was changed massively regarding the relative number of elements per type or the connectedness-by-type relation).



Write the statistics of the last analyze to the specified statistics file (the graph must have been analyzed before this command is called). This way you can save the statistics of a characteristic graph of your domain and compile pattern matchers well adapted to that class of graphs straight from the beginning, saving you the costs of online analysis and matcher compilation. See 29.1 or 25.10 for explanations on how to do this.

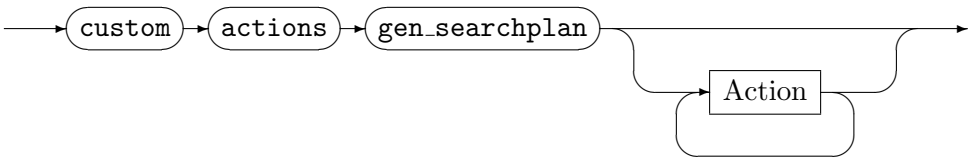


If set to false it prevents deleted elements from getting reused in a rewrite (i.e. it disables a performance optimization). If set to true (default), new elements may not be discriminable anymore from already deleted elements using object equality, hash maps, etc.

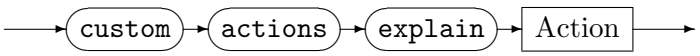


Sets the size of the pool storing deleted elements for reuse to *Number* (default: `10`).

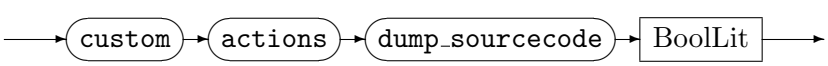
Action related custom commands



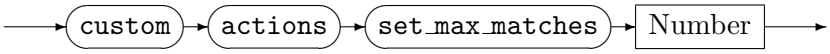
Creates a search plan (and executable code from it) for each rewrite rule *Action* using the data from analyzing the graph (`custom graph analyze`). Otherwise a default search plan is used. If no rewrite rule is specified, all rewrite rules are compiled anew. Analyzing and search plan/code generation themselves take some time, but they can lead to (massively) faster pattern matching, thus overall reduced execution times; the less uniform the type distribution and edge wiring between the nodes, the higher the improvements. During the analysis phase the host graph must be in a shape "similar" to its shape when the main amount of work is done (there may be some trial-and-error steps at different time points needed to get the overall most efficient search plan.) A search plan is available as long as the current rule set remains loaded. Specify multiple rewrite rules with one command instead of using multiple commands, one for each rule, in order to improve search plan generation performance (this avoids superfluous compilation runs).



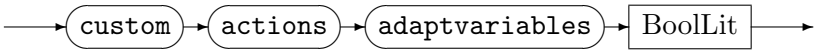
Shows the search plan currently in use for the *Action*, plus the subpatterns called by it. The search plan highlights *how* the pattern gets matched, and thus allows to understand the performance characteristics of executing the action. It is a list of search commands (with commands not doing real matching work shown in parenthesis), which is executed from top to bottom; for more on the search commands have a look at section 35.4. The search plan helps in optimizing for performance, as it shows the effects of pattern changes, e.g. the introduction of parameters, or the annotation of pattern elements with priorities (cf. 34.10), and of changes of the statistical data (stemming from a re-analyzation of a modified graph, or loading of analysis data). This is an inspection tool comparable to the `explain` command offered by SQL-databases, which shows the search plan for a query.



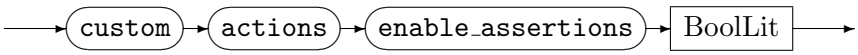
If set to true, C# files will be dumped for the newly generated searchplans (similar to the `-keep` option of the generator; defaults to `false`).



Sets the maximum amount of possible pattern matches to *Number*. This command affects the expression `[Rule]`. If *Number* is less or equal to zero, the constraint is reset.



If set to true (default), variables are cleared (nulled) if they contain graph elements which are removed from the graph, and rewritten to the new element on retypings (only graph-global variables, not sequence variables). This saves us from outdated and dangling variables at the cost of listening to node and edge removals and retypings. Setting it to false improves performance, but can lead to zombie elements hanging out in the graph-global variables (and a lot of confusion in case they are reused, as it is possible when `optimizereuse` is not set to `false`, see graph custom commands above).



If set to true, assertions are enabled (default: `false`), cf. Section 16.7.2.

CHAPTER 26

PERSISTENT STORAGE

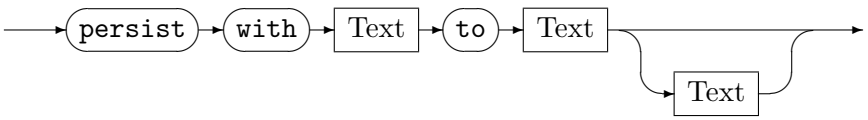
The support for persistent storage can be distinguished into

- the import/export functionality, that allows serializing/deserializing an in-memory graph at a dedicated point in time
- the persistent graph, which is an in-memory graph mirrored to a database (and mirrored back at opening, by a persistence provider), changes to it are written through to the database
- the record/replay functionality, that lies in between the two, starting with a full graph dump to a `.grs` file, then persisting ongoing changes to the `.grs` file.

26.1 Persistent Graph Creation

The `new graph` command (cf. 25.3) may be extended to create or open a persistent graph.

PersistenceSuffix



The first *Text* specifies the persistence provider to use, which has to be given as a dll filename. With `libGrPersistenceProviderSQLite.dll`, the database (management system) SQLite[GPB+22] is used; this dll is as of now the only offered persistence provider. The second *Text* specifies the database connection string to use, which must specify the name of the database file to use, and should set the version. The optional third *Text* may specify additional persistent graph parameters.

EXAMPLE (162)

The command

```
new graph "persistentgraph/persistentgraph_attributed"\
persist with "libGrPersistenceProviderSQLite.dll"\
to "Data Source=testgraphfilename_attributed.db;Version=3;"
```

creates a new graph from the model/actions specified by the `persistentgraph_attributed.grg` file, persisting it with the persistence provider `libGrPersistenceProviderSQLite.dll` to the `testgraphfilename_attributed.db` database file. The command is from the example `persistentgraph_attributed.grs`, it can be found in the `tests/persistentgraph` directory.

NOTE (49)

When using a persistent graph, you are strongly advised to employ persistence provider aka database transactions, at least when non-minor graph or attribute changes occur. Executing a write-heavy sequence without an enclosing database transaction which causes the implicit use of many small transactions is about 500 times slower than the use of an enclosing database transaction. See 22.6 for an explanation of the relevant constructs in the sequences language.

NOTE (50)

The persistent graph also has its price: executing a write-heavy sequence with nearly no matching on a persistent graph with an enclosing database transaction is about 15 times slower than executing the sequence on an in-memory only graph.

Persistent Graph Design and Consequences

The persistent graph is implemented in large parts by a persistence provider, currently only the SQLite based `libGrPersistenceProviderSQLite.dll` is available (the `LGSPPersistentNamedGraph` implementing the `INamedGraph` interface from `libGr` is only a very thin extension of the `LGSPNamedGraph`, the real work is delegated to its persistence provider).

The workflow is: at session begin, when the persistent graph is created, the database is read (or created if it does not exist yet), then ongoing changes to the graph are persisted to the database, until the session ends. At the beginning of the next session, the state at the end of the previous session is still available.

Matching/reading occurs entirely in-memory, the database is read only once, when the persistent graph is loaded. The writing of changes (to the graph structure as well as to the attributes of its entities) is realized by listening to the graph change events (and writing small change sets, typically single values, even in the case of containers). This design allows for quick pattern matching and for quick update writing, but comes at the price of slow loading (due to initial cleanup/container change integration at loading, and esp. compared to database-only operations with on-demand reading/writing, or a lazy loading/caching approach).

A mark and sweep like garbage collector is employed directly after the initial graph reading, it finds out which of the known objects and graphs are still reachable from the host graph, and which of the zombie nodes/edges are still reachable from the host graph. The ones that are not referenced anymore are deleted. Also container compactification occurs: container attributes are stored as a log of change commands (like add and rem), at initial loading, the changes are replayed, and the by-then reached most current state is written anew.

Note that only the objects that are reachable from the host graph on are stored, in contrast to all the objects created, even if they are accessible from e.g. sequence variables. Nodes/edges that are deleted or retyped turn into zombies, references to them dangle afterwards. They may still be available in attributes of node/edge type or attributes of container of node/edge type (in nodes or edges or internal class objects). These still referenced zombies are reported as warnings at persistent graph loading. The attribute containing them should have been assigned null or they should have been removed from the container attribute containing them before the removal from the graph or the retype (a retype keeps the position, name, and unique id, but the retyped element has a different .NET object identity than the previous element).

The persistent graph as it is implemented by the SQLite persistence provider is constrained in certain ways compared to the regular graph:

- when attributes are used that contain nodes/edges (graph element references), the graph elements of the graph must reference the containing graph, i.e. `node edge graph;` must have been declared (cf. 18.4) – in exchange, the persistent graph is capable of coping with references to graph elements from other graphs than the one the reference is contained in – this is *not* supported by the GRS exporter/importer
- the persistent graph does not support the reuse optimization (cf. 25.12, `optimizereuse` that recycles deleted elements)
- the persistent graph does not support parallel sequence execution (cf. 23.2)
- external attribute types (cf. 34.1) are not supported (the external emitting and parsing (cf. 34.6) of the GRS import/export is not supported)

The SQLite persistence provider uses the following table mapping: the graph structure is stored in topology tables (`nodes`, `edges`, `graphs`, `objects`) with one row per entity, the entities (nodes, edges, objects) are stored as rows in per-type table with the attributes as columns, besides container attributes, which have their own tables (additionally, type information is stored in `types` and `attributeTypes` tables, plus a bit of metadata in the `metadata` table). A minimal amount of database indices are used (initial reading occurs by full table scans), as pattern matching is not carried out against the database – the indices to achieve fast lookups and matching are all kept in-memory, the same runtime structures are (re-)used that are used for a non-persistent graph.

You could change the persistent graph offline by carrying out SQL commands, even though this is discouraged (strongly, unless you know what you are doing).

26.1.1 Model Update

Before the graph can be loaded from the database, it may be necessary to update the model stored in the database to the model used by the current host graph (the graph in the database adheres to the (meta) model from the database, not necessarily the current (meta) model from the model file/assembly; a loading is only possible in case of a compatible new model/-compatible changes).

The persistence provider checks for changes by comparing the models, and depending on them

- aborts the update,
- asks for user confirmation,
- auto-updates (or initializes),
- does nothing.

In case entity types were removed from the model that still have instances in the database, and/or enum types were removed from the model that still appear as types of attributes in the database, the update is aborted (and the database left unchanged) — this behavior prevents an accidental loss of data. The update is also aborted when types have changed their kind and the aforementioned conditions apply, e.g. a `node class Foo` was switched to an `edge class Foo` (and the database contains instances of its type), or an enum `Bar` was switched to an object `class Bar` (and the database contains values of its type, determined by attributes of its type), they are modeled as a type deletion in a first step, followed by a type introduction in a second step (types are not only defined by their name, but also their kind (but not their content, content changes are handled as single attribute changes)). You have to revert to an old model in order to delete the instances or the attribute of their types, before you can proceed. Then you can also migrate the data you don't want to lose, see 26.4 for more on

this topic. Removing entity types that don't have instances or enum types that don't appear as types of attributes in entity types is safe and carried out automatically (because you only lose the corresponding meta data/type information).

As an aid for model migration (if you don't have access to the original model anymore), the model stored in the database is written to a (temporary) model file in case of an abort (whose name starts with `partial` and contains the database name) – but note, that this is *not* the original model, for one is it only a partial model (lacking e.g. methods or indices), for the other is the inheritance hierarchy flattened, all classes contain all the attributes from their supertypes, lacking the extends specification.

NOTE (51)

An update may fail directly after you deleted all instances of relevant nodes/edges. You have to open the graph first and wait for completion of garbage collection, then you can delete the types. The next time you open the graph, the model will be updated (successfully).

NOTE (52)

We speak of model updates, while in fact, you may also open a graph with a different model, by accident or intentionally. Still, it is seen as if you modified the previous model to the current state, i.e. the difference from the database to the current models is computed and acted upon (in order to synchronize the database to the current model).

In case attributes or enum cases were deleted, the user is asked to confirm the update, which would remove besides the attribute from the type also all the attribute values in the entities of the type stored in the database, potentially causing data loss, or move the persistent graph to a broken state, if the enum cases still appear in attributes (as attribute values). This holds also for attributes that changed their type, e.g. an attribute `bar:int` being turned into a `bar:string`, in the same `class Foo`, they are modeled as an attribute removal in a first step, followed by an attribute introduction in a second step. When the user accepts the update, it is carried out and the attributes are gone for good, their contained values lost, and the enum cases removed – it may be the graph is in an unreadable state thereafter (unless further measures are taken), if the removed enum cases are still contained in attributes (see 26.4 for how to handle this). When the user denies the update, it is aborted, like in the case of deleted entity types with instances or deleted enum types with attributes, and the graph is neither changed nor opened (so that you can roll back to an older model state and carry out the required migrations). The interactive querying of the user can be prevented by *PersistentGraphParameters*, they will be introduced later in the section.

In case of changes that are only extensions, the update will be automatically carried out. This esp. holds when new entity or enum model types were introduced (they will be added to the database model and can be used thereafter), when new attributes were introduced to entity types (they will be initialized to their GrGen.NET default values in the instances of their containing entity types in the database (*not* to the default initializations specified in the GrGen.NET model file, cf. 4.2.2)), and when new enum cases were introduced (thereafter, they can appear in attributes whose type is that enum type). For kind changes of types and type changes of attributes, the second step of introducing the new types of the name is such a safe extension, after the first step of deleting the old type of the name passed.

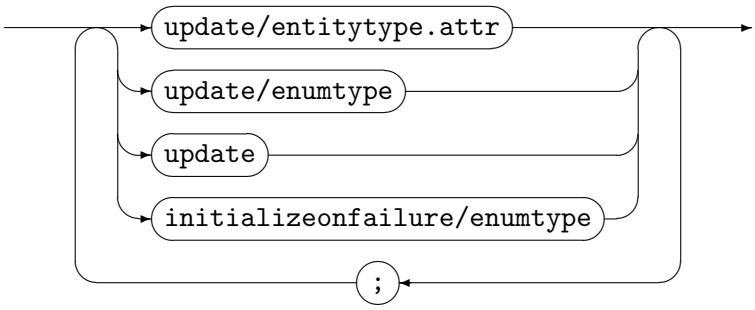
Upon first opening of a persistent graph, the database will be initialized to the model from the (current) graph.

In case the model is unchanged compared to the last time the database was opened, or only parts without relevance to persistent storage (like indices, which are memory-only)

changed, the graph is simply read (this is of course also the follow-up step after the automatic changes and the confirmed changes were carried out).

Persistent Graph Parameters

PersistentGraphParameters



The persistent provider supplying the persistent graph can receive inputs in the form of persistent graph parameters, the ones that are currently supported allow to influence the model update.

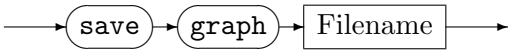
The `update` parameters allow to exclude attributes or enum cases from the update the user has to confirm. The parameter `update/entitytype.attr` causes an auto-confirmation (and thus update) of the deletion/type changing of the attribute `attr` from graph element or class object type `entitytype` during persistent graph opening. The parameter `update/enumtype` causes an auto-confirmation (and thus update) of the enum type `enumtype` for all deleted enum cases. The parameter `update` causes an auto-confirmation (and thus auto-update) of all attribute deletions and type changes and enum case deletions (but is potentially dangerous and thus not recommended). The parameter `initializeonfailure/enumtype` causes an initialization of the attributes of type `enumtype` that fail at loading because their enum case is not available anymore with the default value of the enum type (the first case of value 0 which is typically the first case in the enum specification list, or the first case from the enum specification list if no enum case of value 0 exists).

These parameters allow to deploy a model migration script including a silent model update to user machines (see 26.4 for more on model migration).

NOTE (53)

The persistent graph supports references to nodes/edges that are not contained in the graph of the node/edge that stores them as attributes (so e.g. `graphof(n.a) != graphof(n)` is allowed for `n` and `n.a` being (references to) nodes). This holds neither for the import/export commands, nor for the debugger or the emitting functionality (as of now).

26.2 Graph Input and Output

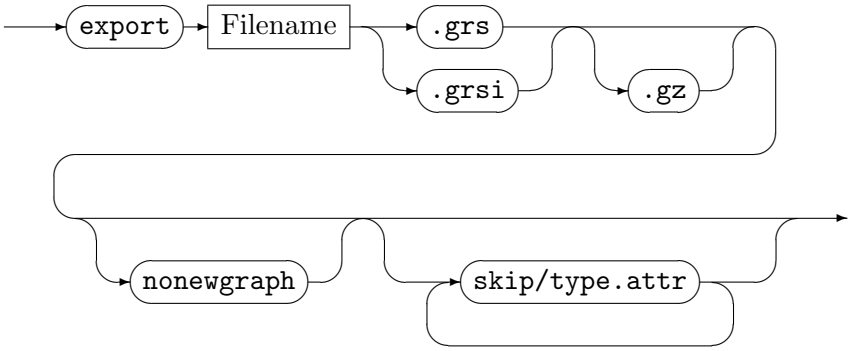


Dumps the current graph as GrShell script into *Filename*. The created script includes

- selecting the backend
- creating a new graph with all nodes and edges (including their persistent names)
- restoring the (graph global) variables

- restoring the visualisation styles

but not necessarily using the same commands you typed in during construction. Such a script can be loaded and executed by the `include` command (see Section 25.7).



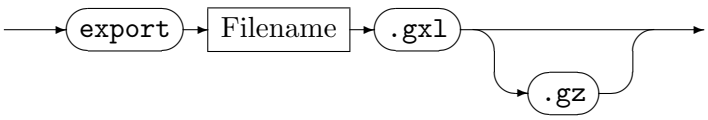
Exports an instance graph in GRS (.grs/.grsi) format, which is a reduced GrShell script (it can get imported and exported on API level without using the GrShell, see Section 33.4). This is the recommended standard format (it is lightweight, human-readable and editable, and supported by an optimized importer). The file contains a `new graph` command, followed by `new node` commands, followed by `new edge` commands. If the `.gz` suffix is given the graph is saved zipped. The export is only complete with the model of the graph given in the `.gm` file. Exporting fails if the graph model contains attributes of `object`-type; you may add support for storing them, too, see 34.6 for more on this.

When the optional parameter `nonewgraph` is given, the initial `new graph` command at file begin is omitted. Such a file cannot be `import`ed, but only `include`d, as it is incomplete. You have to ensure an empty graph of correct model exists before you can include a file exported this way.

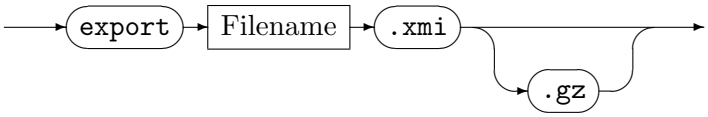
The skip parameters allow to exclude attributes from the node/edge attribute initializer lists of the `new node` or `new edge` commands. The parameter `skip/type.attr` causes omission of attribute `attr` from graph element type `type` during graph serialization. This way you can export an again importable graph if you intend to remove some attributes – otherwise import would fail due to an unknown attribute getting initialized (in the file exported adhering to the old graph model that is not existing in the new graph model).

The native `.grs` format is the best support format, but still limited as of now regarding the full runtime object graph that can be stored in-memory, it does not support node/edge attributes in the nodes/edges of a graph that reference nodes/edges from another graph, or node/edge attributes in an internal class object that can be reached from different graphs (node/edge attributes are considered an ugly duckling). You must refrain from using them in this manner (at least as of now), which could happen quite easily when you store a subgraph created by `insertInduced`/`insertDefined` in the host graph, with node/edge attributed elements, because the elements are shallowly cloned, referencing the original elements – it's your task to overwrite them with `null` or valid versions.

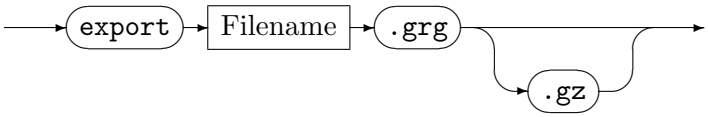
The `save` command from above is for saving a GrShell session including graph global variables and visualization commands, the goal of the `export` command is simply persistent storing of graphs; esp. for applications that use GrGen.NET to get an algorithmic core, but are not built on the GrShell.



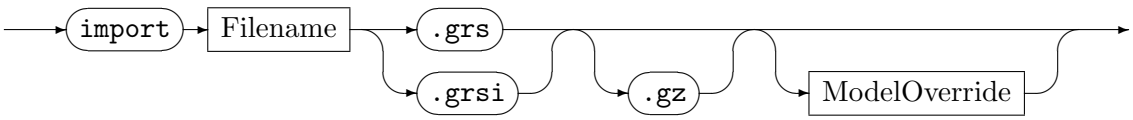
Exports an instance graph and a graph model in GXL format [WKR02, HSESW05], which is somewhat of a standard format for graphs of graph rewrite systems, but suffers from the well-known XML problems – it is barely human-readable and editable, and bloated. It is supported by GrGen.NET as exchange format for inter-tool operability. Exporting fails if the graph model contains attributes of container or `object`-type. If the `.gz` suffix is given the graph is saved zipped.



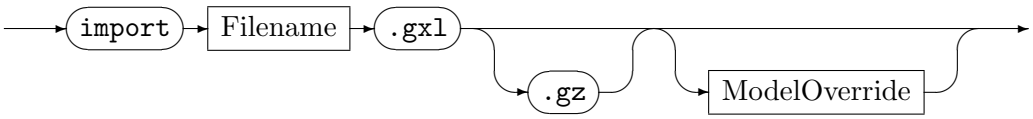
Exports an instance graph in .XMI format. XMI files as written by the Eclipse Modeling Framework (EMF) are a standard format in the model transformation community (together with ecore files for the model). It suffers from the XML problems explained above, in addition it can be even characterized as overly complex and baroque, and it requires some metamodel mapping. It is supported by GrGen.NET as exchange format for inter-tool operability. The metamodel is assumed to stem from a previous import of an ecore file, with its specific way of mapping `.ecore` to `.gm`, i.e. with an underscore prefix, a node type prefix for the edge types, and the `[containment=true]` annotation at the edges that express containment, which is needed so that they are written with XML node containment. If the `.gz` suffix is given the graph is saved zipped.



Exports an instance graph in GRG format, i.e. as one GrGen rule with an empty pattern and a large modify part. There is no importer existing, this format is not for normal use as storage format! If the `.gz` suffix is given the graph is saved zipped.



Imports the specified graph instance in GRS (.grs/.grsi) format (the *reduced* GrShell script, a `saved` graph can only be imported by `include` due to commands not supported by the importer (but an exported graph can be imported by `include`, too)). The graph model referenced in the .grs/.grsi must be available as `.gm`-file. If a model override of the form `Filename.gm` is specified, the given model will be used instead of the model referenced in the GRS file. If a model override of the form `Filename.grg` is specified, the model(s) of the given rule file will be used instead of the model in the GRS file. If the `.gz` suffix is given the graph is expected to be zipped.



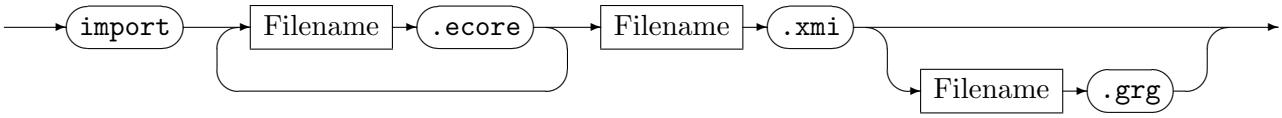
Imports the specified graph instance and model in GXL format. If a model override of the form `Filename.gm` is specified, the given model will be used instead of the model in the GXL file. If a model override of the form `Filename.grg` is specified, the model of the given rule file will be used instead of the model in the GXL file. The `.gxl`-graph must be compatible to the `.gm`-model/`.grg`-model. If the `.gz` suffix is given the graph is expected to be zipped.

NOTE (54)

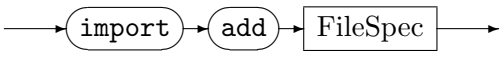
Normally you are not only interested in importing a GXL graph (and viewing it), but you want to execute actions on it. The problem is that the actions are model dependent. So, in order to apply actions, you must use a model override, which works this way:

1. `new graph "YourName.grg"`
   This creates the model library lgsp-YourNameModel.dll and the actions library lgsp-YourNameActions.dll (which depends on the model library generated from the `"using YourName;"`).
2. `import InstanceGraphOnly.gxl YourName.gm`
   This imports the instance graph from the .gxl but uses the model specified in YourName.gm (it must fit to the model in the .gxl in order to work).
3. `select actions lgsp-YourNameActions.dll`
   This loads the actions from the actions library in addition to the already loaded model and instance graph (cf. 25.4).
4. Now you are ready to use the actions.

As of version 3.0beta you can specify a `.grg` as model override; basically it does what the given enumeration does.



Imports the specified graph instance in XMI format and the models in ecore format. They can't be imported directly, as GrGen.NET is not built on EMF. Instead, during the import process an intermediate `.gm` is written which is equivalent to the `.ecore` given – you may inspect it to see how the content gets mapped. (The importer maps packages to GrGen packages, classes to GrGen node classes, their attributes to corresponding GrGen attributes, and their references to GrGen edge classes. Inheritance is transferred one-to-one, and enumerations are mapped to GrGen enums. Edge type names are prefixed by the names of the node types they originate from to prevent name clashes for references of same name, and all types are prefixed by an underscore to prevent name clashes with keywords of the rule language. Edge type declarations are annotated with a `[containment=true]` annotation if they originate from a containment reference.) After this metamodel transformation the instance graph XMI adhering to the Ecore model thus adhering to the just generated equivalent GrGen graph model gets imported. Furthermore, you can specify a `.grg` containing the rules to apply (including further rule and using further model files – this way you can use additional custom graph models). Some examples stemming from old GraBaTs/TTC challenges export XMI with emit statements (e.g. the Program-Comprehension example in `examples/ProgramComprehension-GraBaTs09`), this is not needed anymore with the built-in XMI export.

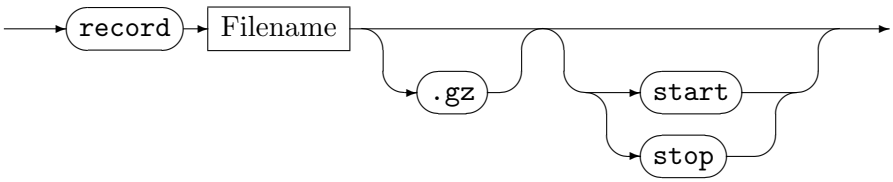


Imports the graph in the specified file and adds it to the current graph (instead of overwriting the old graph with the new graph). The `FileSpec` is of the same format as the file specification in the other import commands.

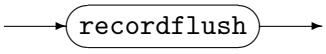
26.3 Graph Change Recording and Replaying

Graph change recording and replaying is available

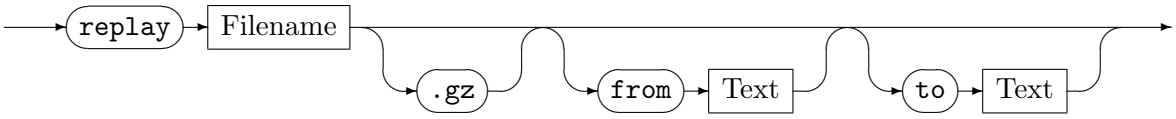
- for one for post-problem debugging, you can execute your transformation, and after a problem occurred inspect the changes that were leading to it
- for the other for ensuring persistence of changes as they happen, in case you are using GrGen.NET as an application-embedded in-memory graph-database



The record command starts or stops recording of graph changes to the specified file. If neither start nor stop are given, recording to the specified file is toggled (i.e. started if no recording to the file is underway or stopped if the file is already recorded to). Recording starts with an export (cf. 26.2) of the instance graph in GRS (.grs/.grsi) format, afterwards the command returns but all changes to the instance graph are recorded to the file until the recording stop command is issued. Furthermore the values given in the `record` statements (cf. 16.7.3) from the sequences are written to the recording (this allows you to mark states). If the `.gz` suffix is given the recording is saved zipped. You may start and stop recordings to different files at different times, every file receives the graph changes and records statements occurring during the time of the recording. Note: As a debugging help a recording does not only contain graph manipulation commands (cf. 25.6) but also comments telling about the rewrites and transaction events which occurred (whose effects were recorded).



Flushes the buffers of the recordings to disk. To be called to guarantee persistence if you use GrGen.NET as a kind of online database, recording the graph changes while running to a redo log.



The replay command plays a recording back: the graph at the time the recording was started is recreated, then the changes which occurred are carried out again, so you end up with the graph at the time the recording was stopped. Instead of replaying the entire GRS file you may restrict replaying to parts of the file by giving the line to start at and/or the line to stop at. Lines are specified by their textual content which is searched in the file. If a *from* line is given, all lines from file begin on including this line are skipped, then replay starts. If a *to* line is given, only the lines from the starting point on, until-excluding this one are executed (i.e. all lines from-including this one until file end are skipped). Normally you reference with `from` and `to` comment lines you write with the `record` statement (cf. 16.7.3) in the sequences, marking relevant states during a transformation process. An example for record and replay is given in `tests/recordreplay`.

26.4 Model Migration

During development, you typically change your graph models (they are called meta models by others – by editing your model files), and want to continue using your existing graphs with the updated models (the existing graphs are contained in an exported file or are stored persistently in a database). Partly this works out of the box (because the graph is compatible to the new model, or because of the update functionality), partly this requires dedicated migration code you have to supply. Also, you may want to carry out the updates on user machines your graph rewrite system is deployed to. We explain here how to handle these use cases / give an overview of the workflow. Note that the model the stored graph adheres to is only implicitly available in case of a serialized `.grs` file (defined by the types and attributes referenced by the `new` graph element commands), and only partially available in case of a persistent graph in the SQLite database (it is explicitly available – albeit in reduced form – in some dedicated types and attributes tables, and implicitly in the layout of the non-topology tables and the values).

Adding types (`node class`, `edge class`, `class`, `enum`) to a model or new attributes to types within the model or new cases to enums works out of the box, the old graph is imported/loaded without problems, the new types simply don't have any instances thereafter, the new attributes are simply initialized to the default values of their types (the GrGen.NET default values, not the values of explicitly specified initializers from the model as introduced in 4.2.2), the new enum cases are not used but just available for usage afterwards. Deletions of entity or enum types and/or attributes and/or enum cases on the other hand does not work out of the box but require preparation, we specify how to handle them in the following (split by file export and database storage).

When you want to delete entity types from the model, you have to first delete their instances from the graph (by graph rewrite rules or sequences or procedures). Then, in case of a serialization file, you have to export the graph, can then remove the types from the model, and import the graph (without issues due to the removed type). In case of a persistent graph in a database, the database has to be closed (the host graph switched to another one or the shell script exited), then re-opened, and closed again. The intermediate re-opening is required so that the garbage collection can remove the deleted entities, which runs only at database opening time but after model updating. Then, you can drop the type from the model, and open the persistent graph (without issues due to the removed type).

When you want to delete enum types from the model, you have to first remove the attributes of that enum type from the entity types containing them (also by graph rewrite rules or sequences or procedures). Then, in case of a serialization file, you have to export the graph, can then remove the types from the model, and import the graph (without issues). In case of a persistent graph in a database, the database has to be closed, then you can remove the enum type from the model, and open the database with the new model (without issues); an intermediate opening is not required in this case.

When you want to delete attributes from the model (i.e. remove them from their containing type), and work with serialization/deserialization, you have to export first with `skip` instructions, see Section 26.2, causing the attributes not to be serialized. Then you can drop them from the model file and import the serialized file without issues (otherwise, import will fail with an unknown attribute message). When you work with a persistent graph stored in a database, you can drop the attributes from the model file, and at the following database opening (persistent graph creation), the persistence provider will ask you for a confirmation that the attributes not available in the model anymore can to be deleted. When confirmed, the database is updated to a version with the attributes dropped, and the graph is loaded (without issues due to the removed attributes). The change has to be confirmed, because the data from the attributes is lost thereafter. You may supply skip-like `update` parameters to the persistent graph at database opening, cf. 26.1.1, which auto-confirm the changes, saving you/the user from the necessity to confirm them (this is esp. helpful if you want to supply a

migration to user machines).

When you want to delete enum cases from their containing enum, you have to first remove all their appearances in attributes of this enum type (in the entities of the entity types that contain these attributes, again with graph rewrite rules or sequences or procedures), and export the graph when working with serialization/deserialization, or close the database when working with a persistent graph. Then you can remove the enum cases from the enum type, and import the exported graph (without issues) or open the database (without issues). In the latter case, the model updater will ask you to confirm the enum case removal, you can safely accept then. You may supply skip-like `update` persistent graph parameters at database opening, cf. 26.1.1, which auto-confirm the changes, saving you from the necessity to confirm them.

In case you forgot to delete before exporting, you will receive enum parsing errors upon import (you could then revert the model / temporarily insert the cases again, or – even if ugly – apply some text processing to remove them from the `.grs` file as a remedy). In case you forgot to delete the enum case values before removing the enum cases from the model specification file and re-opening the database, and you then confirmed prematurely the changes (manually or by auto-confirming persistent graph parameters), you get a database that cannot be loaded anymore (loading will fail at parsing the enum cases that are still in the database but not declared in the enum anymore). By specifying `initializeonfailure` instructions in the persistent graph parameters, cf. 26.1.1, you can make the database load again. The attributes containing removed enum cases get initialized to the default value of the enum type then. But only in-memory, the old values stay in the database, you have to overwrite them by assigning some current value to the corresponding enum attributes and close the database so it is in a valid state again (another – very ugly – solution would be to update the attributes in the corresponding tables by SQL statements — enum attributes are stored as text columns containing the enum case text (at least in the current version)).

A kind change of a type or a type change of an attribute is handled as a deletion followed by an addition, so you have to take care of the deletion part (additions don't cause issues).

In general: if you want to reuse stored graphs when the model changes, you have to take care of the dependencies (supplying migration code that works step-wise along the dependencies). If you forgot to delete the instances / attributes of a type before deleting the type itself (or instances of enum cases before the enum cases), you have to revert the model or temporarily insert the type (the enum cases) again manually, to successfully load the export/database with the model in order to carry out the required deletions (the partial model written by the database could be of help regarding this).

CHAPTER 27

VISUALIZATION AND DEBUGGING

This chapter gives an introduction to the visualization and debugging capabilities of GrGen.NET. They are offered by a

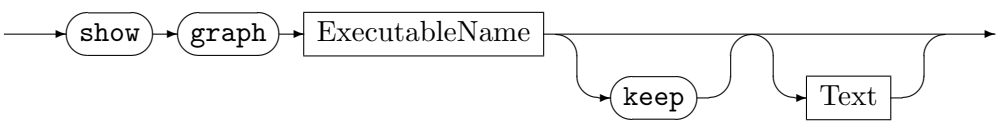
- console debugger (either rendered on a stdout-console or a WindowsForms console like display) or a
- GUI debugger (containing an input/output console).

Together with the

- external graph viewer applications extMSAGLExt or yComp, or an
- internal builtin graph viewer based on the MSAGL library, that can also be used in your own application via an API,

and the configuration commands offered by GrShell (which is the traditional execution host of the console debugger).

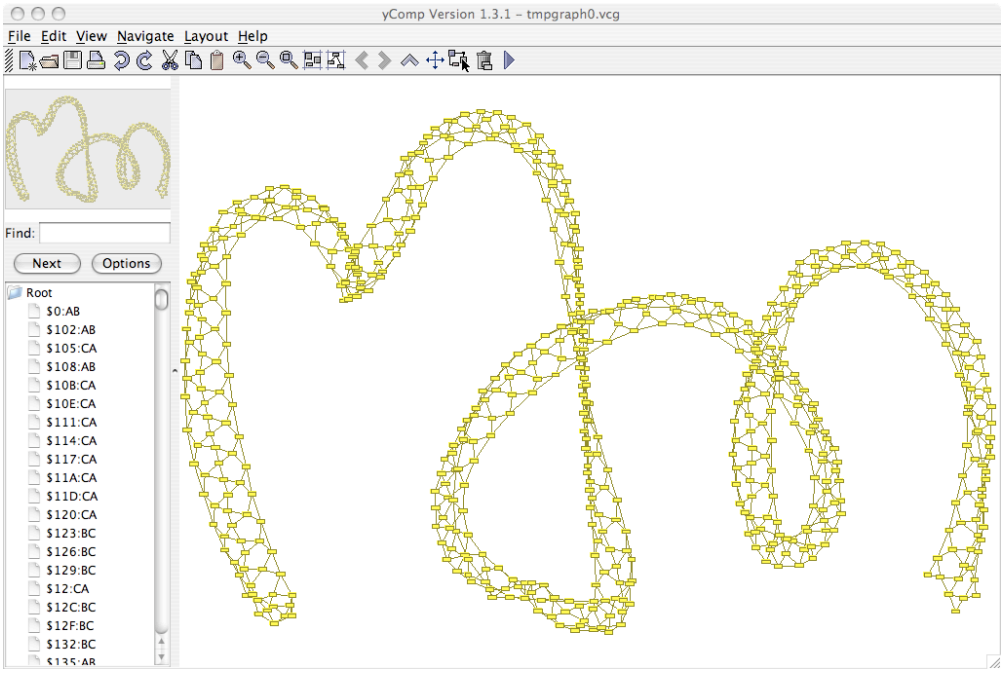
27.1 Graph Visualization Commands (Nested Layout)



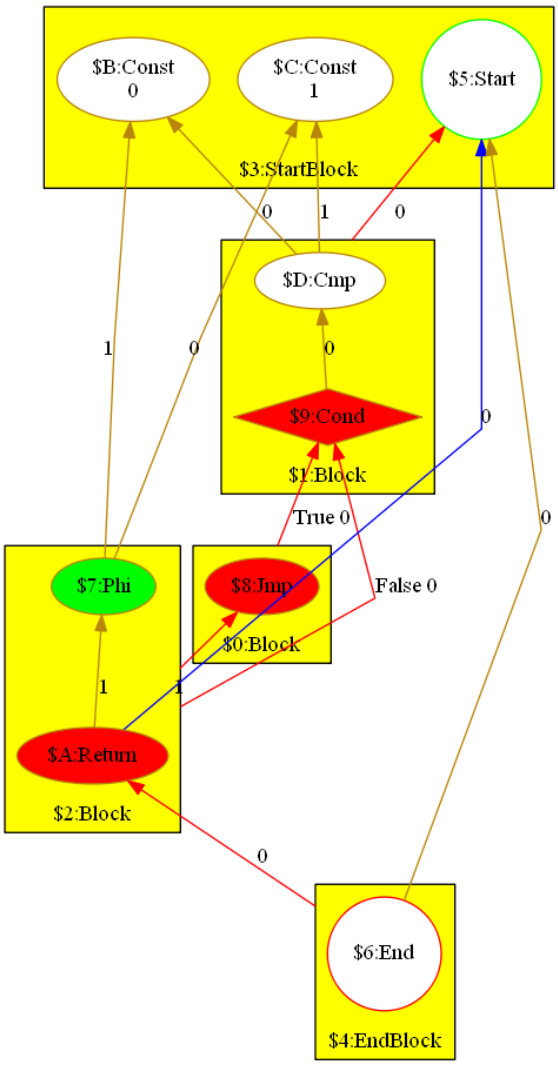
Dumps the current graph in VCG or DOT format into a temporary file. The temporary file will be passed to the program *ExecutableName* as first parameter; further parameters, such as program options, can be specified by *Text*.

If you use yComp[1] as executable (`show graph ycomp`), this may look like

[1] See Section 29.15.



If you use one of the programs of the graphviz package, e.g. dot (`show graph dot`), this may look like

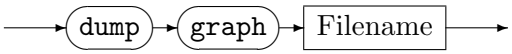


Available programs are `dot`, `neato`, `fdp`, `sfdp`, `twopi`, `circo`. They are used to transform the graph dumped into a `dot`-file to a `png`-file that is then opened with the image viewer of the system. The executables must be available in the search path of the system (so you have to install graphviz and have to add the `bin` folder to the path for this to work).

The temporary file(s) will be deleted when the application *Filename* is terminated, in case GrShell is still running at this time. If the `keep` parameter was specified, the file(s) will be kept instead.

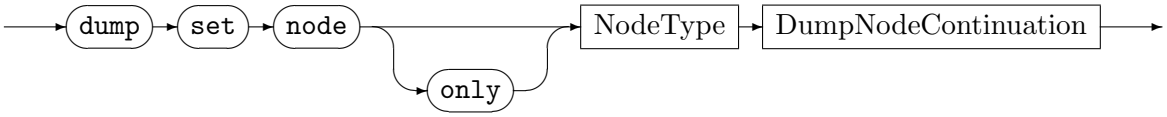
You may also specify `MSAGL` or `MSAGLExt` (see 27.3 for more on MSAGL). Note that no file will be dumped in this case, and that the shell will be frozen until you close the graph viewer window (furthermore, parameters will be ignored).

For that reason, `MSAGL` and `MSAGLExt` are discouraged when it comes to the `show graph` command, but alternatively, you may specify `extMSAGLExt` (which is based on MSAGLExt), then the graph is dumped into a `vcg`-file as in the yComp-case, and displayed in the external graph viewer application extMSAGLExt (see 27.4 for more on this program). This may look like in 27.7 (note that the screenshot shows a situation from debugging, in contrast to the display of a static dumped file).

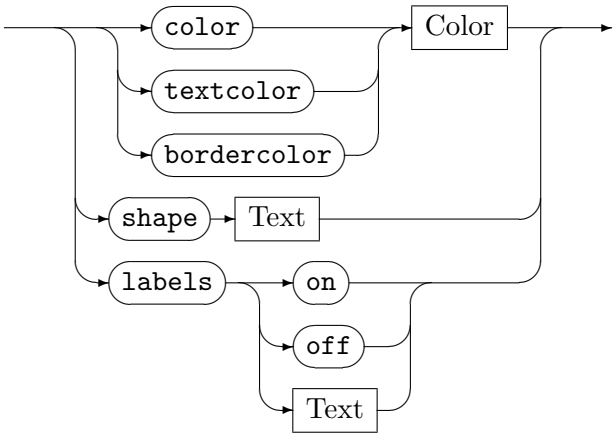


Dumps the current graph in VCG format into the file *Filename*.

The following commands control the style of the VCG output and the online graph display. This affects `dump graph`, `show graph`, and `enable debug`.

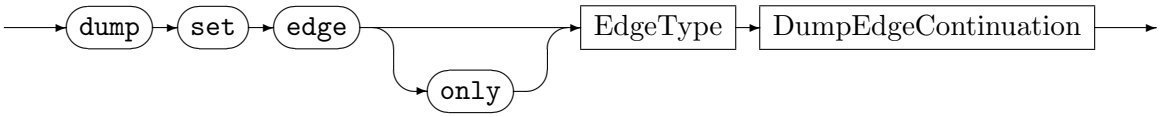


DumpNodeContinuation

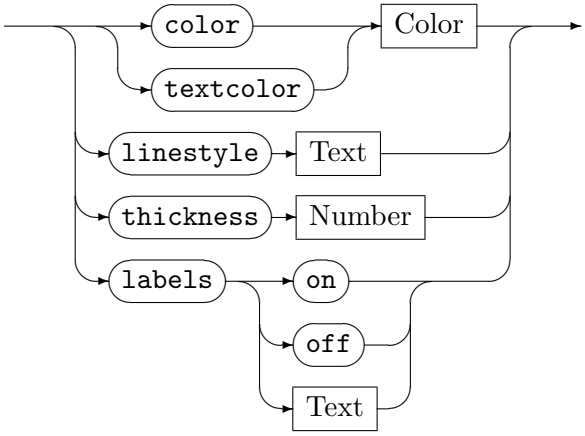


Sets the color, text color, border color, the shape or the label of the nodes of type *NodeType* and all of its subtypes. The keyword `only` excludes the subtypes. The following shapes are supported: `box`, `triangle`, `circle`, `ellipse`, `rhomb`, `hexagon`, `trapeze`, `uptrapeze`, `lparallelogram`, `rparallelogram`. Those are shape names of yComp (for a VCG definition see [San95]). The following colors are supported: `Black`, `Blue`, `Green`, `Cyan`, `Red`, `Purple`, `Brown`, `Grey`, `LightGrey`, `LightBlue`, `LightGreen`, `LightCyan`, `LightRed`, `LightPurple`, `Yellow` (default), `White`, `DarkBlue`, `DarkRed`, `DarkGreen`, `DarkYellow`, `DarkMagenta`, `DarkCyan`, `Gold`, `Lilac`, `Turquoise`, `Aquamarine`, `Khaki`, `Pink`, `Orange`, `Orchid`, `LightYellow`, `YellowGreen`. These are the same color identifiers as in VCG/yComp files (for a VCG definition see [San95]).

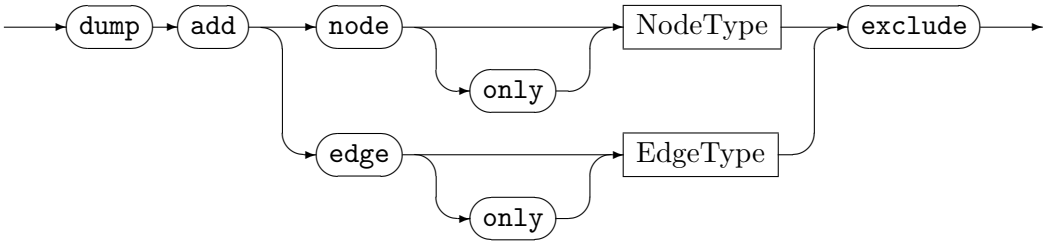
The default labeling is set to `on` with `Name:Type`, it can be overwritten by a specified label string (e.g. a source code line originating a node in a program graph) or switched off.



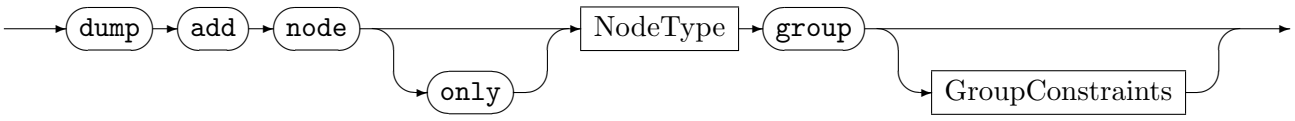
DumpEdgeContinuation



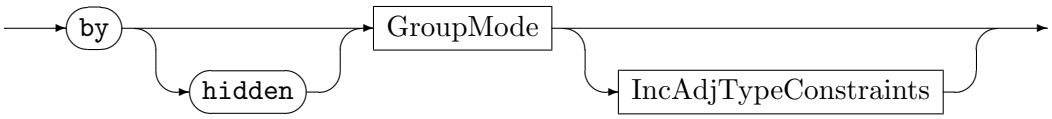
Sets the color, text color, the linestyle, the thickness of the line, or the label of the edges of type *EdgeType* and all of its subtypes. The keyword `only` excludes the subtypes. The available colors are given above with the `dump set node` specification. The default labeling is set to `on` with `Name:Type`, it can be overwritten by a specified label string or switched off. The following linestyles are supported: `continuous` (default), `dotted`, `dashed`. The following thicknesses are supported: `1` (default), `2`, `3`, `4`, `5`.



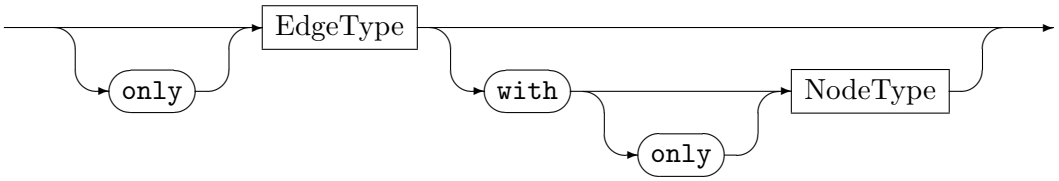
Excludes nodes/edges of type *NodeType*/*EdgeType* and all of its subtypes from output, for a node it also excludes its incident edges. The keyword `only` excludes the subtypes from exclusion, i.e. subtypes of *NodeType*/*EdgeType* are dumped.



GroupConstraints



IncAdjTypeConstraints

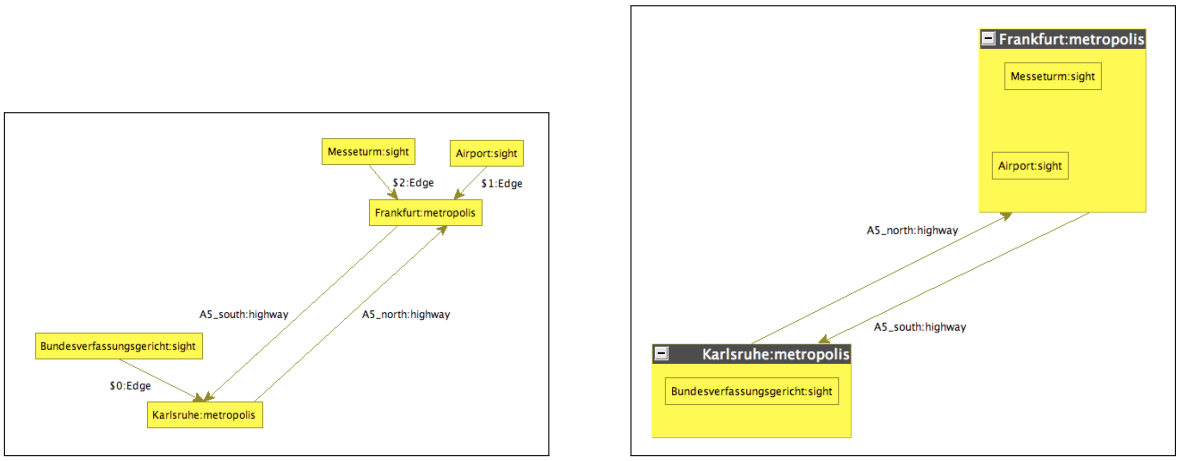


Declares *NodeType* and subtypes of *NodeType* as group node type. All the differently typed nodes that point to a node of type *NodeType* (i.e. there is a directed edge between such nodes) will be grouped and visibly enclosed by the *NodeType*-node (leading to the rendering of a nested graph). `GroupMode` is one of `no`,`incoming`,`outgoing`,`any`; `hidden` causes hiding of the edges by which grouping happens. The `EdgeType` constrains the type of the edges which cause grouping, the `with` clause additionally constrains the type of the adjacent node; `only` excludes subtypes.

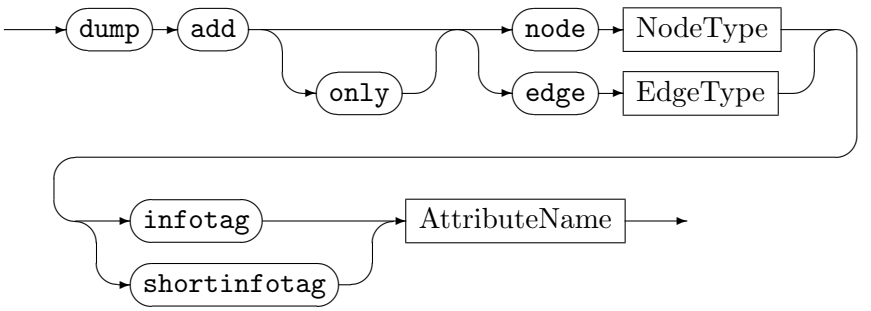
NOTE (55)

Only apply group commands on a graph if they indeed lead to a containment tree of groups. If the group commands would lead to a directed acyclic or even cyclic containment graph, the results are undefined. You may get duplicate edges (and nodes); the implementation is free to choose indeterministically between the possible nestings – it may even grow an arm and stab you in your back. (A conflict resolution heuristic used is to give the earlier executed `add group` command priority. But this mechanism is incomplete – you'd better refine your groups or change the model in that case. Using a model separating edges denoting direct containment from cross-linking edges by type is normally the better design, even disregarding visual node nesting.)

The following example shows *metropolis* ungrouped and grouped:

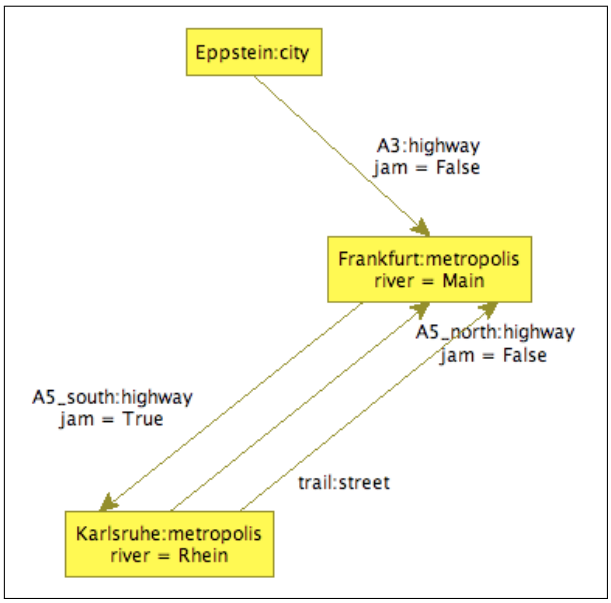


right side: dumped with `dump add node metropolis group`

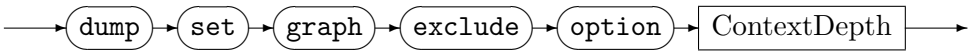


Declares the attribute *AttributeName* to be an "info tag" or "short info tag". Info tags are displayed like additional node/edge labels, in format `Name=Value`, or `Value` only for short info tags. The keyword `only` excludes the subtypes of *NodeType* resp. *EdgeType*.

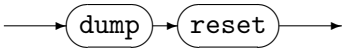
In the following example *river* and *jam* are info tags:







The dump graph exclude commands allow to suppress the display of the graph during debugging. Instead, only the match of the current rule is shown, plus some context up to a certain depth, plus the parent nodes according to the nesting commands. The default depth is 1, i.e. the match plus its direct neighbours are displayed (plus the nesting nodes); you may set it to 0 or a higher value. These commands allow you to still use the debugger if the graph as such is too large to be laid out (or laying it out takes too long to be convenient).



Resets all style options (`dump set...`) and (`dump add...`) to their default values.

NOTE (56)

Small graphs allow for fast visual understanding, but with an increasing number of nodes and edges they quickly lose this property. The group commands are of outstanding importance to maintain readability with increasing graph sizes (e.g. for program graphs it allows to lump together expressions of a method inside the method node and attributes of a class inside the class node). Additional helpers in keeping the graph readable are: the capability to exclude elements from dumping (the less hay in the stack the easier to find the needle), the different colors and shapes to quickly find the elements of interest, as well as the labels/infotags/shortinfotags to display the most important information directly. Choose the layout algorithm and the options delivering the best results for your needs, organic and hierarchic or compiler graph (an extension of hierarchic with automatic edge cutting – marking cut edges by fat dots, showing the edge only on mouse over and allowing to jump to the other end on a mouse click) should be tried first.

The following example consisting of Figures 27.1 and 27.2 shows several of the layout options employed to considerably increase the readability of a program graph (as given in `examples/JavaProgramGraphs-GraBaTs08`), with:

- nesting to show containment
- color coding to distinguish different classes of elements (yellow for classes, magenta for methods, cyan for variables, and green for expressions and statements)
- the typical rendering of the `Hierarchic` layout (its specific version `Compilergraph` aimed at compiler construction to be more exact)

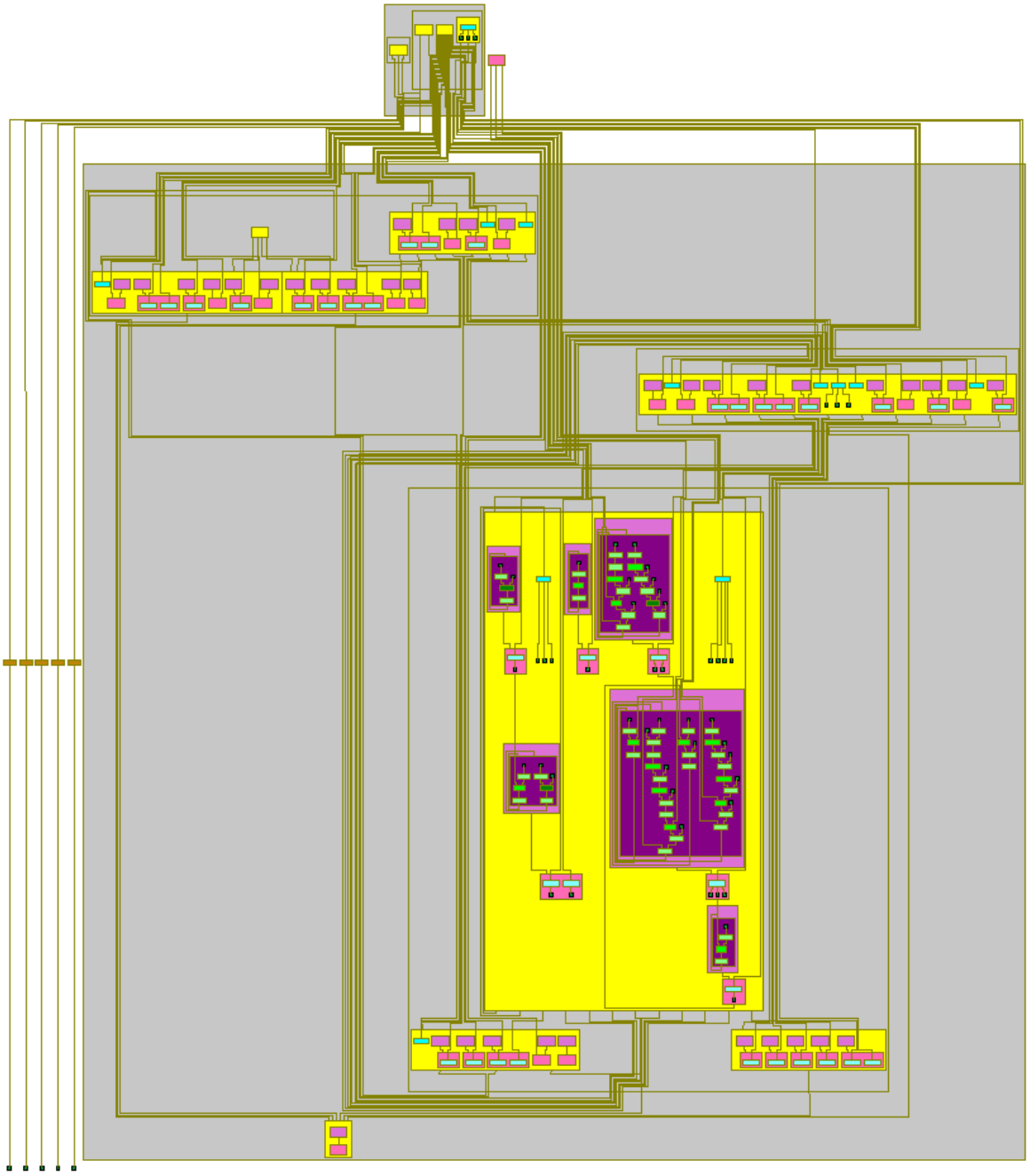


Figure 27.1: Overview of the initial program graph

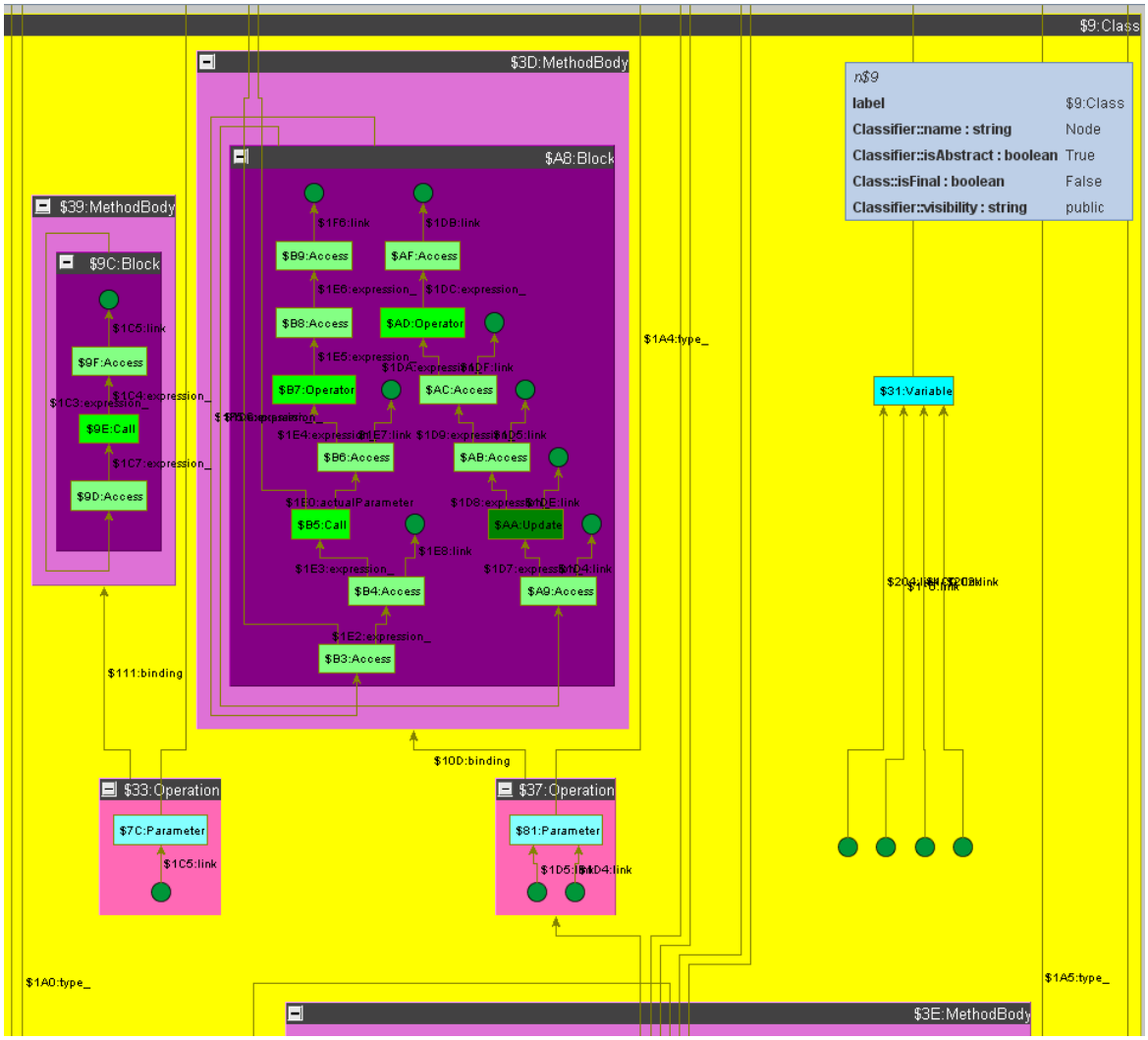


Figure 27.2: Some details of the "Node" class of the initial program graph

You see the `exclude` option applied in Figures 27.3 and 27.4, stemming from the GrGen.NET solution [Jak14] of the TTC14 Movie Database case [HKT14] (you find it under `examples/MovieDatabase-TTC2014`) . There, a rule `couplesWithRating` (whose pattern consists of a `c:Couple` referencing its actors `pn1:Person` and `pn2:Person`) is used in order to fetch the couples of actors who performed together in a movie, for one the couples whose common movies have the highest rating, and for the other the couples with the highest number of common movies. The match with the highest rating or highest number is obtained by ordering with an auto-generated filter (cf. 17) alongside an appropriately filled `def` variable, and throwing away the results below with an auto-supplied filter.

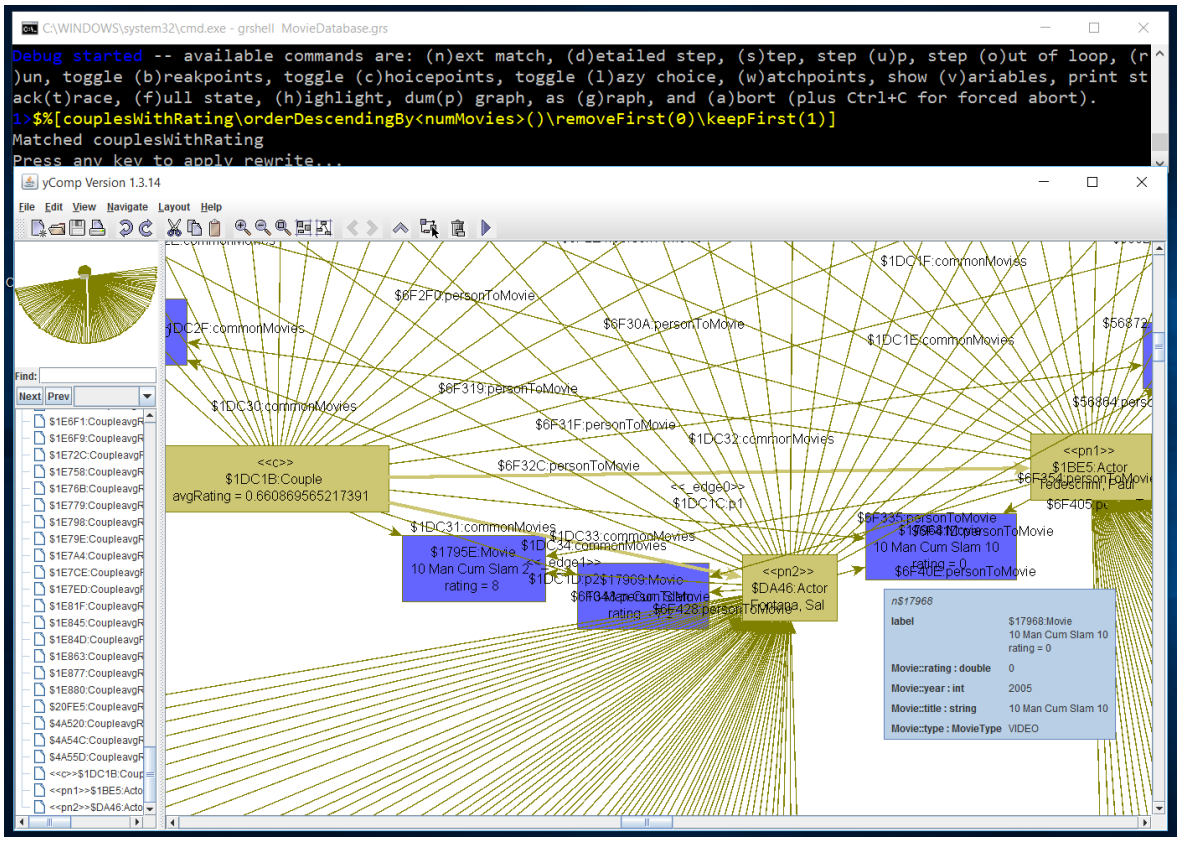


Figure 27.3: `couplesWithRating`, order by `avgRating`, 10,000 movies file

The first screenshot depicted in Figure 27.3 was taken on Windows 10 during processing of the 10,000 movies file from IMDB (98,388 nodes, 124,638 edges). It is displayed with layout `Circular` and shows the match of the couple with the highest rating.

The second screenshot depicted in Figure 27.4 was taken on openSUSE LINUX 13.2 during processing of the generated N=50,000 example (1,000,000 nodes, 1,600,000 edges). It is displayed with layout `Organic` and shows the match of the couple with the highest number of common movies.

The graphs as such are beyond the capabilities of the graph viewer (several thousand graph elements work well, up to a few ten thousand ones), without the `exclude` we'd be blind, this way we're endowed with partial sight (on the important part, the match plus its context). Notice the infotags on the `rating` and `avgRating` attributes.

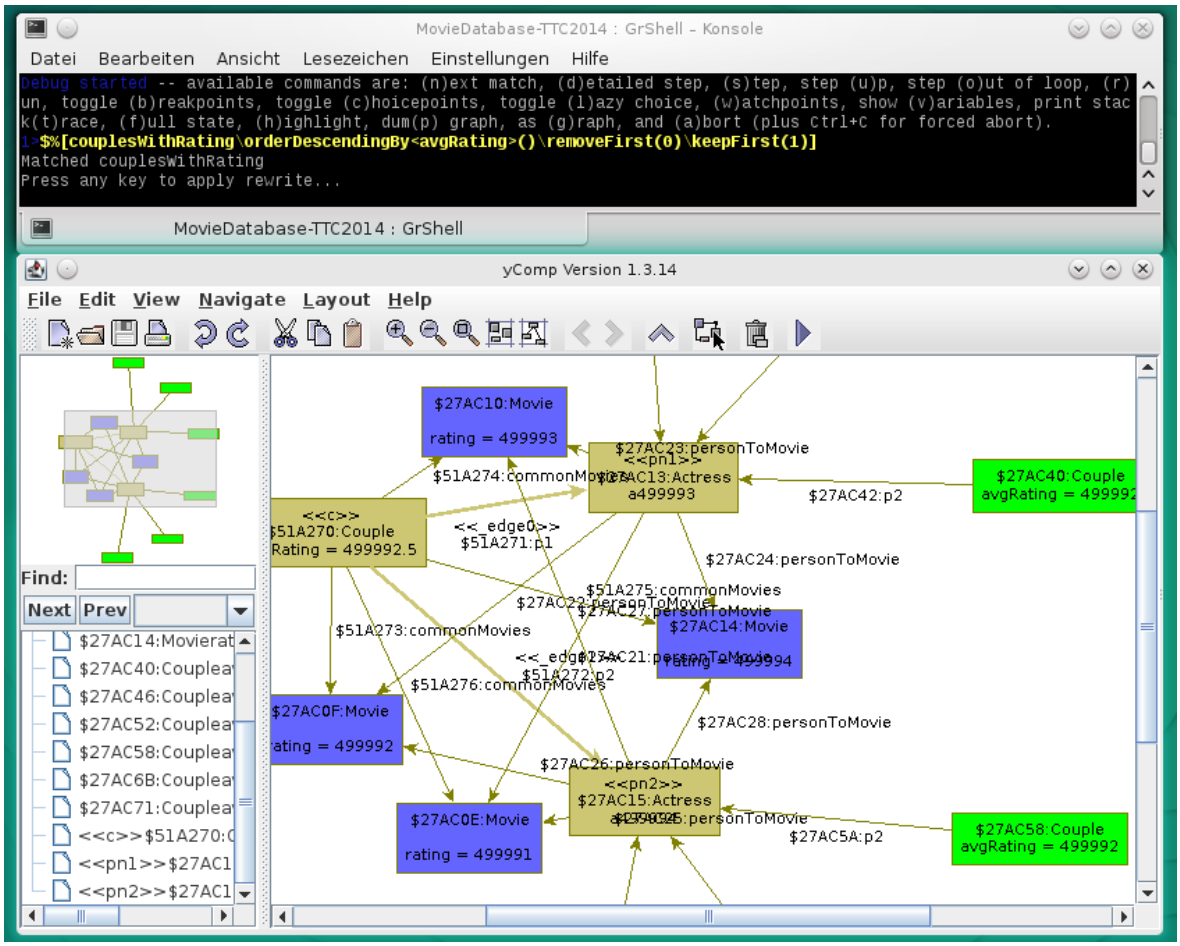


Figure 27.4: `couplesWithRating`, order by `numMovies`, generated with N=50,000

27.2 yComp Usage

yComp[KBG+07] (a graph viewer application originally targeting compiler construction, based on the yFiles[yWo07] library) is the default graph viewer of GrGen, and – when started as a server process – can be controlled by the debugger via a TCP/IP connection.

The available layouts of yComp are: `Random`, `Hierarchic`, `Organic`, `Orthogonal`, `Circular`, `Tree`, `Diagonal`, `Incremental Hierarchic`, as well as `Compilergraph`, see 27.5 for more.

Besides the things already mentioned in 29.15, we want to give the following hints:

- when started on a dump, you must press the rightmost play button to start layout
- play with the layout options offered in the `Layout` menu until you find a good visualization, configure it then in the GrShell; don't forget to press the play button to apply the changes
- you can pane by pressing and holding the middle mouse button while moving the mouse
- you can zoom with the mouse wheel at the position of the cursor
- hovering over graph elements displays the attributes
- you can select graph elements with the left mouse button and delete them with `del` to gain a better overview
- by activating edit mode with the 3rd rightmost button in the toolbar you can move nodes around, which allows you to fix a bad layout (rather seldomly needed)
- the context menu opened by pressing the right mouse button over a graph element allows you to explore the adjacent nodes in non-edit-mode, or delete the element in edit mode
- you can search with `Ctrl-f` or `/` for the persistent name or an attribute value (or by clicking into the left search field), the matching elements get highlighted

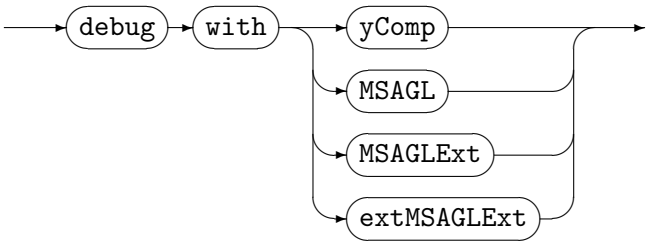
Note that yComp requires a JAVA installation.

27.3 MSAGL and MSAGLExt Usage

MSAGL[NPDH07], an abbreviation of Microsoft Automatic Graph Layout, is a .NET library and tool for graph layout and viewing. MSAGL is not a standalone application, in contrast to yComp, but a library that is linked into GrGen.NET (and used at API level). Technically, the debugger using MSAGL as graph viewer can be embedded into your own .NET program by utilizing the `graphViewerAndSequenceDebugger.dll` and the `graphViewerAndSequenceDebuggerWindowsForms.dll` of GrGen.NET (i.e. you can use it on API level, like MSAGL). Legally, this is possible because MSAGL comes with the very permissive MIT license[oT87], in contrast to the yComp license purchased by IPD Goos from yWorks that restricts its use to academic purposes, and esp. does not allow you to ship it with a release of your own software. GrGen.NET itself is licensed as LGPL, so you can ship it with your own (even commercial) program - the functionality of the MSAGL based graph viewer is limited compared to what yComp offers, though (and under Mono/Linux, sometimes the text labels disappear, you may have to pan with the mouse as a workaround then).

The available layouts of MSAGL are: `SugiyamaScheme`, `MDS`, `Ranking`, `IcrementalLayout`, see 27.5 for more.

In order to use it from GrShell, you have to employ the `debug with` command.

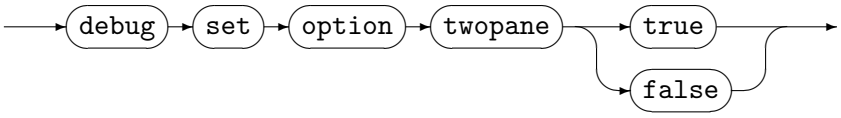


Switches the graphical part of the debugger (the graph viewer) in between yComp (default) or MSAGL or extended MSAGL or extMSAGLExt (external MSAGL extended). Extended MSAGL supplies an additional left pane similar to yComp, with a map-like view, a search pane, a node subgraph nesting tree, and an attributes/properties view. Here, too, you can search with `Ctrl-f` or `/`.

If MSAGL (or its extended version) is used (unless it's the external version), the debugger is displayed with a Windows Forms[Mic02] GUI. It may be reduced to only a console like control, which is functionally similar to the stdout console of the GrShell used to display the debugger in case of yComp or extMSAGLExt, but technically different as it is a Windows Forms control.

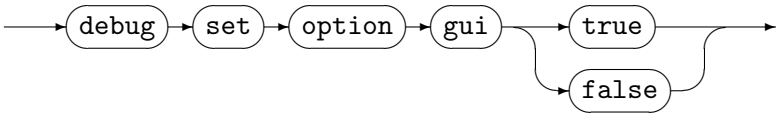
The program graph from `examples/JavaProgramGraphs-GraBaTs08` looks like depicted in Figures 27.5 and 27.6 (compared to the yComp-rendering in Figures 27.1 and 27.2).

The WinForms based debugger that uses MSAGL as graph viewer supplies a two pane mode, it is enabled by default when gui mode is not enabled (when yComp is used, only the single GrShell-console is employed, irrespective of this setting or the gui mode setting), you may disable it (or enable it again) with the configuration option:



In two pane mode of the debugger, two consoles are employed, one for data rendering and one for user dialog. The data rendering console only displays the current work object (at a time – typically the sequence that gets debugged, in its current state), instead of the complete output history, which still holds for the user dialog console, esp. in order to allow you to inspect the subrule traces log. Figure 27.5 shows an example screenshot of the two pane mode.

The debugger also offers a gui mode as an extension of twopane mode, it is enabled by default in case you debug with MSAGL (when yComp is used, only the single GrShell-console is employed, irrespective of this setting), you may disable it (or enable it again) with the configuration option:



In gui mode of the debugger, the two consoles are enriched by a GUI frame offering a menu bar and a toolbar. The menu bar contains the main debugger menu commands (the same that are available in the console main menu), some commands from the submenus/subdialogs, plus the continuation options that are offered during a debugging break; some commands from the submenus/subdialogs on the other hand are only available on the input/output console as of now. The toolbar contains the execution controlling commands plus the continuation options.

The data rendering console view can be switched to a display in graph form, in that case sequences are rendered graphically in form of an abstract syntax tree (and the call stack in

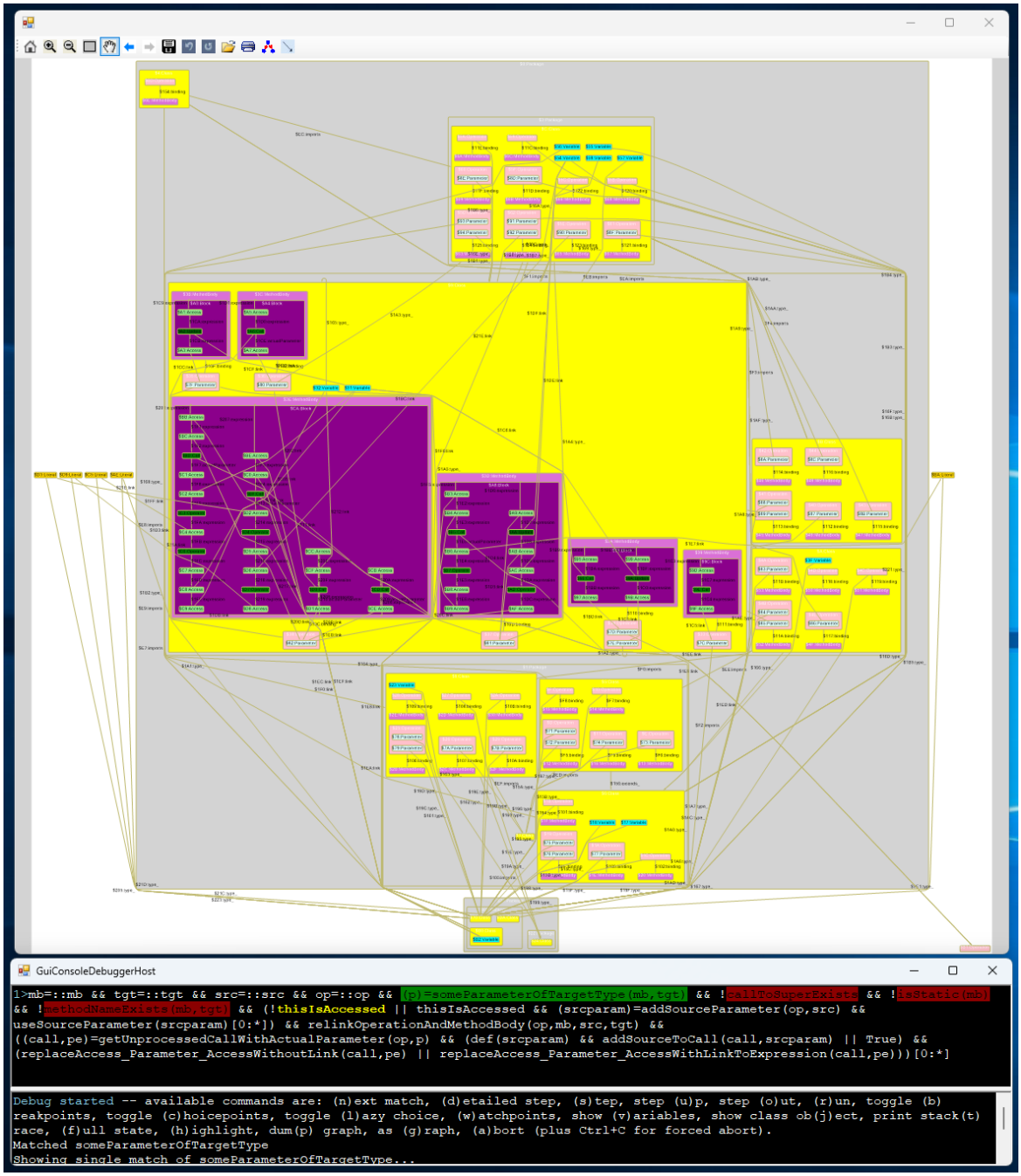


Figure 27.5: The initial program graph rendered with MSAGL and layout SugiyamaScheme, utilized by a debugger with two console panes

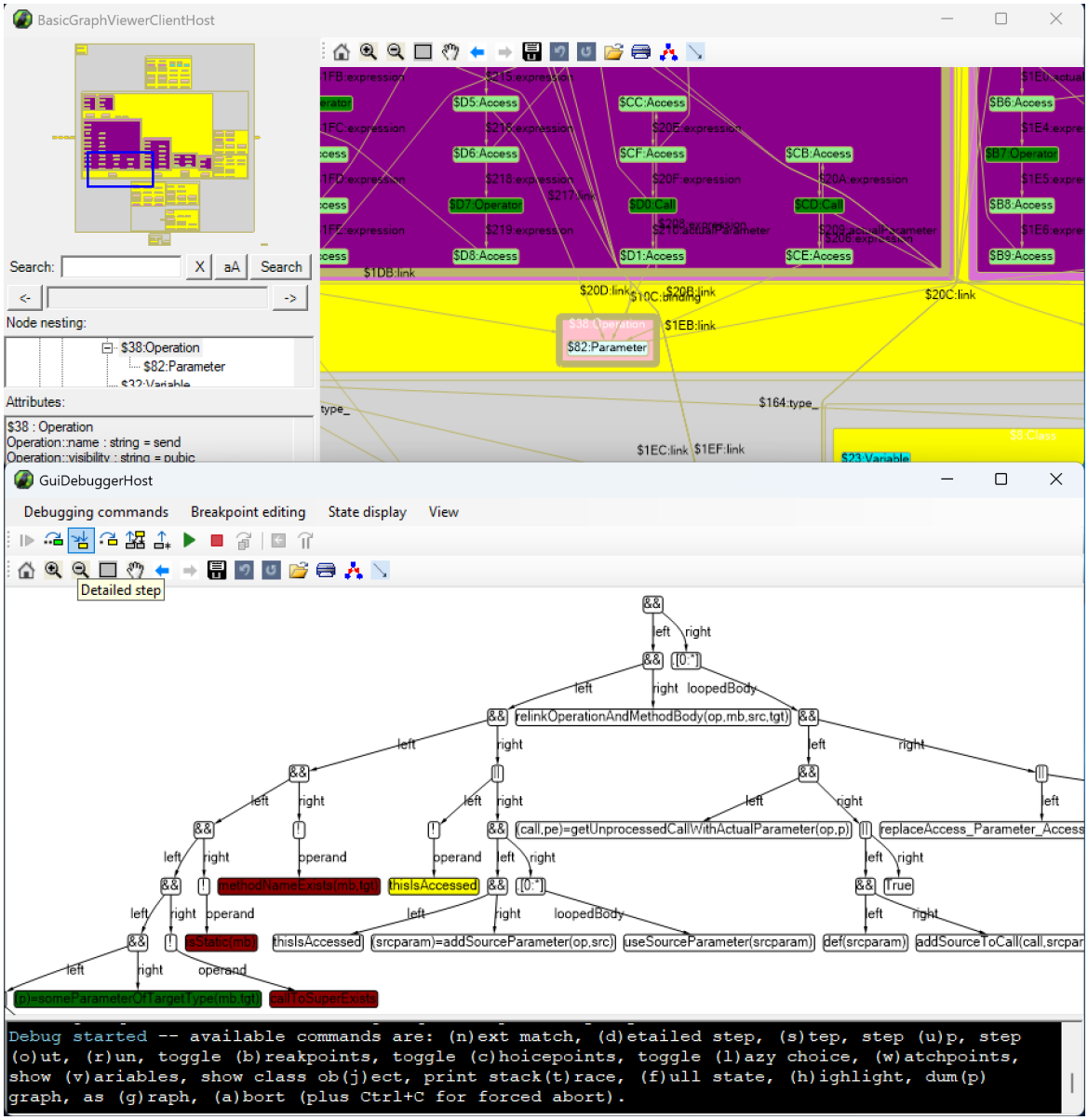
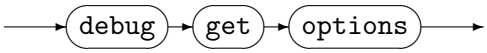


Figure 27.6: Some contents of the "send" method of the initial program graph rendered with MSAGLExt, shown in the GUI debugger, with an upper pane sequence rendered in graph form, i.e. as an abstract syntax tree

form of group nodes representing stack frames, containing their sequences, and local variables in case of a full state dump). Figure 27.6 shows an example screenshot of the gui mode.

You may find out which mode is enabled with the following command:



Lists the available debug options.

27.4 extMSAGLExt Usage

extMSAGLExt is an external graph viewer application based on the internal MSAGLExt graph viewer. When started as a server process, it can be controlled by the console debugger (see 27.7 for an example screenshot) via a loopback/localhost TCP/IP connection like yComp (just to be explicit: the GUI debugger can not be used with extMSAGLExt, it is only employed when `MSAGL` or `MSAGLExt` are specified). When started with a file path of a vcg file as argument, it imports and renders the vcg file. It supports the layouts also supported by MSAGL27.3. The extMSAGLExt app may be also used freely (in contrast to the yComp app).

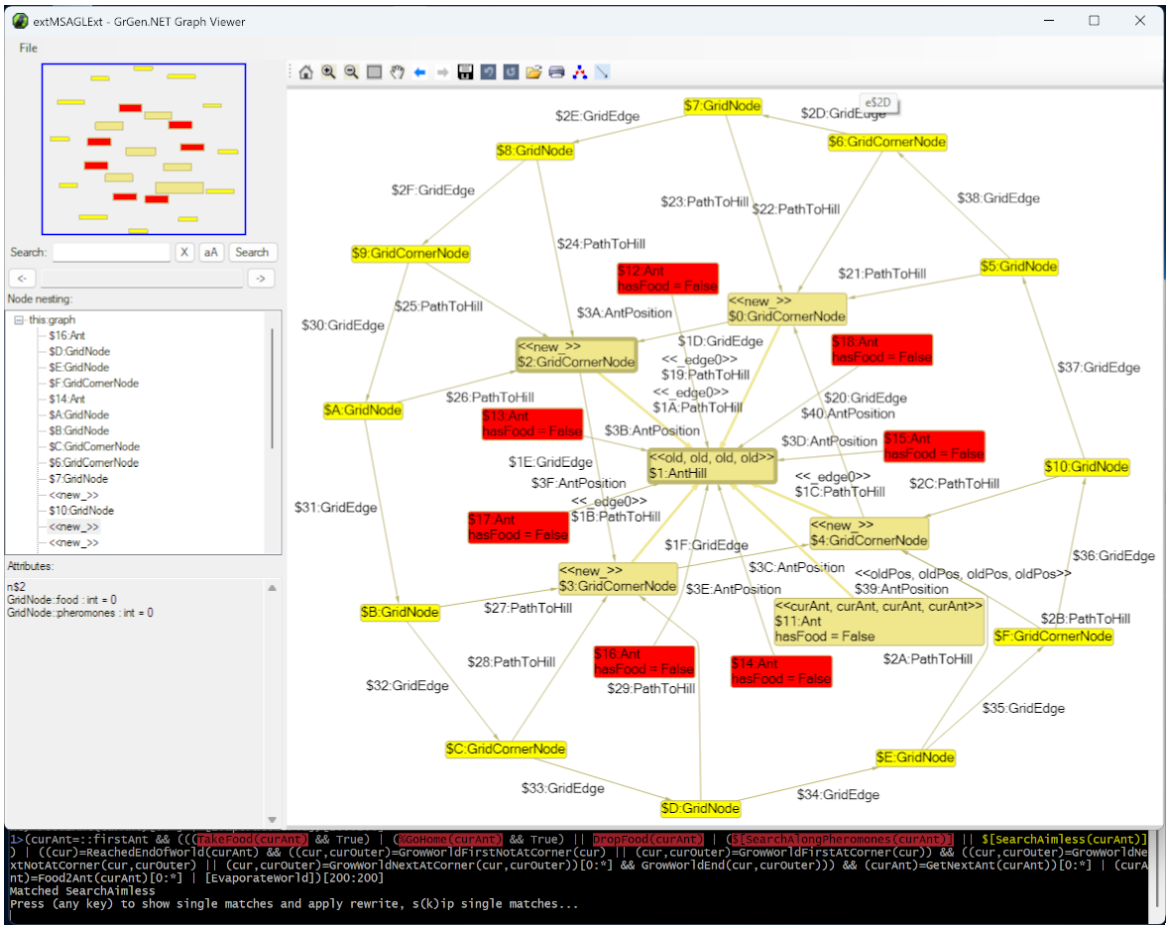
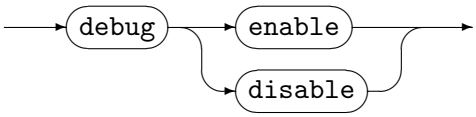
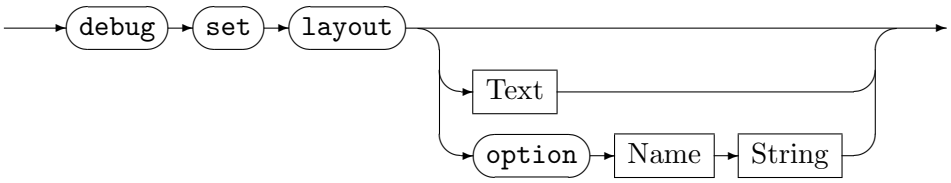


Figure 27.7: A situation from debugging the GrGen.NET-solution of the AntWorld case from GraBaTs[RVG08] that can be found in `examples/AntWorld-GraBaTs08`, rendered with extMSAGLExt and layout MDS

27.5 Debugging Related Commands



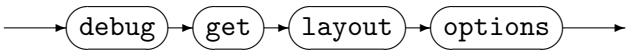
Enables and disables the debug mode. The debug mode shows the current working graph in a yComp or MSAGL window. All changes to the working graph are tracked by the graph viewer immediately.



Sets the default graph layout algorithm to *Text*. If *Text* is omitted, a list of the available layout algorithms is displayed. The `option` version allows you to specify layout options by name value pairs. The available layout options can be listed by the succeeding command.

The following layout algorithms are supported by yComp: `Random`, `Hierarchic`, `Organic`, `Orthogonal`, `Circular`, `Tree`, `Diagonal`, `Incremental Hierarchic`, `Compilergraph`. For technical graphs `Hierarchic` works normally best; `Compilergraph` is a version of `Hierarchic` cutting some edges, it may be of interest if `Hierarchic` contains too many crossing edges. `Organic` is the other general purpose layout algorithm available to be tried out early; the other layout algorithms are rather special, but this should not deter you from using them if they fit to your task ;).

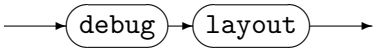
The following layout algorithms are supported by MSAGL or MSAGLExt: `SugiyamaScheme`, `MDS`, `Ranking`, `IcrementalLayout`. We recommend to use `SugiyamaScheme` in place of `Hierarchic`, and `MDS` in place of `Organic`.



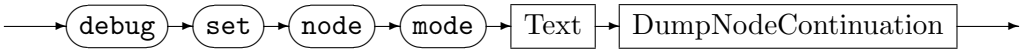
Prints a list of the available layout options of the currently active layout algorithm.

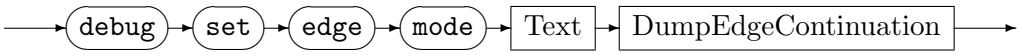


Prints the currently set layout.

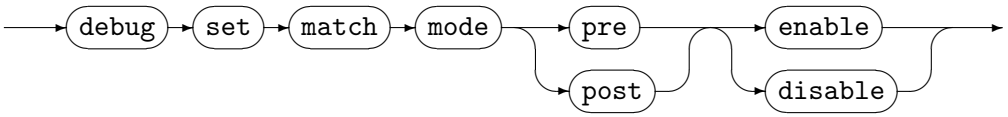


Forces re-layout of the graph shown in yComp (same as pressing the play button within yComp).



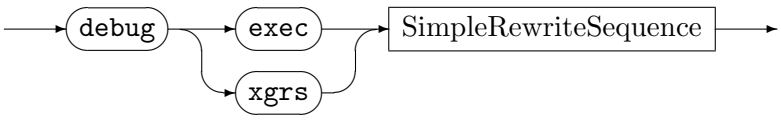


Configures the display of the visual debug states for the nodes/edges. The following modes are supported: `matched`, `created`, `deleted`, `retyped`, `redirected`. Change this if you e.g. want the matched elements to be marked more visibly, or added/deleted elements to be colored green/red (instead of red/grey).



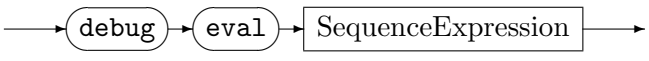
Configures the behavior of the debugger regarding the display of found matches. When pre-match debugging is enabled, the matches of a (multi) rule execution before filtering/selection are displayed in detail steps, the default is disabled. When post-match debugging is enabled, the matches of rules in embedded execs are displayed in detail steps (with post-match disabled, only the match(es) of the very rule followed by the rewrite of the rule), the default is enabled.

GraphRewriteSequence



This executes the graph rewrite sequence *SimpleRewriteSequence* in the debugger. The semantics is the same as for `exec SimpleRewriteSequence` already introduced in Chapter 25, but you can trace the execution step-by-step.

GraphRewriteSequenceExpression



This executes the graph rewrite sequence expression *SequenceExpression* in the debugger. The semantics is the same as for `eval SequenceExpression` already introduced in Chapter 25, but you can watch the matches highlighted in the graph (one (multi) rule query after the other).

27.6 Using the Debugger

Put abstractly, a graph rewrite sequence is executed step-by-step, one rule application after the other, yielding a series of execution states. Each step has a current point of execution in the sequence, referencing the rule that is to be or gets applied. The debugging process can be described as a reduced view on such a series of execution states, consisting of a series of debugging situations, which are a subset of your choice of the execution states reached in time. In each debugging situation or step, the execution state or a subset of it is rendered.

This means concretely, that the debugger – which is a part of the GrShell – prints the debugged sequence with the currently focused/active rule highlighted yellow. What will be shown from executing this rule depends on the chosen debug command, and on the fact whether the focused rule matches or not. An active rule which is already known to match is highlighted green. The rules that matched the last time during sequence execution are shown on dark green background, the rules that failed the last time during sequence execution are shown on dark red background; at the beginning of a new loop iteration the highlighting state of the rules contained in the loop is reset.

During execution, yComp[2]/MSAGL displays the current graph at each single step, and on request highlights a rule application, marking the matches in the graph, and showing the changes to the graph stemming from that rule application.

Besides deciding on what is shown from the application of the current rule, you also determine with the debug commands where to continue debugging, i.e. the rule focused next; but of course this depends strongly on the following flow of execution, esp. the fact whether the execution of the currently active rule succeeds/fails, and the execution state in general.

With the `s` debug command (step) you execute the current rewrite rule (trying to match it, applying the rewrite on success) and then continue with the next rule in the sequence.

With the `n` debug command (next) you fast-forward to the next rule *that matches*, and apply it when pressed again.

With the `r` debug command (run) you continue execution until sequence termination or until a breakpoint is hit.

With the `d` debug command (detailed step) you execute the current rewrite rule, if it matches highlighting its match and then the changes in the graph, and then continue with the next rule in the sequence. In case the rule contains embedded sequences, these are also executed in detail mode (unless switched off with `set match mode post disable`). In case pre match debugging is toggled on (`set match mode pre enable`), all the matches found are displayed in a pre-step, before filtering (which may remove matches) and selection (of e.g. the active rule in a some from set construct).

All the available debug commands are given in Table 27.1. An example debugging run is shown in the following example 163.

[2]Make sure, that the path to your `yComp.jar` package is set correctly in the `ycomp` shell script within GrGen.NET's `/bin` directory.

In addition to the commands for actively stepping or skipping through the sequence execution, there are breakpoints and choicepoints available (toggled with the `b` and `c` commands), which become active if execution reaches them, even if a user command would skip over them in the debugging process.

The break points halt execution, focus the reached sequence, and cause the debugger to wait for further commands (e.g. `d` to inspect the rule execution en detail versus `s` for just applying it).

The choice points halt execution, focus the reached sequence in magenta, and ask for some user input; after the input was received, execution continues according to the command previously issued.

Both break points and choice points are denoted by the `%` modifier. The `%` modifier acts as a break point if it is given before: a rule, an all bracketed rule, a variable predicate, or the constants `true`/`false`. The `%` modifier acts as a choice point if it is appended to the `$` randomize modifier, switching a random decision into a user decision. The dollar randomize modifier can be applied to the binary operators, the random match selector of all bracketed rules, the random-all-of operators and the one-of-set braces. The idea behind this is: you need some randomization for simulation purpose — then use the randomize modifier `$`. You want to force a certain decision, overriding the random decision, in order to try out another execution path while debugging the simulation — then modify the randomize modifier with the user (choice) modifier `%`.

The initial breakpoint and choicepoint assignment is given with the `%` characters in the sequences after the `debug exec` commands in the `.grs` file. The breakpoint and choicepoint commands of the debugger allow to toggle them at runtime, overriding the initial assignment (notationally yielding a sequence with added or removed `%` characters). The user input commands `$%(type)` define choice points which cannot be switched off.

Moreover, there are commands available that allow to print the variables at a given situation, or the sequence call stack, or a full state dump of the call stack and the variables.

Further commands allow to dump the current graph, or highlight elements in the graph, defined by being contained in a (possibly container valued) variable, or being marked according to a visited flag.

In addition, you may crawl the network of objects or transient objects, starting at a variable or object defined by its unique id.

`s`(tep)	Execute the current rewrite rule (matching it, and rewriting in case it matched; the resulting graph is shown).
`d`(etailed step)	Execute the current rewrite rule in a three-step procedure: matching - highlighting the found match, rewriting - highlighting the changing elements, and completion - carrying out the rewrite, showing the resulting graph, then executing subrules from embedded sequences (`exec`), step by step.
(step) `o`(ut)	Continue execution until the end of the current loop. If execution is not in a loop at this moment, but in a sequence called, the called sequence will be executed until its end. If neither is the case, the complete sequence will be executed.
(step) `u`(p)	Ascend one level up within the Kantorowitsch tree of the current rewrite sequence (i.e. rule; see Example 163; at the moment the command is pretty useless because only the serialized form is displayed).
`r`(un)	Continue execution (until the end or a breakpoint).
`a`(bort)	Cancel execution immediately.
`n`(ext)	Go to the next rewrite rule that matches, make it current.
(toggle) `b`(reakpoint)	Toggle a breakpoint at one of the breakpointable locations.
(toggle) `c`(hoicepoint)	Toggle a choicepoint at one of the choicepointable locations.
(edit) `w`(atchpoints)	Allows to edit data breakpoints or the behaviour of programmed debug messages.
`v`(ariables)	Prints the global variables and the local variables of the sequence currently executed, which is the topmost sequence of the sequence call stack. Plus the allocated visited flags. To be more precise: all local variables which were defined (and have not fallen out of scope again) up to the sequence position focussed.
`t`(race)	Prints the stack trace of the current sequence call stack. The stack trace includes the body of each sequence called at its execution state.
`f`(ull dump)	Prints the stack trace including the local variables of each stack frame plus the global variables.
(dum)`p` (graph)	Dumps the current graph as a `.vcg` file and shows it in yComp or extMSAGLExt (depending on the graph viewer configuration). This can be used as a workaround to check the real state in case transaction/backtracking rollback is used on a graph with node nesting, which may lead to a buggy display. In addition, an `undo.log` is written with the undo commands of the open transactions for *change reversal*, which would be applied on rollback.
(as)-`g`(raph)	Asks for the value of the externally defined type that is to be shown in the debugger in graph form (you must implement an extension handler able to return an `INamedGraph` on request for this to work, see 33.4). Alternatively, you may specify a graph value, which is then displayed directly.
`h`(ighlight)	Highlights the elements in the graph that are marked with the visited flag given, or are contained in the variable given (which might be a simple scalar variable containing one graph element, or a container variable potentially containing multiple ones). Multiple variables or visited flags may be given separated by commas.
(show ob)`j`(ect)	Shows the object defined by its unique id or a transient object by a just generated id. You may give a variable name as root element to start crawling at (in order to obtain a transient object id).

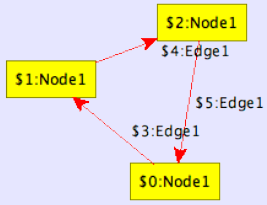
Table 27.1: GrShell debug commands

EXAMPLE (163)

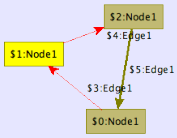
We demonstrate the debug commands by executing a slightly adjusted script for generating the Koch snowflake from GrGen.NET's examples (see also Section 32.1). The graph rewrite sequence is

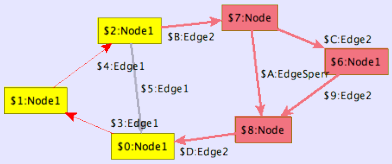
```
1 debug exec (makeFlake1* & (beautify & doNothing)* & makeFlake2* & beautify*)[1]
```

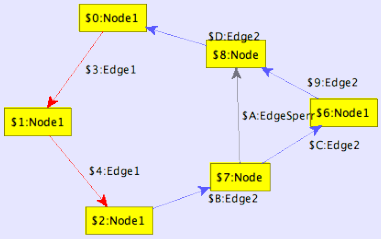
yComp will be opened with an initial graph:



We type `d`(etailed step) to apply `makeFlake1` in detail mode with a matching, rewriting, and completion substep, resulting in the following graphs:







The following table shows the active rule reached after having entered the preceding debug command, for a series of debug commands:

Command	Active rule
s	makeFlake1
o	beautify
s	doNothing
s	beautify
u	beautify
o	makeFlake2
r	—

27.7 Subrule Debugging and Programmed Halts

The basic granularity of debugging in GrGen.NET is the single rule (with its pattern matched and its rewrite carried out) executed from an interpreted sequence. But with embedded execs there are further sequences and actions available outside direct control of and visibility in the debugger. At least some debugging aid is available for them, at least the actions called from the embedded sequences are shown in the debugger – in case detailed mode is used. But the local execution state of imperative code which may be used for tasks where pattern matching is not beneficial is completely invisible to the debugger. Only the effects on the graph become visible.

There is a remedy available for this situation in the form of code-embedded debugging commands, realized by calls to procedures from the built-in package `Debug`. With them you can punch holes into the covering blanket that allow you to peek at what's going on under the covers. The debugging commands are used for one to directly display information, but for the other for handling a stack of debug messages, intended for representing the current state of the call nesting of the executing code.

The available commands are:

`Debug::add(message[,object]*)`
: to be called when a subrule computation or an interesting piece of code is entered. The message is added to the debug messages stack of the debugger. Besides the mandatory `message:string`, an arbitrary number of other parameters may be given (of arbitrary type).

`Debug::rem(message[,object]*)`
: to be called when a subrule computation or an interesting piece of code is exited. The topmost entry message on the messages stack is removed. It is checked that the message of the topmost added entry is identical to the message of current removal – you must always call `add` and `rem` in pairs! The `emit` messages on the way to the topmost `add` are removed. Besides the mandatory `message:string`, an arbitrary number of other parameters may be given (of arbitrary type).

`Debug::emit(message[,object]*)`
: to be called when some interesting points in the code are passed. The message is added to the debug messages stack of the debugger. Besides the mandatory `message:string`, an arbitrary number of other parameters may be given (of arbitrary type).

`Debug::halt(message[,object]*)`
: to be called when some point in the code is reached that is so interesting that you want the execution to break in the debugger. If called, the debugger halts execution, displays the messages stack in its current state, and prints out the halt message with its parameters. Besides the mandatory `message:string`, an arbitrary number of other parameters may be given (of arbitrary type).

`Debug::highlight(message[,object,string]*)`
: to be called when some point in the code is reached that is so interesting that you want the execution to break in the debugger, in order to highlight some nodes or edges graphically in the debugger, in the same way they are highlighted when an action is matched. If called, the debugger halts execution, displays the messages stack in its current state, and displays the nodes and edges passed with the additional parameters highlighted in the graph. The additional parameters must be given in pairs, first the entity to display, then the string that entry will be annotated with in the debugger. The entity to display may be a node or edge, which is then directly highlighted, or a storage containing nodes or edges, all contained nodes/edges will then be highlighted, or a visited flag (integer number), all graph elements that are visited according to that

flag are then highlighted. Besides, as first mandatory parameter, the `message:string` must be given.

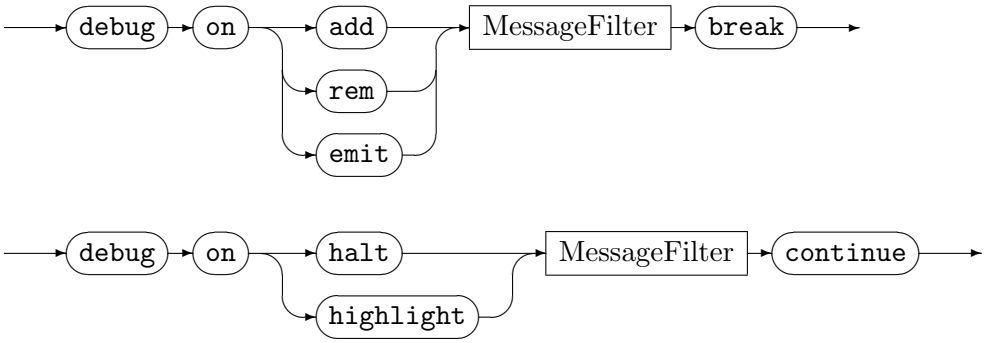
Some `add` and `rem` are automatically inserted by GrGen.NET for you. For one for embedded execs, a debug message is sent when an embedded exec is entered or exited, with a message starting with the name of the containing rule. For the other for procedures and compiled sequence definitions, a debug message is sent when a procedure or defined sequence is entered or exited, with a message being equal to the name of the procedure or defined sequence.

27.8 Watchpoint configuration

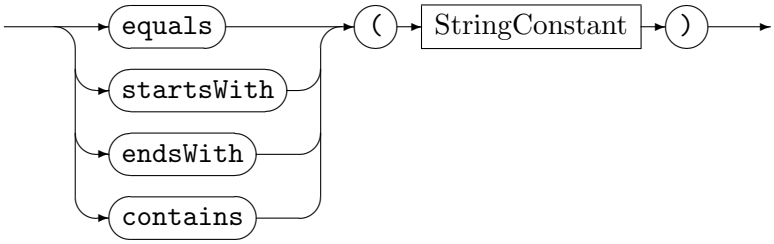
The behaviour of the debugger upon receiving certain events can be defined with configuration rules aka watchpoints. They are contained in a list, ordered by time of insertion. When a subrule debugging event (a debug message as introduced in the previous section), or a graph change event, or an action match event occurs, the list of watchpoints is visited one entry after the other, and one configuration rule checked for a match after the other. When a configuration rule or watchpoint matches, its decision is applied. The decision may be to break execution and display the current execution state, or to continue execution, which is of interest for events that normally break execution but should be better ignored.

Besides defining the configuration rules of the watchpoints beforehand with shell commands, you can edit them interactively with the edit watchpoint command inside the debugger.

Subrule messages

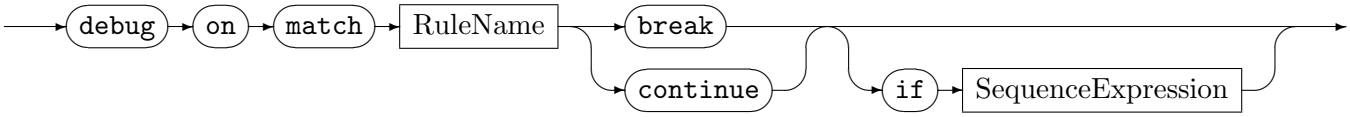


MessageFilter



When the string specified matches the message of the debug event according to the message filter given, execution is interrupted by the debugger in case of a `Debug::add`, or `Debug::rem`, or `Debug::emit`, and the execution state of the code (esp. specified by those commands) is presented in the debugger. Normally, those events are handled silently. When a match happens for a `Debug::halt` or `Debug::highlight`, execution is continued without interruption, while normally those messages break execution in the debugger.

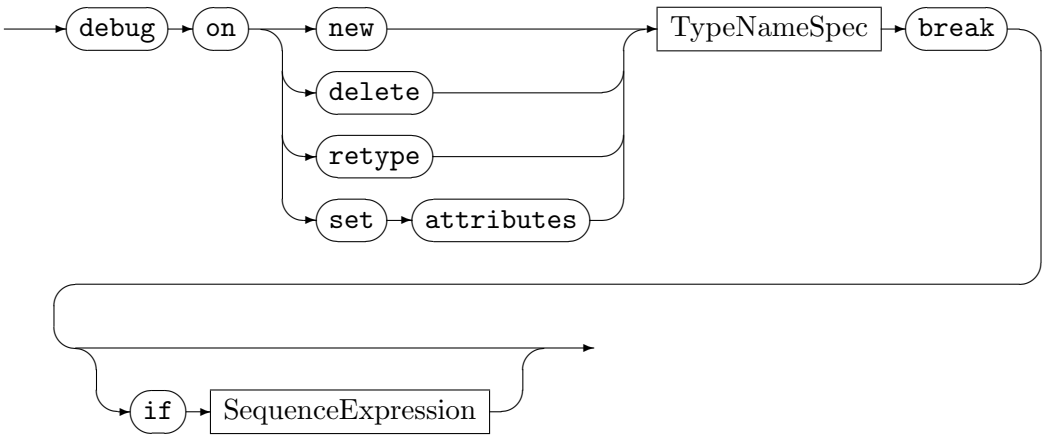
Action match event



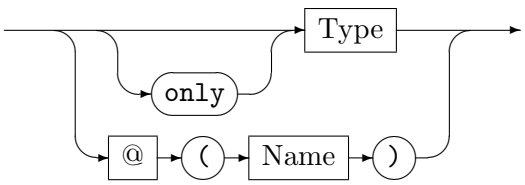
When the action specified was matched, execution is halted in the debugger in case of a `break`. This is of interest for rules executed from execs, as normal breakpoints don't apply to them, this way we can set a breakpoint on a rule irrespective from where it is called (allowing us to just `r`un a sequence until an action match of interest happens). In case of a `continue`, execution is forced to continue. This is of interest for detail mode debugging that normally breaks on each matched rule. It allows us to skip over uninteresting rules in execs without the need to acknowledge them.

The sequence expression finally allows us to decide conditionally. It is evaluated when its corresponding configuration rule is evaluated because its match event occurred, if it returns true the configuration rule matches, otherwise it does not match. The `this` entity is overloaded in the sequence expression (normally it denotes the graph). It gives access to the match found, you can access the entities of the match in dot-Notation (e.g. `this.node1`). The configuration rule is evaluated for all matches in case of an all-bracketed match, if one returns true, the decision is carried out.

Graph change events



TypeNameSpec



When the graph change specified occurred (its corresponding event was fired), execution is halted in the debugger. The supported graph changes are graph element creation, deletion, retyping, and attribute assignment. The *TypeNameSpec* constrains this by type or by name. The first form matches only when the element is of the specified type, in case of `only` only if it is of exactly that type and not a subtype. The second form matches only when the element is of the specified name, given as string constant (rules of that kind are typically created interactively, but due to the persistence of persistent names and the execution of GrGen being as deterministic as possible in between single runs, even static rules make sense – even more so if you assign the names on your own).

The sequence expression finally allows us to decide conditionally. It is evaluated when its corresponding configuration rule is evaluated because its graph change event occurred, if it returns true the configuration rule matches, otherwise it does not match. The `this` entity is overloaded in the sequence expression (normally it denotes the graph). It gives access to the node or edge that was just created, or is getting deleted, or is getting retyped, or was assigned to. So here we find support for conditional data breakpoints.

27.9 Debugging related functionality

Other commands that are of use for debugging were already introduced in the shell chapter:
`show var <Variable>` to print the content of a variable and
`show <GraphElement>.<AttributeName>` to print the content of an attribute.
But pressing the `v` key in the shell debugger is more convenient in the former case, as is searching with `Ctrl-f` or `/` in yComp or MSAGLExt for the persistent name or an attribute value, hovering over the then highlighted graph element, in the latter case.

The commands `record` and `replay` are of interest when you want to follow different paths during a transformation, they allow you to save the graph states before choosing in between the different paths, and to restore them later on (and to inspect the sequence of changes leading to a graph state).

Another development aid comes in the form of validity checking code that gets emitted when the `-debug` option is supplied to `grgen.exe` (cf. 29.1), the usage of that option can be also configured in the shell (cf. 25.10). The validity checking code detects with runtime checks situations in which:

1. a rule is called with a null argument (unless it was specified that null parameters are allowed, in that case the missing element is searched for in the graph)

2. a rule is called with a graph element as parameter that is not contained in the graph anymore

3. a match is about to be rewritten with a matched graph element that is not contained in the graph anymore

The first case typically happens when an initialization of a variable is missing, or when a graph global variable gets auto-nulled, because the graph element was deleted.

The second case typically happens when a node or edge was removed from the graph but is still held in a local variable or a storage.

The third case typically happens during an all-bracketed rule application that matched a graph element multiple times, and now wants to retype the graph element, after having executed a previous rewrite, that deleted that particular graph element (an all-bracketed rule application rewrites one match after the other). The situation may be rather obvious, when the graph element was bound to the same pattern element in different matches, it may be also quite difficult to grasp, because the graph element was bound to different pattern elements in different matches. The check may fail without uncovering a real issue, because the already deleted element is not to be modified with the rewrite of the current match (or only to be deleted again, deletion after deletion is not causing a crash).

Regarding the third case, note that an edge not contained in the graph anymore is not reported as invalid in case its source or target nodes are not contained in the graph anymore (this saves you from spurious error reports, at the price of potentially missing a situation where this is not anticipated). By annotating a node/edge with a `validityCheck=false`, cf. Section 34.10, you can disable the contained-in-graph check for that graph element. By using the same annotation on a rule (cf. Table 34.1), you can disable this validity check for all graph elements that were matched during the execution of the rule.

In addition to the `-debug` flag, you could specify the `-keep` flag, then the generated source code is kept, which allows for easier bug hunting (we recommend you do so when you run into debugging issues).

The persistent graph (cf. 26.1) could also be used as a debugging aide. It allows to find dangling references to zombie nodes/edges, that are contained in attributes of graph element type, or attributes of a container type containing graph elements — they are detected and reported when a persistent graph is loaded (i.e. deferred, not when a node/edge turns into a zombie, because it gets deleted/retyped).

CHAPTER 28

VALIDATION AND INSPECTION

In order to validate, i.e. test your graph rewrite system, be it manually or automatically, you may employ the means described in this chapter (note that due to its well-founded semantics, even formal verification would be possible, cf. Chapter 36).

Available are model and graph queries, they allow you to fetch information of interest from the (meta) model and/or the host graph (instance), be it for validation or just for inspection. Also available are dedicated validation commands, they allow you to check the connection assertions of the model or that a condition expressed as a sequence holds (in contrast to a normal sequence execution that may be used for the same purpose, the sequence may carry out destructive changes, as it operates on a clone of the graph).

When you use GrGen.NET at API level, you are free to employ a .NET test framework of your choice. Alternatively, you may use the GrShell for testing. We use the GrShell together with bash shell scripts for automated testing of the integrity of GrGen.NET itself. You may employ the same scripts for testing your project (if you use Windows, you need the cygwin environment). The testing workflow consists of the application of two scripts:

- the bash shell script `gentest.sh` records the textual output of certain GrShell-commands.
- the bash shell script `test.sh` compares the recorded output with the current output of the GrShell and reports differences as test failures.

You find them in the `tests` folder (of `engine-net-2` when you are working with the sources — the `test` folder (from the `frontend` directory when you are working with the sources) contains further scripts for compilation-only tests — both test suites are quite slow in execution, as they are not unit tests).

Besides the `show` and `validate` commands defined in this chapter in the following sections, also the result of the `show var` command (cf. 25.4) is recorded (and checked against). In addition, the results of sequence `execution` (a boolean value) and sequence expression `evaluation` (cf. 25.4) are recorded (and checked against), the same holds for the number of `matches found` and `rewrites performed` during execution of a sequence (expression). (Even if this is a bit ugly: `emit("the value of variable \"x\" of type dontcare is: ", x, "\n")` — by emitting from your rules or procedures to stdout in a way like the commands do, you could add values to check to a test, the statement just introduced would allow to check that the value of the variable `x` stays the same once the testdatafile was generated (an alternative would be to adapt the test scripts).)

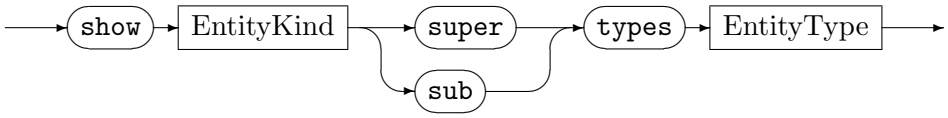
You may also use the return code of the GrShell for testing when it is started with the `-N` parameter, in normal interactive mode it will abort script file execution when an error occurs so the user can continue at this point, in non-interactive mode asked for with the parameter, instead an error code is returned when an error occurs or a validation fails, see Chapter 29 for more on this.

28.1 Model and Graph Queries

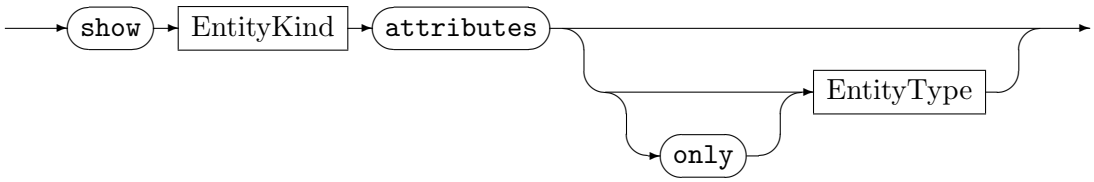
We start with the queries returning information about the static types (from the (meta) model).



Prints the types of the respective entity kind (`node`/`edge`/internal class `object`/`transient` internal class `object`) from the current graph model.

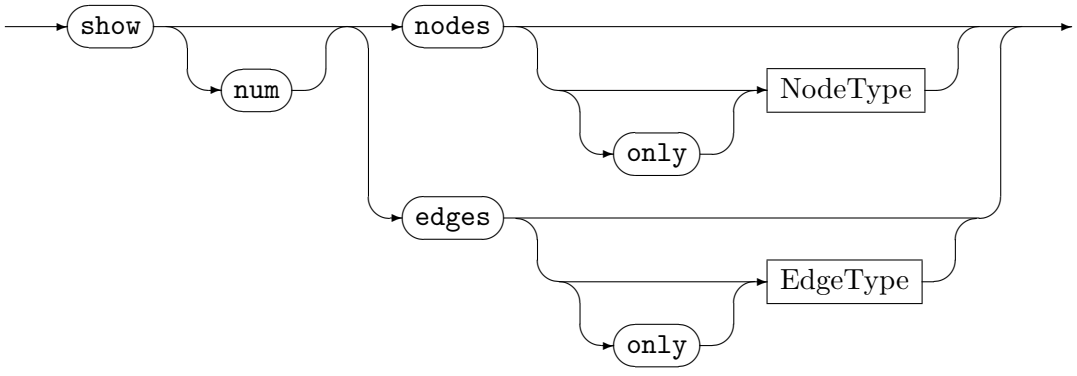


Prints the inherited/descendant types of the given *EntityType* (*NodeType*/*EdgeType*/*ObjectType*/*TransientObjectType*); the kind identifier must fit to the specified type.



Prints the available attribute types. If an *EntityType* is supplied, only attributes defined in the *EntityType* are displayed (otherwise all attributes known to all types of the specified *EntityKind*). The `only` keyword excludes inherited attributes.

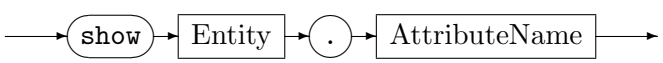
Now to the queries returning information about the dynamic values from the current graph.



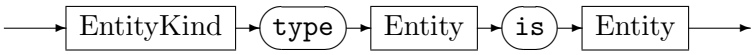
Prints the `num`ber of nodes/edges in the graph, or their actual instances in form of their persistent name together with their type. If a node type or edge type is supplied, only elements compatible to this type are shown/considered. The `only` keyword excludes subtypes.



Prints the attribute types and values of a specific entity (again, the *EntityKind* must match the *Entity*).



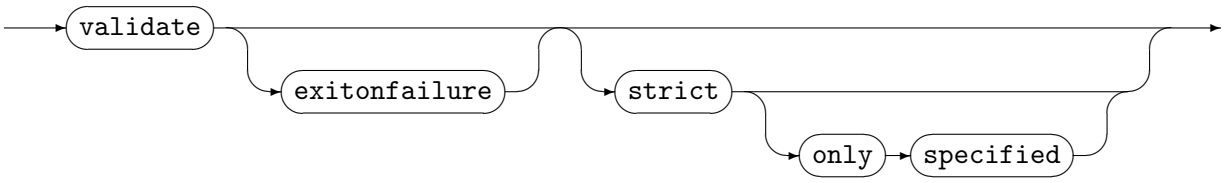
Displays the value of the specified attribute.



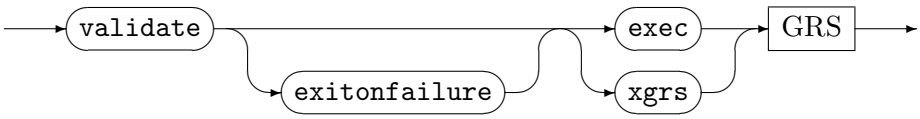
Prints the information whether the first entity is type-compatible to the second entity (the entities must be of the same kind as specified with the *EntityKind*).

28.2 Validation Commands

GrGen.NET offers two different graph validation mechanisms, the first checks against the connection assertions specified in the model, the second checks against an arbitrary graph rewrite sequence containing arbitrary tests and rules.



Validates if the current working graph fulfills the connection assertions specified in the corresponding graph model (cf. 4.2.1). Validate without the strict modifier checks the multiplicities of the connections it finds in the host graph, it ignores node-edge-node connections which are available in the host graph but have not been specified in the model. The *strict* mode additionally requires that all the edges available in the host graph must have been specified in the model. This requirement is too harsh for models where only certain parts are considered critical enough to be checked or might be too big a step in tightening the level of structural checking in an already existing large model. So some form of selective strict checking is supported: The *strict only specified* mode requires strict matching (i.e. that all edges are covered) only of the edges for which connection assertions have been specified in the model.



Validates if the current working graph satisfies the graph rewrite sequence given. Before the graph rewrite sequence is executed, the instance graph gets cloned; the sequence operates on the clone, allowing you to change the graph as you want to, without influence on the host graph. Validation fails iff the sequence fails. This gives a rather costly but extremely flexible and powerful mechanism to specify graph constraints. The GrShell is exited with an error code if `exitonfailure` is specified and the validation fails.

EXAMPLE (164)

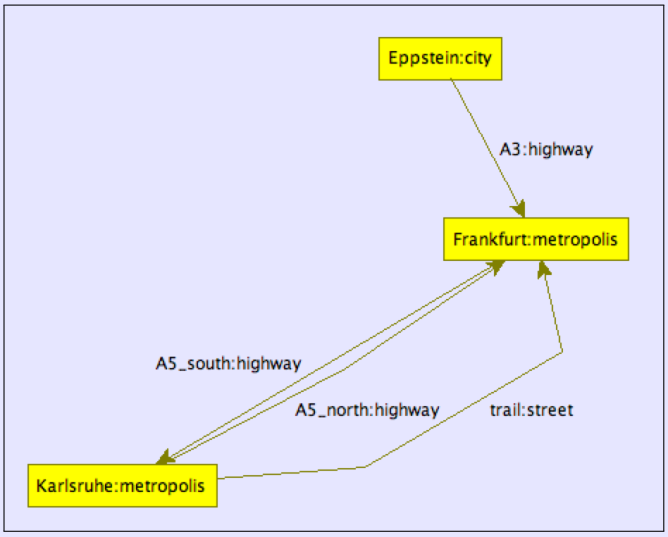
We reuse a simplified version of the road map model from Chapter 4:

```
model Map;

node class city;
node class metropolis;

edge class street;
edge class highway
      connect metropolis [+] --> metropolis [+];
```

The node constraint on *highway* requires all the metropolises to be connected by highways. Now have a look at the following graph:

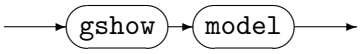


This graph is valid but not strict valid.

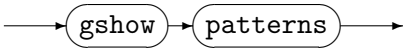
```
> validate
The graph is valid.
> validate strict only specified
The graph is NOT valid:
  CAE: city "Eppstein" -- highway "A3" --> metropolis "Frankfurt" not specified
> validate strict
The graph is NOT valid:
  CAE: city "Eppstein" -- highway "A3" --> metropolis "Frankfurt" not specified
  CAE: metropolis "Karlsruhe" -- street "trail" --> metropolis "Frankfurt" not specified
>
```

28.3 Queries with Graphical Output

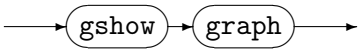
The following queries show information graphically (and thus are not amenable to test automatization, or at least less well suited), with the currently configured graph viewer/debugger.



Shows the graph model as a graph similar to a UML inheritance diagram.



Shows the patterns of the actions as a graph (note that edges are rendered as nodes, so that pattern nesting can be rendered correctly).



Shows the graph (in contrast to the `show graph` command from 27.1 that requires to specify the graph viewer explicitly, this one uses the currently configured graph viewer/debugger).

CHAPTER 29

COMPONENTS AND APPLICATIONS

GrGen.NET is combined from two groups of components: The first consists of the compiler `grgen` and some runtime libraries, which offer the basic functionality of the system; the compiler transforms specifications of declarative graph rewrite rules into highly efficient .NET-assemblies. With one of these libraries, you can optionally enable automatic graph persistence. The second consists of the interactive command line `(G)GrShell` and the graph viewers `yComp` and MSAGL with some further runtime libraries, which offer a rapid prototyping environment supporting graphical and stepwise debugging of rules that are controlled by sequences. These functionalities are also available on API level, you can add graph visualization, stepwise sequence/rule debugging, or even shell scripting to your end user app.

In the following, we are going to visit them component by component.

The programs and libraries of GrGen.NET are open source licensed under LGPL[Fou07]. Notice that the yComp graph viewer is not a part of GrGen.NET; yComp (cf. https://pp.ipd.kit.edu/firm/yComp.html) is closed source and ships with its own license granted by yFiles for academic use – this means above all that you are not allowed to ship it with a release of your own commercial software, in contrast to the GrGen.NET libraries. To be more specific: an academic license restricts the use of the software (yComp) to *non-commercial* purposes (research, teaching, projects, courses and application development). Utilizing the MSAGL based graph viewers saves you from license issues due to the permissive MIT license of the MSAGL library, but they lack the sophistication of yComp. The persistent graph is implemented (by an internal persistence provider) utilizing the SQLite database and its ADO.NET drivers, which are in the public domain.

Executing a generated graph rewrite system requires the .NET Framework[Mic07] version 4.7.2 or mono[Tea07] version 5.10, or later versions (maybe complete packaging); the same holds for compiling/generating it. Building GrGen.NET itself on the other hand and debugging a graph rewrite system with yComp in addition requires JAVA[Ora96] version 1.8 or later. You find the binaries (tool applications and components) in the `bin` subdirectory of your GrGen.NET installation.

29.1 `GrGen.exe`



The `GrGen.exe` assembly implements the GrGen.NET generator. The GrGen.NET generator parses a rule set and its model files and compiles them into .NET assemblies. The compiled assemblies form a specific graph rewriting system together with the GrGen.NET backend.

Usage

```
[mono] GrGen.exe [-keep [<dest-dir>]] [-use <existing-dir>] [-debug]
                 [-b <backend-dll>] [-o <output-dir>] [-r <assembly-path>]
                 [-profile] [-nodebugevents] [-noevents]
                 [-statistics <statisticsfile>]
                 [-lazynic] [-noinline]
                 <rule-set>
```

rule-set is a file containing a rule set specification according to Chapter 5. Usually such a file has the suffix `.grg`. The suffix `.grg` may be omitted. By default GrGen.NET tries to write the compiled assemblies into the same directory as the rule set file. This can be changed by the optional parameter *output-dir*.

Options

`-keep` Keep the generated C# source files. If *dest-dir* is omitted, a subdirectory `tmpgrgen`*n*[1] within the current directory will be created. The destination directory contains:

- `printOutput.txt`—a snapshot of `stdout` during program execution.
- *Name*`Model.cs`—the C# source file(s) of the *rule-set*`Model.dll` assembly.
- *Name*`Actions_intermediate.cs`—a preliminary version of the C# source file of the *rule-set*'s actions assembly. This file is for internal debug purposes only (it contains the frontend actions output).
- *Name*`Actions.cs`—the C# source file of the *rule-set*`Actions.dll` assembly.

`-use` Don't re-generate C# source files. Instead use the files in *existing-dir* to build the assemblies.

`-debug` Compile the assemblies with debug information (also adds some validity checking code).

`-b` Use the backend library *backend-dll* (default is LGSPBackend).

`-o` Store generated assemblies in *output-dir*.

`-r` Link the assembly *assembly-path* as reference to the compilation result.

`-profile` Instruments the matcher code to count the search steps carried out.

`-nodebugevents` Action events (used in debugging) are not fired by the compiled sequences (/rules). Embedded execs are executed faster, but the debugger breaks.

`-noevents` Beside action events, attribute change events are not fired (by the rules). For maximum performance of plain rules, but in addition to debugging, transactions/backtracking breaks, as well as record/replay.

`-statistics` Generate matchers that are optimized for graphs of the class described by the *statisticsfile* (see 25.12 on how to save such statistics).

`-lazynic` Negatives, Independents, and Conditions are only executed at the end of matching (normally asap).

`-noinline` Subpattern usages and independents are not inlined.

Requires

.NET Framework 4.0 (or above) or Mono 3.0 (or above).

[1] *n* is an increasing number.

NOTE (57)

Regarding the column information in the error reports of the compiler please note that tabs count as one character.

NOTE (58)

As .NET supports integration with native code, you may even use a GrGen.NET-generated graph rewriting kernel from a native application.

NOTE (59)

The grgen compiler consists of a frontend `FrontendGrGen.exe` used by the C# backend and driver `GrGen.exe`. The frontend can be executed directly to get a visualization of the model and the rewrite rules, in the form of a dump of the compiler IR as a .vcg file:
`FrontendGrGen.exe -i yourfile.grg`

NOTE (60)

If you run into `Unable to process specification: The system cannot find the file specified` errors, `GrGen.exe` is typically not able to execute the frontend `FrontendGrGen.exe`. Likely the system is not referenced in the search path, the error message should give further hints.

29.2 GrShell.exe



The `GrShell.exe` is a shell application on top of the libGr. GrShell is capable of creating, manipulating, and dumping graphs as well as performing graph rewriting with graphical debug support. For further information about the GrShell language see Chapter 25.

Usage

`[mono] GrShell.exe [-N] [-SI] [-C "<commands>"] <grshell-script>*`
Opens the interactive shell. The GrShell will include and execute the commands in the optional list of *grshell-script*s (usually `*.grs` files) in the given order. The `grs` suffixes may be omitted. GrShell returns 0 on successful execution, or in non-interactive mode -1 if the specified shell script could not be executed, or -2 if a `validate` with `exitonfailure` failed. This allows you to build a test-suite consisting of shell scripts.

Options

`-N` Enables non-debug non-gui mode which exits on error with an error code instead of waiting for user input.

`-SI` Show Includes prints out to the console when includes are entered and left.

`-C` Execute the quoted GrShell commands immediately (before the first script file). If you want the quoted part on a single line (so without line breaks), you have to use a double semicolon `;;` to separate commands. Take care that an exec inside such a command line needs to be exited with `#§` then (to open and immediately close a shell comment, needed as exec terminator in face of missing newline termination).

Requires
.NET Framework 4.7.2 (or above) or Mono 5.10 (or above).

NOTE (61)

The shell supports some `new set` configuration options that map to the `grgen` compiler flags (see 25.10), use them before any `new graph` commands so that matchers are generated according to the compiler flags you would use if you would execute the compiler directly.

EXAMPLE (165)

The shell script `GrgenifyMovieDatabase.sh` from `examples/MovieDatabase-TTC2014` shows how to execute an embedded GrShell-script on all files in a folder, in a version with the `-C` option with line breaks, a version on a single line, and a version employing a here-document circumventing the option this way.

29.3 GGrShell.exe



The Graphical `GrShell.exe` is a shell application on top of the libGr that is implemented as a WindowsForms application (instead of a console application, which is the case for the `GrShell.exe`). It fits better to the internal MSAGL based Windows Forms debugger (that can also be employed from GrShell). The GGrShell supports the same language that is offered by the GrShell see Chapter 25.

29.4 libGrShell.dll

The `libGrShell.dll` contains the shared implementation of the GrShell and the GGrShell (the differences are very small). You may use it in your own applications (build a .NET application on top of it, or even a native one based on .NET-interop) to include graph rewrite shell-like functionality, i.e. offer the user a command line interface that allows to execute graph rewrite scripts.

29.5 graphViewerAndSequenceDebugger.dll

The `graphViewerAndSequenceDebugger.dll` contains code that allows to display graphs (with yComp or extMSAGLExt) and debug sequences on a console. You may use it in your own applications (build a .NET application on top of it, or even a native one based on .NET-interop) to include graph display with an external app and console-based sequence debugging.

29.6 graphViewerAndSequenceDebuggerWindowsForms.dll

The WindowsForms extension of the `graphViewerAndSequenceDebugger.dll` contains code that allows to use the MSAGL graph viewer, and debug sequences with a graphical debugger. You may use it in your own applications (build a .NET application on top of it, or even a native one based on .NET-interop) to include in-app graph display (based on the .NET library MSAGL) and graphical sequence debugging. The MSAGL library can be used nearly freely due to its MIT license. The extMSAGLExt graph viewer application (see below) is built on top of it.

29.7 lgspBackend.dll

The LGSPBACKEND is a .NET assembly containing the libGr SearchPlan backend, the only backend supported by GRGEN.NET as of now[2], implementing together with the generated assemblies the API offered by LIBGR. It allows to analyze the graph in order to create a new search plan and to regenerate the matcher programs at runtime, on user request, see 25.12. For a more detailed introduction have a look at chapter 35.

29.8 libGrPersistenceProviderSQLite.dll

The `libGrPersistenceProviderSQLite.dll` contains code that allows for persistent storage of the host graph to an SQLite database file (including writing of ongoing changes, so they are available at the next program start even in case of a crash). The SQLite assemblies are included in a release of the GRGEN.NET-project as of now; they are in the public domain and thus may be used freely (the LINQ and EntityFramework libraries are covered by the Microsoft Public License, but these are not used by GRGEN.NET). The persistent named graph of the lgspBackend may be also implemented by different persistence providers, but the only persistence provider supported by GRGEN.NET as of now is the SQLite based one.

29.9 libGr.dll

The LIBGR (graph rewrite library) is a .NET assembly defining GRGEN.NET's API. See the extracted HTML documentation for interface descriptions at http://www.grgen.de/doc/API/; a short introduction is given in chapter 33. Besides, it implements some of the functionality (generically, independent of backends or the concrete models). You have to include it in your own applications to access the base functionality of GRGEN.NET (and the code generated from your model and actions specifications, in order to get a graph rewriting based algorithmic core – if you also want graph display or even shell scripting, you have to include further assemblies).

29.10 libConsoleAndOS.dll

The base library `libConsoleAndOS.dll` offers interfaces for text consoles and a basic implementation, as well as some OS abstractions (you have to include it in your own application if you want to use GRGEN.NET, also the libGr requires it).

29.11 libConsoleAndOSWindowsForms.dll

The WindowsForms extension of the `libConsoleAndOS.dll` contains a GUI console implementation for WindowsForms apps (that is used by the debugger from the `graphViewerAndSequenceDebuggerWindowsForms.dll` and the GGRSHELL, you have to include it in your own WindowsForms application if you want to use GRGEN.NET).

29.12 MSAGL assemblies/libraries

In order to use the MSAGL graph viewer on API level, you have to ensure that the following libs are available in your search path (you find them in the `bin` directory):
`Microsoft.Msagl.Drawing.dll`, `Microsoft.Msagl.GraphViewerGdi.dll`, `Microsoft.Msagl.dll`, `System.Buffers.dll`, `System.Memory.dll`, `System.Numerics.Vectors.dll`, `System.Resources.Extensions.dll`, `System.Runtime.CompilerServices.Unsafe.dll`

[2] the compiler frontend supports a non-.NET backend targeting the FIRM compiler, generating C code

29.13 SQLite assemblies/libraries

In order to use the persistent graph, or better the SQLite persistence provider on API level, you have to ensure that the `System.Data.SQLite.dll` is available in your search path, besides the `libGrPersistenceProviderSQLite.dll`, and the `e_sqlite3.dll` in case of Windows or the `libe_sqlite3.so` in case of LINUX (note that the latter two libraries are native x64-builds of SQLite, and that the release of GrGen.NET is a 64bit-only version by now, you may build the x86-version(s) on your own, see Section 35.1). You find them in the `bin` directory.

29.14 `extMSAGLExt.exe`

extMSAGLExt is a graph visualization application based on the `MSAGLExt` control from `graphViewerAndSequenceDebuggerWindowsForms.dll` (and intended to be the open-source successor of yComp in GrGen.NET). extMSAGLExt implements several graph layout algorithms (the ones offered by the Microsoft Automatic Graph Layout graph drawing library MSAGL[**?**]) and supports files in VCG format, which is originating from the VCG[San95] tool, which in turn is based on the graph representation language GRL from the EDGE[PT90] tool.

Usage

Usually extMSAGLExt will be loaded by the GrShell, and esp. by its debugger, it will run in server mode in the latter case. You might want to open extMSAGLExt manually by typing

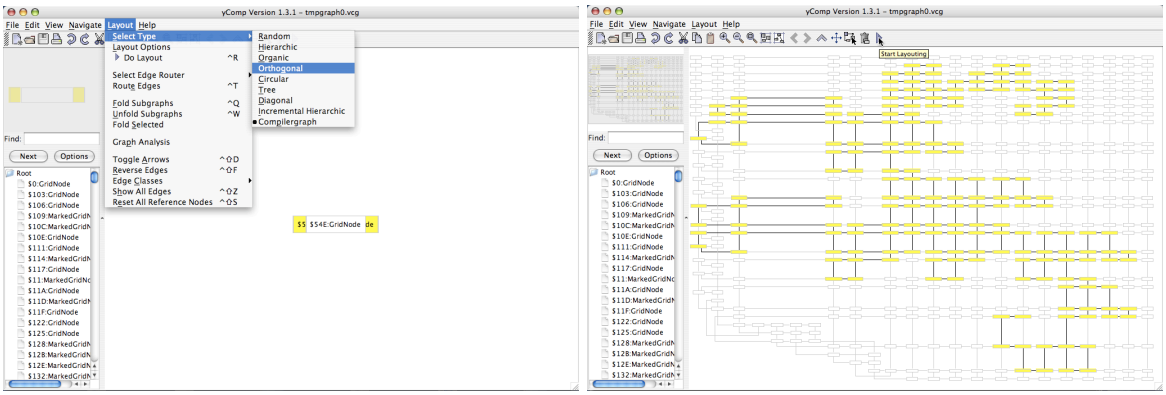
```
extMSAGLExt.exe [<graph-file>]
```

Or by executing the batch file `extMSAGLExt` under Linux, which will start extMSAGLExt on the given file adding the `mono` prefix as needed.

The program offers a menu entry to show an error log console (and even a `--console` argument you may give at startup), this is primarily for Windows, under Linux, error message are printed on the parent console. When you add a `--autoreload` option before the filename, extMSAGLExt is going to watch the file and reload it when it changes (this may be used when you `dump` multiple times into the same file). You may use the `--help` option to list all available arguments.

29.15 `yComp.jar`

yComp [KBG+07] is a graph visualization tool based on yFiles [yWo07]. It is well integrated and shipped with GrGen.NET, but it's not a part of GrGen.NET; it is supplied by IPD Snelting, the successor institute of IPD Goos.yComp implements several graph layout algorithms and has file format support for VCG, GML and YGF among others.



Usage

Usually yComp will be loaded by the GrShell. You might want to open yComp manually by typing

`java -jar yComp.jar [<graph-file>]`

Or by executing the batch file `ycomp` under Linux / `ycomp.bat` under Windows, which will start yComp on the given file with increased heap space. The *graph-file* may be any graph file in a supported format. yComp will open this file on startup.

Hints

The layout algorithm compiler graph (yComp's default setting, a version of `hierarchic` optimized for graph based compiler intermediate representations) may not be a good choice for your graph at hand. Instead `organic` or `orthogonal` might be worth trying. Use the rightmost blue play button to start the layout process. Depending on the graph size this may take a while.

Requires

Java Runtime Environment 1.5 (or above).

You may use the `--autoreload` option to watch a file for changes or the `--help` option to list all available arguments.

NOTE (62)

If yComp doesn't start: check whether your path contains a zombie java variable from an old installation pointing to nowhere; remove it in this case. Normally just installing a JRE is sufficient, but it may help to install a JDK to a non system path and adding the bin folder of the JDK to the path variable. It could be necessary to utilize the OpenJDK in an old version (e.g. `1.8`) to get a running yComp.jar because of incompatibilities introduced later on (or may be necessary to use JAVA from Oracle, which is normally the better choice regarding compatibility anyway).

29.16 Birds eye view

Figure 29.1 gives an overview of the GrGen.NET system components and applications (with their dependencies).

Figure 29.2 depicts how an application you build with GrGen.NET (at API level) would look like when fit into a traditional three layer architecture (TLA), composed of a presentation layer facing the user, the application logic layer (with the algorithmic core), and the persistence layer.

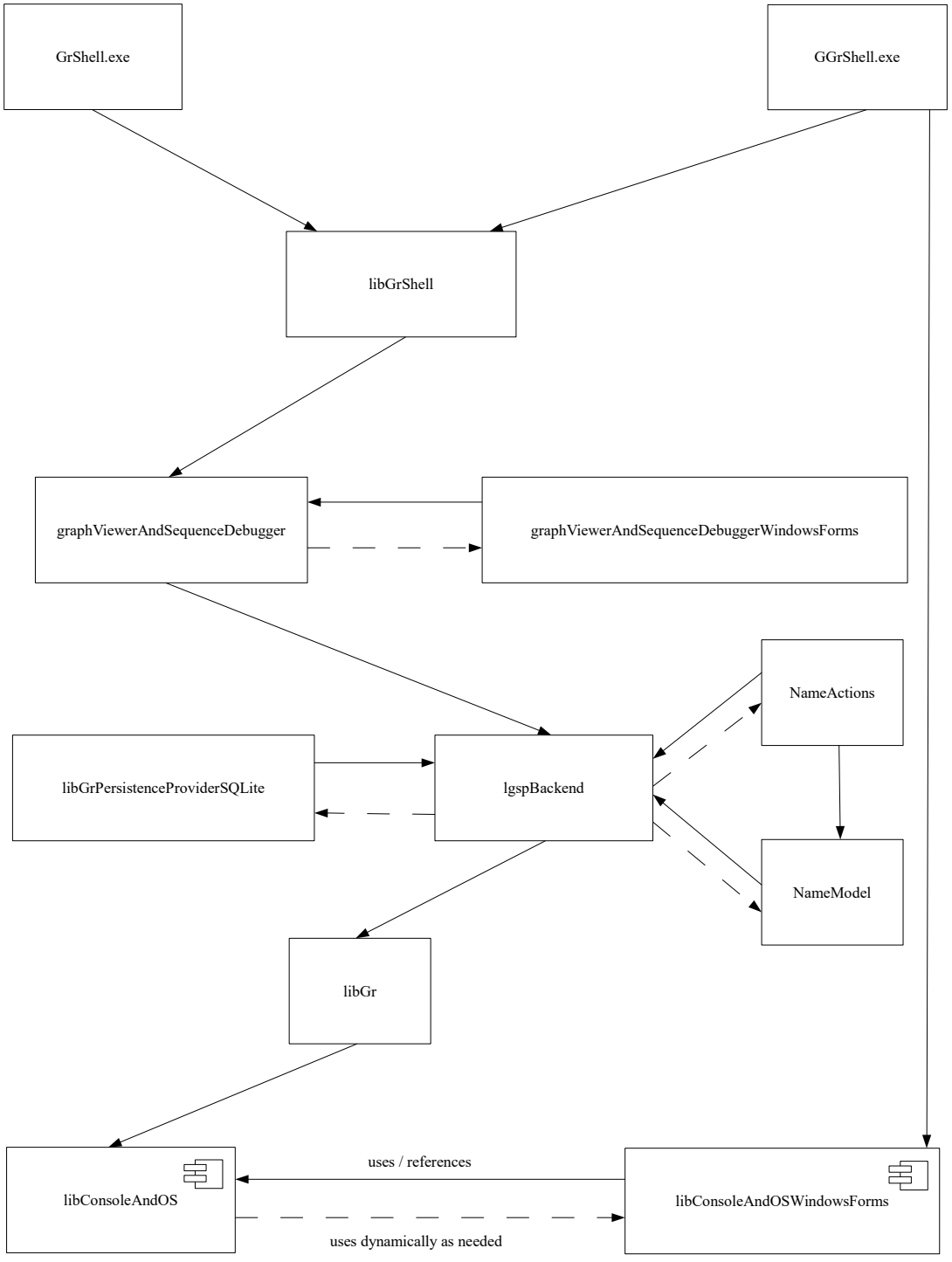


Figure 29.1: GrGen.NET system components and applications (draft)

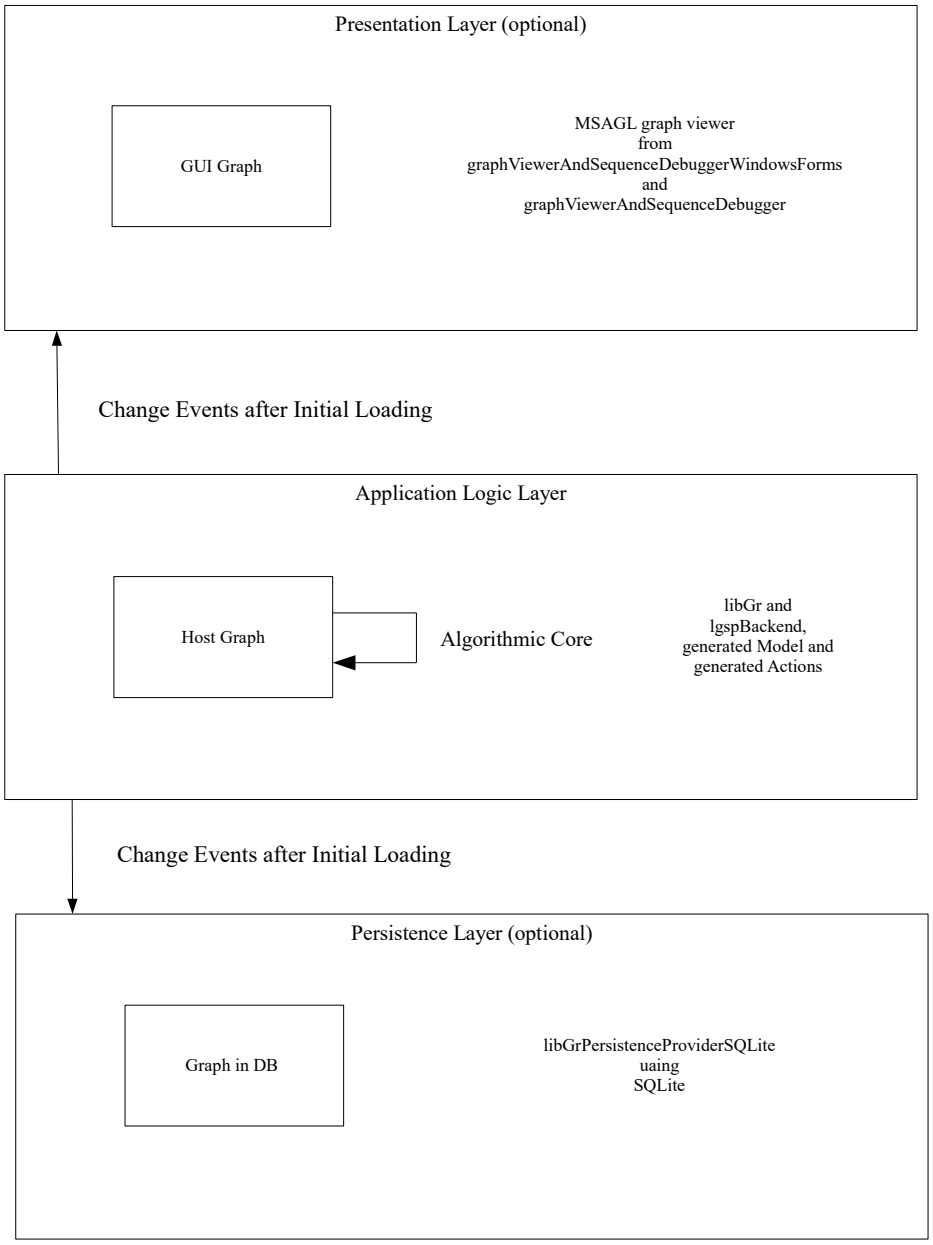


Figure 29.2: GrGen.NET three layer architecture (draft)

CHAPTER 30

INDICES

Matching patterns in a graph is based on searching for the elements of interest. It is better to find elements with certain characteristics straight ahead without search, utilizing a data structure that allows telling the elements that follow the characteristic apart from the ones that do not, than to search for them, traversing each and every node or edge in the graph, subjecting it to a test for the characteristic.

Welcome to indices as known from database parlance. In GrGen.NET the following types of indices are supported:

1. Type indices
2. Neighbourhood indices
3. Attribute indices
4. Incidence count indices
5. Name index
6. Uniqueness index

30.1 Type Indices

All the nodes or edges in GrGen.NET of a certain type are contained in a list that can be accessed in $O(1)$ and iterated in $O(k)$ with k being the number of elements in that list (the nodes or edges of same type), in contrast to n, being the number of nodes or edges in the graph. The first node or edge in a pattern is typically bound by iterating such a type list. In case the pattern is disconnected, a lookup is needed per connected component. In case multiple types have very few members, the search planner may decide to use several lookups even in case of a connected pattern. (This can be especially the case for reflexive marker edges pointing to the current point of processing/focus of attention in the graph.)

30.2 Neighbourhood Indices

All the edges outgoing from a node are contained in a list that can be accessed in $O(1)$ and iterated in $O(k)$ with k being the number of elements in that list (the outgoing edges), in contrast to n, being the number of edges in the graph. All the edges incoming to a node are contained in a list that can be accessed in $O(1)$ and iterated in $O(k)$ with k being the number of elements in that list (the incoming edges). So following neighbouring elements, crawling alongside the structure is very cheap in GrGen.NET – and possible in both directions, following the edges as well as in their opposite direction.

It is cheap in graph databases, too, but absolutely not in relational databases. To follow a relation, there you have to join two complete tables, building the cartesian product of two tables (materializing only the rows where they agree) – you must inspect all edges in the

graph to find the neighbours. Typically you optimize this with a database index that allows to find the elements of the second table matching your focused element of the first table in $O(log(n))$ – but for large n or for complex queries this global product is way less efficient than direct local access to exactly the neighbouring elements. (Furthermore, a table join is conceptually and notationally much heavier than an edge in a graph pattern).

In case of undirected edges, or arbitrary directed edges in the pattern, both directions are searched, both lists – the outgoing list as well as the incoming list – are crawled.

30.3 The Costs

The type and neighbourhood indices are built into each and every GrGen.NET graph, wired into a system of ringlists (cf. 35.3). They are implicitly used by GrGen.NET, pattern matching is carried out alongside them, in an order decided upon by the search planner.

You always benefit from them during matching (most if you are using statistics-based search planning), but you must always pay their price: The memory consumption of a GrGen.NET node without attributes is 32 bytes using a 32bit CLR (5 pointers a 4 bytes, a 4 bytes flags field/bitvector, plus 8 bytes .NET object overhead), it is 60 bytes for a 64bit CLR (5 pointers a 8 bytes, a 4bytes flags field, plus 16 bytes .NET object overhead). The memory consumption of a GrGen.NET edge without attributes is 48 bytes using a 32bit CLR (9 pointers a 4 bytes, a 4 bytes flags field/bitvector, plus 8 bytes .NET object overhead), it is 92 bytes for a 64bit CLR (9 pointers a 8 bytes, a 4 bytes flags field, plus 16 bytes .NET object overhead). Attributes are cheap, they only increase this by the .NET-memory-footprint of their underlying objects.

The runtime price of maintaining those indices during graph manipulation is low, adding a graph element is done in $O(1)$, removing a graph element is done in $O(1)$, too. Furthermore, those indices allow optimizing loops by letting the pattern matching of the next iteration start where the previous iteration left off (search state space stepping – a further built-in contribution to the motto "Find, Don't Search")).

We consider this the best you can get for the task of efficient pattern matching and rewriting on typed, attributed multigraphs, with multiple inheritance and attributes on nodes as well as edge types – those indices are simply worth their price in the general case. You can only improve on this by exploiting the specifics of a less general graph model, a less general processing strategy supporting less change operations optimally out-of-the box, or by restricting yourself to less general and non-automatic pattern matching. E.g. by using only single-linked ringlists for the types and incident elements, we could reduce memory consumption considerably and thus increase query performance, but deletion and retyping would require to iterate the full list to get to the predecessor to link it to the new successor, so rewriting would not be $O(1)$ anymore. Or by always entering the graph at some elsewhere stored root nodes we could save the type ringlists (that give us the modularity of the rules). Or by using less general edges/edges that appear only with a statically fixed association cardinality, we could reduce them to mere pointers stored in the nodes (not supporting attributes, not allowing to wire nodes at runtime flexibly with an arbitrary number of edges).

GrGen.NET was built for general-purpose graph pattern matching and rewriting on a general model, to perform well on *any* task, with *minimum programming effort* for the user; regarding this it is implemented in a way we consider optimal and not improvable. If you are willing to invest extra programming and understanding effort to exploit the specifics of your task at hand, you *can* manually code a higher-performance solution, by stripping away features from the model you don't need. Saving on the memory needed for the generality and programming productivity is what gives our competitor FUJABA a lead performance-wise for tasks where a simpler graph model is sufficient and the user is willing to invest more development time into (and no iterative search-replace-operations are carried out, search state space stepping wins then).

30.4 Attribute Indices

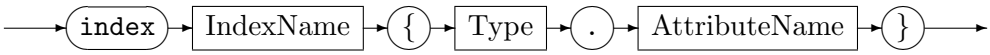
In addition to the built-in type and neighbourhood indices, you may declare attribute indices. An attribute index allows to execute a graph element lookup based on an attribute value of interest, or a range of attribute values. This stands in contrast to the default behaviour of carrying out a lookup on a type, visiting all n elements of the type, filtering them down to the elements of interest (within range) with an attribute condition. If this means you have to inspect a lot of values while searching only for a few ones, you should use an attribute index and benefit from its massively improved selectivity for the lookup. It requires only $O(log(n))$ to search for the first element, and $O(k)$ for the k elements within the bounds specified. It is implemented with a balanced binary search tree (an AA-tree[And93] to be exact) that requires three pointers plus one 4 byte integer per element contained in the index (two pointers to the left and right tree nodes, and one to the graph element as value), which is really cheap. But it must be maintained on graph changes, which is less cheap. On each and every graph element insertion and removal, but esp. attribute assignment, the index has to be updated, which is an $O(log(n))$ operation. That's only logarithmic, but clearly worse than the default $O(1)$ behaviour of GrGen.NET, so if you do a lot of graph manipulations and only few lookups based on it, an index may in fact degrade performance.

Declaration in the model

In contrast to type and neighbourhood indices that are always available (they define the core of a GrGen.NET-graph) and implicitly used (when matching a specified pattern), all other indices have to be worked with explicitly.

An attribute index must be declared in the model.

AttributeIndexDecl

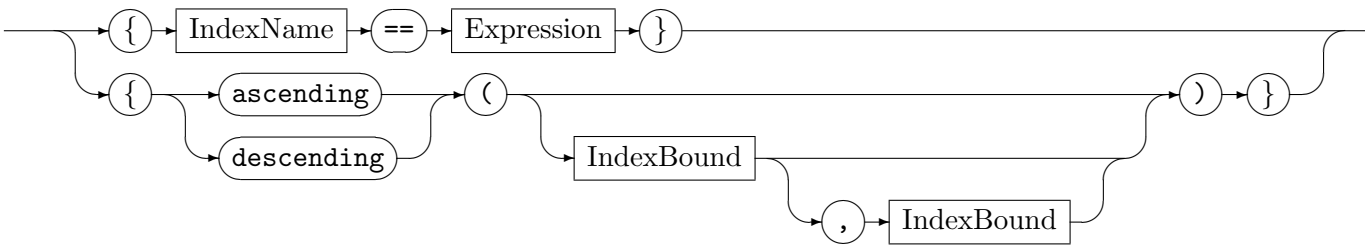


Following the `index` keyword, a name for the index is specified; in the body of the index, the type and name of the attribute to be indexed are given.

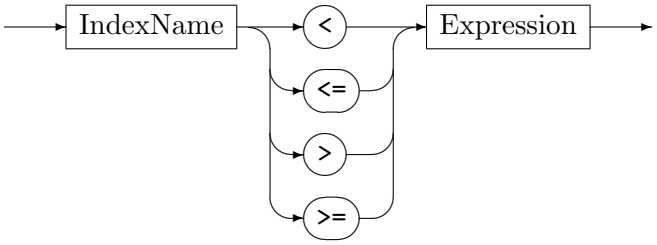
Usage in the rules

In the pattern part you may ask for an element to get bound to an element from an index; this is syntactically specified by giving the index access enclosed in left and right braces (curly brackets) after the element declaration. If the type of the element retrieved from the index is not compatible to the type of the pattern element specified, the retrieved candidate is not bound to the pattern element, and matching continues with the next candidate from the index, if the available candidates fitting to the attribute condition are exhausted or if the index is empty, matching fails. The elements from the index are successively bound to the pattern element, then the rest of the pattern is bound and checked, until the requested number of matches is found.

IndexAccess

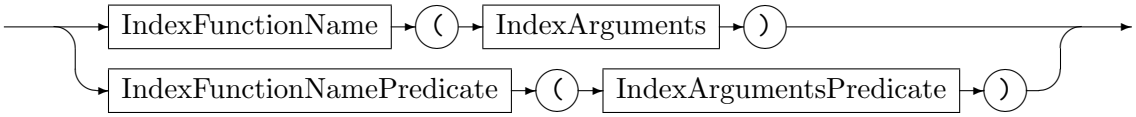


IndexBound

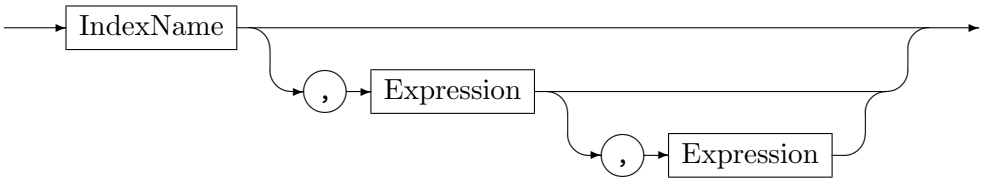


The pattern element may be bound to the elements from the index with an attribute value equal to a specified value, or to the elements from the index that lie within up to two bounds, in ascending or descending order (ordered access). In case of ordered access, you may specify no bound, this is typically only of interest when a single match is requested, the one with the highest or lowest attribute value (satisfying the other constraints of the pattern), or you may specify a lower bound, or an upper bound, or a lower *and* an upper bound.

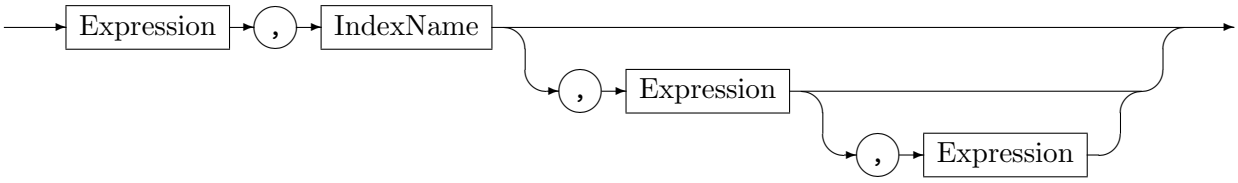
IndexFunctionCall



IndexArguments



IndexArgumentsPredicate



A second way to access an index is from index query functions which are available in the expressions of the rule language, as well as in the expressions of the sequence computations. There are multiple ones available, the index query functions returning a set of nodes are (the non-index-name arguments are of the type of the indexed attribute):

nodesFromIndex(.)
: returns all nodes from the given index, as set.

nodesFromIndexSame(.,.)
: returns all nodes from the given index that are equal to the value specified with the second argument, as set.

nodesFromIndexFrom(.,.)
: returns all nodes from the given index that are greater or equal than the second argument, as set.

nodesFromIndexFromExclusive(.,.)
: returns all nodes from the given index that are greater than the second argument, as set.

nodesFromIndexTo(.,.)
: returns all nodes from the given index that are lower or equal than the second argument, as set.

nodesFromIndexToExclusive(.,.)
: returns all nodes from the given index that are lower than the second argument, as set.

nodesFromIndexFromTo(.,.,.)
: returns all nodes from the given index that are greater or equal than the second argument and lower or equal than the third argument, as set.

nodesFromIndexFromToExclusive(.,.,.)
: returns all nodes from the given index that are greater or equal than the second argument and lower than the third argument, as set.

nodesFromIndexFromExclusiveTo(.,.,.)
: returns all nodes from the given index that are greater than the second argument and lower or equal than the third argument, as set.

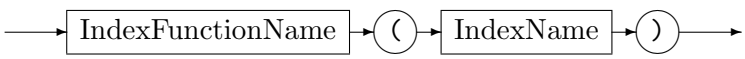
nodesFromIndexFromExclusiveToExclusive(.,.,.)
: returns all nodes from the given index that are greater than the second argument and lower than the third argument, as set.

The very same ones exist also for edges (their name starts then with `edgesFromIndex` instead of `nodesFromIndex`, and they return a `set<AEdge>` instead of a `set<Node>`).

The index query functions can be used in five possible ways (all functions support/require all the arguments listed above, in the following description we only give the index for brevity):

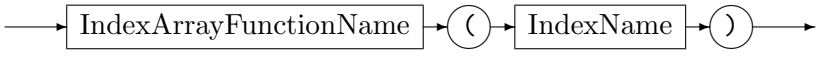
Set functions:
: The index query function returns a set of contained entities. It builds an unordered set that is likely thrown away thereafter, so esp. for large sets this function as such is less efficient than the other versions that inspect the index directly without building an intermediate data structure.

  Expression



Array functions:
: The index query function returns an array of contained entities. It builds an ordered array that is likely thrown away thereafter, so esp. for large arrays this function as such is less efficient than the other versions (with exception of the set version, as set building is more expensive than array building). It allows to specify whether the entities are to be ordered ascendingly alongside the indexed attribute or descendingly. Please note that the descending version requires the upper bound as first post-index argument and the lower bound as second post-index argument, whereas the ascending version like the unordered versions expects the lower bound as first post-index argument and the upper bound as second post-index argument.

  Expression

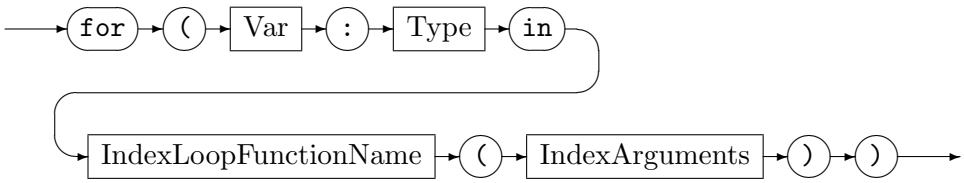


  The *IndexArrayFunctionName* is built from the *IndexFunctionName* by appending `AsArrayAscending` or `AsArrayDescending`. In case of the `nodesFromIndexSame` function only `AsArray` is appended.

Iteration loops:

The index function is employed in a for loop that allows to iterate the entities from the index. No set/array needs to be built here (but a statement is used/you must be in statement context). Please note that the descending version requires the upper bound as first post-index argument and the lower bound as second post-index argument, whereas the ascending version like the unordered versions expects the lower bound as first post-index argument and the upper bound as second post-index argument.

ForIndexFunctionHeader



The *IndexLoopFunctionName* is built from the *IndexFunctionName* by appending `Ascending` or `Descending`. In case of the `nodesFromIndexSame`/`edgesFromIndexSame` function the original *IndexFunctionName* stays.

So the index can be accessed from the statements of the rule language in the form of a (for function) loop, cf. 14.3. The iteration variable is bound to the graph element retrieved from the index, and then, for each such element, the body is executed.

Counted functions:

The counted index function returns the count of entities from the index. This is more efficient than calling `size()` on the resulting set/array of the plain index function; note that for fetching the count of all the entities in the index, `indexSize` is most efficient and should be preferred over `countNodesFromIndex`/`countEdgesFromIndex` ($O(1)$ instead of $O(n)$).

Expression

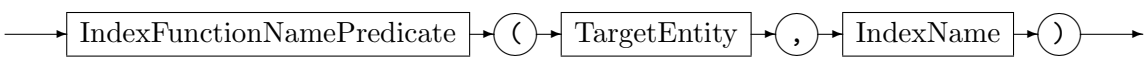


The *CountedIndexFunctionName* is built from the *IndexFunctionName* by prepending `count` and switching the first character of *IndexFunctionName* to upper case.

Boolean functions:

The index function is employed in a boolean predicate version that allows to check whether a target entity is in the queried index. This is more efficient than calling `in` on the resulting set (and even more so on the resulting array) of the plain index function. The computation stops as soon as a positive result is obtained.

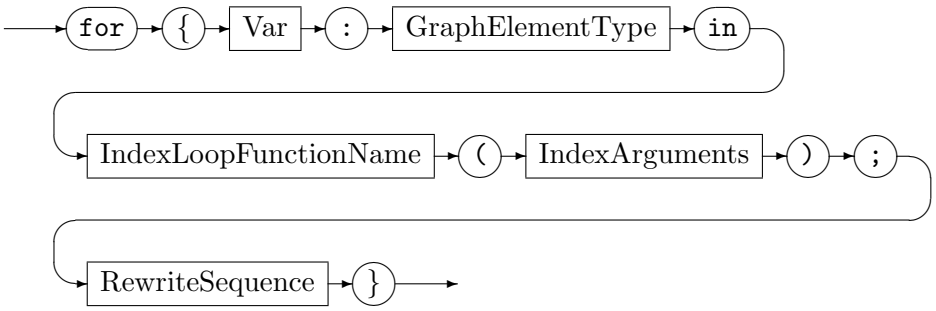
Expression



The *IndexFunctionNamePredicate* is built from the *IndexFunctionName* by prepending `isIn` and switching the first character of *IndexFunctionName* to upper case.

Moreover, the index can be queried from the sequence expressions with the same index access functions as in the rule language, and can be iterated over in the sequences with loops utilizing the same access functions.

IndexAccessLoopSequence



EXAMPLE (166)

The following index `foo` allows to quickly fetch nodes of type `N` (or a subtype) based on the value of its attribute `i`.

```
node class N {
  i : int;
}
index foo { N.i }
```

Fetching typically occurs in the rules, binding a pattern element based on the constraints specified with the lookup. The test `t` only matches if a node is found whose attribute `i` matches the parameter `j`. The rule `r` ascendingly binds the nodes whose attribute `i` is greater than or equal 5 and lower than 13. The ascendingly means that the node `n` with smallest attribute `i` is matched that satisfies the other constraints of the pattern (if there are no further constraints like in this example, it is the node with the smallest attribute satisfying the limits of the index access).

```
test t(var j:int) {
  n:N{foo==j};
}

rule r {
  n:N{ascending(foo>=5, foo<13)};

  modify {
  ---
    emit("The value of attribute i is ", n.i, "\n");
  }
}
```

EXAMPLE (167)

A similar index access (similar to the previous Example 166) from the expressions may look like:

```
test t(var j:int) {
  n:N;
  if { isInNodesFromIndexSame(n, foo, j); }
}

rule r {
  modify {
  ---
    emit("The␣nodes␣with␣5␣<=␣foo␣<␣13␣are:␣",
        nodesFromIndexFromToExclusiveAsArrayAscending(foo, 5, 13), "\n");
  }
}
```

An index may also be queried with a `for` loop from the statements, accepting the same bounds; or with a `for` loop from the sequences.

```
rule rd {
  modify {
  ---
    eval {
      for(n:N in nodesFromIndexFromExclusiveToDescending(foo, 13, 5)) {
        emit("The␣value␣of␣attribute␣i␣is␣", n.i, "\n");
      }
    }
    exec (
      for{n:N in nodesFromIndexFromDescending(foo, 13);
        { emit("The␣value␣of␣attribute␣i␣is␣", n.i, "\n") }
      }
    );
  }
}
```

Non-attribute-comparing functions

An index may also be queried directly for the entities with the lowest or the highest attribute value, as well as for its size; with the following index query functions which are available in the expressions of the rule language, as well as in the expressions of the sequence computations.

minNodeFromIndex(.)
: returns the node from the given index with the lowest attribute value (the starting node, which is one of the nodes with the lowest attribute value in case several ones exist), or null if the index is empty.

maxNodeFromIndex(.)
: returns the node from the given index with the highest attribute value (the ending node, which is one of the nodes with the highest attribute value in case several ones exist), or null if the index is empty.

minEdgeFromIndex(.)
: returns the edge from the given index with the lowest attribute value (the starting edge, which is one of the edges with the lowest attribute value in case several ones exist), or null if the index is empty.

maxEdgeFromIndex(.)
: returns the edge from the given index with the highest attribute value (the ending edge, which is one of the edges with the highest attribute value in case several ones exist), or null if the index is empty.

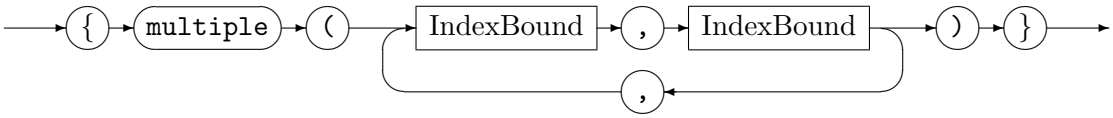
indexSize(.)
: returns the size of the given index, as `int`.

Multiple index queries and spatial queries

In the pattern you may ask for a pattern element to get bound to a graph element from multiple indices; the graph element must be contained in all the specified indices, lying within the specified bounds for each of them (i.e. a multi index join is carried out, which can be seen as a result set intersection, implemented in an efficient way).

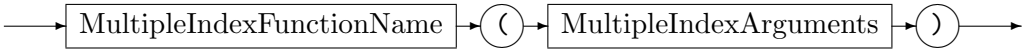
This is specified syntactically similar to an *IndexAccess* (see above). If the type of the element retrieved from the indices is not compatible to the type of the pattern element specified, the retrieved candidate is not bound to the pattern element, and matching continues with the next candidate from the indices, if the available candidates fitting to the multiple attribute conditions are exhausted or if the index join is empty, matching fails. The elements from the indices are successively bound to the pattern element, then the rest of the pattern is bound and checked, until the requested number of matches is found.

MultipleIndexAccess

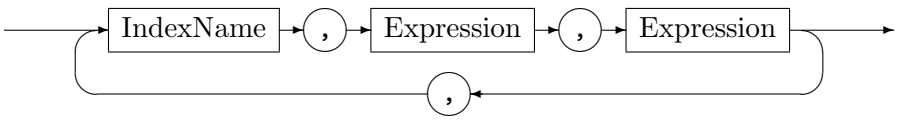


You must always specify a lower and an upper bound, for each index queried. You may use the `minNodeFromIndex`/`maxNodeFromIndex`/`minEdgeFromIndex`/`maxEdgeFromIndex` functions (see above) to obtain lowest lower and highest upper bounds, so you can emulate single index accesses with only one bound or even no bound (the same holds for the following multiple index access functions).

IndexFunctionCall



MultipleIndexArguments



A second way to access multiple indices is from multiple index access functions which are available in the expressions of the rule language, as well as in the expressions of the sequence computations.

nodesFromIndexMultipleFromTo(.,.,.[,.,.,.])*
: returns all nodes from the join of the given indices as a set, i.e. the intersection of the result sets of the simple index queries, which are of the form: the indexed attribute of the index given as kth argument must be greater or equal than the $k+1$ argument and lower or equal than the $k+2$ argument, where $k \in 1, 3, 6, \ldots$

edgesFromIndexMultipleFromTo(.,.,.[,.,.,.])*
: returns all edges from the join of the given indices as a set, i.e. the intersection of the result sets of the simple index queries, which are of the form: the indexed attribute of the index given as kth argument must be greater or equal than the $k+1$ argument and lower or equal than the $k+2$ argument, where $k \in 1, 3, 6, \ldots$

EXAMPLE (168)

The following index `foo` allows to quickly fetch nodes of type `N` (or a subtype) based on the value of their attribute `i`, the index `bar` inspects the value of its attribute `j` towards this purpose.

```
node class N
{
  i : int;
  j : string;
}

index foo { N.i }
index bar { N.j }
```

Fetching typically occurs in the rules, binding a pattern element based on the constraints specified with the lookup. The rule `r` carries out a `multiple` index access (multi index join), it binds the nodes whose attribute `i` is greater than or equal 5 and lower than or equal 13, and whose attribute `j` is greater than or equal "a" and lower than or equal "i".

```
rule r {
  n:N{multiple(foo>=5, foo<=13, bar>="a", bar<="i")};

  modify {
  ---
    emit("The␣value␣of␣attribute␣(i,j)␣is␣(" + n.i + "," + n.j + ")\n");
  }
}
```

EXAMPLE (169)

A similar index access (similar to the previous Example 168) from the expressions may look like:

```
rule r {
  n:N;
  if { n in nodesFromIndexMultipleFromTo(foo, 5, 13, bar, "a", "i"); }

  modify {
  ---
    emit("The␣nodes␣with␣5␣<=␣foo␣<=␣13␣&&␣a␣<=␣bar␣<=␣i␣are:␣",
        nodesFromIndexMultipleFromTo(foo, 5, 13, bar, "a", "i"), "\n");
  }
}
```

Multiple indices may also be queried with a `for` loop from the statements, or with a `for` loop from the sequences.

```
rule rd {
  modify {
  ---
    eval {
      for(n:N in nodesFromIndexMultipleFromTo(foo, 5, 13, bar, "a", "i")) {
        emit("The␣value␣of␣attribute␣(i,j)␣is␣(" + n.i + "," + n.j + ")\n");
      }
    }
    exec (
      for{n:N in nodesFromIndexMultipleFromTo(foo, 5, 13, bar, "a", "i");
        { emit("The␣value␣of␣attribute␣(i,j)␣is␣(" + n.i + "," + n.j + ")\n") }
      }
    );
  }
}
```

Spatial queries

The multiple index access can be used to carry out spatial queries (emulating a spatial index by utilizing one index per dimension).

The fabricated index examples geared at testing from which excerpts were shown above can be found in the `tests/index` directory. The following more interesting spatial query examples can be found in the `tests/indexSpatialQuery` directory.

EXAMPLE (170)

The following indices `ix`, `iy`, `iz` allow to quickly fetch neighboring nodes in a 3 dimensional space (so each coordinate has its own index).

```
node class Point3dNode {
  x : double;
  y : double;
  z : double;
}

index ix { Point3dNode.x }
index iy { Point3dNode.y }
index iz { Point3dNode.z }

class Point3d {
  x : double;
  y : double;
  z : double;
}
```

You may use them in rules to match pattern elements that are unconnected by nearness in space (fetching all nodes that are within a bounding box around the node of interest; note that the nodes are not matched ordered by distance, you have to apply a post-matches filter in order to achieve this goal).
The rule `matchNeighboring` applied with all-bracketing matches all nodes `neighbor` that are within `delta`-difference to all the coordinates of the node `n` in 3-dimensional space. The `multiple` indicates a multiple index access.

```
var ::delta:double;

// match neighbor that is nearby in space even if unconnected
rule matchNeighboring : (Point3dNode, Point3dNode) {
  . --> n:Point3dNode;
  neighbor:Point3dNode{multiple(ix >= n.x - delta, ix <= n.x + delta, iy >= n.y - delta, iy
      <= n.y + delta, iz >= n.z - delta, iz <= n.z + delta)} --> .;

  modify {
    return(n, neighbor);
  }
}
```

You may use them in the rules to check matched pattern elements for nearness in space with the functions.

```
// reduce cartesian product of unconnected components to the ones nearby in space
rule checkNeighboring : (Point3dNode, Point3dNode) {
  . --> n:Point3dNode;
  neighbor:Point3dNode --> .;

  if {
    neighbor in nodesFromIndexMultipleFromTo(ix, n.x - delta, n.x + delta, iy, n.y - delta,
        n.y + delta, iz, n.z - delta, n.z + delta);
  }

  modify {
    return (n, neighbor);
  }
}
```

Table 30.1 and table 30.2, as well as table 30.3 and table 30.4 list the global built-in index functions of the rule language at a glance.

`nodesFromIndexSame(Index,T):set<Node>`
`nodesFromIndexSameAsArray(Index,T):array<Node>`

`nodesFromIndex(Index):set<Node>`
`nodesFromIndexFrom(Index,T):set<Node>`
`nodesFromIndexFromExclusive(Index,T):set<Node>`
`nodesFromIndexTo(Index,T):set<Node>`
`nodesFromIndexToExclusive(Index,T):set<Node>`
`nodesFromIndexFromTo(Index,T,T):set<Node>`
`nodesFromIndexFromExclusiveTo(Index,T,T):set<Node>`
`nodesFromIndexFromToExclusive(Index,T,T):set<Node>`
`nodesFromIndexFromExclusiveToExclusive(Index,T,T):set<Node>`

`nodesFromIndexAsArrayAscending(Index):array<Node>`
`nodesFromIndexFromAsArrayAscending(Index,T):array<Node>`
`nodesFromIndexFromExclusiveAsArrayAscending(Index,T):array<Node>`
`nodesFromIndexToAsArrayAscending(Index,T):array<Node>`
`nodesFromIndexToExclusiveAsArrayAscending(Index,T):array<Node>`
`nodesFromIndexFromToAsArrayAscending(Index,T,T):array<Node>`
`nodesFromIndexFromExclusiveToAsArrayAscending(Index,T,T):array<Node>`
`nodesFromIndexFromToExclusiveAsArrayAscending(Index,T,T):array<Node>`
`nodesFromIndexFromExclusiveToExclusiveAsArrayAscending(Index,T,T):array<Node>`

`nodesFromIndexAsArrayDescending(Index):array<Node>`
`nodesFromIndexFromAsArrayDescending(Index,T):array<Node>`
`nodesFromIndexFromExclusiveAsArrayDescending(Index,T):array<Node>`
`nodesFromIndexToAsArrayDescending(Index,T):array<Node>`
`nodesFromIndexToExclusiveAsArrayDescending(Index,T):array<Node>`
`nodesFromIndexFromToAsArrayDescending(Index,T,T):array<Node>`
`nodesFromIndexFromExclusiveToAsArrayDescending(Index,T,T):array<Node>`
`nodesFromIndexFromToExclusiveAsArrayDescending(Index,T,T):array<Node>`
`nodesFromIndexFromExclusiveToExclusiveAsArrayDescending(Index,T,T):array<Node>`

`nodesFromIndexMultipleFromTo(Index,T,T[,Index_k,T_k,T_k]*):set<Node>`

Table 30.1: Index functions returning nodes at a glance

`countNodesFromIndex(Index):int`
`countNodesFromIndexSame(Index,T):int`
`countNodesFromIndexFrom(Index,T):int`
`countNodesFromIndexFromExclusive(Index,T):int`
`countNodesFromIndexTo(Index,T):int`
`countNodesFromIndexToExclusive(Index,T):int`
`countNodesFromIndexFromTo(Index,T,T):int`
`countNodesFromIndexFromExclusiveTo(Index,T,T):int`
`countNodesFromIndexFromToExclusive(Index,T,T):int`
`countNodesFromIndexFromExclusiveToExclusive(Index,T,T):int`

`isInNodesFromIndex(Node,Index):boolean`
`isInNodesFromIndexSame(Node,Index,T):boolean`
`isInNodesFromIndexFrom(Node,Index,T):boolean`
`isInNodesFromIndexFromExclusive(Node,Index,T):boolean`
`isInNodesFromIndexTo(Node,Index,T):boolean`
`isInNodesFromIndexToExclusive(Node,Index,T):boolean`
`isInNodesFromIndexFromTo(Node,Index,T,T):boolean`
`isInNodesFromIndexFromExclusiveTo(Node,Index,T,T):boolean`
`isInNodesFromIndexFromToExclusive(Node,Index,T,T):boolean`
`isInNodesFromIndexFromExclusiveToExclusive(Node,Index,T,T):boolean`

`minNodeFromIndex(Index):Node` `maxNodeFromIndex(Index):Node`
`indexSize(Index):int`
`countFromIndex(Index, Node):int`

Table 30.2: Index functions on nodes at a glance

edgesFromIndexSame(Index,T):set<AEdge>
edgesFromIndexSameAsArray(Index,T):array<AEdge>
edgesFromIndex(Index):set<AEdge> edgesFromIndexFrom(Index,T):set<AEdge> edgesFromIndexFromExclusive(Index,T):set<AEdge> edgesFromIndexTo(Index,T):set<AEdge> edgesFromIndexToExclusive(Index,T):set<AEdge> edgesFromIndexFromTo(Index,T,T):set<AEdge> edgesFromIndexFromExclusiveTo(Index,T,T):set<AEdge> edgesFromIndexFromToExclusive(Index,T,T):set<AEdge> edgesFromIndexFromExclusiveToExclusive(Index,T,T):set<AEdge>
edgesFromIndexAsArrayAscending(Index):array<AEdge> edgesFromIndexFromAsArrayAscending(Index,T):array<AEdge> edgesFromIndexFromExclusiveAsArrayAscending(Index,T):array<AEdge> edgesFromIndexToAsArrayAscending(Index,T):array<AEdge> edgesFromIndexToExclusiveAsArrayAscending(Index,T):array<AEdge> edgesFromIndexFromToAsArrayAscending(Index,T,T):array<AEdge> edgesFromIndexFromExclusiveToAsArrayAscending(Index,T,T):array<AEdge> edgesFromIndexFromToExclusiveAsArrayAscending(Index,T,T):array<AEdge> edgesFromIndexFromExclusiveToExclusiveAsArrayAscending(Index,T,T):array<AEdge>
edgesFromIndexAsArrayDescending(Index):array<AEdge> edgesFromIndexFromAsArrayDescending(Index,T):array<AEdge> edgesFromIndexFromExclusiveAsArrayDescending(Index,T):array<AEdge> edgesFromIndexToExclusiveAsArrayDescending(Index,T):array<AEdge> edgesFromIndexToAsArrayDescending(Index,T):array<AEdge> edgesFromIndexFromToAsArrayDescending(Index,T,T):array<AEdge> edgesFromIndexFromExclusiveToAsArrayDescending(Index,T,T):array<AEdge> edgesFromIndexFromToExclusiveAsArrayDescending(Index,T,T):array<AEdge> edgesFromIndexFromExclusiveToExclusiveAsArrayDescending(Index,T,T):array<AEdge>
edgesFromIndexMultipleFromTo(Index,T,T[,Index_k,T_k,T_k]*):set<AEdge>

Table 30.3: Index functions returning edges at a glance

`countEdgesFromIndex(Index):int`
`countEdgesFromIndexSame(Index,T):int`
`countEdgesFromIndexFrom(Index,T):int`
`countEdgesFromIndexFromExclusive(Index,T):int`
`countEdgesFromIndexTo(Index,T):int`
`countEdgesFromIndexToExclusive(Index,T):int`
`countEdgesFromIndexFromTo(Index,T,T):int`
`countEdgesFromIndexFromExclusiveTo(Index,T,T):int`
`countEdgesFromIndexFromToExclusive(Index,T,T):int`
`countEdgesFromIndexFromExclusiveToExclusive(Index,T,T):int`

`isInEdgesFromIndex(Edge,Index):boolean`
`isInEdgesFromIndexSame(Edge,Index,T):boolean`
`isInEdgesFromIndexFrom(Edge,Index,T):boolean`
`isInEdgesFromIndexFromExclusive(Edge,Index,T):boolean`
`isInEdgesFromIndexTo(Edge,Index,T):boolean`
`isInEdgesFromIndexToExclusive(Edge,Index,T):boolean`
`isInEdgesFromIndexFromTo(Edge,Index,T,T):boolean`
`isInEdgesFromIndexFromExclusiveTo(Edge,Index,T,T):boolean`
`isInEdgesFromIndexFromToExclusive(Edge,Index,T,T):boolean`
`isInEdgesFromIndexFromExclusiveToExclusive(Edge,Index,T,T):boolean`

`minEdgeFromIndex(Index):AEdge` `maxEdgeFromIndex(Index):AEdge`
`indexSize(Index):int`

Table 30.4: Index functions on edges at a glance

30.5 Incidence Count Indices

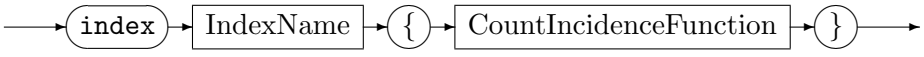
Attribute indices can be seen as a general-purpose device, that extends the built-in ability of quick lookup by-type and of quick lookup by-neighbourhood with a quick lookup by-attribute, thus giving complete quick-lookup coverage for all foundational elements of the graph model. Incidence count indices in contrast are more of a special-purpose device for a certain abstraction that can be applied to a graph – the *count* of incident edges – and are beneficial only if that abstraction is of importance. They allow you to quickly look up nodes based on their number of incident edges. This is especially beneficial for algorithms that work best when they traverse the nodes from the ones with the highest number of incident edges to the ones with the lowest number of incident edges (or the other way round).

The index furtheron allows to fetch the incidence count for a node quickly with just an index lookup, but typically no gain can be made from this, as counting the incident edges of a node is commonly cheap (an $O(log(n))$ index lookup versus an $O(k)$ counting enumeration, with k being the number of edges incident to the focused node).

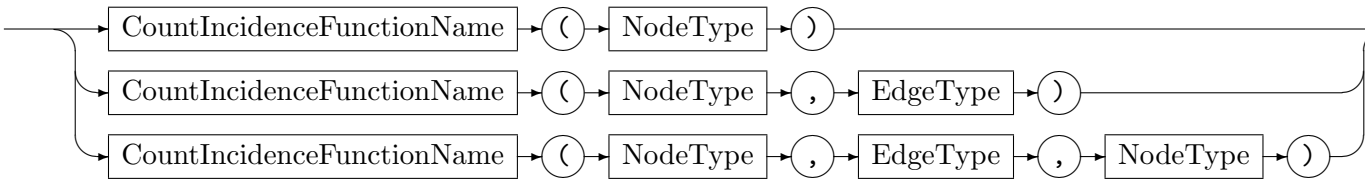
Declaration in the model

An incidence count index must be declared in the model.

IncidenceCountIndexDecl



CountIncidenceFunction



Following the `index` keyword, a name for the index is specified; in the body of the index, the incidence function and its types are given. The admissible count incidence functions are `countIncident`, `countIncoming`, and `countOutgoing`, with semantics as already introduced in 16.2.3. (The count of the edges complying to the specified incidence function is stored in an index, for all nodes in the graph of the type of the first node in the count incidence function.)

Usage in the rules

An incidence count index can be used in the pattern in exactly the same way as an attribute index, see above 30.4.

In addition to the index lookup, the count may be queried from the rule language expressions (note that while the attributes can be accessed directly for a pattern element, the incidence count would have to be obtained by counting in contrast, but this only saves time for heavily connected nodes).

`countFromIndex(.,.)`
: returns the count of incident edges as `int` of the node given as second argument, obtained from the incidence count index given as first argument.

The index function is also available in the sequence expressions; the indexer notation (similar to an array access) that was available in a previous version of GrGen.NET is not available anymore.

EXAMPLE (171)

The following index `bar` allows to quickly fetch nodes based on the number of `outgoing` edges, `qux` allows to quickly fetch nodes of type `N` based on the number of `incoming` edges of type `E`, stemming from a source node of type `N`.

```
node class N {
   i:int = 0;
}
edge class E;
index bar { countOutgoing(Node) }
index qux { countIncoming(N, E, N) }
```

Fetching typically occurs in the rules, binding a pattern element based on the constraints specified with the lookup. The test `t` returns the node `n` in the graph with the maximum number of outgoing edges. The rule `r` ascendingly binds the nodes whose number of incoming edges of type `E` from a node of type `N` is greater than or equal the value of attribute `i` of node `l` matched before (constrained to values below 42 by the `if`); it `emit`s the count of outgoing edges of `n`, queried from the index `bar` with array access notation.

```
test t() {
   n:Node{descending(bar)};
}

rule r() {
  l:N; if{ l.i<42;}
  n:N{ascending(qux>=l.i)};

  modify {
  ---
    emit("count␣outgoing(Node):␣", countFromIndex(bar, n), "\n");
  }
}
```

30.6 Name Index

A named graph is a graph where each element bears a unique name and can be looked up quickly by that name. So basically it works as a graph with an integrated key-value store from a string to a graph element, and one from a graph element to a string. It is implemented with two hash maps, one from the names to the elements, and the other from the elements to the names. Lookup is carried out in $O(1)$ (either way). Maintaining the index on graph insertions and removals is carried out in $O(1)$, too. So in contrast to the attribute index and the incidence count index, a name index does not allow multiple elements per indexed property (which is here the name).

No declaration needed

The name index does not need to be declared, it is implemented with the named graph, which is always generated together with its sibling, the non-named graph. If you are working with the GrShell, this is the only kind of graph you'll be working with, as the shell always instantiates the named graph (it delivers the *persistent names* you see in the debugger). At API level you may opt for the non-named graph that is considerably cheaper regarding memory usage – a named graph requires not much less than about twice the amount of memory of a plain graph (unless heavily attributed).

But please note in that case that the export and import capabilities require uniquely named elements and thus only work with named graphs. If given a plain graph, the exporters just create a named graph. A graph is always created as a named graph by the importers. So non-named graphs are typically only of interest if you don't need to persist them into a serialization format.

Usage in the rules

The name index can be used in the pattern to look up graph elements by their name, with the same syntax as used in the GrShell for accessing elements by their (persistent) name.

NamedIndexAccessExpr



There is at most one element existing with the name asked for; if no graph element with that name exists, matching fails and backtracks, if an element exists, the pattern element is bound to it, and matching continues. Please note that the (string-)expression used to compute the name is only allowed to reference at most one other pattern element (not handed in as parameter).

The name index may be further queried from the rule language expressions or the sequence expressions with the `nameof` as well as the `nodeByName` and the `edgeByName` functions:

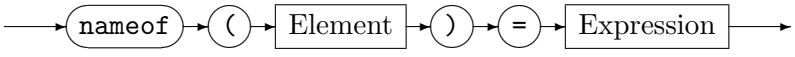
`nameof(.)`
: returns the name (type string) of the given node or edge (or (sub)graph, a missing argument amounts to the host graph).

`nodeByName(.)`
: returns the node of the given name, or null if no node of that name exists. Optionally, a target node type may be given as second argument, then you receive a node of that type as result, or null if no node of that name *and type* exists.

`edgeByName(.)`
: returns the edge of the given name, or null if no edge of that name exists. Optionally, a target edge type may be given as second argument, then you receive an edge of that type as result, or null if no edge of that name *and type* exists.

The names are normally automatically assigned (computed from a global counter increased with each element, prepended by the dollar symbol – unless you specify a name with an attribute initialization list, cf.12.8), but you may assign a different name to an attribute element. This can be only done in an `eval`-block, or a `procedure`, with a `nameof`-assignment.

NameofAssignment



When you do so, it is your responsibility to ensure that the name does not already exist, otherwise the assignment will cause a runtime exception. The same syntax may be used to assign the name of a (sub)graph, a missing element amounts to the host graph.

EXAMPLE (172)

The test `t` succeeds if there is a node of name "foo" existing in the graph, with a reflexive edge of the name defined by input parameter `s`, and there is a further node of name "bar" existing in the graph.

```
test t(var s:string) {
  n:Node{@("foo")} -e:Edge{@(s)}-> n;
  if{ nodeByName("bar") != null; }
}
```

The rule `r` `yield`s the name of the node matched to `n` out to `oldname`, and changes the name of the node in the `modify` part by prepending the name of the host graph. Furthermore, it prints the name of the edge bound to `e`.

```
rule r() {
  n:Node -e:Edge->;
---
  def var oldname:string;
  yield { yield oldname = nameof(n); }

  modify {
    eval {
      nameof(n) = nameof() + "_" + oldname;
    }
  ---
    emit("The␣name␣of␣the␣graph␣element␣bound␣to␣e␣is␣", nameof(e), "\n");
  }
}
```

30.7 Uniqueness Index

Uniqueness can be applied in two parts, the i) uniqueness *constraint*, and ii) the uniqueness *index*.

The uniqueness *constraint* ensures that each element in the graph has a unique id. It allows in addition to fetch the unique id from a graph element. You may see it as a key-value store from graph elements to their ids – it is a very efficient one, as the unique-id is stored in the graph elements. This incurs an additional cost of a 4 bytes unique id field per graph element.

The uniqueness *index* allows in addition to fetch a graph element by its unique id, extending the uniqueness property to an index allowing for quick graph element lookup. You may see it as a key-value store from the unique ids to their graph elements. This one is realized with an array/list of unique ids to graph elements (thus the cost is at most about twice the number of elements ever held multiplied by 4 bytes on a 32bit CLR and 8 bytes on a 64bit CLR).

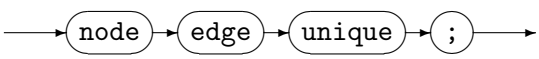
So lookup is carried out efficiently in $O(1)$ either way. Maintaining the uniqueness information is in $O(log(n))$, but much less on average, as the ids of deleted elements have to be stored in a heap (the data structure, not the memory model) for quick reuse of the lowest one upon graph element addition (this way we ensure a maximally packed id range, which keeps the index array and the is-matched-bit-arrays of the parallelized matchers small and packed).

A uniqueness index does not allow multiple elements per indexed property (which is here the unique id), like the name index, and in contrast to the attribute and incidence count indices.

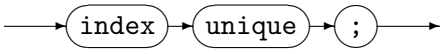
Declaration in the model

The space for the unique id is always reserved in the graph elements, but unique ids are only assigned if you declare an uniqueness constraint. Or if you declare another index in the model, as indices depend on the unique ids, or if you use parallelize annotations, as the parallelized matchers depend on the unique ids, too. If you need to fetch graph elements by their unique ids, you must declare an unique index (in addition or instead).

UniqueConstraintDecl



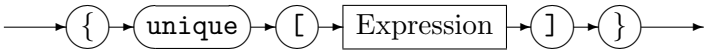
UniqueIndexDecl



Usage in the rules

The unique index can be used in the pattern to lookup graph elements by their unique id.

UniqueIndexAccessExpr



There is at most one element existing with the unique id asked for; if no graph element with that unique id exists, matching fails and backtracks, if an element exists, the pattern element is bound to it, and matching continues. Please note that the expression used to compute the unique id is only allowed to reference at most one pattern element (not handed in as parameter).

The unique constraint may be further queried from the rule language expressions or the sequence expressions with the `uniqueof` function, the unique index with the `nodeByUnique` and `edgeByUnique` functions:

`uniqueof(.)`
: returns the unique id (type int) of the given node or edge (or (sub)graph, a missing argument amounts to the host graph).

`nodeByUnique(.)`
: returns the node of the given unique id, or null if no node of that unique id exists. Optionally, a target node type may be given as second argument, then you receive a node of that type as result, or null if no node of that unique id *and type* exists.

`edgeByUnique(.)`
: returns the edge of the given unique id, or null if no edge of that unique id exists. Optionally, a target edge type may be given as second argument, then you receive an edge of that type as result, or null if no edge of that name *and type* exists.

The unique ids are automatically assigned (utilizing an id pool that is enlarged with newly allocated ids when an element is added to the graph but the pool is empty, or reusing an already existing id; when an element is removed from the graph, its id returns to the id pool). In contrast to the names, you cannot change the id assignment.

EXAMPLE (173)

The following model declares a node class `N`, specifies the uniqueness constraint, and requests the uniqueness index.

```
1 node class N;
2 node edge unique;
3 index unique;
```

The test `t` succeeds if there is a node of unique id 0 existing in the graph, with a reflexive edge of the unique id defined by input parameter `i`, and there is a further node of unique id 42 existing in the graph. It is unlikely you will really use code like that as you cannot assign a unique id.

```
1 test t(var i:int) {
2   n:N{unique[0]} -e:Edge{unique[i]}-> n;
3   if{ nodeByUnique(42) != null; }
4 }
```

The rule `r` allows to match `n` and `m` to the same graph element homomorphically, just to forbid it then with the condition requesting that their unique-ids must be different. The point here is that the `hom` allows the engine to omit the isomorphy check, yielding slightly better performance. By using `<` instead of `!=` we ensure that we get only one match for the automorphic pattern in case of an application with all-bracketing – the one in which the element with the smaller id is bound to `n`. Furthermore, the rule prints the unique id of the node bound to `n`.

```
1  rule r() {
2    n:N -- m:N;
3    hom(n,m);
4    if{ uniqueof(n) < uniqueof(m); }
5
6    modify {
7    ---
8      emit("The␣unique␣id␣of␣the␣graph␣element␣bound␣to␣n␣is␣", uniqueof(n), "\n");
9    }
10 }
```

CHAPTER 31

PERFORMANCE OPTIMIZATION

The most important point to understand when optimizing for speed is that the expensive task is the search carried out during pattern matching. The effort of rewriting (the dominant theme in graph rewriting literature) is negligible.

Searching is carried out with a backtracking algorithm following a search plan in a fixed order, binding one pattern element after another to a graph element, checking if it fits to the already bound parts. If it does fit search continues trying to bind the next pattern element (or succeeds building the match object from all the elements bound if the last check succeeds), if it does not fit search continues with the next graph element; if all graph element candidates for the currently focused pattern element are exhausted, search backtracks to the previous decision point and continues there with the next element.

Typically, first a graph element is determined with a lookup operation reaching into the graph, binding the element to a graph element of the required type (the fewer elements of that type exist, the better) – then neighbouring elements are traversed following the graph structure (the fewer neighbouring elements exist, the better), until a match of the entire pattern is found.

31.1 Search Plans

A search plan for the pattern in figure 31.1 is:
`lkp(v1:A); out(v1,e1:a); tgt(e1,v2:B); out(v2,e3:b); tgt(e3,v3:C); out(v3,e2:a)`
The search operation `lkp` denotes a node (or edge) lookup in the graph, `out` follows the outgoing edges of the given source node and `in` follows the incoming edges of the given target node, while `src` fetches the source from the given edge and `tgt` fetches the target from the given edge.

For some graphs the search plan might work well, but for the graph given in figure 31.2 it is a bad search plan. Why so can be seen in the search order sketched in figure 31.3. Due to the multiple outgoing edges of `v1` of which only one leads to a match it has to backtrack several times.

This schedule in contrast is a good one:
`lkp(v3:C); out(v3,e2:a); tgt(e2,v2:B); out(v2,e3:b); in(v2,e1:a); src(e1,v1:A)`
corresponding to the search order depicted in figure 31.4.

It is crucial for achieving high performance to prevent following graph structures splitting into breadth as given in this example, and especially to avoid lookups on elements which are available in high quantities.

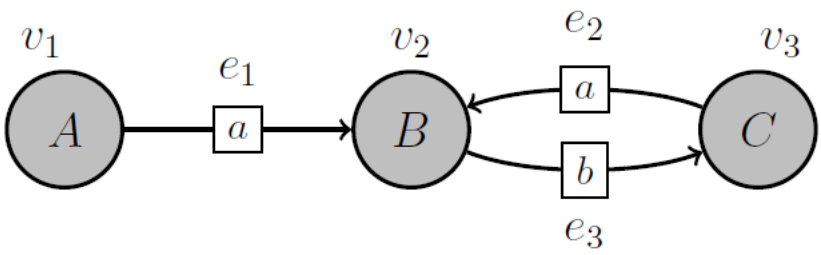


Figure 31.1: Pattern to search

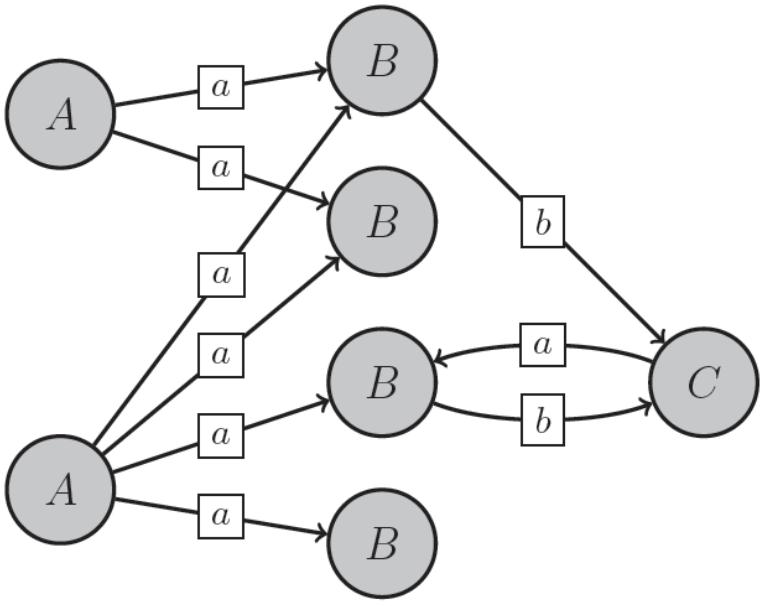


Figure 31.2: Host graph to search in

31.2 Find, Don't Search

It is better to find elements with certain characteristics straight ahead without search, utilizing indices as already introduced in Chapter 30.

31.2.1 Consequences For Optimization

The consequences of the automatically used type indices 30.1 and neighbourhood indices 30.2 for optimization are:

Use types!

Fine grain types cause a fast lookup of the first pattern element(s), or more exactly: they cause a small list of candidate elements for the first pattern element to be tried. And they cause quicker pruning of search branches because of early failing type checks. Using fine-grain types is easy in GrGen, as multiple inheritance on node and edge types (cf. 4.2.1) is supported. Besides, the more fine grain the graph is typed, the better are the statistics, allowing GrGen.NET to find better search plans (see more for this below).

Prune early

Attribute conditions are evaluated as soon as all needed graph elements are matched (saving us from enumerating futile match extensions). But the unit of scheduling are the full attribute condition expressions. If an expression can be separated into different parts that depend on less pattern elements than the entire expression (such a separation is trivial for boolean expressions joined by logical conjunctions), split it into those parts. The parts are then checked earlier in the search process. You could even introduce checks that are logically not needed because a later check removes all incorrect matches anyway, just to achieve better performance, which is the case if the checks allow to prune some branches from the search space earlier. This holds as long as the checks are cheap, if they are expensive you could do the opposite and introduce artificial dependencies to all pattern elements instead, to ensure the attribute condition is evaluated as late as possible.

Prefer directed edges

Non-directed edges in the pattern are searched in both directions, considerably increasing the search space. Use non-directed edges only if you really need them. For some problems this is the case, you then simply have to pay the price of the increased effort for the symmetry. But some problems where undirected edges are more natural can be easily encoded with directed ones – in case of performance problems, refactor and optimize them to employ only directed edges, imposing an arbitrary but deterministic direction on the edges. Besides, the vstructure statistics are more discriminating in case of directed edges, leading to better search planning results; in case of undirected ones the information from both directions is coalesced.

Beware of Disconnected Patterns

Disconnected patterns cause a combinatorial explosion of the matches, because the overall number of matches equals the cartesian product of the partial matches of the disconnected parts.

This is inevitable and a price that must be simply paid if this is really needed. But it is normally not needed, and, most importantly, is not specified accidentally, as a disconnected pattern in a single flat pattern is typically immediately visible. But take care of nested patterns or subpatterns. You might overlook that nested patterns and subpatterns are matched strictly one after the other and especially after their containing pattern. You cannot ignore

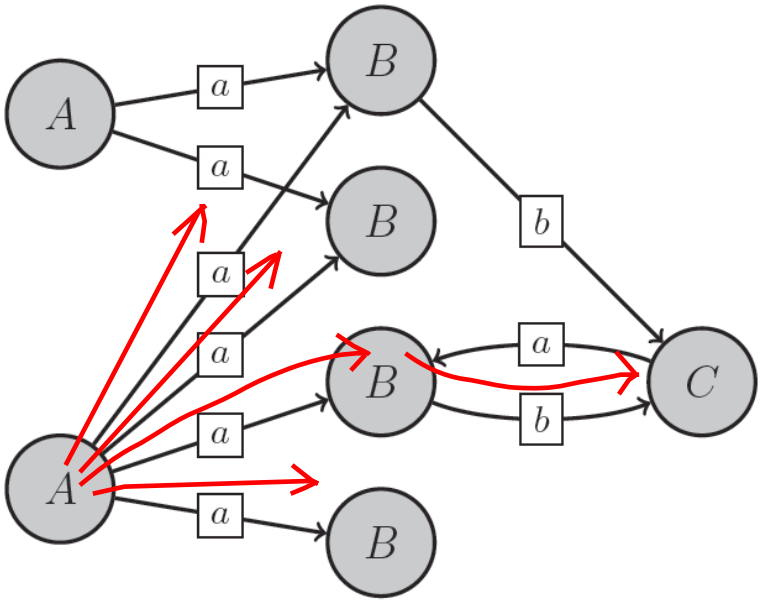


Figure 31.3: Bad search order

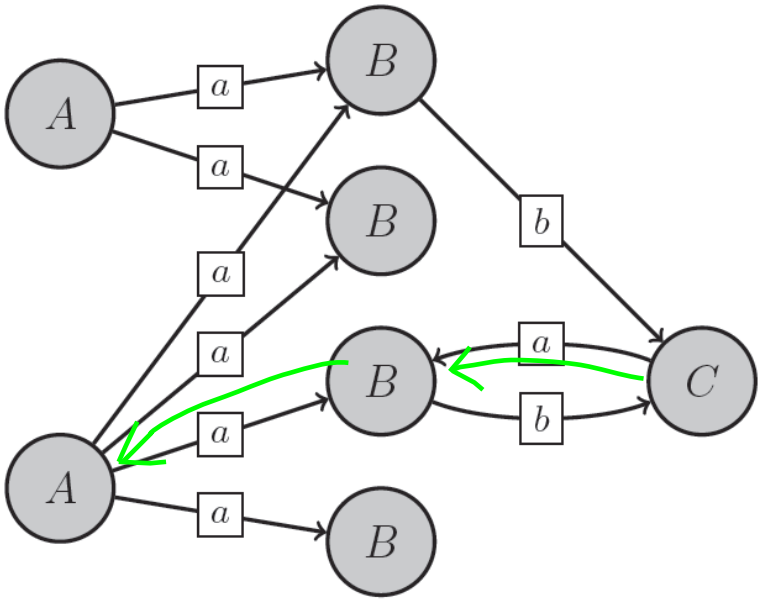


Figure 31.4: Good search order

block borders and subpattern calls, they disconnect otherwise connected components. Esp. take care of not disconnecting patterns when factoring out a common part into a subpattern to improve the code. But due to inlining, things look better than what was said until now.

EXAMPLE (174)

Take a look at pattern nesting, when the pattern is disconnected, you run into issues due to combinatorial explosion. This holds for sure for alternative and iterated patterns.
So you must take care of pattern cardinality ...

```
test bad {
  n1:Node; n2:Node; // builds cartesian product of all nodes in the graph (O(n*n))
  multiple {
    n1 --> n2; // then filters it doum to the connected nodes
  }
}
```

... and alternatives ...

```
test bad {
  n1:Node; n2:Node; // builds cartesian product of all nodes in the graph (O(n*n))
  alternative {
    single {
      n1 --> n2; // then filters it doum to the connected nodes
    }
  }
}
```

EXAMPLE (175)

Take a look at pattern nesting, when the pattern is disconnected, you run into issues due to combinatorial explosion. This may hold for independent and subpatterns – but inlining is of help here.
The edge from the independent in the example is typically inlined into the pattern removing the issue...

```
test bad {
  n1:Node; n2:Node; // builds cartesian product of all nodes in the graph (O(n*n))
  independent {
    n1 --> n2; // then filters it doum to the connected nodes
  }
}
```

... as is the subpattern body inlined into the pattern, removing the performance issue.

```
test bad {
  n1:Node; n2:Node; // builds cartesian product of all nodes in the graph (O(n*n))
  :P(n1,n2);
}
pattern P(n1:Node, n2:Node) {
  n1 --> n2; // then filters it doum to the connected nodes
}
```

Take a look at the output of the `explain` command to check whether inlining occurred.

The need to take nested pattern borders and subpattern calls into account is due to the

recursive descent matching with a multi-pushdown machine as described in Section 10.3 and Section 35.3.

Subpatterns are matched top-down, from the input parameters on. If the input arguments are disconnected in the pattern containing the subpattern, the containing pattern enumerates the cross product of the matches of the disconnected parts, which is only later on filtered for the ones which are connected. This will likely wreak havoc on search performance. Even if you don't search for all matches, if you only compute a single overall match — the calling pattern must enumerate a lot of combinations of its parts (worsened by the fact that those are typically found often because of their simplicity), until the nested pattern finally is able to connect one of the disconnected pairs fed into it. It might be more efficient to just search from a start parameter towards a connected end location, and yield the found one out (cf. 10.4); or to search from a start parameter on all connected end locations, collecting the found ones in a result set – and then to check the ones found alongside connectedness in a second step.

Nested patterns are also matched top-down, from the input parameters on. But parameter passing is implicit here, the elements from the containing pattern that are referenced in the nested pattern are passed in automatically as arguments. This holds especially for the `alternative` and `iterated` constructs, which are matched with the pushdown machine (cf. 35.3), too, but also for the `negative` and `independent` constructs, which are matched with nested local code embedded into the matcher code of their containing pattern.

But don't shy away from using subpatterns or independents too early, inlining is of help here!

The elements from an independent that are linking a disconnected pattern are typically inlined into their using pattern. The elements from a subpattern are often inlined into their using pattern, causing the pattern to get connected (again), but also removing the pushdown machine overhead. But you must take into account that the *inlining* implemented in GrGen is limited to depth one. If a pattern is disconnected over two or more levels of subpattern usage (which might happen statically with one subpattern using another subpattern, and will for sure dynamically on a subpattern recursion path), it will hit performance. You may have a look at the output of the `explain` command (cf. 25.12) to see if the subpatterns are disconnected. This is typically indicated by multiple lookups in the containing pattern, for fetching the disconnected starting points, which are then handed down with preset parameters to the nested or subpatterns, and only get connected there, with search commands following their outgoing or incoming edges.

31.2.2 Search Planning On Request

Search planning at runtime is only carried out on request! You must analyze the graph and then re-generate the matchers manually, with `custom graph analyze` and `custom actions gen_searchplans` issued on the command line, or with calls of the `Custom` methods of the `IGraph` and the `IActions` objects. Unless you do so, the static search plan is kept in place.

You can display the search plan currently employed with the `custom actions explain <actionname>` command, in order to inspect how the pattern elements are matched. Issue it before search planning to show the statically generated search plan, issue it afterwards to show the dynamically re-generated search plan. See subsection 25.12 for more on this.

The initial static search plan typically starts with a lookup of an arbitrary edge, and then arbitrarily follows the pattern graph structure. Interestingly, in many cases this still works quite well because of the quick pruning by the type checks in a well-typed graph. In addition, often the rules match from some parameter nodes onwards, which are typically well-suited starting points for searching.

Beware: the `analyze` command is costly. It helps in improving the matching performance, but requires execution time on its own. Don't use it freely.

It is costly because it has to visit all the nodes and edges in the graph, in order to gather

statistical data about the amount of breadth-splitting per node and edge-types. For each ($NodeType, EdgeType, NodeType*$)-triple (for all node and edge types), it counts for each start node of $NodeType$ the number of edges of $EdgeType$, which are leading to an end node of $NodeType*$. Given that information, search planning is able to compute a search plan that avoids to follow structures splitting into breadth (V-structures) that occur often in the host graph. Given the information about the element counts per type (which is directly available from the host graph), search planning is able to compute a search plan that avoids lookups on populated types.

Such a dynamically generated search plan typically leads to minimum matching durations, but not always – sometimes you may fare better by manually assigning priorities to the pattern elements, thus influencing the search order in the initial static search plan (high-prio elements are matched first).

Again: Use types! The more fine grain a graph is typed, the better are the statistics regarding the breadth-splitting and the number of elements of a certain type, and the better are then in consequence the search plans in their ability of evading breadth-splitting structures and avoiding lookups of often occurring types.

31.2.3 Consequences For Optimization, II

The most important consequence of the attribute, incidence count, unique id, and name indices for optimization is that you have to use them explicitly in order to achieve a speed-up (in contrast to the type and neighbourhood indices which are used implicitly):

If you need to find nodes/edges quickly by attribute value, then an attribute index 30.4 will do the trick. It is used similarly to a database index on a column of choice. But unfortunately not applied automatically upon an attribute constraint in an if part, you have to explicitly use a node/edge with an index access suffix; or an index function call in an if part. Remember you can query by multiple indices that get joined, which allows for spatial querying, see Example 170 for more on this. Note that an index may degrade performance when you carry out a lot of graph manipulations (assignments to the indexed attribute, and element addition to or removal from the graph that also counts as attribute change, as the index has to be maintained then which is an $O(log(n))$ operation), and only a few lookups on it. The implementations for unique ids and names (discussed later) could be used to implement database-like keys, but because of their special semantics you may prefer an attribute index.

If you need to find nodes by the count of incident edges, which is typically used with algorithms that repeatedly process the nodes with the highest amount of connections or the nodes with the lowest amount of connections, an incidence count indices 30.5 will be your friend. It could be also helpful with densely connected graphs when counts are needed (counting incident edges is faster for sparsely connected graphs). But this index also needs to be maintained at the price of an $O(log(n))$ operation per element addition to or removal from the graph.

If you need a unique id similarly to a database key, the uniqueness index 30.7 may be of interest. It is more efficient than an attribute index on a corresponding integer attribute ($O(1)$), but does not allow for range queries (it can be seen as a key-value store). The uniqueness constraint already ensures you get graph elements with a unique id, but does not allow to fetch the elements quickly based on the id. You may also need to employ an attribute index when you have to ensure that a once assigned id is never re-assigned – the unique ids from the unique index are reused by new graph elements when their original graph elements are removed from the graph.

Another way to obtain a unique id similarly to a database key is the name index 30.6. It is more efficient than an attribute index on a corresponding string attribute ($O(1)$), but does not allow for range queries (it too can be seen as a key-value store, which is built-in and automatically used when you work with the GrShell). There is no distinction into

a constraint and an index, you always get both, i.e. the ability to quickly fetch the name of a graph element, and the ability to quickly fetch a graph elements by its name. Names are automatically assigned and incremented (unless you overwrite them explicitly), but not automatically re-used during the lifetime of the host graph (in contrast to the unique ids).

With a low number of nodes and edges, working without indices may be the fastest approach - you evade the cost of building and maintaining the index (as well as the additional memory cost, as always, you trade time against space), but have to pay more during searches, as you have to inspect more elements subjecting them to a test for the property that is answered quickly by the index. With a low number of nodes and edges, working without types or explicit indices may be also the fastest approach regarding the combined speed of development and execution, as development is faster and execution fast enough. But this does not scale, the higher the amount of nodes and edges, the more do you need the implicit and the explicit indices (so that your solution stays feasible as the problem size increases). (Of course, it would be best regarding development time if *all* available indices would be used implicitly/automatically by the search planner (like the type and neighbourhood indices) instead of explicitly, but well, this level of declarative convenience of the language would have to be implemented imperatively-manually first in the runtime-system(/the compiler)...)

31.3 Location Passing and Memorization

Often in a transformation, you know the location that needs processing from a previous step. In this case, return these locations out from the rules, store them in variables of node (or edge) type, and hand them in again via rule arguments to the follow-up rules. Or call embedded rules, directly handing in elements from the containing rule, saving you the intermediate assignments.

Commonly, this is just the right way of handling this situation, as you must continue processing the very spot you just processed with the previous rule(s) to achieve a valid result. Sometimes you don't need to, but gain higher performance when you do so. Optimize this way then, use *rooted* pattern matching, with roots defined by previous rules, it's cheaper as it saves you the lookup in the graph.

A different way of parameter passing is adding reflexive edges to the graph to mark the spot of processing. This is normally working well, too, but the parameter passing is typically easier and does not "pollute" the graph with processing information.

In case of a statically not known number of locations as they appear e.g. in a wavefront algorithm, you may store the locations in storages (cf. chapter 15), i.e. collection valued variables (which are iterated over in the sequences or passed to the rules as `ref` parameters). Search will then start at these parameters, instead of looking up a value in the graph by type (unless search planning taking the statistics about the graph into account comes to the conclusion that a lookup on a super-seldom type is still the better approach).

A sophisticated way of remembering facts about non-local properties is to compute them with data flow analyses (see section 24.6) and store them as attributes in the graph elements. This allows to replace searching for distant values, or global properties like reachability, by checking a local property, at the price of re-running an analysis every time the graph changes in an important way.

Variables and storages can be seen as indices into the graph, indices that are typically more selective than the automatically supplied indices; but you must maintain them by hand in contrast to the automatically supplied ones. Beware of elements already deleted from the graph still hanging out in your storage because you forgot to remove them.

The approach of remembering state instead of searching when needed has a clear caveat: the code becomes susceptible to ordering effects (more brittle) and less readable. As in normal programming, you must balance performance optimizations against maintainability.

Traversal passes

In traditional graph programming, typically functionality is smeared into (generic) graph traversal code of global passes (and orchestrated via processing state), carried out by recursive visiting calls. One pass taking care of different aspects in the performance optimized version, for each aspect its own pass in the easier to maintain and understand version. Index structures or state in helper variables may be used to get quicker to the parts of interest.

In graph rewriting with local rules, index structures are a must, otherwise each rule application would amount to a traversal pass (an extreme waste of performance). In pure graph rewriting, they would be queried with every rule application. In a performance optimized – and often easier to specify version – parameters are used instead to process a certain spot in the graph. The price for these benefits is the need to synchronize the processing state with the graph state (to prevent stale elements after graph changes), or the computations with the processing state (all using rules have to be adapted when the processing state is changed).

NOTE (63)

An aspect of the notion of graph rewriting is that no state is stored outside of the graph (a similar idea applies to database design), with every rule application, the graph is queried (/searched) for the parts of interest. The benefit is that no stored state (elementary variables, storages, match-arrays) can get stale at changes, which excludes an entire class of hard-to-find errors, but it comes at the price of repeated searches to get again to (/find) the parts of interest. With its indices, GrGen.NET is well suited to this style of programming, as getting to the part of interest is quick, but when beneficial or needed, you can revert to a more imperative programming style based on an intermediate processing state.

NOTE (64)

If a simple specification is fast enough, keep it simple. If the program executes fast enough, let it carry out unneeded work. You must weight programming time esp. including the long term costs of maintenance against execution time. Be as declarative as possible, become only as imperative as needed. This may mean not to use state memorization and passing for faster execution, but to parallelize pattern matching instead (see following section).

31.4 Profile and Parallelize

A basic and often sufficient means of profiling is built into GrShell: After each sequence execution, the time required to carry it out is printed.

Sometimes you need further information about what's going on. Use the built-in profiling[1] then, by specifying the `profile` compiler option (cf. 29.1) to request search code instrumentation, or better the `set profile on` shell command (cf. 25.10). You receive two kinds of results when executing matchers instrumented this way.

1. After each sequence execution, the shell prints out the number of search steps that were carried out, in addition to the time. A search step consists of binding a graph element to a pattern element.

2. You can request a detailed per-action profile with the `show profile` shell command (cf. 25.4).

The detailed per-action profile contains a value that gives you a hint regarding the expected use of matcher parallelization, the higher the number the better is the rule suited.

The means available to parallelize – hopefully boosting performance – are described in Chapter 23 in detail. These are:

- parallel action matching with an annotation,
- parallel isomorphy checking with a dedicated operation,
- parallel execution in the sequences (explicitly programmed).

[1] You may of course apply a profiler on the generated code, but then you descend to the level of implementation of GrGen.NET

31.5 Compilation and Static Knowledge

Use saved graph analysis data

The different instance graphs for a certain graph-based problem you work with often show similar characteristics regarding the types and their connectedness. You can analyze such a characteristic graph, once, and save the results, with the `custom graph statistics save` command, cf. 25.12. And build the static matchers based on it, at each following static action generation, by loading that statistics file. This is possible with the `statistics` compiler option, cf. 29.1, and with the shell `new set statistics` command, cf. 25.10. It is seldom that up-to-the-point dynamic information about the host graph makes a real difference, while graph analysis and matcher re-generation at runtime are *costly* – push this effort from runtime to compilation time.

Use Compiled Sequences

The compiled sequences from the rules file are executed a good deal faster than the interpreted sequences from the shell. So if you have to optimize for performance, replace interpreted sequences by compiled ones. The price you pay is a loss of debuggability, a compiled sequence can only be executed as one big step (the introduction of subrule debugging – see 27.7 – improved on this, but the differences regarding state introspection and control are still considerable).

Use Pre-Compiled Code

For shortly-running tasks the JIT-compiling overhead dominates the execution runtimes. The times in the range of a few dozen milliseconds printed out for most of our tests are the times needed for just-in-time compilation of the .NET bytecode of the matchers into machine code — the matching itself is typically several orders of magnitude faster, it is in the range of microseconds or below (for the small example graphs and simple patterns used). This effect can be seen when a sequence takes some hundred milliseconds to execute when executed for the first time, but afterwards completes immediately (for a task of the same size). You may cut down on this overhead by utilizing the `ngen` tool of .NET for pre-compiling the GrGen.NET-dlls (the supplied as well as the generated ones, cf. 2.1), or the `--aot` option of mono, for ahead-of-time compilation.

31.6 Miscellaneous Things

Besides the things discussed in the following, you may employ compiler flags to disable certain features evading their performance impact (e.g. debugging of embedded execs), cf. Subsection 29.1; they can be also configured in the GrShell, then also firing of the action/debug events in the interpreted sequences gets disabled, see Section 25.10.

Use persistence provider transactions

When you use a persistent graph, it writes through every state change immediately to the database, i.e. each modification is contained in its own database transaction. This is perfect regarding the durability of each and every state change, but wreaks havoc on performance, cf. 26.1. Use persistence provider transactions to define more coarse grain synchronization points, i.e. wrap your sequence in persistence provider transaction angles `<: seq :>`, see 22.6 for more on this.

Visited Flags versus Storages versus Reflexive Edges

Visited flags are the most efficient way of marking elements, in case a large number of elements has to be marked, or if all elements – irrespective whether marked or not – need to be iterated. (This holds because they are stored in a flags variable of the graph elements themselves (the first k flags, afterwards they need to be stored outside of the graph)).

Otherwise they are inefficient because they do not allow to access(/lookup) the marked elements based on the marking information — a lot of elements need to be iterated for a lookup, just to filter the visited ones out. Storages that allow you to access their contained elements quickly are better then.

An alternative are reflexive marker edges of a special type in the graph, search planning favors them (assuming they appear in much smaller quantities than normal edges), so search starts at those elements. Besides they can be visualized directly in the graph (emulating a kind of "cursor").

Loops versus All-Bracketing versus Iterated

Regarding performance, you should prefer all-bracketing (cf. 11.1) to sequence loops and those to iterated patterns. Applying a rule on all matches is the most efficient way of processing multiple spots in a graph. This holds for normal matchers, but even more so for parallelized matchers. For a normal rule call, typically a part of the concurrent search effort is wasted because some workers already passed the point where the match was found; when all matches are sought, no effort is wasted – and typically the entire search takes longer in that case, which lowers the break-even point regarding the overhead incurred by the parallel matcher.

Loops are not much slower due to the search state space stepping, continuing where the last iteration left off. But they are semantically considerably different: each step of the loop operates on the then-changed graph, in contrast to applying a rule on all matches available at a certain point in time. If you have to explore new match possibilities created by applying the rule you need a loop, but even then you'll likely benefit from applying an all-bracketed rule in the loop. On the other hand you have to use a loop in case the parts of the matches which are to be modified, e.g. retyped, are not disjoint (an element can only be changed once).

Iterated patterns are similar to all-bracketed rules insofar that they match all spots in the graph existing at a certain point in time. They can be embedded directly in a rule (this is considerably less heavyweight than declaring a rule and passing the attachment points as parameters), and they allow for easy yielding of elements to the pattern (typically with an accumulating yield; this is also less heavyweight than returning elements back to a calling rule). But the overhead of the three pushdown machine must be paid for them, and they can't be matched in parallel. Using an iterated as the outermost block of a rule is wasteful, all-bracketing is to be preferred in this case. But don't replace iterateds unless really needed, they are simply much more convenient.

Reachable versus Subpattern Recursion

Prefer the reachable predicate over subpattern recursion. The reachable predicate has a tight implementation, while subpattern recursion must pay for graph parsing with the three pushdown machine (whose execution overhead is not horrible but can be clearly felt).

Note the semantic difference: elements already matched in the pattern cannot be matched again in case of subpattern recursion due to the isomorphy constraint. The reachable predicate does not know about the pattern it is called from, so elements already matched in the pattern can get matched again. This is often what is wanted anyway, and leads together with the increased convenience of having only to call a pre-implemented functionality instead of having to program an iterated path to the clear advice to favor the reachable predicate. You

have to use subpattern recursion, though, if you want to impose a certain pattern of type alternation on the iterated path, or if you want to match more than chains of single nodes linked by single edges (esp. attached tree-like structures), or if you want to somehow change the elements on the path – not searching merely for existence.

Helper Programs versus Patterns

GrGen.NET search planning can be compared to searching straw stars on a freshly harvested field, looking at the places where the ground is only slightly covered, only reaching into the haystacks when they can't be circumvented at all, and only for peripheral parts. A pattern matcher is generated based on the assumption that search planning worked well in circumventing those haystacks. It is based on nested loops for binding the pattern elements to the graph elements; and for a pattern node that is reached via multiple pattern edges on comparison code to check whether the node bound to the pattern node with the first loop is the same as the node that is reached later on from the other parts.

That approach works very well for sparse graphs with low incidence counts. But for graphs with massively linked parts that need to be reached on different paths you may need to overwrite this behaviour with hashset based connectedness checks.

The comparison code as such is a simple object identity comparison, so it is very cheap, but what weights in here are the cycles needed to iterate through the many different edge candidates until one is found that indeed is incident to the current node candidate. Hashset based connectedness checks are a good deal more expensive than iteration and reference comparison (esp. due to hashset building and filling), but they can be done in $O(1)$ in contrast. For a handful of elements, they straight out lose against the default code emitted by GrGen.NET, but when we speak of hundreds or thousands of elements, they win.

So when you have to find all nodes that are adjacent to two different nodes at the same time, and both of those have a considerable amount of adjacent nodes in addition, you should switch to the following approach: store the adjacent nodes of one node in a hash set, and query it with the adjacent nodes from the other node. This is only $O(n)$ instead of $O(n*n)$ due to the $O(1)$ hash set lookup (applied as query, or implicitly used in a hash set intersection).

This especially holds when you are only interested in the fact whether a certain minimum number is reached – in that case you may write a helper function that returns as soon as this amount is reached; see [Jak14] for an example of this. Using a helper function is also commonly more lightweight than using a helper rule (in case the helper function does not get large, which is the case when a pattern containing more than a handful of elements is needed).

Helper Programs versus Patterns on an Example

In the example [Jak14] mentioned above which is solving the TTC14 Movie Database Case [HKT14], we started optimizing the rule shown in Fig. 31.6 with the search plan printed by the `explain` command (cf. 25.12) in Figure 31.5.

```
findCouples:
    lookup -_edge0_inlined_idpt_0:personToMovie-> in graph
    from <-_edge0_inlined_idpt_0- get source pn1:Person
    from -_edge0_inlined_idpt_0-> get target m1_inlined_idpt_0:Movie
    from m1_inlined_idpt_0 incoming <-_edge1_inlined_idpt_0:personToMovie-
    from <-_edge1_inlined_idpt_0- get source pn2:Person
    independent {
        (preset: pn1)
        (preset: m1 after independent inlining)
        (preset: pn2)
        (preset: _edge0 after independent inlining)
        (preset: _edge1 after independent inlining)
        from pn1 outgoing -_edge2:personToMovie->
        from -_edge2-> get target m2:Movie
        from pn1 outgoing -_edge4:personToMovie->
        from -_edge4-> get target m3:Movie
        from pn2 outgoing -_edge3:personToMovie-> check m2 connected to _edge3
        from pn2 outgoing -_edge5:personToMovie-> check m3 connected to _edge5
    }
```

Figure 31.5: Initial search plan

```
rule findCouples
{
    pn1:Person; pn2:Person;
    independent {
        pn1 -:personToMovie-> m1:Movie <-:personToMovie- pn2;
        pn1 -:personToMovie-> m2:Movie <-:personToMovie- pn2;
        pn1 -:personToMovie-> m3:Movie <-:personToMovie- pn2;
    }

    modify {
        c:Couple;
        c -:p1-> pn1;
        c -:p2-> pn2;
    ---
        exec(addCommonMoviesAndComputeAverageRanking(c, pn1, pn2));
    }
} \ auto
```

Figure 31.6: findCouples rule

The graph is entered with a lookup of a `personToMovie` edge, then the candidates for the source node `pn1` and the target node `m1` are extracted from it. Afterwards, the other `personToMovie` edge is matched in reverse from the movie to `pn2`. Several elements in the `independent` are handed in as already matched presets from the outer pattern, then the `personToMovie` edges are taken from `pn1` to the movies `m2` and `m3`, and finally the `personToMovie` edges from `pn2` are matched, with an implicit check that the target movie is the same as the one already matched.

We see here an automatically applied optimization, the movie `m1` was inlined from the independent pattern to its containing pattern. Without this optimization, `pn1` and `pn2`

would have to be enumerated in the main pattern *unconnected*, resulting in the unfolding of the cartesian product of all `Person` nodes — before handing it in to the matcher of the nested independent pattern in order to purge the actors without a connecting movie.

Note that connectedness checks for nodes that are reached via multiple edges are carried out as early as possible. In the example search plan, take a look at the two latest lines:
`from pn2 outgoing -_edge3:personToMovie-> check m2 connected to _edge3`
`from pn2 outgoing -_edge5:personToMovie-> check m3 connected to _edge5`
For those two lines, after the edges were matched, there are no get target operations to fetch `m2` and `m3` existing, because the targets `m2` and `m3` were already matched from other edges on. Instead, the connectedness check is carried out here: it is checked that the target node of `_edge3` is equal to the target node already bound to `m2`, and that the target node of `_edge5` is equal to the target node already bound to `m3`.

Helper Programs versus Patterns on an Example, Optimized

In the example case [Jak14] we applied a succession of optimization steps. The final optimized rule is shown in Figure 31.8, and its helper functions in Figure 31.9 and Figure 31.10. Its search plan is listed in Figure 31.7 below.

```
findCouplesOpt:
    parallelized lookup pn1:Person in graph
    if { depending on findCouplesOpt_node_pn1 }
    from pn1 outgoing -p2m1_inlined_idpt_0:personToMovie->
    from -p2m1_inlined_idpt_0-> get target m1_inlined_idpt_0:Movie
    from m1_inlined_idpt_0 incoming <-p2m2_inlined_idpt_0:personToMovie-
    from <-p2m2_inlined_idpt_0- get source pn2:Person
    if { depending on findCouplesOpt_node_pn2 }
    if { depending on findCouplesOpt_node_pn1,findCouplesOpt_node_pn2 }
    independent {
        (preset: pn1)
        (preset: m1 after independent inlining)
        (preset: pn2)
        if { depending on findCouplesOpt_node_pn1,findCouplesOpt_node_pn2 }
        (preset: p2m1 after independent inlining)
        (preset: p2m2 after independent inlining)
    }
```

Figure 31.7: Final search plan

Note the parallelized first lookup that is spreading matching work over multiple threads. In addition, the computation of the common movies is executed in parallel, materializing the results to a common movies set, per match. Only a part of the original pattern is used here, because the connectedness check was moved to a helper function, called from an attribute condition.

```
rule findCouplesOpt[parallelize=16]
{
    pn1:Person; pn2:Person;
    hom(pn1,pn2);
    independent {
        pn1 -p2m1:personToMovie-> m1:Movie <-p2m2:personToMovie-
            pn2;
        hom(pn1,pn2); hom(p2m1,p2m2);
        if{ atLeastThreeCommonMovies(pn1, pn2); }
    }
    if{ uniqueof(pn1) < uniqueof(pn2); }
    if{ countFromIndex(countPersonToMovie, pn1) >= 3; }
    if{ countFromIndex(countPersonToMovie, pn2) >= 3; }
---
    def ref common:set<Node>;
    yield {
        yield common = getCommonMovies(pn1, pn2);
    }

    modify {
        c:Couple;
        c -:p1-> pn1;
        c -:p2-> pn2;
    ---
        eval {
            for(movie:Node in common) {
                add(commonMovies, c, movie);
            }
        }
    }
}
```

Figure 31.8: findCouplesOpt rule

```
function atLeastThreeCommonMovies(pn1:Person, pn2:Person) : boolean
{
    if(countFromIndex(countPersonToMovie, pn1) <=
       countFromIndex(countPersonToMovie, pn2)) {
        def var common:int = 0;
        def ref movies:set<Node> = adjacentOutgoing(pn1,
           personToMovie);
        for(movie:Node in adjacentOutgoing(pn2, personToMovie)) {
            if(movie in movies) {
                common = common + 1;
                if(common >= 3) {
                    return(true);
                }
            }
        }
    } else {
        def var common:int = 0;
        def ref movies:set<Node> = adjacentOutgoing(pn2,
           personToMovie);
        for(movie:Node in adjacentOutgoing(pn1, personToMovie)) {
            if(movie in movies) {
                common = common + 1;
                if(common >= 3) {
                    return(true);
                }
            }
        }
    }
    return(false);
}
```

Figure 31.9: atLeastThreeCommonMovies rule

```
function getCommonMovies(pn1:Person, pn2:Person) : set<Node>
{
    def ref common:set<Node> = set<Node>{};
    if(countFromIndex(countPersonToMovie, pn1) >=
       countFromIndex(countPersonToMovie, pn2)) {
        def ref movies:set<Node> = adjacentOutgoing(pn2,
           personToMovie);
        for(movie:Node in adjacentOutgoing(pn1, personToMovie)) {
            if(movie in movies) {
                common.add(movie);
            }
        }
    } else {
        def ref movies:set<Node> = adjacentOutgoing(pn1,
           personToMovie);
        for(movie:Node in adjacentOutgoing(pn2, personToMovie)) {
            if(movie in movies) {
                common.add(movie);
            }
        }
    }
    return(common);
}
```

Figure 31.10: getCommonMovies rule

Helper Programs versus Patterns on an Example, Profiling

When you enable profiling for the `imdb-0005000-50176.movies.xmi` example, cf. 31.4, you see a drastic reduction in search operations for the optimized version compared to the non-optimized version (and a corresponding reduction in runtime), compare 31.11 to 31.12.

Execution was carried out on a double core. The number of matches differs because automorphic matches filtering was executed during matching in one case, and after matching in the other case. Also, in the optimized version there are no matches occurring from an embedded *exec*, as all search work was shifted to the parallelized matcher.

```
Executing Graph Rewrite Sequence done after 23174.7 ms with result True:
 - 138718180 search steps executed
 - 34958 matches found
 - 34958 rewrites performed
> show profile findCouples
profile for action findCouples:
  calls total: 1
  steps of first loop of pattern matcher total: 5000
  search steps total (in pattern matching, if checking, yielding): 136430986
  search steps total during eval computation): 0
  search steps total during exec-ution (incl. actions called): 2287194
  search steps until one match: 0.00
  loop steps until one match: 0.00
  search steps per loop step (until one match): 0.00
  search steps until more than one match: 136430986.00
  loop steps until more than one match: 5000.00
  search steps per loop step (until more than one match): 4441.29
  parallelization potential: 474.49
```

Figure 31.11: Profile of findCouples

```
Executing Graph Rewrite Sequence done after 1764.0 ms with result True:
 - 2962438 search steps executed
 - 17479 matches found
 - 17479 rewrites performed
> show profile findCouplesOpt
profile for action findCouplesOpt:
  calls total: 1
  steps of first loop of pattern matcher total: 5000
  search steps total (in pattern matching, if checking, yielding): 2962438
  search steps total during eval computation): 0
  search steps total during exec-ution (incl. actions called): 0
for thread 0:
  search steps total: 373345
  loop steps total: 2193
  search steps until one match: 0.00
  loop steps until one match: 0.00
  search steps per loop step (until one match): 0.00
  search steps until more than one match: 373345.00
  loop steps until more than one match: 2193.00
  search steps per loop step (until more than one match): 33.64
for thread 1:
  search steps total: 374142
  loop steps total: 2807
  search steps until one match: 0.00
  loop steps until one match: 0.00
  search steps per loop step (until one match): 0.00
  search steps until more than one match: 374142.00
  loop steps until more than one match: 2807.00
  search steps per loop step (until more than one match): 15.91
```

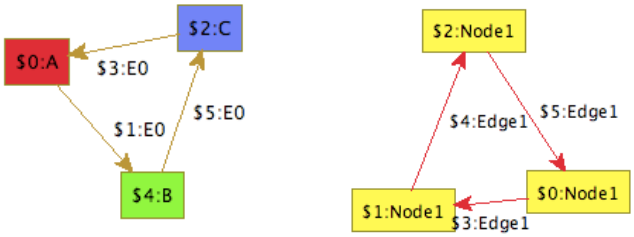
Figure 31.12: Profile of findCouplesOpt

CHAPTER 32

EXAMPLES

32.1 Fractals

The GrGen.NET package ships with samples for fractal generation. We will construct the Sierpinski triangle and the Koch snowflake. They are created by consecutive rule applications, starting at initial host graphs.



First of all, we have to compile the model and rule set files. So execute in GrGen.NET's `examples` directory

```
GrGen.exe Sierpinski\Sierpinski.grg
GrGen.exe Snowflake\Snowflake.grg
```

or

```
mono GrGen.exe Sierpinski/Sierpinski.grg
mono GrGen.exe Snowflake/Snowflake.grg
```

respectively (assuming grgen is contained in your search path, otherwise you have to execute it from the `bin` directory, adapting the paths to the specification files). We can increase the number of iterations to get even more beautiful graphs by editing the file `Sierpinski.grs` or `Snowflake.grs`, respectively. Just follow the comments. But be careful: the running time increases exponentially with the number of iterations, also yComp's layout algorithm might need some time and attempts to lay it out nicely.

We execute the Sierpinski script by

```
GrShell.exe Sierpinski\Sierpinski.grs
```

or

```
mono GrShell.exe Sierpinski/Sierpinski.grs
```

respectively.

Because both of the scripts are using the debug mode, we complete execution by typing `r`(un). See Section 25.4 for further information. The resulting graphs should look like Figures 32.1 and 32.2.

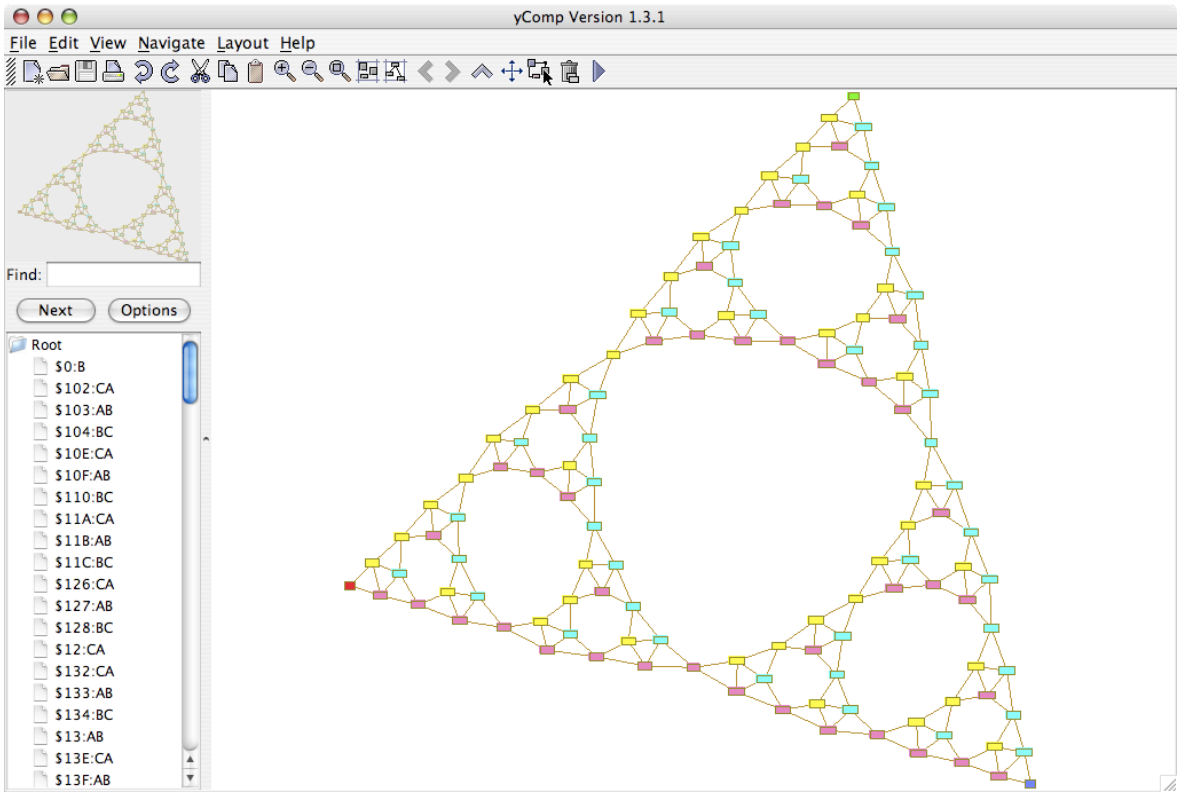


Figure 32.1: Sierpinski triangle

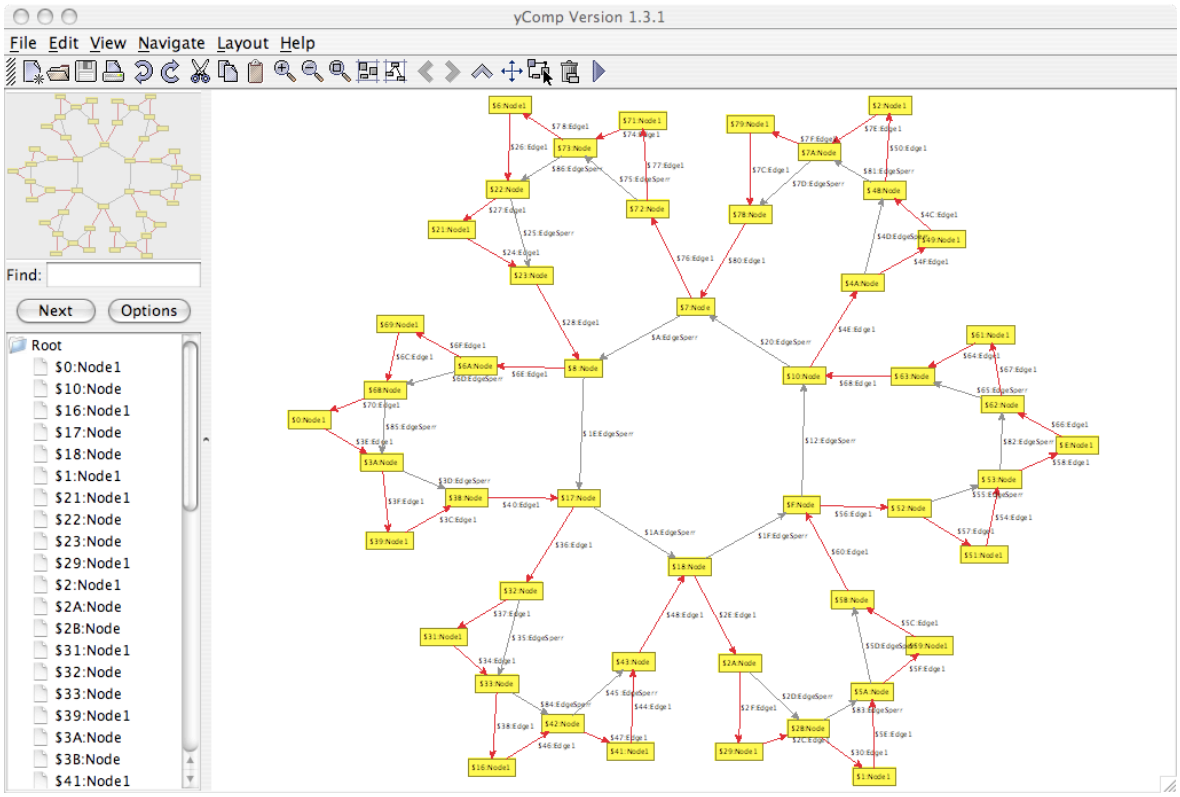


Figure 32.2: Koch snowflake

32.2 Busy Beaver

We want GrGen.NET to work as hard as a busy beaver [Kro07, Dew84]. A busy beaver is a Turing machine, in our case with five states plus a "halt"-state; it writes 1,471 bars onto the tape and terminates [MB00]. We use the graph model and the rewrite rules to define a general Turing machine. Our approach is basically to draw the machine as a graph. The busy beaver logic is implemented by rule applications in GrShell. Besides giving an example, this shows that GrGen.NET is Turing complete.

32.2.1 Graph Model

So first of all, we design a Turing machine as graph model. The tape will be a chain of `TapePosition` nodes connected by right edges. A cell value is modeled by a reflexive `value` edge, attached to a `TapePosition` node. The leftmost and the rightmost cells (`TapePosition`) do not have an incoming and outgoing edge, respectively. Therefore, we have the node constraint $[0:1]$.

```
21 node class TapePosition;
22 edge class right
23   connect TapePosition[0:1] --> TapePosition[0:1];
24
25 edge class value
26   connect TapePosition[1] --> TapePosition[1];
27 edge class zero  extends value;
28 edge class one   extends value;
29 edge class empty extends value;
```

Furthermore, we need states and transitions. The machine's current configuration is modeled with an `RWHead` edge pointing to a `TapePosition` node. `State` nodes are connected with `WriteValue` nodes via `value` edges, a `moveLeft`/`moveRight`/`dontMove` edge leads from a `WriteValue` node to the next state (cf. the picture on page 425).

```
30 node class State;
31
32 edge class RWHead;
33
34 node class WriteValue;
35 node class WriteZero extends WriteValue;
36 node class WriteOne extends WriteValue;
37 node class WriteEmpty extends WriteValue;
38
39 edge class moveLeft;
40 edge class moveRight;
41 edge class dontMove;
```

32.2.2 Rule Set

Now the rule set: We begin the rule set file `Turing.grg` with

```
1 #using "TuringModel.gm"
```

We need rewrite rules for the following steps of the Turing machine:

1. Read the value of the current tape cell and select an outgoing edge of the current state.
2. Write a new value into the current cell, according to the sub type of the `WriteValue` node.
3. Move the read-write-head along the tape and select a new state as current state.

As you can see a transition of the Turing machine is split into two graph rewrite steps: Writing the new value onto the tape, and performing the state transition. We need eleven rules: Three rules for each step (for "zero", "one", and "empty") and two rules for extending the tape to the left and the right, respectively.

```
3  rule readZeroRule {
4     s:State -h:RWHead-> tp:TapePosition -:zero-> tp;
5     s -:zero-> wv:WriteValue;
6     modify {
7        delete(h);
8        wv -:RWHead-> tp;
9     }
10 }
```

We take the current state `s` and the current cell `tp` which is implicitly given by the unique `RWHead` edge and check whether the cell value is zero. Furthermore, we check if the state has a transition for zero. The replacement part deletes the `RWHead` edge between `s` and `tp` and adds it between `wv` and `tp`. The remaining rules are analogous:

```
11 rule readOneRule {
12    s:State -h:RWHead-> tp:TapePosition -:one-> tp;
13    s -:one-> wv:WriteValue;
14    modify {
15       delete(h);
16       wv -:RWHead-> tp;
17    }
18 }
19
20 rule readEmptyRule {
21    s:State -h:RWHead-> tp:TapePosition -:empty-> tp;
22    s -:empty-> wv:WriteValue;
23    modify {
24       delete(h);
25       wv -:RWHead-> tp;
26    }
27 }
28
29 rule writeZeroRule {
30    wv:WriteZero -rw:RWHead-> tp:TapePosition -:value-> tp;
31    replace {
32       wv -rw-> tp -:zero-> tp;
33    }
34 }
35
36 rule writeOneRule {
37    wv:WriteOne -rw:RWHead-> tp:TapePosition -:value-> tp;
38    replace {
39       wv -rw-> tp -:one-> tp;
40    }
41 }
42
43 rule writeEmptyRule {
44    wv:WriteEmpty -rw:RWHead-> tp:TapePosition -:value-> tp;
45    replace {
46       wv -rw-> tp -:empty-> tp;
47    }
48 }
49
```

```
rule moveLeftRule {
   wv:WriteValue -:moveLeft-> s:State;
   wv -h:RWHead-> tp:TapePosition <-r:right- ltp:TapePosition;
   modify {
      delete(h);
      s -:RWHead-> ltp;
   }
}

rule moveRightRule {
   wv:WriteValue -:moveRight-> s:State;
   wv -h:RWHead-> tp:TapePosition -r:right-> rtp:TapePosition;
   modify {
      delete(h);
      s -:RWHead-> rtp;
   }
}

rule dontMoveRule {
   wv:WriteValue -:dontMove-> s:State;
   wv -h:RWHead-> tp:TapePosition;
   modify {
      delete(h);
      s -:RWHead-> tp;
   }
}

rule ensureMoveLeftValidRule {
   wv:WriteValue -:moveLeft-> :State;
   wv -:RWHead-> tp:TapePosition;
   negative {
      tp <-:right-;
   }
   modify {
      tp <-:right- ltp:TapePosition -:empty-> ltp;
   }
}

rule ensureMoveRightValidRule {
   wv:WriteValue -:moveRight-> :State;
   wv -:RWHead-> tp:TapePosition;
   negative {
      tp -:right->;
   }
   modify {
      tp -:right-> rtp:TapePosition -:empty-> rtp;
   }
}
```

Have a look at the negative conditions within the `ensureMove...` rules. They ensure that the current cell is indeed at the end of the tape: An edge to a right/left neighboring cell must not exist. Now don't forget to compile your model and the rule set with `GrGen.exe` (see Section 32.1).

32.2.3 Rule Execution with GrShell

Finally we construct the busy beaver and let it work with GrShell. The following script starts with building the Turing machine that is modeling the six states with their transitions in our Turing machine model:

```
select backend "../bin/lgspBackend.dll"
new graph "../lib/lgsp-TuringModel.dll" "Busy␣Beaver"
select actions "../lib/lgsp-TuringActions.dll"

# Initialize tape
new tp:TapePosition($="Startposition")
new tp -:empty-> tp

# States
new sA:State($="A")
new sB:State($="B")
new sC:State($="C")
new sD:State($="D")
new sE:State($="E")
new sH:State($ = "Halt")

new sA -:RWHead-> tp

# Transitions: three lines per state and input symbol for
#   - updating cell value
#   - moving read-write-head
# respectively

new sA_0: WriteOne
new sA -:empty-> sA_0
new sA_0 -:moveLeft-> sB

new sA_1: WriteOne
new sA -:one-> sA_1
new sA_1 -:moveLeft-> sD

new sB_0: WriteOne
new sB -:empty-> sB_0
new sB_0 -:moveRight-> sC

new sB_1: WriteEmpty
new sB -:one-> sB_1
new sB_1 -:moveRight-> sE

new sC_0: WriteEmpty
new sC -:empty-> sC_0
new sC_0 -:moveLeft-> sA

new sC_1: WriteEmpty
new sC -:one-> sC_1
new sC_1 -:moveRight-> sB

new sD_0: WriteOne
new sD -:empty-> sD_0
new sD_0 -:moveLeft->sE

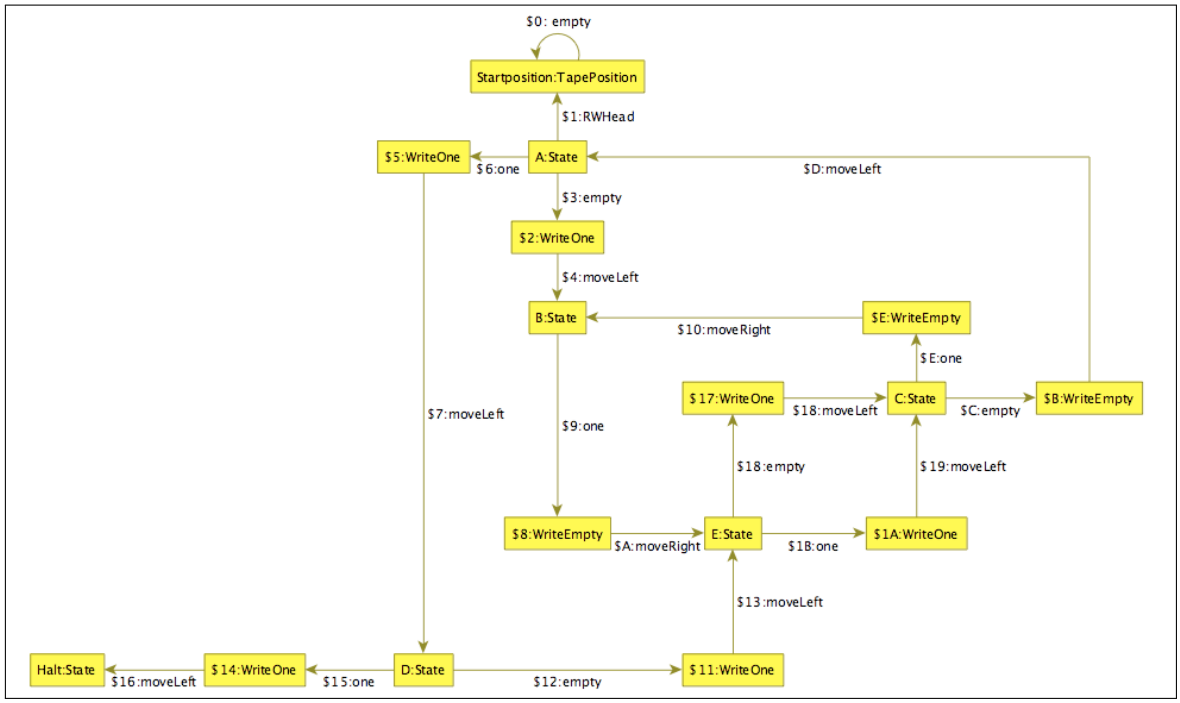
new sD_1: WriteOne
new sD -:one-> sD_1
new sD_1 -:moveLeft-> sH

new sE_0: WriteOne
new sE -:empty-> sE_0
new sE_0 -:moveRight-> sC

```

```
60 new sE_1: WriteOne
61 new sE -:one-> sE_1
62 new sE_1 -:moveLeft-> sC
```

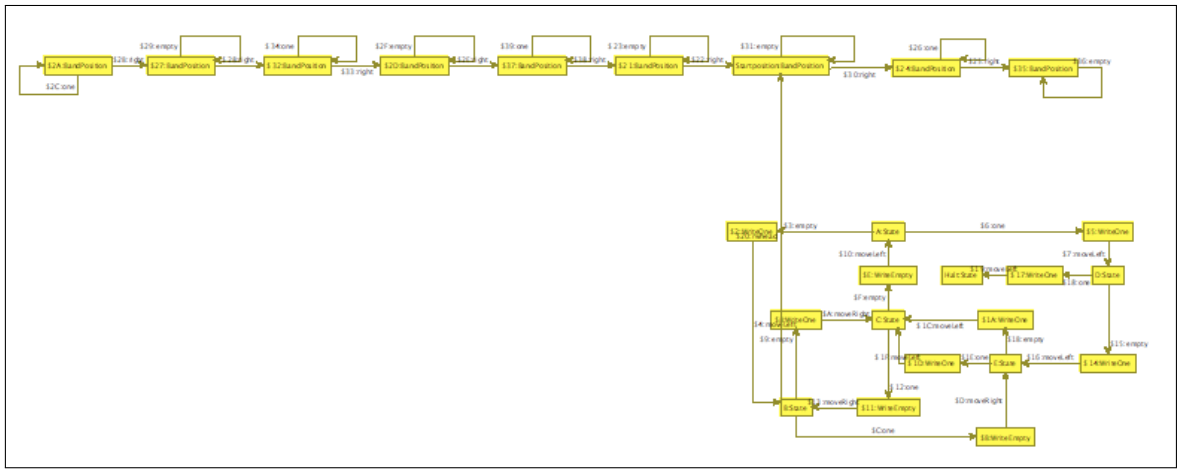
Our busy beaver looks like this:



We have an initial host graph now. The graph rewrite sequence is quite straight forward and generic to the Turing graph model. Note that for each state the "`...Empty...` — `...One...`" selection is unambiguous.

```
63 exec ((readOneRule | readEmptyRule) & (writeOneRule | writeEmptyRule) &
       (ensureMoveLeftValidRule | ensureMoveRightValidRule) & (moveLeftRule |
       moveRightRule))[32]
```

We interrupt the machine after 32 iterations and look at the result so far:



In order to improve the performance we generate better search plans. This is a crucial step for execution time: With the initial search plans the beaver runs for 1 minute and 30 seconds. With improved search plans after the first 32 steps it takes about 8.5 seconds[1].

[1] On a Pentium 4, 3.2Ghz, with 2GiB RAM.

```
64 custom graph analyze
65 custom actions gen_searchplan readOneRule readEmptyRule writeOneRule writeEmptyRule
       ensureMoveLeftValidRule ensureMoveRightValidRule moveLeftRule moveRightRule
```

Let the beaver run:

```
66   exec ((readOneRule | readEmptyRule) & (writeOneRule | writeEmptyRule) &
         (ensureMoveLeftValidRule | ensureMoveRightValidRule) & (moveLeftRule |
         moveRightRule))*
```

You can see the difference between the search plans causing that improvement in execution time by utilizing the `explain` command.

```
1 custom actions explain moveRightRule
```

We'll take a look at the search plan of the `moveRightRule`, first the initial version:

```
static search plans
moveRightRule:
    lookup -r:right-> in graph
    from <-r- get source tp:TapePosition
    from -r-> get target rtp:TapePosition
    from tp incoming <-h:rwHead-
    from <-h- get source wv:WriteValue
    from wv outgoing -_edge0:moveRight->
    from -_edge0-> get target s:State
```

After graph analysis and search plan regeneration it looks like this:

```
moveRightRule:
    lookup -h:rwHead-> in graph
    from <-h- get source wv:WriteValue
    from -h-> get target tp:TapePosition
    from wv outgoing -_edge0:moveRight->
    from -_edge0-> get target s:State
    from tp outgoing -r:right->
    from -r-> get target rtp:TapePosition
```

The crucial difference is the changed lookup, the rule begins matching at the single `rwHead` edge instead of one of the many `right` edges on the tape.

CHAPTER 33

APPLICATION PROGRAMMING INTERFACE

This chapter describes the Application Programming Interface of GrGen.NET, i.e. of the system runtime, explaining how to use the supplied or generated code from your own applications – the user interface of the compiler and shell applications with the command line arguments, the languages, and the GUI parts was already introduced in the previous chapters. The main part of the interface is defined by the libGr, which is implemented by the lgspBackend that also defines parts of the interface, as well as the assemblies generated from the model and rule specification files – but also other components come into play like e.g. the graphViewerAndSequenceDebugger assembly.

We'll have a look at

- the interface to the graph and the model
- the interface to the rules and the matches
- the interface of the graph processing environment (esp. for executing sequences)
- the porter module for importing and exporting of graphs, and miscellaneous stuff
- the interface to the debugger and the graph viewer
- the interface to the shell functionality
- the interface to the persistent graph and the persistence provider
- how to implement external class and function declarations
- how to implement external match filter and external sequence declarations
- the events fired when the graph is changed
- the events fired during action execution

The compiler `grgen.exe` generates from the input file `Foo.grg` the output files `FooModel.cs` for the model and `FooActions.cs` for the actions,

- defining the exact interface,
- implementing the exact interface with generated code and code from the lgsp backend, i.e. entities from `de.unika.ipd.grGen.lgsp` available from lgspBackend.dll,
- and implementing the generic interface from `de.unika.ipd.grGen.libGr` using the entities mentioned in both points above.

Normally you want to use the type-safe interface of the generated code as it is much more convenient. Only if your application has to get along with models and actions that are unknown before it is compiled, do you have to fall back to the generic interface. An extensive

example showing how to cope with the latter is shipped with GrGen.NET in form of the GrShell.

At runtime, you could generate a new rule set file, apply the compiler, and dynamically link the resulting assemblies/dlls (similar to what GrShell does), or re-generate the pattern matchers of some of the existing rules, dynamically compile them, and replace the old pattern matchers. But you cannot extend an existing rule set at runtime with new rules.

When working with the API, you could reference the generated binary dlls in your project, in the same way as you have to reference the `libGr.dll` and `lgspBackend.dll`. For easier debugging though, especially when you are integrating the generated code with own extensions (as described in chapter 34), it is recommended to directly include the source code generated (which is thrown away normally after it was compiled into the assemblies `lgsp-FooModel.dll` and `lgsp-FooActions.dll`). For this, use the `-keep` option when you call `grgen.exe`, and include the model and the actions file (excluding the intermediate actions file) directly as C# source code files (preferably, you also specify the `-debug` option then).

We recommend that you read this chapter together with one or several examples from the `examples-api` folder, and access to the API documentation at https://grgen.de/doc/API/index.html (including comments in addition to the raw type specifications) or even better the source code from https://github.com/ejaku/grgen, as we only replicate small parts of the classes and interfaces written about – the contained examples should be sufficient to give a first impression, though.

33.1 Interface to the Host Graph

The generated file `FooModel.cs` opens the namespace `de.unika.ipd.grGen.Model_Foo` containing all the generated entities. It contains for every node or edge class `Bar` an interface `IBar`, which offers C# properties giving access to the attributes, and is inheriting in the same way as specified in the model file. This builds the exact interface of the model (with sharp types), it is implemented by a sealed class `Bar` with generated code and with code from the lgsp backend. Furthermore, the namespace contains a model class `FooGraphModel` implementing the interface `de.unika.ipd.grGen.libGr.IGraphModel`, which supports iteration over the entities defined in the model, using further generic/inexact (we use generic in a wide sense here, not in the narrow sense meaning template methods or generic collections) interfaces from libGr (plus gives access to some model properties). Finally, the namespace contains a class `FooGraph` defining an `LGSPGraph` of a model equivalent to `FooGraphModel`; it contains convenience functions to easily create nodes and edges of exact type in the graph. Some examples with code will be given in the following section.

In addition, a class `FooNamedGraph` is available, which defines an `LGSPNamedGraph` of a model equivalent to `FooGraphModel`. The named graph offers persistent names (cf. 158) for all of its graph elements, but besides that, it is identical to an `LGSPGraph`. The named graph is the one utilized by GrShell, and the one recommended to be used for its ability to uniquely denote graph elements, but be aware that the storage of the names requires about the same amount of memory as an unnamed graph (without attributes) as such.

NOTE (65)

If you want to use the type-safe interface, your entry points are the `CreateNodeBar`-methods of the graph class `FooGraph` or the `CreateNode`-method of node class `Bar`, returning a node of the type `IBar`. If you want to use the generic interface, your entry point is the `IGraphModel`, with a call `INodeModel.GetType("Bar")` returning a `NodeType`, which is then used in a call `IGraph.AddNode(NodeType)`, returning a node of the type `INode`.

The interface types for the node and edge root types are already supplied in libGr and

are thus common to all graph rewrite systems (this allows for the "generic" handling); they are available as the `INode` interface for the `Node` root node type, as the `IEdge` interface for the `AEdge` root edge type, as well as the `IDEdge` interface for the directed `Edge` root edge type and the `IUEdge` interface for the undirected `UEdge` root edge type (in signatures where `IEdge` is used, the more concrete interface types allow to obtain directedness information directly from the .NET type).

The interfaces of the types you specified in the model are generated, as well as the backing implementation types of all these interface types.

EXAMPLE (176)

We start by including the namespaces of the libGr and the lgsp backend shipped with GrGen.NET plus the namespaces of our model, generated from `Foo.grg`.

```
using de.unika.ipd.grGen.libGr;
using de.unika.ipd.grGen.lgsp;
using de.unika.ipd.grGen.Model_Foo;
```

Then we create the model, followed by the graph, as an instance of the generic `LGSPGraph`, plus the global variables. We create a single node of type `Bar` in the graph, which we assign to a variable `b`. Afterwards, we read and write its attribute `value`.

```
IGraphModel model = new FooGraphModel();
IGraph graph = new LGSPGraph(model, new LGSPGlobalVariables());
Bar b = Bar.CreateNode(graph);
b.value = b.value + 1;
```

We could have created a named graph instead, that is also allowing to access graph elements by their persistent names:

```
INamedGraph graph = new LGSPNamedGraph(model, new LGSPGlobalVariables(), 0);
INode node = graph.GetNode("\$1");
```

EXAMPLE (177)

GrGen.NET also generates a specific graph class with the model bound to it at generation time, offering convenience functions based on the model. This is an example showing how 176 is realized with that graph version.

```
FooGraph graph = new FooGraph(new LGSPGlobalVariables());
Bar b = graph.CreateNodeBar();
```

Also a dedicated named graph class is available.

```
FooNamedGraph graph = new FooNamedGraph(new LGSPGlobalVariables());
```

EXAMPLE (178)

This is an example showing how to use the `IGraph` interface.

```
// query the IGraph graph, its model and the INode n
foreach(INode node in graph.Nodes) doSomething(node);
NodeType nt = graph.Model.NodeModel.GetType("Bar");
    //nt = de.unika.ipd.grGen.Model_Foo.NodeType_Bar.typeVar;
foreach(INode node in graph.GetCompatibleNodes(nt)) doSomething(node);
foreach(INode node in graph.GetExactNodes(nt)) doSomething(node);
foreach(IEdge edge in n.Outgoing)
{
    if(edge.Source == edge.Target) doSomething(edge.Source);
}

// update the graph by commands
INode n = graph.AddNode(nt);
IEdge e = graph.AddEdge(graph.Model.EdgeModel.GetType("Baz"), n, n);
n = Retype(n, graph.Model.NodeModel.GetType("Qux"));
graph.Remove(n);
```

33.2 Interface to the Rules

The generated file `FooActions.cs` opens the namespace `de.unika.ipd.grGen.Action_Foo` containing all the generated entities. It contains for every rule or test `bar`

- a class `Rule_bar` inheriting from `de.unika.ipd.grGen.lgsp.LGSPRulePattern`, which contains the exact match interface `IMatch_bar`, which in turn defines how a match of the rule looks like, extending the generic rule-unspecific `IMatch` interface. Have a look at section 34.4 for an extended introduction to the matches interfaces. Furtheron available are (but meant only for internal use): a match class `Match_bar` implementing the exact and inexact interface, a description of the pattern to match (the rule representation generated by the compiler frontend), and the modify methods for carrying out the rewriting.

- an exact action interface `IAction_bar`, which contains the methods:

  - `Match`, to match the pattern in the host graph, with in-parameters corresponding to the in-parameters of the rule (name and type), returning matches of the exact type `Rule_bar.IMatch_bar`.
  - `Modify`, to modify a given match according to the rewrite specification, with out-parameters corresponding to the out-parameters of the rule.
  - `Apply`, to match and modify the found match, with in-parameters corresponding to the in-parameters of the rule, and with ref-parameters corresponding to the out-parameters of the rule.

  Furtheron available is (but meant only for internal use) the class `Action_bar`, implementing the exact action interface as well as the generic `IAction` interface from LIBGR; it contains the generated matcher code searching for the patterns.

Moreover, the namespace contains an action class `FooActions` implementing the abstract class `BaseActions` from `de.unika.ipd.grGen.libGr` (in fact `LGSPActions` from the namespace `de.unika.ipd.grGen.lgsp`), which supports iteration over the entities defined in the

actions using further, generic (i.e. inexact) interfaces from libGr. Additionally, it contains the instances of the actions singletons, as member `bar` of the exact type `IAction_bar`.

NOTE (66)

If you want to use the type-safe interface, your entry point is the member `bar` of type `IAction_bar` from `FooActions` (or `Action_bar.Instance`). Actions are used with named parameters of exact types. If you want to use the generic interface, your entry point is the method `GetAction("bar")` of the interface `BaseActions` implemented by `FooActions` returning an `IAction`. Actions are used with `object`-arrays for parameter passing.

NOTE (67)

The old generic interface of names specified within strings and entities of node-, edge-, and object-type is implemented with the new interface of named and exactly typed entities. Thus you will receive runtime exceptions from the generic interface when you carry out operations that are not type-safe, in contrast to GrGen.NET < v2.5. If you need the flexibility of the old input parameter semantics of a silently failing rule application upon receipt of a wrong type, you must declare it explicitly with the syntax `r(x:ExactType<InexactType>)`; then the rule parameter in the exact interface will be of type `InexactType`.

EXAMPLE (179)

Here we'll show a short example of how to use GrGen.NET with the type-safe API, an extended version of the model-only 177.
We start by including the namespace of our actions, generated from `Foo.grg`, assuming the namespaces of the libGr, the lgspBackend, and the model were already included like in Example 176.

```
using de.unika.ipd.grGen.Action_Foo;
```

Then we create a graph with a model bound at generation time, followed by the actions to operate on this graph (the graph has to be initialized with the global variables). Finally, we apply the action `bar(Bar x) : (Bar)` on the graph via the graph processing environment `procEnv` (introduced in the following section), with `b` as input, also receiving the output. The rule is taken from the actions via the member named as the action.

```
FooGraph graph = new FooGraph(new LGSPGlobalVariables());
FooActions actions = new FooActions(graph);
Bar b = graph.CreateNodeBar();
actions.bar.Apply(procEnv, b, ref b); // input of type Bar, output of type Bar
```

EXAMPLE (180)

This is an extended version of the previous example 179, based on 176. Again, we create the actions to apply on the graph (here with a graph not bound to a model). Then we get the action from the actions and save it to an action variable `bar`. Afterwards, we use the action for finding all available matches of `bar` with input `b`, and remember the found matches in the exactly typed `matches` variable. Finally, we take the first match from the matches and execute the rewrite on it. We could have inspected the nodes and edges of the match or their attributes before (using element names prefixed with `node_/edge_` or attribute names to get exactly typed entities).

```
LGSPGraph graph = new LGSPGraph(model, new LGSPGlobalVariables());
FooActions actions = new FooActions(graph);
Bar b = Bar.CreateNode(graph);
IAction_bar bar = Action_bar.Instance;
IMatchesExact<Rule_bar.IMatch_bar> matches = bar.Match(procEnv, 0, b);
if(!someCheck(matches.FirstExact.node_someNodeFromRuleBar.GetAttribute("a")))
    return;
bar.Modify(procEnv, matches.First, out b);
```

EXAMPLE (181)

If you want to work with the generic (object and string based) interface not knowing the exact generated types, your entry point is the `LGSPBackend` instance implementing the `IBackend` interface.

```
IBackend backend = LGSPBackend.Instance;
IGlobalVariables globalVariables = LGSPBackend.Instance.CreateGlobalVariables();
INamedGraph graph;
IActions actions;
ProcessSpecFlags flags = ProcessSpecFlags.UseNoExistingFiles;
backend.CreateNamedFromSpec("Foo.grg", globalVariables, "graph-name", null,
                    flags, new List<String>(), 0,
                    out graph, out actions);
```

Regarding actions generation: The matcher code generated contains the initial, static search plans. When you analyze the graph at runtime and generate new matchers, see section 25.12 for more on this, you can request the dumping of the source code of the improved matchers. The custom commands are available at API level via the `Custom` operation of the `IGraph` interface for the graph commands and via the `Custom` operation of the `BaseActions` class for the actions commands, just handing in the same parameters as otherwise specified on the command line. If the intended workflow of "i) loading a typical graph or doing a warm-up run creating a typical graph, ii) analyzing that graph, iii) compiling new matchers that are better suited to the graph" is not easily achievable, or you want to start with the optimized matchers straight from the beginning, you may copy and paste the dumped dynamic matchers of an example run to the existing static code. But this is only a last resort, as the price is that your manual editing is overwritten again with the static search plans at the next time you call `grgen`. Besides, by now you could also request to generate initial matchers based on a statistics file with the compiler parameter `-statistics` and graph analyzation data saved in a previous session with the `custom graph statistics save` shell command.

33.3 Interface of the Graph Processing Environment

The graph processing environment – which is by now needed for all transformations – offers most of the additional processing-related functionality of GrGen.NET exceeding what is offered by the graph and the actions. It is made available by the interface `IGraphProcessingEnvironment`, which is implemented in the `LGSPGraphProcessingEnvironment` class.

EXAMPLE (182)

The graph processing environment `procEnv` is constructed on top of the graph and the actions, both of which are handed in. It esp. allows to carry out transformations by executing sequences, combining actions with operators (for control-flow) and variables (for data-flow).

```
IGraphProcessingEnvironment procEnv =
    new LGSPGraphProcessingEnvironment(graph, actions);
procEnv.ApplyGraphRewriteSequence("<(::x)=foo && (::y)=bar(::x) | bla(::y)>");
```

The `IGraphProcessingEnvironment` inherits from the `ISubactionAndOutputAdditionEnvironment` which in turn inherits from the `IActionExecutionEnvironment`. The `IActionExecutionEnvironment` contains certain configuration options, multiple helpers for action execution which are used by the sequences and could/should be preferred to using the actions directly if tight integration with the framework is targeted, and gives access to the most important parts of a graph rewrite system or transformation. The `ISubactionAndOutputAdditionEnvironment` enriches it by subgraph handling and output emitting plus graph change recording. The `IGraphProcessingEnvironment` contains the application logic `TransactionManager` and gives access to the `PersistenceProviderTransactionManager` iff one is available because a persistent graph is used, it contains helper code to interact with the graph global variables and allows to parse and apply sequences. All environments offer events, you find more on them in Section 33.11 following below.

The most important of the helper classes is the transaction manager which is utilized when GrGen.NET is used for crawling through a search space or for enumerating a state space, see section 22.2. The operations mentioned there are implemented by calling the functions given in example 183.

EXAMPLE (183)

```
LGSPGraphProcessingEnvironment procEnv = ...;
ITransactionManager tm = procEnv.TransactionManager;
public interface ITransactionManager
{
    int Start();
    void Pause();
    void Resume();
    void Commit(int transactionID);
    void Rollback(int transactionID);
    bool IsActive { get; }
    void ExternalTypeChanged(IUndoItem item);
}
```

The `Start` starts a transaction and returns its id; it may be called multiple times returning different ids for the transactions (i.e. nested transactions are supported, a failing outer one

rolls back the changes of an inner transaction which succeeded). Changes to the graph are recorded thereafter into an undo log, unless a `Pause` was called not yet followed by a `Resume`. When the changes of interest were carried out, the transaction identified by its id is either `Commit`ed, which causes the changes recorded since the corresponding `Start` to stay in the graph, or rolled back by calling `Rollback`, in that case all the changes recorded since the corresponding `Start` are undone. The `ExternalTypeChanged` allows you to include external attribute types in transaction handling, you must supply an undo item capable of rolling back the changes to the transaction manager, for each type a change is to be carried out.

33.4 Import/Export and Miscellaneous Stuff

GrGen natively supports the following formats:

GRS/GRSI
: Graph Rewrite Script files, which are a reduced version of the GrShell script files (also ending with `.grs`), limited to graph creation commands. They expect a model in a `.gm` file. This is the recommended standard format.

GXL
: Graph eXchange Language files, ending with `.gxl`, see http://www.gupro.de/GXL/.

ECORE/XMI
: Ecore(`.ecore`) model files and XMI(`.xmi`) graph files, which are formats defined and used by the Eclipse Modelling Framework. In an intermediate step, a `.gm` file is generated for the model.

GRG
: A GrGen rule file containing one rule with an empty pattern and a large rewrite part. Export only [1], not for normal use.

While both GRS and GXL importers expect one file (the GXL importer allows to specify a model override, see GrShell import, Note 54), the EMF/ECORE importer expects first one or more `.ecore` files, and following optionally an `.xmi` file and/or a `.grg` file (see Note 26.2 for more on this).

To import a graph model and/or a graph instance, you can use `Porter.Import()` from the libGr API (the GrShell command `import` is mapped to it). The file format is determined by the file extensions. To export a graph instance you can use `Porter.Export()` from the libGr API (the GrShell command `export` is mapped to it). For an example of how to use the importer/exporter on API level see `examples-api/JavaProgramGraphsExample/JavaProgramGraphsExample.cs` For more on import/export and its potential limitations, have a look at the corresponding GrShell Section 26.2.

The GRS(I) importer (the `.grsi` ending stands for graph rewrite script include) returns an `INamedGraph`; if you don't need the persistent names, get rid of them by casting to the `LGSPNamedGraph` implementing the interface, (copy-)constructing a `LGSPGraph` from it, and removing any references to the named graph. You may do so because naming is rather expensive (even though worthwhile in the general case): a `LGSPNamedGraph` supplying the name to element and element to name mappings normally uses up about twice the amount of memory of the `LGSPGraph` defining the graph alone.

External Emitting and Parsing

If `external emit class;` or `external emit graph class;` (see 34.6) are specified in the model file `Foo`, GrGen.NET generates a file `FooModelExternalFunctions.cs` located beside the model and rule files, containing the following functions.

[1] Original German Pisswasser, for export only :)

```
/// <summary>
/// Called during .grs import, at exactly the position in the text reader where the
    attribute begins.
/// For attribute type object or a user defined type, which is treated as object.
/// The implementation must parse from there on the attribute type requested.
/// It must not parse beyond the serialized representation of the attribute,
/// i.e. Peek() must return the first character not belonging to the attribute type
    any more.
/// Returns the parsed object.
/// </summary>
object Parse(TextReader reader, AttributeType attrType, IGraph graph);

/// <summary>
/// Called during .grs export, the implementation must return a string
    representation for the attribute.
/// For attribute type object or a user defined type, which is treated as object.
/// The serialized string must be parseable by Parse.
/// </summary>
string Serialize(object attribute, AttributeType attrType, IGraph graph);

/// <summary>
/// Called during debugging or emit writing, the implementation must return a string
    representation for the attribute.
/// For attribute type object or a user defined type, which is treated as object.
/// The attribute type may be null.
/// The string is meant for consumption by humans, it does not need to be parseable.
/// </summary>
string Emit(object attribute, AttributeType attrType, IGraph graph);
```

The `Parse` function is called when an attribute of an external or object type is to be imported from a grs file.

The `Serialize` function is called when an attribute of an external or object type is to be exported to a grs file.

The `Emit` function is called when a value of external or object type is to be emitted, or displayed in the debugger (including yComp).

The functions forward the calls to `ParseImpl`, `SerializeImpl`, and `EmitImpl` functions, respectively, which have to be implemented in a file named `FooModelExternalFunctionsImpl.cs`, located in the folder of the `FooModelExternalFunctions.cs` file. Implementing them is *your* task, in exchange you get a tight integration of your own datatypes into GrGen.NET. For an example, you may have a look at `ExternalAttributeEvaluation` in the `tests` folder or `ExternalAttributeEvaluationExample` in the `examples-api` folder.

The functions are called from GrShell, too, insofar as possible – the shell parses a single or double quoted text or a word or a number at an attribute position and hands that over then to the user defined parser.

```
/// <summary>
/// Called when the grs importer or the shell hits a line starting with "external".
/// The content of that line is handed in.
/// This is typically used while replaying changes containing a method call of an
    external type
/// -- after such a line was recorded, by the method called, by writing to the
    recorder.
/// This is meant to replay fine-grain changes of graph attributes of external type,
/// in contrast to full assignments handled by Parse and Serialize.
```

```
/// </summary>
void External(string line, IGraph graph);
```

The `External` function is called when a line starting with `external` is seen by the shell or the grs importer, in this case the content up to the end of the line is munched and handed in to this function. These lines are typically recorded by calls of the `public void External(string value)` function supplied in the `IRecorder` interface, from external attribute method calls (cf. 34.1), for persisting changes of the external method calls. For assignments of complete attribute values, you must implement `Parse` and `Serialize`. The `External` function is existing to support fine-grain changes of external attribute types.

```
/// <summary>
/// Called during debugging on user request, the implementation must return a named
    graph representation for the attribute.
/// For attribute type object or a user defined type, which is treated as object.
/// The attribute type may be null. The return graph must be of the same model as
    the graph handed in.
/// The named graph is meant for display in the debugger, to visualize the internal
    structure of some attribute type.
/// This way you can graphically inspect your own data types which are opaque to
    GrGen with its debugger.
/// </summary>
INamedGraph AsGraph(object attribute, AttributeType attrType, IGraph graph);
```

The `AsGraph` function is called from the debugger on user request, to visually inspect an external attribute type, rendered as graph. It will be called when `external emit graph class;` was specified. It forwards the call to `AsGraphImpl`, which needs to be implemented by *you*, in the same way as specified above for the functions called when `external emit class;` is given.

External Copying and Comparing

If `external copy class;` or `external ~~ class;` or `external < class;` (see 34.7) are specified in the model file `Foo`, GrGen.NET generates a file `FooModelExternalFunctions.cs` located beside the model and rule files, containing a *partial* class `AttributeTypeObjectCopierComparer`. You have to implement the `Copy` or the `IsEqual` or the `IsLower` functions in the same partial class in `FooModelExternalFunctionsImpl.cs`. For every external type defined, another function showing that type in the input parameters is expected.

```
// Called when a graph element or internal (transient) object bearing attributes of
    external object type is to be copied.
// Also called when a top-level external object is to be cloned or copied.
// If "copy class" is not specified, object attributes are copied by copying the
    reference, i.e. they are identical afterwards (top-level objects cannot be
    copied/cloned in this case).
// If "copy class" is specified:
// If the old to new element dictionary is null, objects are to be cloned, i.e.
    top-level object (of the very call) is to be cloned and others are just assigned
    by reference.
// Otherwise, they are to be copied by-value (so changing one attribute later on has
    no effect on the other).
public static object Copy(object, IGraph, IDictionary<object, object>);

// Called during comparison of graph elements from graph isomorphy comparison, or
    during deeply equal attribute comparisons.
```

```
// For attribute type object.
// If "~~ class" is not specified, objects are equal if their references are
    identical.
// The visited objects dictionary contains the already visited objects, insert your
    object here to detect multiple appearances/cycles (and check against it).
public static bool IsEqual(object, object, IDictionary<object, object>);

// Called during attribute comparison.
// For attribute type object.
// If "< class" is not specified, objects can't be compared for ordering, only for
    equality.
public static bool IsLower(object, object, IDictionary<object, object>);
```

The `Copy` function is called when one of the copy operations offered in the rule language or the computation statements is executed, on an attribute bearing type (node class or edge class or object class or transient object class) that bears attributes of object or a user-defined external type. The dictionary `oldToNewObjectMap` is `null` in case of a `clone` operation, it contains a mapping of old to new elements in case of a `copy` operation (which is used to assign the same element upon multiple appearances, and prevent endless recursions in case of cyclical structures) – pull from it and add to it as required.

The `IsEqual` function is called when values (attributes) of object or a user-defined external type are compared with the structural (deep) equality operator `~~` (`==` only checks for reference identity), or when two graphs are compared for isomorphy and the graph elements contain attributes of object or a user-defined external type. The `IsLower` function is called when values (attributes) of object or a user-defined external type are compared with one of the relational operators. A comparison for `u<=v` is mapped to an expression `IsLower(u,v) || IsEqual(u,v)`, for this reason a `< class;` specification requires a preceding `~~ class;` specification. The dictionary `visitedObjectsMap` contains the already visited elements (as keys, the values are null), it is used to detect multiple appearances of objects (esp. due to cycles). You have to follow the same protocol, i.e. you have to check whether the currently compared objects are already contained (this has to cause a crash because of an unsupported shortcut/cycle), and you have to add them upon entry and have to remove them upon exit.

Implementing these functions is *your* task, in exchange you get a tight integration of your own datatypes into GrGen.NET. You may have a look at `tests/ExternalAttributeEvaluation` or `examplesapi/ExternalAttributeEvaluationExample` for an example.

Further Examples

There are further examples available in the `examples-api` folder of the GrGen.NET-distribution:

- How to use the terse graph rewrite sequences on API level is shown in (cf. also 33.3) `examples-api/BusyBeaverExample/BusyBeaverExample.cs`.
  Alternatively, you could directly use your favourite .NET programming language for combining single rule applications, employing the type-safe interface.
- How to use the old and new interface for accessing a match on API level is shown in `examples-api/ProgramGraphsExample/ProgramGraphsExample.cs`.
- How to use the visited flags on API level is shown in `examples-api/VisitedExample/VisitedExample.cs`.
- How to analyze the graph and generate matchers based on this information – which are (hopefully) performing better – is shown in `examples-api/BusyBeaverExample/BusyBeaverExample.cs`.

- How to compile a `.grg`-specification at runtime and dump a graph for visualization in `.vcg` format on API level is shown in `examples-api/HelloMutex/HelloMutex.cs`.

- How to access the annotations at API level is shown in `examples-api/MutexDirectExample/MutexDirectExample.cs`.

NOTE (68)

While C# allows input argument values to be of a subtype of the declared interface parameter type (OO), it requires the argument variables for the `out` parameters to be of exactly the type declared (non-OO). Although a variable of a supertype would be fully sufficient – the variable is only assigned. So for `node class Bla extends Bar;` and action `bar(Bar x) : (Bla)` from the rules file `Foo.grg` we cannot use a desired target variable of type `Bar` as `out`-argument, but are forced to introduce a temporary variable of type `Bla` and assign this variable to the desired target variable after the call.

```
using de.unika.ipd.grGen.libGr;
using de.unika.ipd.grGen.lgsp;
using de.unika.ipd.grGen.Action_Foo;
using de.unika.ipd.grGen.Model_Foo;
FooGraph graph = new FooGraph();
FooActions actions = new FooActions(graph);
Bar b = graph.CreateNodeBar();
IMatchesExact<Rule_bar.IMatch_bar> matches = actions.bar.Match(graph, 1, b);
//actions.bar.Modify(graph, matches.First, out b); // wont work, needed:
Bla bla = null;
actions.bar.Modify(graph, matches.First, out bla);
b = bla;
```

33.5 Graph Viewer and Debugger

The `graphViewerAndSequenceDebugger` project contains convenience interfaces that allow you to access the graph viewer and the sequence debugger easily from API level (you have to include the `graphViewerAndSequenceDebugger.dll` in order to do so, and in case you want to use the MSAGL based debugger, you have to make the `graphViewerAndSequenceDebuggerWindowsForms.dll` available beside the aforementioned dynamic link library, note its package reference to the NuGet package `Microsoft.Msagl.GraphViewerGDI` requiring the inclusion of `System.Resources.Extensions` with a `bindingRedirect` in the `app.config` of your application (the `GrShell.csproj` from the sources may serve as an example)).

The `GraphViewer` class esp. offers methods to

- show a graph until it is closed, adapting to changes (`ShowGraph`, `EndShowGraph`, `UpdateDisplayAndSync`)

- dump a graph to a file in a graph visualization format and show it (`DumpAndShowGraph`)

The `examples-api/YCompExample/YCompExample.cs` shows how to use the `GraphViewer` class with YComp (or alternatively MSAGL) in order to display a graph (and changes to it), as well as to dump a graph as a `.vcg` file and render it (utilizing parts of the `Mutex` example).

The `Debugger` class allows you to execute a graph rewrite sequence under debugger control (`ApplyGraphRewriteSequence`) from your own application, once it is configured properly esp.

with the required environments (DebuggerEnvironment and DebuggerGraphProcessingEnvironment) by utilizing debugger.InitNewRewriteSequence(sequence, true); and the procEnv.ApplyGraphRewriteSequence(sequence);.

The examples-api/DebuggerExample/DebuggerExample.cs shows how to execute a sequence in the debugger (in a stdout command line window in case of GraphViewerTypes.YComp or GraphViewerTypes.ExtMSAGLExt, or in a WindowsForms command line window emulation in case of GraphViewerTypes.MSAGL, or GraphViewerTypes.MSAGLExt if you specify it) (utilizing parts of the Mutex example; and a GuiConsoleControl, hosted in a GuiConsoleDebuggerHost in case of the WindowsForms example).

33.6 Shell functionality

The libGrShell project allows you to include GrShell-functionality (this means esp. executing a .grs script file, but you could also use it as an in-place interactive command line) into your own application (you have to reference the libGrShell.dll (and the graphViewerAndSequenceDebugger.dll in order to do so)).

The examples-api/ShellExample/ShellExample.cs shows how to achieve this (utilizing parts of the Mutex example).

The examples-api/ShellExampleWindowsForms/Program.cs and examples-api/ShellExampleWindowsForms/ShellForm.cs show how to utilize the libGrShell from a Windows Forms application (note that it requires the libConsoleAndOSWindowsForms.dll).

The examples-api/ApplicationExample/ApplicationExampleForm.cs shows how to use the Debugger and the GrShell from a GUI application (utilizing parts of the Mutex example).

The console application GrShell and the Windows Forms application GGrShell could also serve as examples, but then you'd need the source code of GrGen.NET.

33.7 Persistent Graph and Persistence Provider

In order to use a persistent graph, you have to create an LGSPPersistentNamedGraph with one of its constructors (or with one of the CreateXXX functions from the IBackend, when you want to work with the libGr-API only, without calling methods of the lgspBackend directly). You have to specify the persistenceProviderDllName, as of now only libGrPersistenceProviderSQLite.dll is supported, and the connectionParameters, these are the same that you give with the new graph persist with to shell command (cf. 26.1 – there, you also find some words on the design in a subsection). Also, persistentGraphParameters may be given, see 26.1.1 for more on them.

The persistent graph is a thin extension of an LGSPNamedGraph, the major part of the work is delegated to the persistence provider. You may implement a persistence provider on your own, it only has to be in an assembly and contain a class that implements the IPersistenceProvider interface.

When using a persistent graph, you may be interested in the PersistenceProviderTransactionManager field of the IGraphProcessingEnvironment, that allows to access an IPersistenceProviderTransactionManager object, which – when available – gives access to database transaction processing.

EXAMPLE (184)

```
public interface IPersistenceProviderTransactionManager
{
    void Start();
    void CommitAndRestart();
    void Commit();
    void Rollback();
    bool IsActive { get; }
}
```

The `Start` starts a transaction if none exists yet (nested transactions are *not* supported). The `CommitAndRestart` carries out an intermediate commit, afterwards the changes up to that point are persistently stored (for sure), and a transaction is active; this is what is typically used to balance performance against the granularity of persistence – for real transactions, use the application layer transaction manager. The `Commit` commits the transaction, afterwards the changes up to that point are persistently stored (for sure), and a transaction is not active anymore. The `Rollback` rolls the changes since that last CommitAndRestart() or the last Start() if no restart occurred back, afterwards the transaction is not active anymore. Pending changes are forgotten when no commit occurs.

The persistence provider depends on an SQLite ADO.NET driver, which is included by a `System.Data.SQLite` package reference, and on the SQLite itself, a build of it is included by a `SourceGear.sqlite3` package reference. The release is a `x64` version only by now, which contains Windows and LINUX binaries of the native SQLite (`.dll` and `.so`). See 35.1 for more on this (e.g. if you want to build an `x86` version).

33.8 External Class, Function and Procedure Implementation

For a model file `Foo` that contains

- external classes (cf. 34.1), or
- external functions (cf. 34.2), or
- external procedures (cf. 34.3),

GrGen.NET generates a file `FooModelExternalFunctions.cs` located beside the model and rule files, which contains

- within the model namespace public partial classes named as given in the external class declaration, inheriting from each other as stated in the external class declarations, containing a body of comments giving the expected function and procedure method prototypes.
- within the `de.unika.ipd.grGen.expression` namespace a public partial class named `ExternalFunctions` with a body of comments giving the expected function prototypes.
- within the `de.unika.ipd.grGen.expression` namespace a public partial class named `ExternalProcedures` with a body of comments giving the expected procedure prototypes.

The generated partial classes come without implementation, you must implement them in a file named `FooModelExternalFunctionsImpl.cs`, located in the folder of the `FooModelExternalFunctions.cs` file, by replicating them, and fleshing them out:

- the partial classes skeletons with attributes containing data of interest and maybe helper methods

- the `ExternalFunctions` partial class skeleton with the functions you declared in the external function declarations, obeying the function signatures as specified; here you can access the attributes or methods of the external classes (which are known at this point in the code, they are not known in the specification language), or carry out complicated custom computations or graph queries, based on the values you receive with the function call.

- the `ExternalProcedures` partial class skeleton with the procedures you declared in the external procedure declarations, obeying the procedure signatures as specified; here you can access the attributes or methods of the external classes (which are known at this point in the code, they are not known in the specification language), or carry out complicated custom computations or graph manipulations, based on the values you receive with the procedure call.

Don't forget that the source code file `FooModelExternalFunctionsImpl.cs` is an integral part of your GrGen.NET graph transformation project, in contrast to the other C# files that are generated (and get overwritten). You find a fabricated example showing how to use the external classes and functions in `examples/ExternalAttributeEvaluationExample` and `examples-api/ExternalAttributeEvaluationExample`.

When you use third-party assemblies in your source code, you must inform GrGen.NET about them, so references to them are included into the assembly generated; this can be done with the `-r` parameter when calling `grgen.exe` directly (cf. 29.1) or with the `new add reference` command of GrShell (cf. 25.10). Using the `keepdebug` configuration option of the `new` command is recommended as it allows for easier debugging.

33.9 External Filter and Sequence Implementation

For an actions file `Bar` that contains

- external match filter declarations (cf. 34.4), or

- external sequence declarations (34.5),

GrGen.NET generates a file `BarActionsExternalFunctions.cs` located beside the model and rule files, which contains within the action namespace

- a public partial class named `MatchFilters` with a body of comments giving the expected function prototypes, and for the `auto` filter even the implementation.

- public partial classes, named `Sequence_foo` for a sequence `foo`, with a body containing a comment specifying the expected function prototype of the sequence application function.

The partial classes come without implementation, you must implement them in a file named `BarActionsExternalFunctionsImpl.cs`, located in the folder of the `BarActionsExternalFunctions.cs` file, by replicating them, and fleshing them out:

- the `MatchFilters` partial class skeleton with the match filter functions you declared, obeying the function signatures as specified; you might want to convert the received matches object to an `IList` in case you want to reorder the list and reinject it into the matches object afterwards.

- the partial class skeletons of the external sequences with the required `ApplyXGRS_foo` functions, obeying the signature of the sequence application function as specified.

Don't forget that the source code file `BarActionsExternalFunctionsImpl.cs` is an integral part of your GrGen.NET graph transformation project, in contrast to the other C# files that are generated (and get overwritten). You find a fabricated example showing how to use the external classes and functions in `examples/ExternalFiltersAndSequencesExample` and `examples-api/ExternalFiltersAndSequencesExample`.

When you use third-party assemblies in your source code, you must inform GrGen.NET about them, so references to them are included into the assembly generated; this can be done with the `-r` parameter when calling `grgen.exe` directly (cf. 29.1) or with the `new add reference` command of GrShell (cf. 25.10). Using the `keepdebug` configuration option of the `new` command is recommended as it allows for easier debugging.

NOTE (69)

libGr allows to split a rule application into two steps: Firstly, searching for the subgraphs of the host graph that match the pattern, and secondly rewriting of some of these matches. This occurs by employing the following methods of the `IAction` (or better `IActionCore`) interface:

```
IMatches Match(IActionExecutionEnvironment actionEnv, int maxMatches, object[]
    parameters);
object[] Modify(IActionExecutionEnvironment actionEnv, IMatch match);
```

In C#, this might look like:

```
IMatches myMatches = myAction.Match(myProcEnv, -1, null); // -1: get all the matches
for(int i=0; i<myMatches.NumMatches; ++i)
{
   if(inspectCarefully(myMatches.GetMatch(i)))
   {
      myAction.Modify(myProcEnv, myMatches.GetMatch(i));
      break;
   }
}
```

The external match filters are executed in place of the `inspectCarefully`, in between the calls to the `Match` and the `Modify` functions. They allow you to write a complex inspection routine in a .NET language, that can then still be used from GrGen.NET internal sequences. Many interesting filters can be even generated automatically for you, e.g. the `auto` filter for filtering symmetric matches of automorphic patterns, see 34.4 and 17 for more on this. You can call the (internal) filters by employing the following method of the `IAction` interface:

```
void Filter(IActionExecutionEnvironment actionEnv, IMatches matches,
    FilterCallWithArguments filter);
```

33.10 Graph Events

Before or after the host graph is changed, events are fired, notifying listeners about the changes. The debugger, the graph viewers, the transaction handler, the graph change recorder, and the persistent graph implement their functionality by listening and reacting to these events. You may add own event handlers to insert custom-made, event-based functionality; or may even implement an event-driven rule execution engine on top of it.

The events are fired automatically by the methods of the graph class on modifications (excluding attribute changes), and by the actions which get applied. If you operate with external code, it's your responsibility to fire the events for attribute changes before changing

an attribute. Otherwise the changes won't be visible in the debugger, they won't be rolled back at the end of a transaction or during backtracking, and they won't be recorded in case of change recording.

Quite a lot of a three layer architecture graph rewrite application is available out of the box due to the events, see also 29.16 (a user application you supply, working at API level, or the GrShell/GGrShell system applications).

The events available that are fired automatically on graph changes (excluding attribute value changes) are:

```
// Fired after a node has been added
event NodeAddedHandler OnNodeAdded;

// Fired after an edge has been added
event EdgeAddedHandler OnEdgeAdded;

// Fired after an object has been created (there is no event when it gets destroyed)
event ObjectCreatedHandler OnObjectCreated;

// Fired before a node is deleted
event RemovingNodeHandler OnRemovingNode;

// Fired before an edge is deleted
event RemovingEdgeHandler OnRemovingEdge;

// Fired before all edges of a node are deleted
event RemovingEdgesHandler OnRemovingEdges;

// Fired before the whole graph is cleared
event ClearingGraphHandler OnClearingGraph;

// Fired before the type of a node is changed.
event RetypingNodeHandler OnRetypingNode;

// Fired before the type of an edge is changed.
event RetypingEdgeHandler OnRetypingEdge;

// Fired before an edge is redirected (causing removal then adding again).
event RedirectingEdgeHandler OnRedirectingEdge;
```

The events available that are fired automatically on attribute value changes of graph elements by the code generated to implement the actions, but not by the graph, and thus need to be fired by you if you change attribute values on API level, are:

```
// Fired before an attribute of a node is changed.
event ChangingNodeAttributeHandler OnChangingNodeAttribute;

// Fired before an attribute of an edge is changed.
event ChangingEdgeAttributeHandler OnChangingEdgeAttribute;

// Fired before an attribute of an object is changed.
event ChangingObjectAttributeHandler OnChangingObjectAttribute;

// Fired before each rewrite step (also rewrite steps of subpatterns) to indicate
// the names of the nodes added in this rewrite step in order of addition.
```

```
event SettingAddedElementNamesHandler OnSettingAddedNodeNames;

// Fired before each rewrite step (also rewrite steps of subpatterns) to indicate
// the names of the edges added in this rewrite step in order of addition.
event SettingAddedElementNamesHandler OnSettingAddedEdgeNames;
```

If you listen to or fire the action application events (cf. 33.11), you may be interested in the added names event from the list above, too, which tells about the names of the elements which will get added immediately thereafter (this is used in the debugger to display the names of the elements as defined in the rule modify part).

For a container type this means single element additions or removals, that's the reason why the interface for attribute changes is more complex than what you'd expect intuitively:
`void ChangingNodeAttribute(INode node, AttributeType attrType, AttributeChangeType changeType, Object newValue, Object keyValue)`

The chang*ing* attribute events from above are fired before the attributes are assigned, with the single element change that will occur. They are used e.g. by the transaction manager to record the changes (note that rollback of changes of attributes of container type requires single element notifications, complete container copies would be prohibitively costly).

The conditional watchpoints that may be configured in the debugger in contrast hook into the chang*ed* attribute events that are fired after an attribute was assigned. These are:

```
// Fired after an attribute of a node is changed; for debugging purpose.
event ChangedNodeAttributeHandler OnChangedNodeAttribute;

// Fired after an attribute of an edge is changed; for debugging purpose.
event ChangedEdgeAttributeHandler OnChangedEdgeAttribute;
```

The events available that are fired automatically on visited flag changes are:

```
/// Fired after a visited flag was allocated.
event VisitedAllocHandler OnVisitedAlloc;

/// Fired after a visited flag was freed.
event VisitedFreeHandler OnVisitedFree;

/// Fired before a visited flag is set.
event SettingVisitedHandler OnSettingVisited;
```

33.11 Action Events

When actions (esp. rules) are executed, events are fired, notifying listeners about the changes. The GrShell-debugger implements its functionality by listening and reacting to these events. You may add own event handlers to insert custom-made, event-based functionality.

The events are fired automatically by the code generated by GrGen for the actions or sequences. If you operate with external code, it's your responsibility to fire the events – in case you want to simulate rules or sequences. Otherwise the changes won't be visible in the debugger.

The action based events that are declared by the `IActionExecutionEnvironment` are:

```
// Fired when execution of a pattern matching construct begins.
event BeginExecutionHandler OnBeginExecution;

// Fired after all requested matches of a multi rule or rule have been matched,
    before filtering of the matches.
```

```
// Allows a lookahead on the matches, on all found matches, in contrast to
    OnMatchedAfter that only reports the ones that are applied in the end.
event MatchedBeforeFilteringHandler OnMatchedBefore;

// Fired after all requested matches of a multi rule or rule have been matched
    (after filtering of matches).
event MatchedAfterFilteringHandler OnMatchedAfter;
```

The events follow a bracketing structure, the opening outer events seen so far are fired once per (multi) rule application construct, the inner events following are fired for each match.

```
// Fired when a match was selected for execution (after filtering/selection of
    matches).
event MatchSelectedHandler OnMatchSelected;

// Fired before the selected match is rewritten (comparable to the old OnFinishing
    and OnRewritingNextMatch, now one event, fired per selected match).
event RewriteSelectedMatchHandler OnRewritingSelectedMatch;

// Fired after the selected match was rewritten (but before embedded sequences are
    executed).
event SelectedMatchRewrittenHandler OnSelectedMatchRewritten;

// Fired after the selected match was rewritten and embedded sequences executed.
event FinishedSelectedMatchHandler OnFinishedSelectedMatch;
```

The inner events just listed are followed by the following closing part of the outer events.

```
// Fired after the rewrite step of a rule/after rule execution has completed.
event FinishedHandler OnFinished;

// Fired when execution of a pattern matching construct ends.
event EndExecutionHandler OnEndExecution;
```

The subrule based events that are declared by the `ISubactionAndOutputAdditionEnvironment` extending the `IActionExecutionEnvironment` are:

```
// Fired when a debug entity is entered.
event DebugEnterHandler OnDebugEnter;

// Fired when a debug entity is left.
event DebugExitHandler OnDebugExit;

// Fired when a debug emit is executed.
event DebugEmitHandler OnDebugEmit;

// Fired when a debug halt is executed.
event DebugHaltHandler OnDebugHalt;

// Fired when a debug highlight is executed.
event DebugHighlightHandler OnDebugHighlight;
```

A special kind of sequence based events are the graph change events that are fired when processing enters a subgraph or leaves a subgraph (fired when the subgraph usage stack is altered):

```
// Fired when graph processing (rule and sequence execution) is switched to a
    (sub)graph.
```

```
// (Not fired when the main graph is replaced by another main graph, or initialized.)
event SwitchToSubgraphHandler OnSwitchingToSubgraph;

// Fired when graph processing is returning back after a switch.
// (To the main graph, or a subgraph previously switched to.)
event ReturnFromSubgraphHandler OnReturnedFromSubgraph;
```

The sequence based events that are declared by the `IGraphProcessingEnvironment` extending the `ISubactionAndOutputAdditionEnvironment` are:

```
// Fired when a sequence is entered.
event EnterSequenceHandler OnEnteringSequence;

// Fired when a sequence is left.
event ExitSequenceHandler OnExitingSequence;

// Fired when a sequence iteration is ended.
event EndOfIterationHandler OnEndOfIteration;
```

Another special kind of sequence based events that are declared by the `IGraphProcessing-Environment` are the spawn and join events that are fired when a parallel sequence is spawned or joined:

```
// Fired when sequences have been spawned (split off) from their parent thread.
event SpawnSequencesHandler OnSpawnSequences;

// Fired when sequences have joined (again) their parent thread.
event JoinSequencesHandler OnJoinSequences;
```

CHAPTER 34

EXTENSIONS

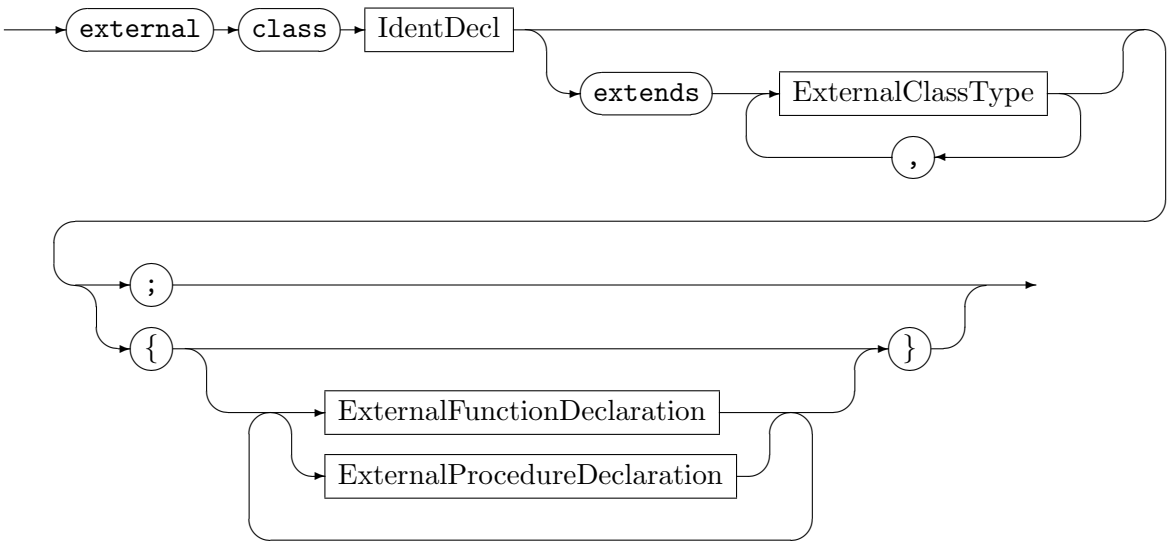
This chapter explains how to customize GrGen.NET with external code, or how to tightly integrate GrGen.NET with external code. It lists the ways you can interact with the external world outside of GrGen.NET. The primary means available are: external attribute types and their methods, external functions and procedures, external match filters, and external sequences; the secondary helpers available are annotations, command line parameters, and external shell commands.

You typically want to use them for integrating functionality outside of graph representation processing, i.e. subtasks GrGen.NET was *not* built for. You may want to use them for performance reasons, to implement linked lists or trees more efficiently than with the nodes and edges supplied by GrGen.NET{ – but regarding this, you can use internal object classes18.1 by now.

The languages of GrGen.NET shield you from the details of the runtime and the framework below it; you can combine their constructs freely. But when you plug into the existing framework with your code, you must obey the expectations of that framework. When you extend GrGen.NET, you must play according to the rules of the GrGen.NET-components. That said, there are plenty of possibilities to extend the framework with your own functionality.

34.1 External Attribute Types

ExternalClassDeclaration



Registers a new attribute type with GrGen.NET. You may declare the base types of the type, and you may specify external function methods or procedure methods. Attributes cannot be specified. The attribute type (and the declared methods) must be implemented

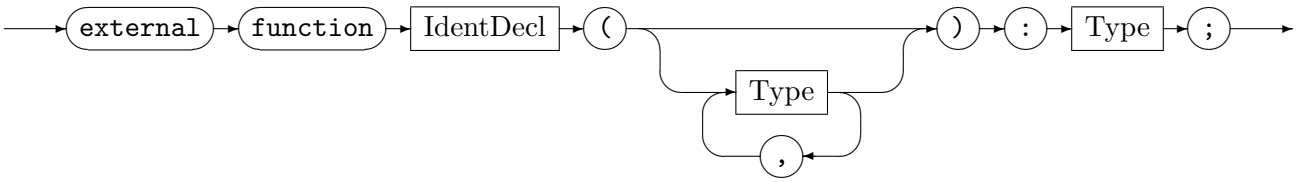
externally, see 33.8. For GrGen.NET the type is opaque, only the function/procedure methods or external functions/procedures can carry out computations on it.

You may extend GrGen.NET with external attribute types if the built-in attribute types (cf. 6.1) are insufficient for your needs because you need functionality not related to graph-rewriting (in order to just implement lists or trees more efficiently than with the node and edge types you may use the internal (object) class type instead). The external types can be explicitly cast to `object` but are not implicitly cast. (The methods may register own undo items with the transaction manager to realize rollback behaviour for external attributes (cf. 183)).

The external functions and procedures declared within the body of the external class are *methods*, in contrast to the external functions and procedures from the global namespace that use the same declaration syntax that are introduced in the following sections.

34.2 External Function Types

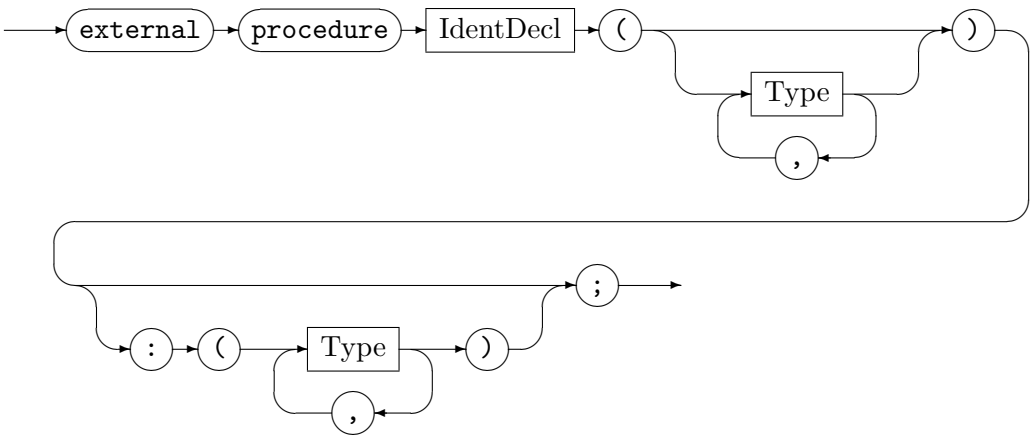
ExternalFunctionDeclaration



Registers an external function with GrGen.NET to be used in attribute computations. An external function declaration specifies the expected input types and the delivered output type. The function must be implemented externally, see 33.8. An external function call (cf. 6.8) may receive and return values of the built-in (attribute) types, as well as of the external attribute types. The real arguments on the call sites are type-checked against the declared signature following the subtyping hierarchy of the built-in – as well as of the external – attribute types. You may extend GrGen.NET with external functions if the built-in attribute computation capabilities (cf. 6.2) or graph querying capabilities are insufficient for your needs.

34.3 External Procedure Types

ExternalProcedureDeclaration

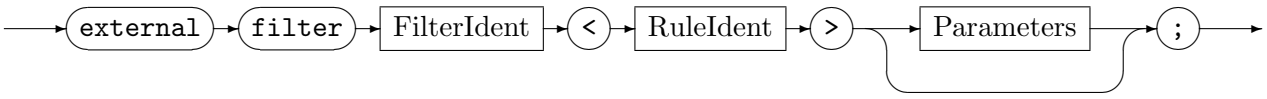


Registers an external procedure with GrGen.NET to be used in attribute computations. An external procedure declaration specifies the expected input types and the delivered output types. The procedure must be implemented externally, see 33.8. An external procedure call (cf. 6.8) may receive and return values of the built-in (attribute) types as well as of the external attribute types. The real arguments on the call sites are type-checked against

the declared signature following the subtyping hierarchy of the built-in – as well as of the external – attribute types. You may extend GrGen.NET with external procedures if the built-in attribute computation capabilities (cf. 6.2) or graph manipulation capabilities are insufficient for your needs.

34.4 External Filter Functions

FilterFunctionDefinition



The *FilterFunctionDefinition* prepended with an `external` and deprived of its body (cf. 17 for more on filters) registers an external filter function with GrGen.NET, for the rule (or test) specified in angles. It can then be used from applications of that rule (cf. *FilterCalls* in 11.1). A filter function name – irrespective whether external or internal – must be globally unique (in contrast to the predefined filters — each rule may offer one with the same name). The match filter function must be implemented externally, see 33.9. You may extend GrGen.NET with match filters if you need to inspect the matches found for a rule, in order to decide which one to apply (see note 69), or if you just need a post-match hook which informs you about the found matches.

Match object

For a rule or subpattern `r`, a match interface `IMatch_r` is generated, extending the generic `IMatch` interface from `libGr`. The basic constituents, i.e. nodes, edges and variables are mapped directly to members of their name and type, containing the graph element matched or the value computed.

For alternatives nested inside the pattern, a common base interface `IMatch_r_altName` is generated, plus for each alternative case an interface `IMatch_r_altName_altCaseName`. When you walk the matches tree using the type exact interface (which is recommended), you must type switch on the match object in the alternative variable, to determine the case which was finally matched, and cast to its match type.

For iterated patterns nested inside the pattern, an iterated variable of type `IMatchesExact<IMatch_r_iteratedName>` is created in the pattern match object; the `IMatchesExact` allows you to iterate over the patterns finally found.

A subpattern usage is mapped directly to a variable in the match object typed with the match interface of the subpattern.

An independent pattern is mapped directly to a variable in the match object typed with the match interface of the independent pattern.

Negative patterns do not appear in the match objects, for they prevent the matching and thus the building of a match object in the first place.

34.5 External Sequences

ExternalSequenceDeclaration

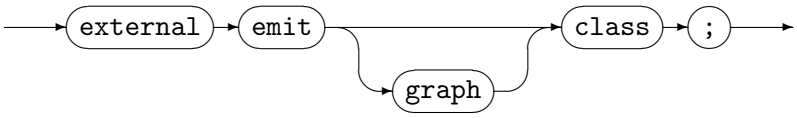


Registers an external sequence similar to a defined sequence (cf. 22.1), but in contrast to that one, it must or can be implemented outside in C# code. An external sequence declaration specifies the expected input types and the delivered output types. The sequence must be

implemented externally, see 33.9. You may extend GrGen.NET with external sequences if you want to call into external code, in order to interface with libraries outside of the domain of graph rewriting, or if the GrGen.NET-languages are not well suited for parts of the task at hand.

34.6 External Emitting and Parsing

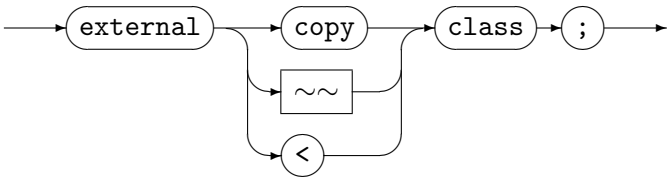
EmitParseDeclaration



Tells GrGen.NET that GRS importing and exporting is to be extended, as well as emitting and debugger display, so that these can handle external types, or the type `object`. This allows for a tight integration of external types with the built-in functionality. It allows inspecting types not defined in GrGen.NET in the debugger (incl. yComp), and it allows serializing/deserializing graphs with types not defined in GrGen.NET. See 33.4 for an explanation on what needs to be implemented in this case.

34.7 External Cloning and Comparison

CopyCompareDeclaration

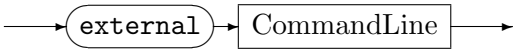


Tells GrGen.NET that graph element copying or attribute type comparisons are to be extended, so that they can handle external types, or the type `object`. This allows for a tight integration of external types with the built-in functionality, and esp. allows implementing value semantics (objects are normally compared by checking reference identity and cloned by copying the reference). See 33.4 for an explanation on what needs to be implemented in this case.

34.8 Shell Commands



CommandLine is executed as-is by the shell of the operating system.



A method in an extension file is called with the *CommandLine* as string parameter. This is only possible after `external emit class` was specified, see 34.6 above. See 33.4 for an explanation of the method named `External` that needs to be implemented in this case.

34.9 Shell and Compiler Parameters

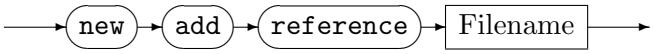
When you want to include external code, you likely have to reference external assemblies. You can do so by compiler parameters, that are also available as configuration options in the shell. Debug symbols and source code availability are also of increased importance in this case.

When executing the compiler `GrGen.exe`, amongst others, the following parameters are admissible:

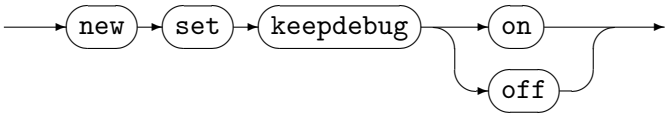
```
[mono] GrGen.exe  [-keep [<dest-dir>]] [-debug] [-r <assembly-path>]
```

The assembly *assembly-path* is linked as reference to the compilation result with the `-r` parameter. The `-keep` parameter causes the generated C# source files to be kept. If *dest-dir* is omitted, a subdirectory `tmpgrgen`*n*[1] within the current directory will be created. When `-debug` is supplied, the assemblies are compiled with debug information (also some validity checking code is added).

These compiler parameters can be configured in the GrShell, too:



Configures a reference to an external assembly *Filename* to be linked into the generated assemblies, maps to the `-r` option of `grgen.exe` (cf. 29.1).

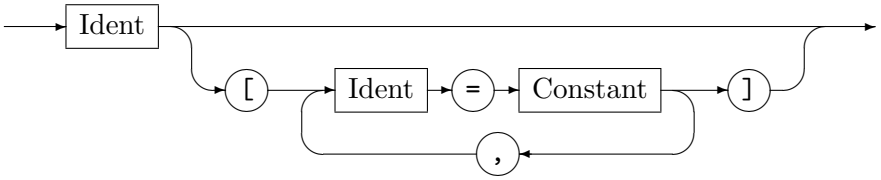


Configures the compilation of the generated assemblies to keep the generated files and to add debug symbols. Maps to the `-keep` and the `-debug` options of `grgen.exe`.

34.10 Annotations

Identifier definitions can be annotated by pragmas. Annotations are key-value pairs.

IdentDecl



You can use any key-value pairs between the brackets. For GrGen.NET only the identifiers prio, maybeDeleted, containment, parallelize, and validityCheck have an effect, cf. Table 34.1. But you may use the annotations to transmit information from the specification files to API level where they can be queried.

[1] *n* is an increasing number.

Key	Value Type	Applies to	Meaning
`prio`	int	node, edge	Changes the ranking of a graph element for search plans. The default is `prio`=1000. Graph elements with high values are likely to appear prior to graph elements with low values in search plans.
`maybeDeleted`	boolean	node, edge	Prevents a compiler error when a graph element gets accessed that may be matched homomorphically with a graph element that gets deleted.
`containment`	boolean	edge type	Used for XMI export; typically defined by the ecore(/XMI) import. Declares an edge type to be an edge type defining a containment relation, which causes XML element containment in the exported XMI. Default is `containment`=false.
`parallelize`	int	rule, test	Causes parallelization of the pattern matcher of the action; the value specifies the number of threads to use.
`validityCheck`	boolean	node, edge	In case of false, skips the contained-in-graph checks for that node/edge, which are executed in case the debug compiler flag (or shell option) is supplied, before a match is rewritten, esp. in between matches of an all-bracketed rule execution.
		rule	In case of false, skips the contained-in-graph checks for all nodes/edges in the pattern(s) of the rule.

Table 34.1: Annotations

EXAMPLE (185)

We search the pattern `v:NodeTypeA -e:EdgeType-> w:NodeTypeB`. We have a host graph with about 100 nodes of `NodeTypeA`, 1,000 nodes of `NodeTypeB` and 10,000 edges of `EdgeType`. Furthermore, we know that between each pair of `NodeTypeA` and `NodeTypeB` there exists at most one edge of `EdgeType`. GrGen.NET can use this information to improve the initial search plan if we adjust the pattern with `v[prio=10000]:NodeTypeA -e[prio=5000]:EdgeType-> w:NodeTypeB`.

CHAPTER 35

UNDERSTANDING AND EXTENDING GRGEN.NET

This chapter describes the inner workings of GrGen.NET to allow you

- to understand how GrGen.NET works
- esp. in order to extend it with new features [1]

It starts with a section that describes how to build GrGen.NET, followed by an introduction into the generated code, then an introduction into the mechanism of search planning, and ends with a section giving some details of the structure of, and the data flow in the GrGen.NET-code generator. Here we repeat some content from chapter 31, refining it with more details.

35.1 How to Build

In case you want to build GrGen.NET on your own (which is available from the public git repository `https://github.com/ejaku/grgen`), you should recall the system layout 2.1. The graph rewrite generator consists of a frontend and a backend written in C# [HWG03] (the `engine-net-2`). The frontend orginally written in Java [GJSB05] was extended and changed since its first version, but only recently replaced by a C#-port, which is somewhat of an one-to-one mapping of the Java version. In contrast to the backend, which has seen multiple engines and versions: a MySQL database based version, a PSQL database based version, a C version executing a search plan with a virtual machine, a C# engine executing code generated from a search plan and finally the current C# engine version 2 featuring attributes stored in the graph elements instead of in an extra object, and being capable of matching nested and subpatterns (the SQLite persistence provider is pretty different, it only stores the graphs persistently, the database is *not* used in pattern matching for performance reasons).

The frontend is available in the `frontend` subdirectory. It contains a VisualStudio 2022 solution file `Frontend.sln`, that results in a `FrontendGrGen.exe` when built.

Before you can build the solution, you have to execute `./genparser.bat` under Windows, or you have to execute `./genparser.sh` under Linux in order to get the ANTLR parsers for the model and actions languages, and the embedded/compiled sequences generated.

The backend is available in the `engine-net-2` subdirectory. It contains a VisualStudio 2022 solution file `EngineNet2.sln` (if you want to use VisualStudio 2017, you have to remove the comments around the binding redirect of `System.Resources.Extensions` in the `app.config`s of the application projects), containing projects for the `libConsoleAndOS` environment abstraction library, the `libGr`, the `lgspBackend` (libGr-Search-Plan-Backend), the `graphViewerAndSequenceDebugger` library, the `libGrShell`, the `GrGen` compiler (backend) as well as the `GrShell`.

[1] which may be special extensions for a dedicated task, but also may be of general interest to other users; note that GrGen.NET is licensed under LGPL

Plus the WindowsForms versions/extensions `libConsoleAndOSWindowsForms`, the `graphViewerAndSequenceDebuggerWindowsForms` library employing MSAGL, as well as the WindowsForms `GGrShell` application and the graph viewer application `extMSAGLExt`. Also available is a `libGrPersistenceProviderSQLite` (implementing the persistent graph by mapping to an SQLite database — in contrast to the previous database-based backends it is a strictly optional extension and only read once at session start, i.e. it is esp. not used for runtime pattern matching).

Before you can build the solution, you have to execute `./genparsers.bat` under Windows (which will call `./src/libGr/genparser.bat`, `./src/graphViewerAndSequenceDebugger/genparser.bat`, `./src/libGrShell/genparser.bat` and `./src/extMSAGLExt/genparser.bat`), or you have to execute `./genparsers.sh` under Linux (which will call `./src/libGr/genparser.sh`, `./src/graphViewerAndSequenceDebugger/genparser.sh`, `./src/libGrShell/genparser.sh` and `./src/extMSAGLExt/genparser.sh`) in order to get the CSharpCC parsers for the rewrite sequences, the constant literals, and the shell generated, as well as the ANTLR parser for the VCG importer of the extMSAGLExt graph viewer.

Under Windows you may use `make_cygwin_windows.sh` to get a complete build (regenerating the parsers each time, which takes long for the sequences parser). But note that it employs `dotnet` for building, while regular releases are built manually with VisualStudio, i.e. `msbuild`. To get the API examples generated you must execute `./genlibs.bat`.

Under LINUX you may use `make_linux.sh` to get a complete build. It was able to carry out a complete build with `mdtool` from MonoDevelop (the old way), but this does not work anymore since the switch to SDK-style projects (after version 8.0 of GrGen.NET) — now `dotnet build` is used. The new way of development under LINUX is to employ VisualStudio Code with the C#-extensions and the Mono Debug extension; the `dotnet` tool is used then also. Note that for running/debugging, you still have to use `mono` (GrGen.NET still targets the .NET framework, largely because of the WindowsForms GUI (a few adaptations yet missing would be needed regarding the compilation of the generated code so that the system core could also work with .NET standard 2.0)).

On both systems, you may use `GrGen.sln` to build the complete code (frontend plus engine-net-2), but without parser generation.

The VisualStudio solutions contain build configurations for `x64`, `x86`, and `Any CPU`. The binary release contains the binaries from the `x64` build only. If you want to run GrGen.NET in 32bit mode, or esp. want to include it in a native 32bit application, you have to build the assemblies on your own. Before version 8.0, the `Any CPU` build was released, this changed because of the introduction of the persistent graph. The SQLite used by the persistent graph cannot be built as `Any CPU`, most likely because the native binaries for the matching platform have to be fetched (the rest of the project can be built as `Any CPU`, you can still use it, only the persistent graph may be unavailable). The corresponding nuget-package contains builds for quite some platforms, so it should be not much of an issue to get e.g. an ARM-version of including the persistent graph running.

The `doc` subdirectory contains the sources of the user manual, for building on Windows enter `./build_cygwin.sh grgen` in `cygwin-bash` or on Linux `./build grgen` in `bash`. The `engine-net-2` subdirectory contains a `GrGen-API.shfbproj` file that is used by the Sandcastle Help File Builder to build the API documentation. The `syntaxhighlighting` subdirectory contains syntax highlighting specifications for the `.gm`,`.grg`, and `.grs`-files for `Notepad++`, `vim`, and `Emacs` (the `N++` versions are typically more up-to-date/complete than the others).

35.2 How to Test

You can check the result of your build with the test suite we use to check against regressions. It is split into syntactic (/frontend) tests checking language acceptance by the compiler and semantic (/backend) tests checking the execution semantics of the languages.

The syntactic tests reside in `frontend/test` (the `test` folder in the binary release), they are checking that example model and rule files can get compiled by `grgen` (or not compiled, or only compiled emitting warnings) and that the resulting code can get compiled by `csc`. The tests get executed by calling `./test.sh` from `bash` or `cygwin-bash`; deviations from a gold standard are reported (at the time of writing, 3 mathematics tests are failing under mono/LINUX because of minor numeric differences).

The semantic tests reside in `engine-net-2/tests` (the `tests` folder in the binary release), they are checking that executing example shell scripts on example models and rules yields the expected results (utilizing the outputs of the commands described in Chapter 28). They get executed by calling `./test.sh`.

The examples from `engine-net-2/examples` can be used as smoke tests by the contained `./test.sh` (under Windows).

For a very thorough test coverage, you could manually force parallelization of all pattern matchers in the frontend code (search for `uncomment` in `Unit.cs`, or you could manually force profiling instrumentation of all pattern matchers (search for `uncomment` in `GrGen.cs`).

A unit test suite was recently added to the `engine-net-2` folder (in the `unittests` subfolder) and to the solution file, you may run it directly in VisualStudio 2022 or use `dotnet test`. It is based on NUnit3 [2] and very limited in what it covers as of now.

A unit test suite was also recently added to the `frontend` folder (in the `unittest` subfolder), you may run it directly in VisualStudio 2022 or use `dotnet test`. It is based on NUnit3 and comprises a small set of quick-check acceptance tests as of now (so also limited coverage).

The application tests checking that the languages work as expected (on the outside) had priority over checking internal components — but the unit tests are to be extended, as they allow finding errors much earlier than the syntax/semantics test suites that take a long time to execute.

35.3 The Generated Code

In this section we'll have a look at what is generated by GrGen.NET out of your specifications: firstly the implementation of the model, secondly the implementation of the rules, better their pattern matchers, ending thirdly with an explanation of the matching of nested and subpatterns.

Internal Graph Representation

The graph structure is maintained in an `LGSPGraph` built of `LGSPNode` and `LGSPEdge` objects, basically without type and attribute information. The types defined in the model are realized by generated node and edge interfaces and generated node and edge classes. The interfaces define the user visible types and attributes, the classes inherit from and work like `LGSPNode`s and `LGSPEdge`s in the graph, additionally they implement the type and attribute bearing interfaces. The nodes and edges are contained in a system of ringlists.

At the top level you find type ringlists, every node or edge is contained in a linked list of its specific type. Each list contains a dummy head node or head edge that serves as entry point, these are accessible from an array in the graph, storing as many heads as there are types. Every node or edge contains a field `typeNext` giving the next element of the type. These

[2] you may need NUnit2 when you want to execute your own tests from a native application like AutoCAD hosting your graph rewrite system — but beware of outdated knowledge

lists allow to quickly look up all elements from the graph bearing a certain type. Figure 35.1 gives an example for a graph with 3 node types, one having no instance nodes, one having only one instance node, and one having 5 instance nodes; the same holds for the edge types.

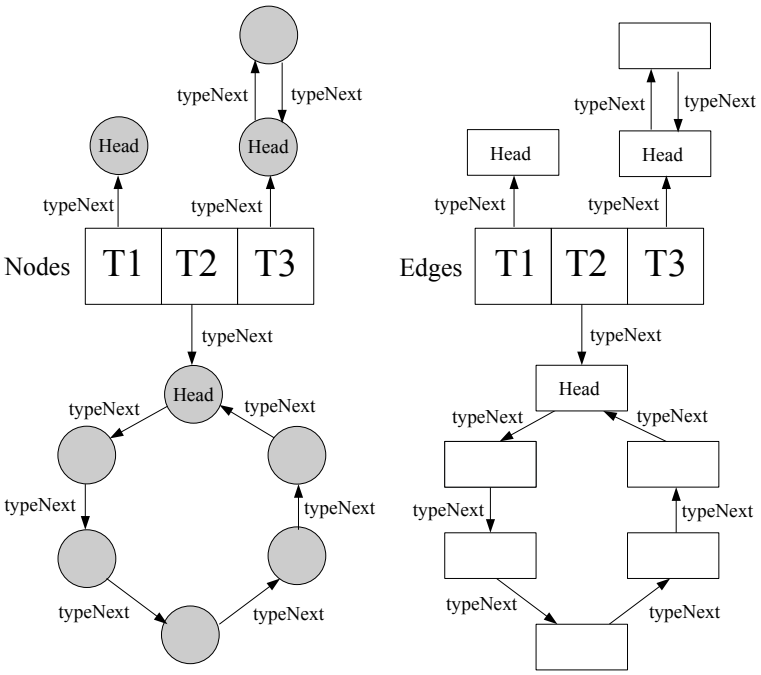


Figure 35.1: Example for type ringlists

The connectedness information is also stored in ringlists, two ringlists per node object, one containing the incoming edges and one containing the outgoing edges. The node object contains a field `inHead` referencing an arbitrary edge object of the incoming edges (or `null` if there is none) and a field `outHead` referencing an arbitrary edge object of the outgoing edges (or `null` if there is none). These lists allow to quickly retrieve all incoming or all outgoing edges of a node.

The edge objects contain fields `source` and `target` referencing the source and the target node. They give instant access to the source and target nodes of an edge. Furthermore, edges contain a field `inNext` referencing the next edge in the incoming ringlist they are contained in, and a field `outNext` referencing the next edge in the outgoing ringlist they are contained in.

All the 3 ringlists (type, in, out) are doubly linked to allow for fast insertion and deletion. For every `next` field there is also a `prev` field available that references the previous element in the ringlist (`typePrev`, `inPrev`, `outPrev`). The figure 35.3 gives the ringlist implementation of the example graph depicted in figure 35.2.

In case attribute indices were declared, a balanced search tree (an AA-tree) is created for each of them. The tree is maintained by event handlers that listen to graph element creation and deletion, but esp. for attribute assignment.

Naming is implemented by the `LGSPNamedGraph`, the uniqueness constraints/the index by the `LGSPUniquenessEnsurer`/`LGSPUniquenessIndex`. The `LGSPGraphStatistics` class stores the graph analyzation results (element count and vstruct statistics).

Pattern Matching and Search Programs

The pattern of an action (rule or test) is matched with a backtracking algorithm that is binding one pattern element after another to a graph element, while checking if it fits to the already bound parts. If it does fit, search continues trying to bind the next pattern element, or succeeds in case the last check succeeded, building the match object from all the elements

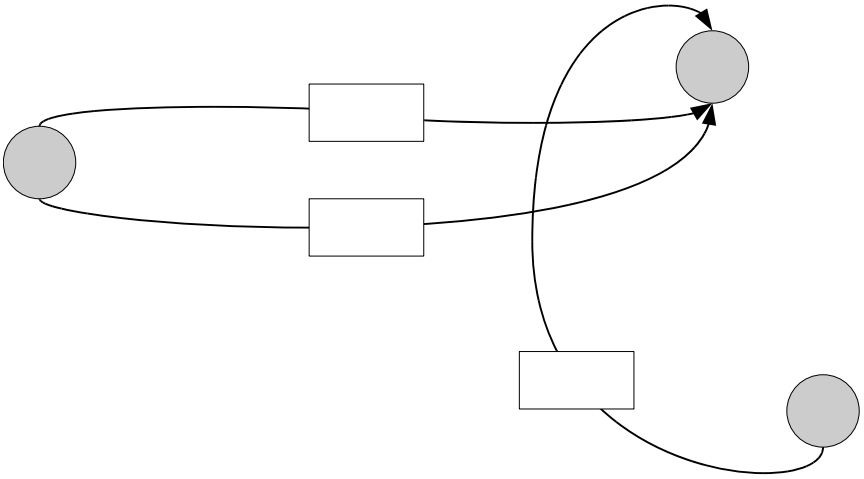


Figure 35.2: Incidence example situation

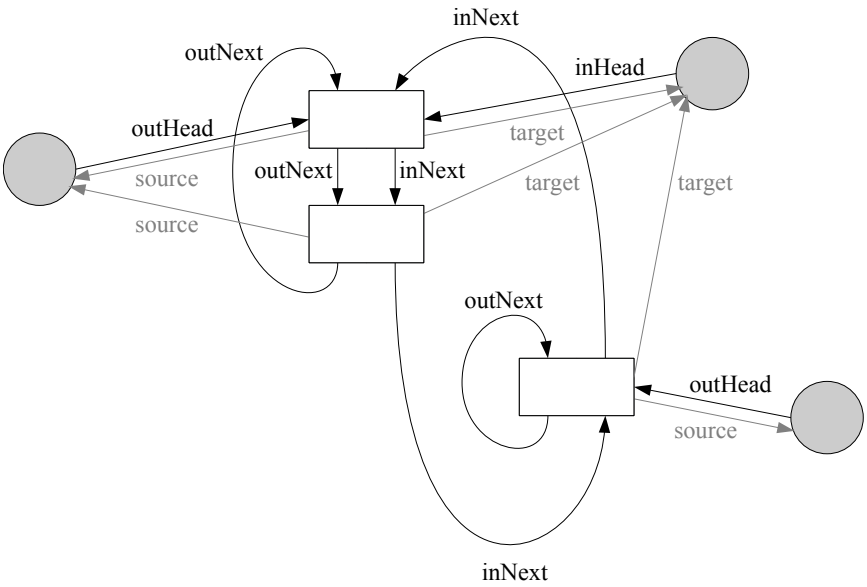


Figure 35.3: Ringlist implementation of incidence example

bound. If it does not fit, search continues with the next graph element candidate for the pattern element. If all graph element candidates for this pattern element are exhausted, search backtracks to the previous decision point and continues there with the next graph element candidate, or fails in case the first check failed.

For every pattern that is to be matched, a search program implementing this algorithm is generated, basically consisting of nested loops iterating the available graph element candidates for each pattern element, and condition checking code that continues search with the next graph element candidate in case of a failing check. Figure 35.4 shows a pattern and a search program generated for it.

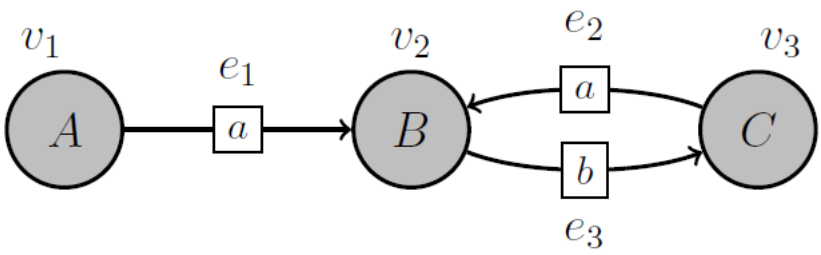


Figure 35.4: Pattern to search

```
1  foreach(v1:A in graph) {
2     foreach(e1 in outgoing(v1)) {
3        if(type(e1)!=a) continue;
4        v2 = e1.tgt;
5        if(type(v2)!=B) continue;
6        foreach(e3 in outgoing(v2)) {
7           if(type(e3)!=b) continue;
8           v3 = e3.tgt;
9           if(type(v3)!=C) continue;
10          foreach(e2 in outgoing(v3)) {
11             if(type(e2)!=a) continue;
12             if(e2.tgt!=v2) continue;
13             // v1,e1,v2,e3,v3,v2 constitute a match
14          }
15       }
16    }
17 }
```

Search Programs Refined And Rewriting

If a non-leaf-type (regarding the inheritance hierarchy) is to be matched with a graph *lookup* (the outermost loop in the example, defining the root element for pattern matching), then an additional loop is used that is iterating all the subtypes of the type of the pattern element. After an initial lookup, search normally follows the pattern structure (matching it against graph structure), by iterating the *incoming* or *outgoing* edges of a node, or determining the *source* or *target* node of an edge. If a pattern is given one or more parameters, search normally continues from the parameters on, picking those as roots, instead of looking up by type in the graph. If a pattern consists of several unconnected components, several lookups are needed.

Undirected or arbitrarily directed pattern edges are searched in both ringlists, the ringlist of the incoming and the ringlist of the outgoing edges (undirected edge types are otherwise implemented like directed edge types). Other matching operations besides graph lookup and incoming/outgoing or source/target following are storage access, storage attribute access and storage mapping as well as attribute index access and name/unique index access. The condition checking code may be targeted at different constraints: a graph element candidate may get rejected because of its type, missing structural connections to already bound elements, or because the graph element is already matched to another pattern element (isomorphy checking). Furthermore, attribute conditions may have to be checked, negative patterns for non-matches, and independent patterns for matches (these conditions are normally depending on multiple elements).

Compared with pattern matching, a search task that may need a long time, pattern rewriting is a simple task that executes quickly. It consists of a sequence of operations, most notably: i) new nodes creation, then new edge creation, ii) attribute evaluation and assignment, iii) edge deletion, then node deletion, iv) embedded sequences execution (see table 10.2 for more on this).

A notable performance optimization allowed by the graph model is search state space stepping: after a pattern was matched, the list heads of the type ringlists, the incoming ringlists and the outgoing ringlists are moved to the position of the matched entries with `MoveHeadAfter`, `MoveInHeadAfter` and `MoveOutHeadAfter`. With this optimization, the pattern matching during an iteration `r*` will start where the previous iteration step left, saving the cost of iterating again through all the elements that failed in the previous iteration.

Pushdown Machine for Nested and Subpatterns

Every subpattern (as introduced in chapter 8) is matched with a search program that is generated from its pattern, in the way introduced in the previous section. The interesting part is how the subpatterns get combined, this is carried out with a $2+n$ pushdown machine.

It consists of a call stack, containing the subpattern instances found, an open tasks stack containing the subpatterns to match, and n result stacks containing the (partial) match object trees. The subpattern instances on the call stack consist of the currently bound elements in the local variables of the stack frame of the search program. A task denotes the pattern to match, and the parameters where to start; when a subpattern was matched, its contained subpatterns are pushed to the open tasks stack, then the top of the stack gets processed. For a normal rule application holds $n = 1$, while the number of matches is unbound for an all-bracketed rule. A simulation of this machine, i.e. the matching process of a pattern using subpatterns is shown on the following pages.

Alternatives are handled like a subpattern with several possible patterns that are tried out, the first pattern that is matching is accepted. Iterateds are handled like subpatterns whose tasks are not removed from the open tasks stack when they get matched, but only if matching failed or the maximum amount of matches was reached. In case of a failure the minimum required amount of matches is inspected, if the amount of found matches is larger

or equal then matching partially succeeds and continues with the next open task (or plain succeeds if there are no open tasks left), otherwise matching of the given partial match fails, causing matching to backtrack (to the previous decision point).

The advantages of this design linearizing the pattern tree on the call stack are the rather low usage of heap memory, the ability to reuse the matcher programs for the patterns as introduced in the previous section, and the ability to find all matches (the n in $2+n$ pushdown machine stands for the number of matches found).

Rewriting is carried out by a depth-first traversal of the match object tree, creating new elements before descending, evaluating attributes and deleting old elements before ascending. For each pattern piece matched, the modification specified in the rewrite part is carried out, i.e. the overall rewrite gets combined from the rewrites of the pattern pieces (cf. table 10.2 for the order of rewriting).

The subpattern usage parameters are computed during matching from matched elements and call expressions (inherited attribution), LHS yielding is carried out after the match was found during match object building with the yield statements (synthesized attribution), RHS yielding is carried out during match object tree rewriting with the eval statements (left attribution, with a user defined left-relation).

In the following, an example run of the $2 + n$ pushdown machine is given:

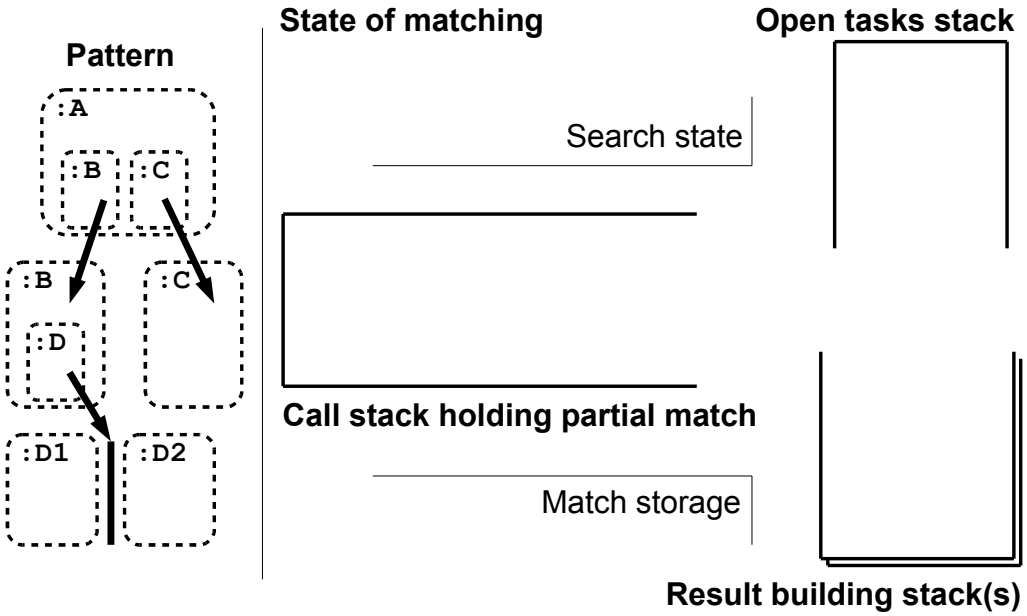


Figure 35.5: 1. Start state

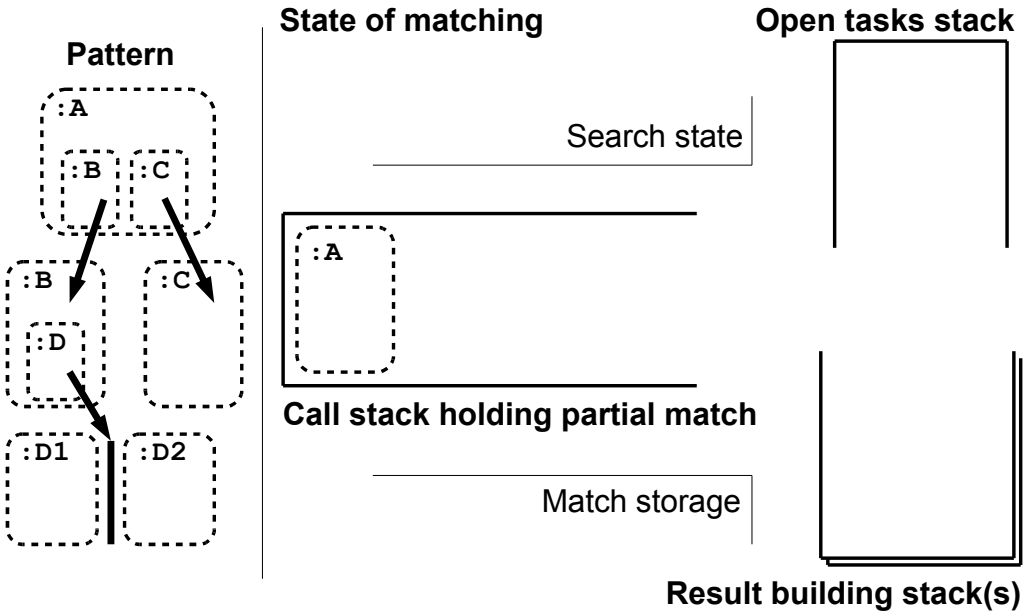


Figure 35.6: 2. The terminal part of pattern A was matched

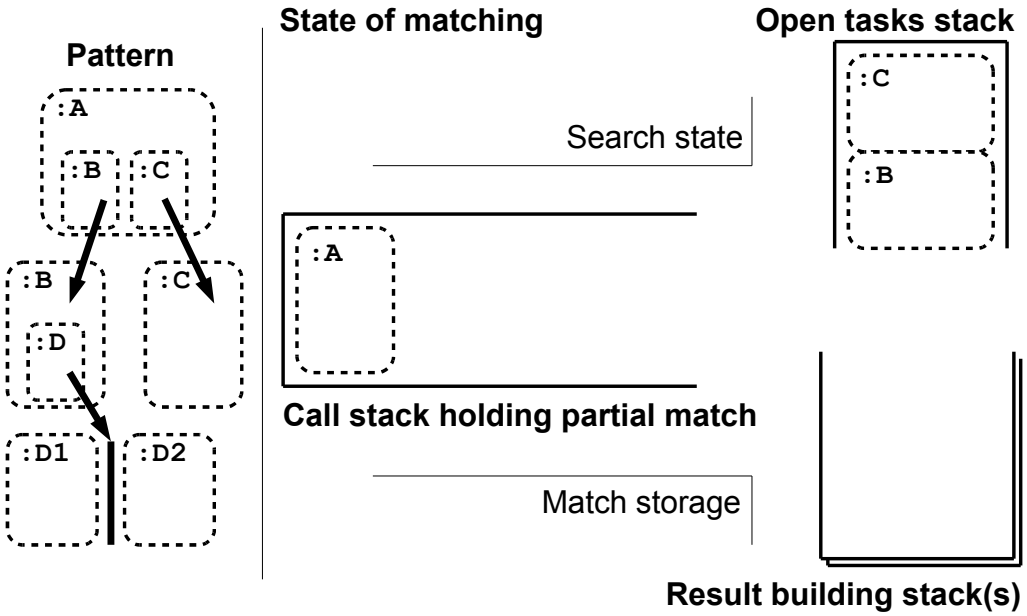


Figure 35.7: 3. The tasks for subpatterns B and C are pushed

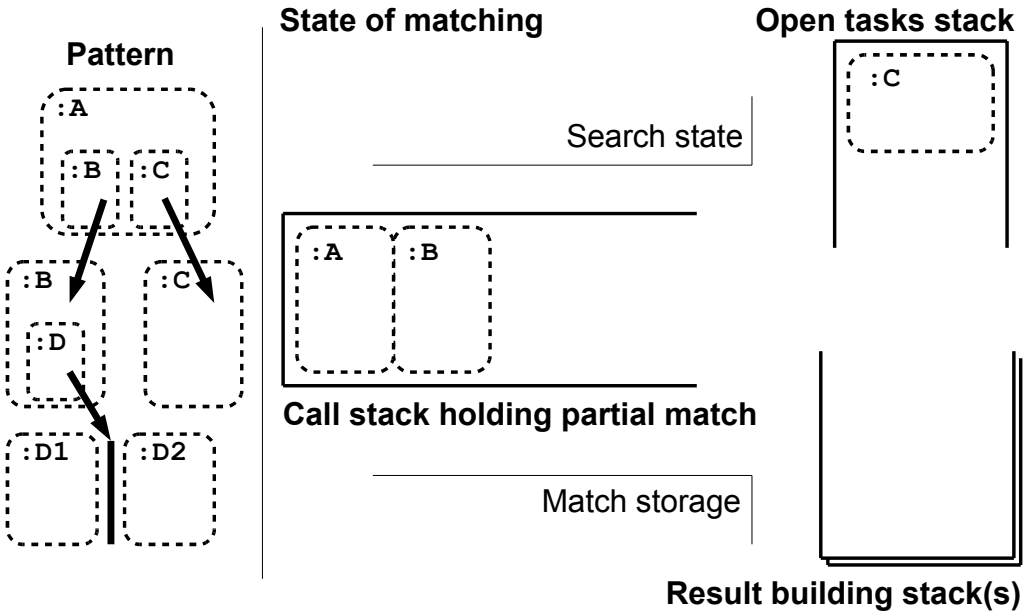


Figure 35.8: 4. The task for B gets executed, the terminal part of B was matched

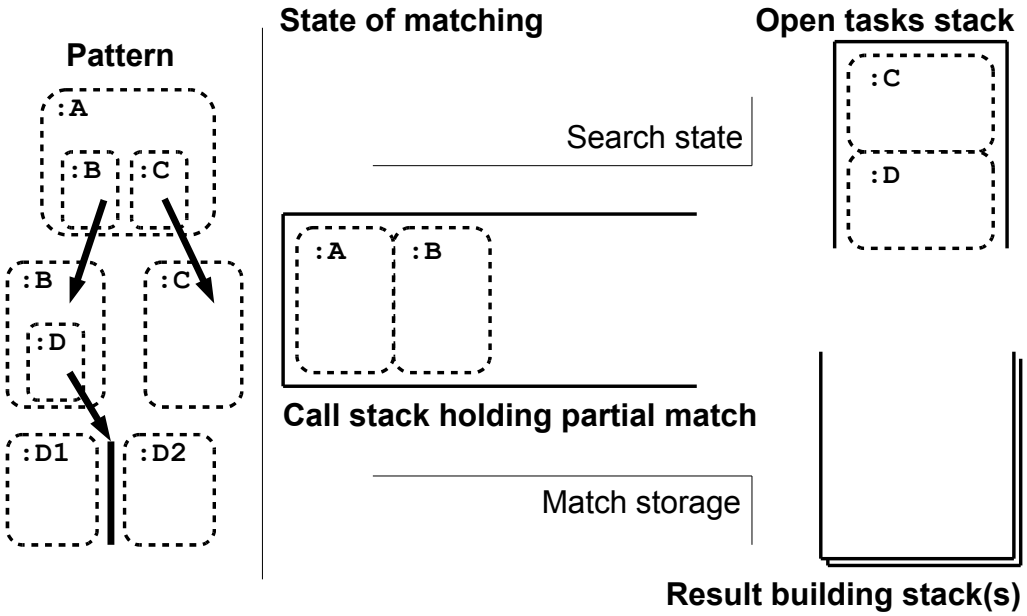


Figure 35.9: 5. The task for alternative D is pushed

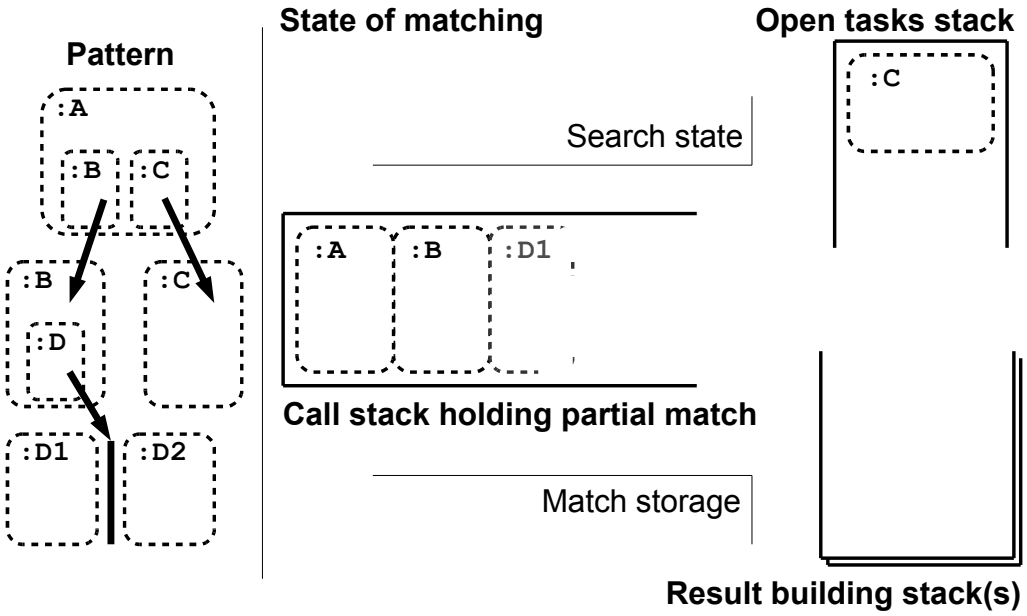


Figure 35.10: 6. The task for D gets executed, D1 is tried, but matching fails

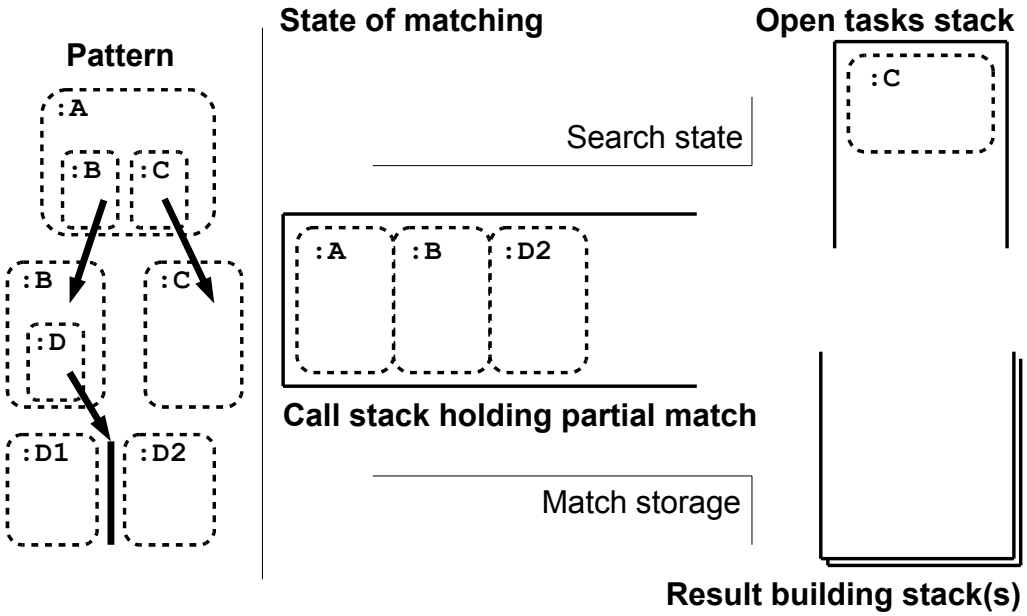


Figure 35.11: 7. The task for D gets executed, D2 was matched

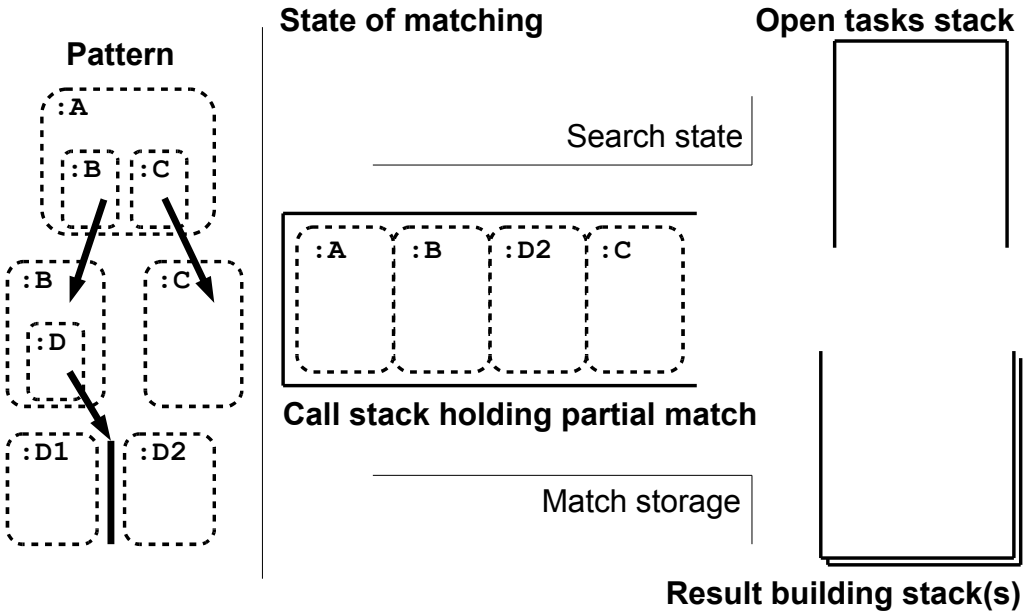


Figure 35.12: 8. The task for C gets executed, C was matched, a match for the overall pattern was found, but is contained only on the call stack

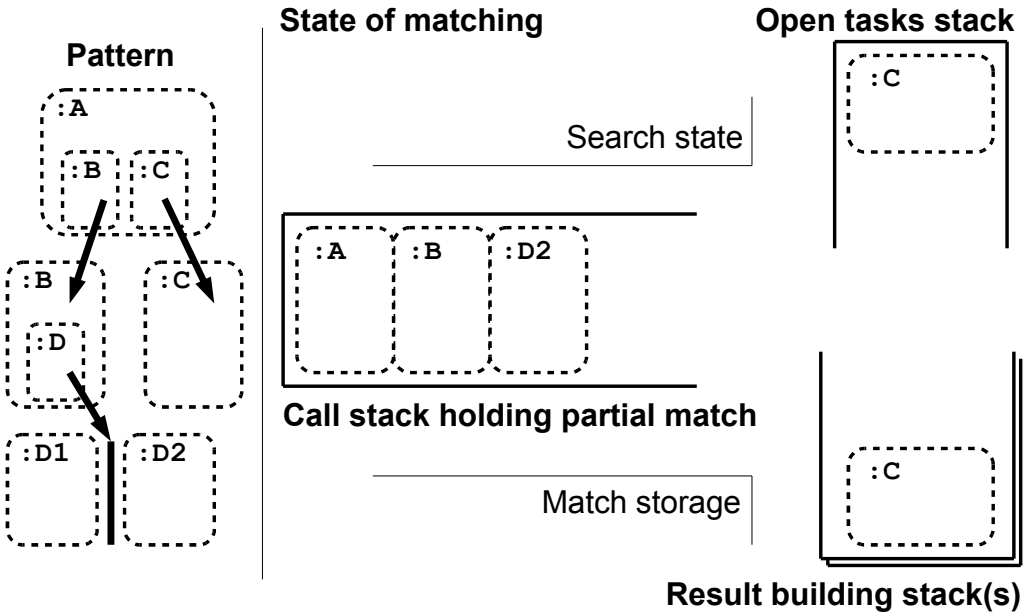


Figure 35.13: 9. The match of C is popped from the call stack and pushed to the result stack

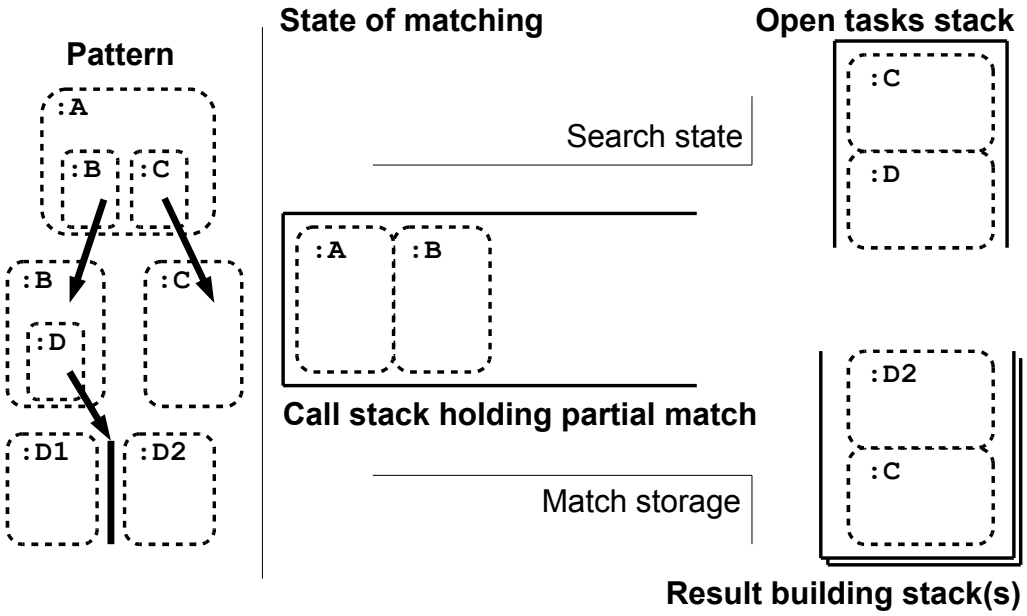


Figure 35.14: 10. The match of D2 is popped from the call stack and pushed to the result stack

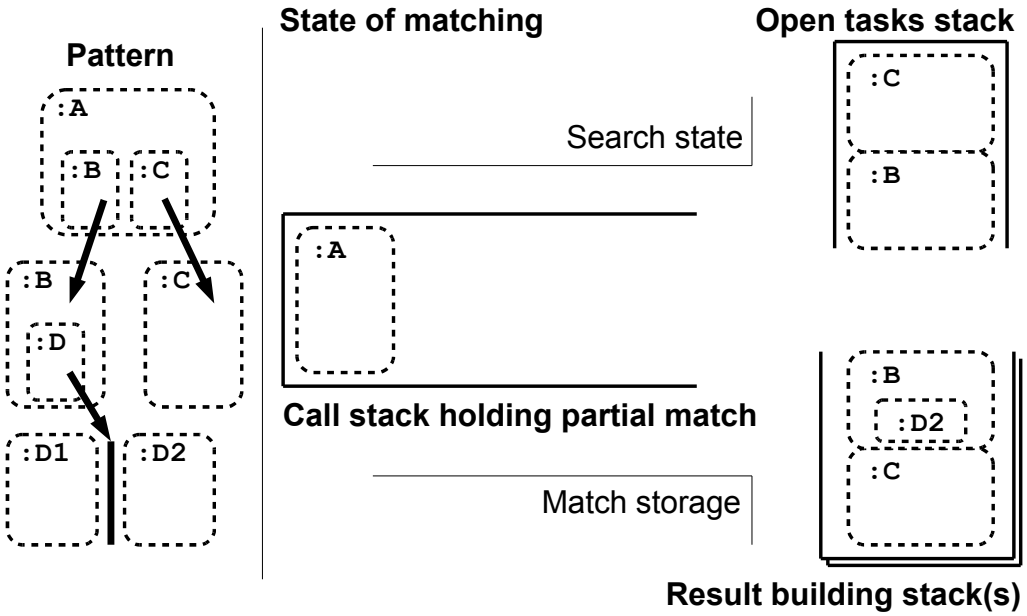


Figure 35.15: 11. The match of B is popped from the call stack, D2 from the result stack is added, the combined match is pushed to the result stack

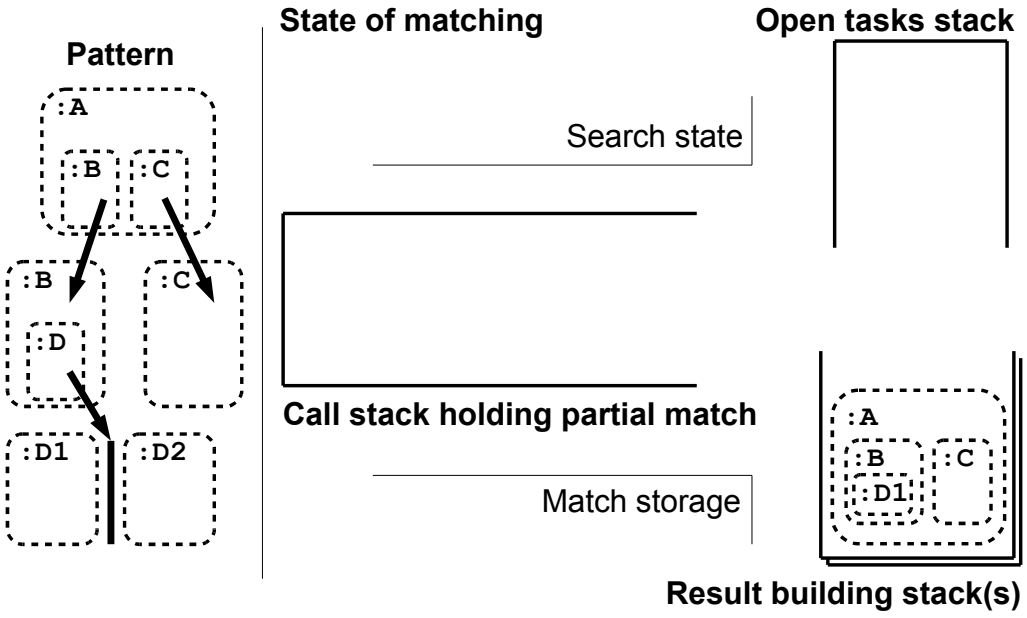


Figure 35.16: 12. The match of A is popped from the call stack, B and C from the result stack are added, now we got the combined match of the overall pattern

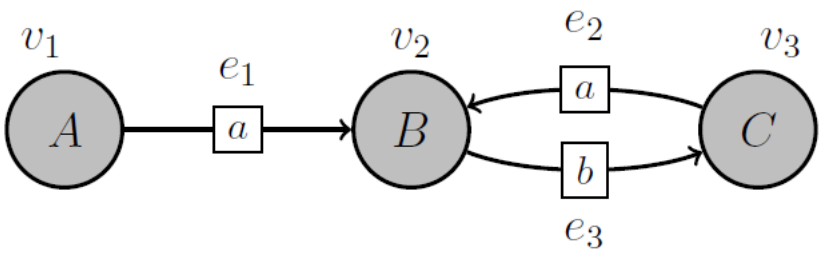


Figure 35.17: Pattern to search

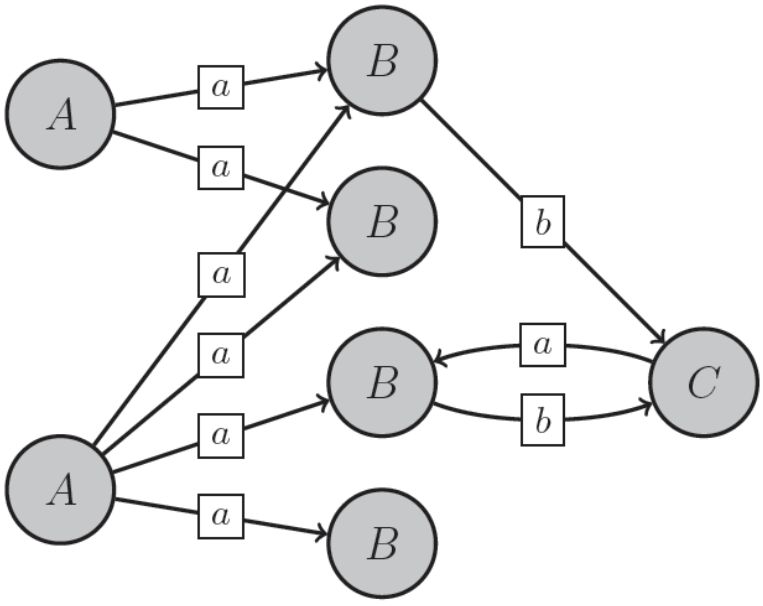


Figure 35.18: Host graph to search in

35.4 Search Planning in Code Generation

In the previous section, a search program for the pattern repeated in figure 35.17 was given. It follows the schedule
`lkp(v1:A); out(v1,e1:a); tgt(e1,v2:B); out(v2,e3:b); tgt(e3,v3:C); out(v3,e2:a)`

A *schedule* is a more abstract version of a search program, consisting of a list of search operations. Available are the search operations `lkp`, denoting node (or edge) lookup in the graph by iterating the type list, `out` iterating the outgoing edges of the given source node and `in` iterating the incoming edges of the given target node, as well as `src` fetching the source node from the given edge and `tgt` fetching the target node from the given edge.

The schedule might work well for some graphs, but for the graph given in figure 35.18 it is a bad schedule. Why so can be seen on inspecting the search order illustration in figure 35.19. Because only one of the multiple outgoing edges of `v1` leads to a match for `e1`, it has to backtrack several times. A better search order would be one matching edge `e1` from `v2` on in reverse direction (this is possible in GrGen.NET because the graph model contains a list of outgoing as well as a list of incoming edges, so each edge can be traversed in either direction).

And indeed, the search planning component of GrGen.NET chooses such a schedule:
`lkp(v3:C); out(v3,e2:a); tgt(e2,v2:B); out(v2,e3:b); in(v2,e1:a); src(e1,v1:A)`

It leads to the search order depicted in figure 35.20, and the search program:

```
1  foreach(v3:C in graph) {
2     foreach(e2 in outgoing(v3)) {
3        if(type(e2)!=a) continue;
4        v2 = e2.tgt;
5        if(type(v2)!=B) continue;
6        foreach(e3 in outgoing(v2)) {
7           if(type(e3)!=b) continue;
8           if(e3.tgt!=v3) continue;
9              foreach(e1 in incoming(v2)) {
10             if(type(e1)!=a) continue;
11             v1 = e1.src;
12             if(type(v1)!=A) continue;
13             buildMatchObjectOfPatternWith(v3,e2,v2,e3,e1,v1);
14          }
15       }
16    }
17 }
```

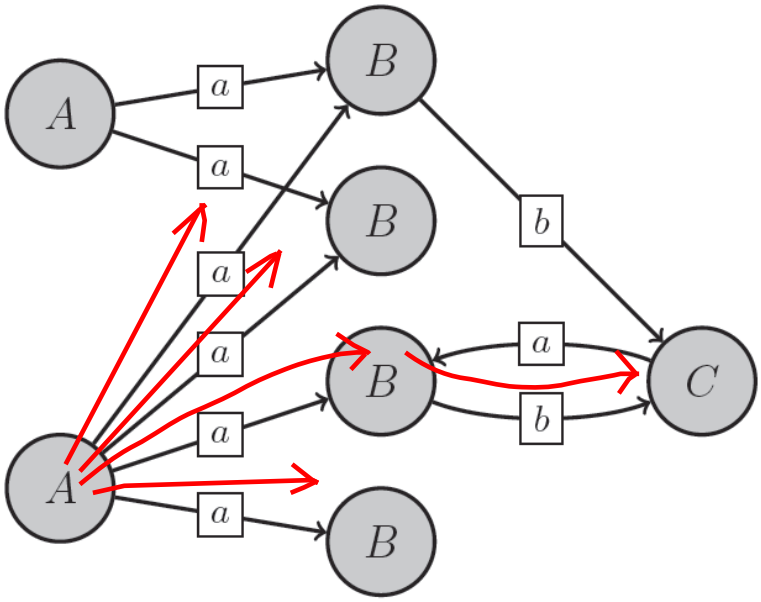


Figure 35.19: Bad search order

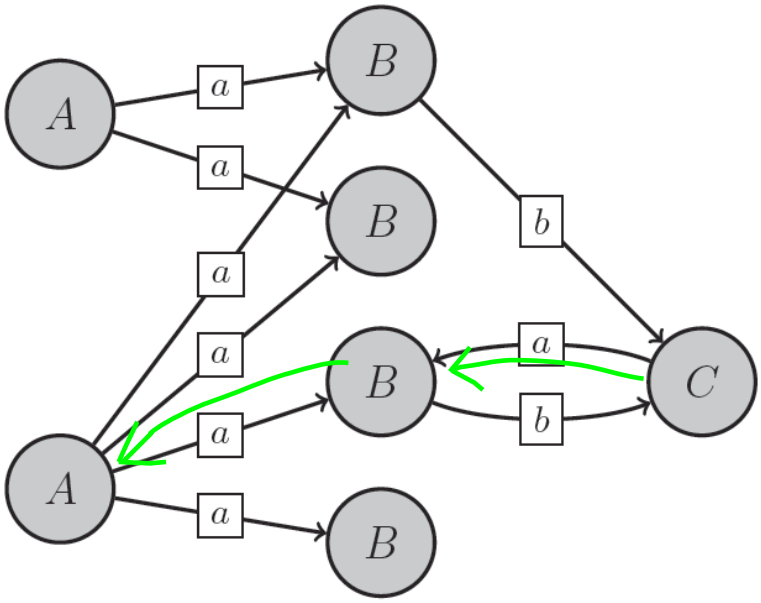


Figure 35.20: Good search order

For every pattern, there are normally multiple potential search programs existing, each capable of finding all the matches, but with vastly different performance characteristics. In order to improve performance, GrGen.NET executes a search planning phase. With the goal of finding a good schedule, it tries to prevent

- following graph structures splitting into breadth as given in this example, or
- lookups on types that are available in high quantities.

In more detail, the mechanism of search planning works by constructing a *search plan graph* from the pattern graph. A search plan graph is an edge-weighted graph, with nodes corresponding to the pattern elements – both nodes and edges – and edges representing operations to match them, with weight attributes giving an estimated cost of the operation.

A search plan graph contains an additional root node, with an outgoing edge to each other node defining a `lookup` operation. From plan nodes created for nodes, edges are leading to plan nodes created for edges, denoting `outgoing` and `incoming` operations. From plan nodes created for edges, edges are leading to plan nodes created for nodes, denoting `source` and `target` operations. The cost of the operations is determined by analyzing the amount of splitting between adjacent nodes for every triple of (*nodetype*, *edgetype*, *nodetype*) in the graph – called a V-Structure – and by counting the number of elements of every node type or edge type.

Then, a *spanning arborescence*[3] of minimum overall cost is selected from the search plan graph. The spanning arborescence is further linearized into a *schedule*, a list of the selected search operations (as already introduced with the good and the bad schedules).

The details of search planning and some evaluation how well it works are given in [Bat06] and in [BKG08].

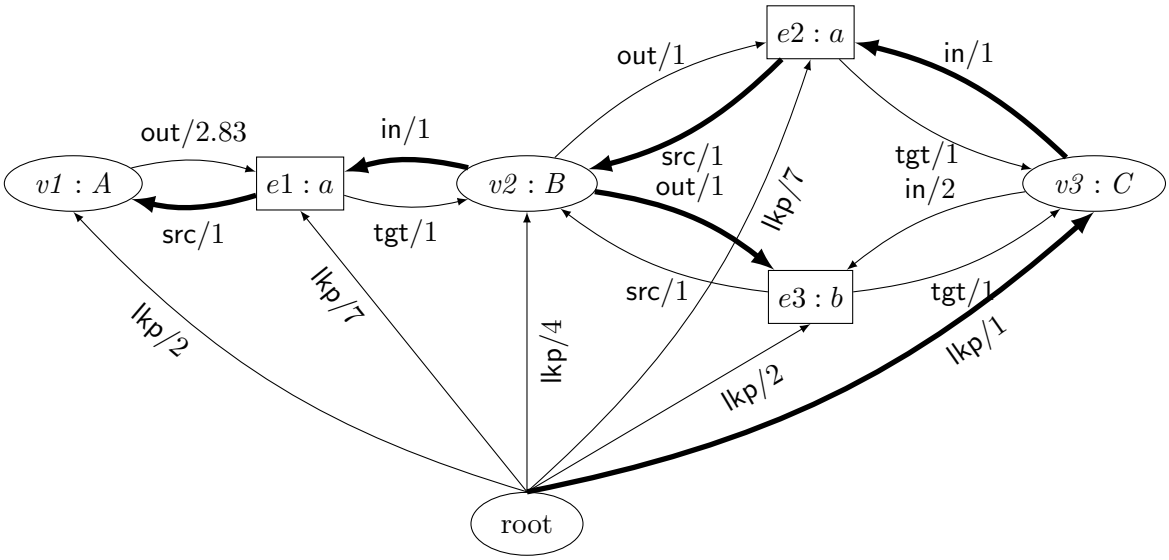


Figure 35.21: The search plan graph for the pattern graph of Figure 35.17 with estimated backtracking costs induced by the host graph of Figure 35.18. The found minimum directed spanning tree is denoted by thick edges.

Afterwards, a search program is constructed from the schedule. It is situated at a level of abstraction in between the schedule and the code finally emitted. Structurally, it is a tree resembling the syntax tree of the code to generate, as sketched in figure 35.22 (with list entries maybe containing further lists).

It contains explicit instructions for isomorphy checking and connectedness checking (which depend on the exact schedule). Furthermore, it contains exact locations where to continue at; these target search operations may be different from the directly preceding search operation (because that one does not define a choice point of influence).

Connectedness checking can be seen in the search program of the good example in the check that the target of `e3` is indeed `v3`; a target matching operation `tgt(e3,v3:C)` is not used because `v3` was already matched by the lookup operation, and is already contained in the spanning tree.

A remark on isomorphy checking that ensures that two pattern elements are not bound to the same graph element: as a consequence of the simple loop-after-loop based assignment of graph elements to pattern elements, the pattern elements would get matched homomorphically to each other by default, if an explicit check against would not be inserted (because assignment starts with the same graph elements, the homomorphic matches often *are* the first

[3] directed tree

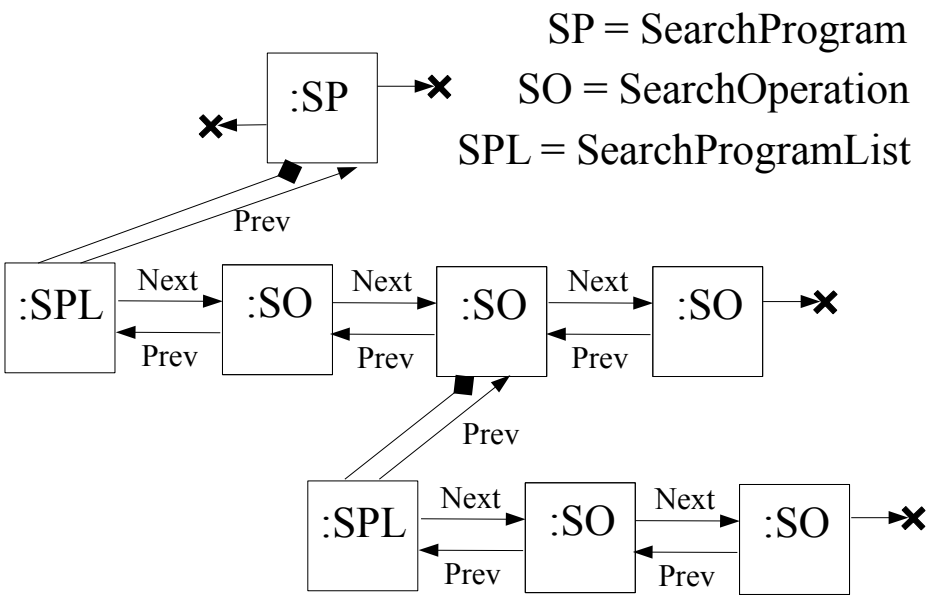


Figure 35.22: A simple Search Program

ones found). Isomorphy checking is implemented by flags that are contained in the graph elements, and are set when a graph element is bound to a pattern variable, and reset when the binding is given up again (one flag for each negative/independent nesting level, until the implementation defined limit of flags available in the graph elements is reached, then a list of dictionaries is used). The flags are then checked in the following, nested matching operations. An iso-check scheduling pass ensures that checks are only inserted if the elements must be isomorphic and their types do not already ensure that they cannot get matched to the same elements.

35.5 The Code Generator

35.5.1 Frontend

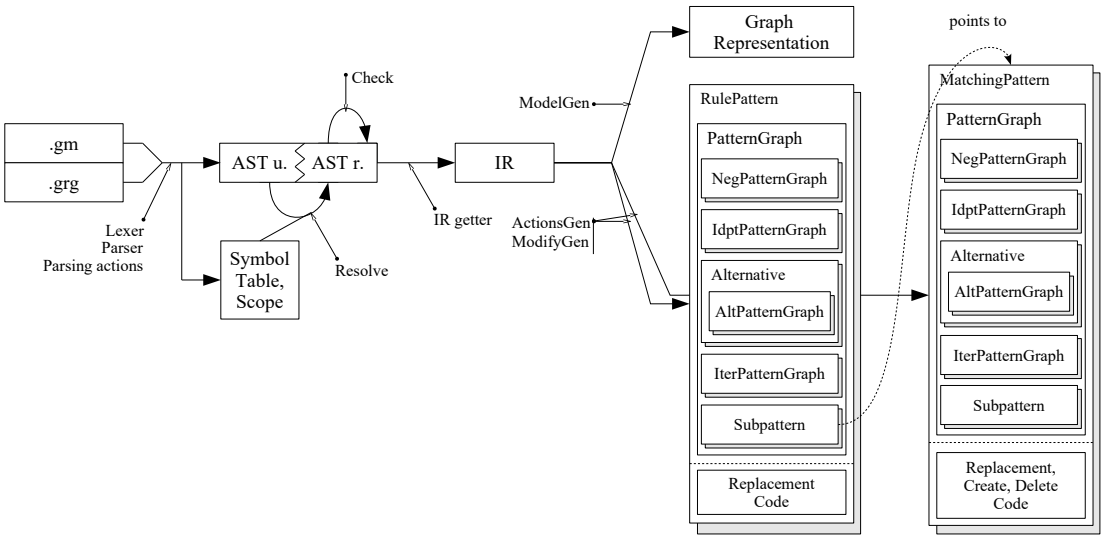


Figure 35.23: Frontend Code Generation

The frontend is spread over the directories `parser`, `ast`, `ir`, `be` and `util`, with their code being used from `Main.cs`.

Syntax and Static Semantics

The directory `parser` contains parser helpers like the symbol table and scopes and within the `antlr` subdirectory the ANTLR parser grammar of GrGen.NET in the files `Grgen.g` and `EmbeddedExec.g`. The semantic actions of the parser build an abstract syntax tree consisting of instances of the classes given in the directory `ast`, with the base class `BaseNode`. The AST is operated upon in three passes, first resolving by `Resolve` and `ResolveLocal`, mainly replacing identifier nodes by their declaration nodes during a (largely) preorder walk. Afterwards, the AST is checked by `Check` and `CheckLocal` during a (largely) postorder walk for type and other semantic constraints. Finally, an intermediate representation is built from the abstract syntax tree by reading from the `IR` property and calling `ConstructIR` methods.

Intermediate Representation

The IR classes given in the `ir` folder can be seen as more lightweight AST classes; their name is often the same as for their corresponding AST classes, but without the `Node`-suffix which is appended to all AST classes. The most interesting classes are `Rule` used for rules, alternative cases and iterateds, as well as `PatternGraphLhs` used for all left hand side pattern graphs including negatives and independents, and `PatternGraphRhs` used for all right hand side pattern graphs; several data flow analyses are contained in `PatternGraphLhs`, some covering the nesting of patterns, some being even global.

A particularly interesting one is `EnsureDirectlyNestingPatternContainsAllNonLocalElementsOfNestedPattern` from `Unit.cs`, it ensures that, from a pattern which contains a certain entity the first time, up to every pattern that references this entity, all intermediate patterns contain that entity, too. It is implemented with a recursive walk over the nested patterns in the IR-structure. Each pattern receives the set of already known elements as parameter. Before descending, the elements available in the current pattern are added to the set of already known elements, then recursive descent into the directly nested patterns follows, handing down the already known elements. On ascending, elements are added to the current pattern if they are contained in a directly nested pattern, but not in the current pattern, although they are known in the current pattern. This function allows keeping the backends simple.

The IR classes are the input to the backend of the frontend, which can be found in the `be/Csharp` folder [4]

Backend (of the Frontend)

The directory `be/Csharp` contains the code generator for the C# based backend of GrGen.NET. It generates the source code for the model with the node and edge classes, in `FooModel.cs` for a rule file named `Foo.grg`, and the intermediate source code for the rules, consisting of a specification of the patterns to match, a listing of the embedded graph rewrite sequences, and the rewriting code, in `FooActions_intermediate.cs`. This is done in several recursive passes over the nesting structure of the patterns in the IR.

The backend of the frontend does *not* generate the complete source code for the rules including the matcher code or the code for the embedded rewrite sequences — this is done by `grgen.exe`, which calls the `FrontendGrGen.exe` (formerly `grgen.jar`) of the frontend

[4] The directory `be/C` is only contained in the SCM history anymore, it contained the code generator for the C based backend, which is integrated into the IPD C compiler. (The compiler transforms a C program into a graph and SSA based compiler intermediate representation named FIRM using libFirm (see libfirm.org, [TLB99], [Lin02]) and further on to x86 machine code.)

(generating a `FooActions.cs`). You may call it on your own in order to get a visualization of the model and rewrite rules, in the form of a `.vcg`-dump of the IR, cf. Note59.

Model Generation

The code generation emitting the model is implemented in `ModelGen` (entry point `GenModel`), it is rather straight forward:

1. first, code for the user defined enums is generated,
2. then the node classes are generated,
3. then the edge classes are generated,
4. and then the object and transient object classes are generated.

This occurs for the global namespace and also by-package for the available model packages. This is followed by the generation of

1. the node model,
2. the edge model,
3. the object and transient object model,
4. and finally, the graph model.

For the nodes as well as for the edges, three classes are generated:

1. the first is the interface visible to the user, giving access to the attributes,
2. the second is the implementation of the interface, also inheriting from `LGSPNode` or `LGSPEdge`,
3. and the third is a type representation class, which is inheriting from `NodeType` or `EdgeType`, that is giving information about the type and its attributes, and is used in the node/edge model and thus the graph model.

Additionally, the (database-like) indices are generated by `ModelIndexGen`, and functionality regarding external extensions by `ModelExternalGen`. `ModelIndexGen` generates first the index types with `GenIndexTypes` (they inherit from `IAttributeIndex` or `IIncidenceCountIndex`, and contain diverse `Lookup` functions), then the index implementations with `GenIndexImplementations` (each being an implementation of an AA-tree consisting of event handlers that are notified about graph changes, the maintenance functions for insertion, deletion and tree rebalancing, and the implementation of the lookup functions from the interface returning data from the tree), and finally the overall index set with `GenIndexSetType`.

Rule Representation Pass

The code generation emitting the actions – better their representation – is implemented in `ActionsGen` (entry point `GenActionlike`), with the main functionality (the LHS pattern descriptions and the RHS pattern implementations, in this section we concentrate on the former) in the code called from `GenSubpattern` for the subpatterns and `GenAction` for the rules and tests.

The main functionality is complemented by the code called from `GenSequence`, `GenFunctions`, `GenProcedures`, `GenFilterFunctions`, `GenMatchClassFilterFunctions`, and finally `GenMatchClasses`. This occurs for the global namespace and also by-package for the available action packages.

Then the representation classes of the external functions and procedures are generated with `GenExternalFunctionInfos` and `GenExternalProcedureInfos`, and finally the actions representation class (which is inheriting from `LGSPRuleAndMatchingPatterns`).

Regarding the main functionality: in a prerun from `GenRuleOrSubpatternClassEntities` on, some needed entities are generated, e.g. type arrays for the allowed types of the pattern elements (the arrays are only filled if the base type was constrained), or indexing enums, which map the pattern element names to their index in the arrays of the matched host graph elements in the match objects of the generic match interface.

In the rule representation pass, from `GenRuleOrSubpatternInit` on, the subpattern- or rule representations of type `MatchingPattern` or `RulePattern` are generated, including the contained `PatternGraph`-objects for the (nested) pattern graph(s), by mutually recursive calls of `GenPatternGraph` and `GenElementsRequiredByPatternGraph` (the former assembling the rule representation with the elements generated by the latter).

The method `GenElementsRequiredByPatternGraph` generates for a pattern graph its contained elements, the pattern nodes, the pattern edges, the used subpatterns as `PatternGraphEmbedding` members, the contained alternatives, and the iterateds; the representations of contained negative and independent patterns as well as alternative cases are generated by `GenPatternGraph` before the containing pattern.

A graph element contained in a pattern, but defined in a nesting pattern, is saved in the pattern as a reference to the element in the nesting pattern. The `PatternGraph` in which it was used first is remembered in a `PointOfDefinition` member variable (in the generated code, but a similar scheme is used in the compiler frontend). The source and target nodes of edges are saved in the `PatternGraph` in hash tables (so that an edge with an undefined source or target can be refined in a nested pattern); source and target nodes are determined by ascending alongside the pattern nesting until a definition is found.

As all graph elements, including the ones of the nested patterns, are created flatly in the `initialize`-method of the `MatchingPattern` (in the generated code), name prefixes are needed to prevent name clashes; this is ensured by the parameter `pathPrefix`. Correct naming of already declared elements is ensured with an `alreadyDefinedEntityToName`-hashtable. The split into constructor and `initialize` method is needed because a recursive `MatchingPattern` must reference itself at construction.

Furthermore, local and global homomorphy tables for the nodes and the edges are generated, defining which pattern elements are allowed to match the same host graph element (the global tables specify the homomorphy in between elements from the `PatternGraph` of an alternative case or an iterated, and an enclosing `PatternGraph`).

Additionally, expression trees for the attribute checks and statement blocks for the yieldings are generated, by `GenExpressionTree` and `GenYield` from `ActionsExpressionOrYieldingGen`, called by `GenPatternCondition` and `GenPatternYielding`; the entities referenced in them are gathered by `CollectNeededEntities`. The real code for these representations is emitted in the `engine-net-2`-backend as is the real code (the pattern matchers) for the pattern representations.

Then the rewrite code pass is executed (explained in more detail further below), followed by the handling of embedded sequences.

For embedded sequences, representations in form of `LGSPEmbeddedSequenceInfo` objects are generated, which contain the sequence as string, plus additional parameter information; the real sequence code is generated in the `engine-net-2` backend. For sequences embedded in alternatives, iterateds, or subpatterns, additionally closure classes inheriting from `LGSPEmbeddedSequenceClosure` are generated. They store the graph elements the pattern elements from the pattern containing the exec were bound to; the sequence execution function is then called on this closure, which is stored in a queue, and executed from the top level rule. This is carried out in `ActionsExecGen`; the representations of the non-embedded defined sequences bearing a name are directly emitted in `GenSequence` of `ActionsGen`.

Finally (the exact) match classes are generated by calls of `GenMatch` utilizing `ActionsMatchGen`, for every pattern besides negative patterns an interface and a class implementing this interface (they are not to be confused with the match class types defined by the user directly, getting emitted by calls of `GenMatchClass`, utilizing the same `ActionsMatchGen`). The match classes are instantiated after a match of the corresponding pattern was found, giving a highly convenient and type safe interface to the matched entities at API level.

Note that `GenFunction`, `GenProcedure`, `GenFilterFunction`, and `GenMatchClassFilterFunction` are largely using `ModifyEvalGen` to generate the body with their functionality (besides their interfaces/representations), the procedures that may contain embedded sequences also utilize `ModifyExecGen`.

For the actions in the narrow sense (rules and tests), some further (match processing) helper code is emitted by `GenExtractor`, `EmitConvertAsNeededHelper` and `GenArraySortBy`.

Rewrite Code Pass

In the rewrite pass (a virtual pass woven into the code generation, not being a separate top-level pass in its strict sense in the code generation), the rewrite code gets generated, via `GenModify` of `ModifyGen`. The nesting of negative or independent patterns does not need to be taken care of here, as they come without rewrite parts.

For every pattern piece and its rewrite role (dependent rewriting, creation, deletion) are local rewrite methods generated, carrying out the complete rewriting by calling each other at runtime. The `ModifyGenerationState` holds the state during generation, while the `ModifyGenerationTask`s allow to give a left and right graph independent of the rewriting specified with the rule (so the creation and deletion code can be generated in addition to the dependent rewrite as explained in the following). The code for subpattern creation is generated by using an empty graph as left graph and the pattern graph as right graph. The code for subpattern deletion is generated by using the pattern graph as left graph and an empty graph as right graph. The code for dependent rewriting is generated by simply referencing the left and right pattern graphs from `IR::Rule`. For a test, the pattern graph is used as left and right graph. For alternatives and iterateds, dispatching methods are generated, calling the rewrite methods of the instances of the patterns matched. Rewrite parameters are mapped to local parameters of the rewrite methods.

The core method of rewrite code generation named `GenModifyRuleOrSubrule` generates the rewriting of the given pattern and the calls to the rewrite methods of the nested patterns. The rewrite methods of the nested patterns are prefixed with their nesting path, in order to avoid name conflicts, which may occur as they are placed flatly in their `MatchingPattern` or `RulePattern`. Their role is distinguished by one of the name postfixes `Modify`, `Create` or `Delete`; for keeping a graph unmodified obviously no methods are needed.

Also during the rewrite pass, the statements of the `eval` blocks are generated, this happens in `ModifyEvalGen`, and the *calls of the routines* implementing the embedded `exec`s, in `ModifyExecGen` (the non-embedded sequences are called explicitly, no implicit calls have to be inserted automatically).

Note that the non-pattern-matching code (the rewrite part plus functions, procedures and alike) that will be executed is emitted directly (in `ModifyGen` and `ModifyEvalGen`), instead of only a representation that is the input to further processing in the `engine-net-2` backend, as is the case for the rule/test/subpattern descriptions including the attribute conditions and yieldings (the parts that can be recompiled at runtime; also the real sequences implementation is part of the `engine-net-2` backend). The code to emit the expression code is contained in `CSharpBase`, the `ModifyEvalGen` extends it by the code for the statements. A lot of non-trivial functionality is mapped to calls of functions supplied by the runtime (many helper classes from the `libGr`, some from the `lgspBackend`); besides, some further code generation exists, e.g. to fill constant sets or maps or to implement the functions with lambda-expression

parameters.

35.5.2 Backend

The real matcher code is generated by the backend given in `engine-net-2`, in the `src/lgsp-Backend` subdirectory. The generation is carried out in several passes, use as entry points

- `GenerateAndInsertMatcherSourceCode` from `lgspGrGen.cs` in case of the static compilation of the actions dll
- the `GenerateActions` function from the `lgspMatcherGenerator.cs` in case of dynamic matcher recompilation at runtime

The base data structure is the `PatternGraph`, resp. the nesting of the `PatternGraph`-objects, contained in the `RulePattern`-objects of the rules/tests or the `MatchingPattern`-objects of the subpatterns.

1. Step

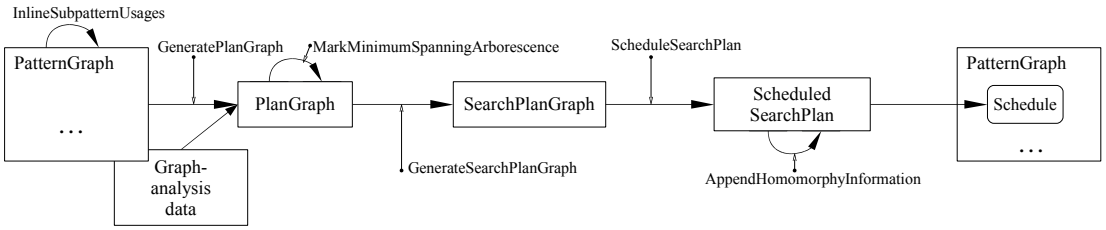


Figure 35.24: Backend Code Generation, step 1

First, the subpattern usages are inlined when inlining is assumed to be beneficial. (This allows the user to extract common patterns into subpatterns increasing readability and modularity without losing performance (but as of now only one level of inlining is supported, so excessive subpattern extraction has to be avoided, especially if the containing pattern would become disconnected by this)). Before and after this kind of pattern rewriting, the patterns and their relations are analyzed by the `PatternGraphAnalyzer` – the analyzation results are used for emitting better code later on (choosing more efficient but more limited implementations of language features, which are not sufficient in the general case but are so for the specification at hand). The `PatternGraphAnalyzer` that also contains the inlining functionality is called from `AnalyzeAndInlineMatchingPatterns` in `lgspGrGen.cs` in case of static compilation based on the results emitted by the frontend, it is called from the generated actions code in `InitActions` at runtime, its results are then used in case of dynamic matcher recompilation.

Then a `PlanGraph` is created from the `PatternGraph` and data from analyzing the host graph (for generating the initial matcher some default data from the frontend is used). Independents are inlined during plan graph creation unless the feature is disabled, the main part of independent inlining is occurring in `GenerateStaticPlanGraph` or `GeneratePlanGraph` of the `PlanGraphGenerator`, when `inlineIndependents` is requested.

A minimum spanning arborescence is computed defining a hopefully optimal set of operations for matching the pattern (the hopes are founded, see [BKG08]). A `SearchPlanGraph` is built from the arborescence marked in the `PlanGraph` and used thereafter for scheduling the operations into a `ScheduledSearchPlan`, which gets completed by `AppendHomomorphy-Information` with isomorphy checking information. The `ScheduledSearchPlan` is then

stored in the PatternGraph it was computed from. This is carried out in GenerateScheduledSearchPlans in lgspGrGen.cs during compilation or in GenerateScheduledSearchPlans in lgspMatcherGenerator.cs during runtime recompilation.

2. Step

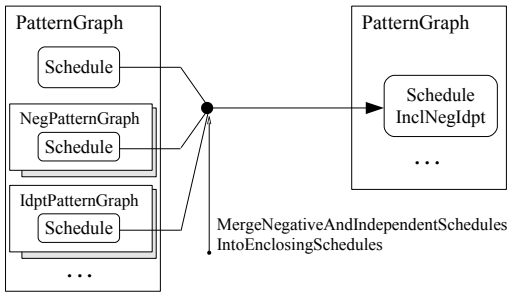


Figure 35.25: Backend Code Generation, step 2

In a second step, the Schedules of the negative or independent PatternGraphs are integrated into the Schedule of the enclosing PatternGraph, by MergeNegativeAndIndependentSchedulesIntoEnclosingSchedules (in the ScheduleEnricher-class), in a recursive run over the nesting structure of the PatternGraphs in the MatchingPatterns or RulePatterns. Due to nested negative or independent graphs, this may happen spanning several nesting levels; the result is saved in the ScheduleIncludingNegativesAndIndependents field of the non-negative/independent PatternGraphs.

3. Step

Finally, in a third step, code is generated by GenerateActionAndMatcher (from lgspMatcherGenerator.cs), again in a recursive run over the nesting structure of the PatternGraphs, using the ScheduleIncludingNegativesAndIndependents stored in them.

For each RulePattern, an Action-class is generated, and for each MatchingPattern, a SubpatternAction-class is generated. Additionally, for each alternative an AlternativeAction-class is generated, the alternative matcher will contain code for all alternative cases. For each iterated, an IteratedAction-class is generated. The real matching code is then generated into these classes.

The SearchProgramBuilder builds a SearchProgram tree data structure resembling the syntax tree of the code to generate out of the ScheduleIncludingNegativesAndIndependents in the PatternGraphs. To be more precise: the SearchProgramBuilder builds the SearchProgram matcher routine heads and then employs a SearchProgramBodyBuilder to build the bodies of the matcher routines – consisting of diverse SearchProgramOperations – in a run over the schedule with BuildScheduledSearchPlanOperationIntoSearchProgram, one input SearchOperation after the other. In a further pass, the SearchProgram is completed by the SearchProgramCompleter, determining the locations to continue at when a check fails, writing undo code for the effects which were applied from that point on to the current one. Finally, the C# code gets generated by calling the Emit methods of the SearchProgram. If you want to extend this code, you may be interested in the Dump methods which dump the SearchProgram in an easier readable form into text files.

Besides this pattern-matching related core functionality of GrGen.NET, do you find in the lgspBackend/ExpressionOrYielding folder the classes emitting the code for the condition expressions and yielding statements. The generated code typically calls one of the

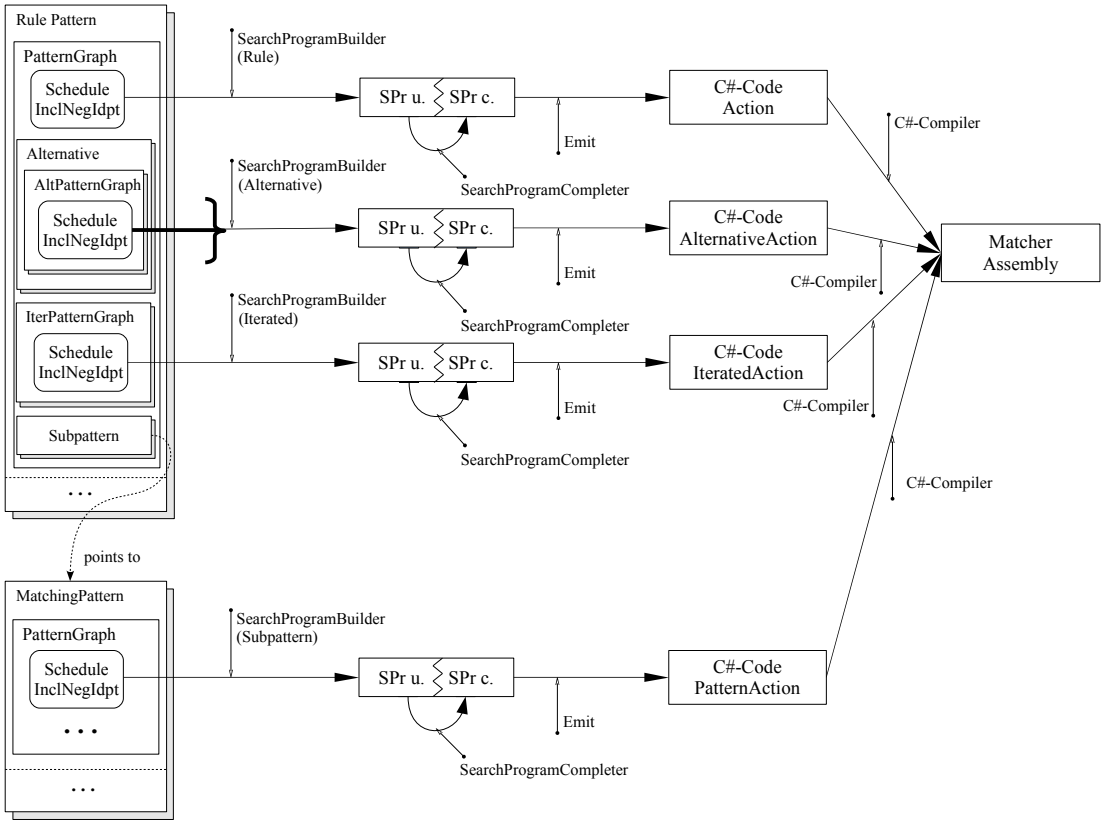


Figure 35.26: Backend Code Generation, step 3

helper method available in one of the `XXXHelper.cs` files in the `libGr` folder (defining the runtime library for the GrGen.NET-languages).

Nested and Subpattern Matching

The `SubpatternAction`-classes (cf. 35.26) do not only contain the matcher code, but are at the same time the tasks of the $2 + n$ pushdown machine, which are pushed on the open tasks stack; they contain the subpattern parameters as member variables. The same holds for the `AlternativeAction`- and `IteratedAction`-classes, which do not hold parameters but entities from the nesting pattern they reference.

The higher levels of code generation are in large parts independent of nested and subpattern matching and control of the $2 + n$ pushdown machine. Only on the level of search programs does it become visible, with an `InitializeSubpatternMatching`-search program operation at the begin of a search program and a `FinalizeSubpatternMatching`-search program operation at the end of a search program; but mainly with a call to `buildMatchComplete` when the end of the schedule is reached during search program building. This corresponds to the innermost location in the search program, the location at which the local pattern was just found (during execution); now the control code is inserted by `insertPushSubpatternTasks`, pushing the tasks for the subpatterns used from this pattern, as well as alternatives and iterateds nested in this pattern. To execute the open tasks a `MatchSubpatterns` operation is inserted into the search program. Afterwards, `insertPopSubpatternTasks` inserts the operations for cleaning the task stack, `insertCheckForSubpatternsFound` the operations to handle success and failure, and `insertMatchObjectBuilding` the code for maintaining the result stack.

35.6 Further Backend Parts and Applications

The `engine-net-2` folder contains besides the pattern matching parts of the backend code generator also the implementation of the graph rewrite sequences, the language runtime, the GrShell with the debugger, and further libraries.

libGr and lgspBackend

The `src/libGr` subfolder contains the libGr, offering the base interfaces you see on the API level for the model, the actions and the pattern graphs, as well as the host graph. The interfaces are implemented by code from the libGr search plan (lgsp) backend in `src/lgspBackend` and by the generated code. The generated code defines a type-safe interface of named and typed entities, which is more convenient to use at API level, but bound to a specification. The libGr in contrast offers a "generic", name string and object or root type based interface.

In addition, it offers the graph rewrite sequence parser, which gets generated from `SequenceParser.csc`, and is building a graph rewrite sequence AST out of a sequence string, from the classes in `Sequence.cs` (and `SequenceBase.cs` plus `SequenceComputation.cs` and `SequenceExpression.cs`), further utilizing a `SymbolTable` and the `SequenceParserEnvironment`. After building, the sequence AST is type checked by the `SequenceCheckingEnvironment`, which – like the parser environment – comes in a version for the interpreted sequences and a version for the compiled sequences, abstracting away the differences in available (type) information while generating the action code (compiled) compared to just using the generated action code (interpreted).

The graph rewrite sequence classes contain a method `ApplyImpl(IGraphProcessingEnvironment procEnv)`, which is called for executing/interpreting the sequence. The compiled graph rewrite sequences instead are emitted by the `lgspSequenceGenerator` as C# source code. The `lgspSequenceGenerator` works as driver, employing esp. the `SequenceGenerator` for generating the rule controlling sequences, which in turn employs esp. the

`SequenceComputationGenerator` for generating the sequence computations, and the `SequenceExpressionGenerator` for generating the sequence expressions.

When GrGen rules are matched, as well as when the graph is changed, events are fired. They allow to display the matches and graph changes in the debugger, to record changes to a file for later playback, or record changes to a transaction undo log for later rollback. You can register event handlers to them, in order to execute your own code (allowing you to build an event based graph rewriting mechanism on API level). The graph delegates fired are given in `IGraph.cs`. Graph events recording is implemented in `Recorder.cs`, replaying amounts to a normal `.grs` execution. Graph transaction handling is implemented in `lgspTransactionManager.cs`, with a list of undo items, which know how to undo the effect that created them. They are purged on commit or executed on rollback. Backtracking is implemented with nested transactions. If you are changing the graph programmatically at API level (not using GrGen rules), you have to fire some of the events (esp. the ones for attribute assignment) on your own in case you want to use any of the mechanisms (graphical debugging, record and replay, transactions and backtracking, event based programming) above.

The runtime library `libGr` further contains the interface `IGraphProcessingEnvironment` to the graph processing environment (built up from a series of environment interfaces with increasing functionality), offering e.g. graph global variables, sequence execution, convenient action execution, as well as graph change recording and transaction handling (implemented by the aforementioned classes in the `lgspBackend`). Additionally, it implements helper code in `XXXHelper.cs` files.

The `ContainerHelper.cs` contains code that is common for all container types (cf. Chapter 15), the `DictionaryHelper.cs` contains code that is common for sets and maps, while the `SequentialTypeHelper.cs` contains code that is common for array and deques; code specific to each container type is available in correspondingly named files. The `GraphHelper.cs` contains code that implements the diverse graph query functions and graph manipulation procedures (cf. Chapter 16), while the `GraphHelperParallel.cs` contains code that implements the diverse graph query functions in versions that can be employed from parallelized pattern matchers (which are not allowed to use the isomorphy checking flags contained in the graph elements that are employed by the non-parallel pattern matchers). The `EmitHelper.cs`, `AttributeTypeEmitHelper.cs` and `AttributeChangeTypeEmitHelper.cs` contain code for uniform emitting/to-string-conversion of the diverse types supported by GrGen.NET, the `GrGenTypesHelper.cs`, `DotNetTypesHelper.cs`, and `TypesHelper.cs` files contain type querying and type representation conversion functions (predominantly for internal use).

The `libGr` offers also several importers and exporters in the `src/libGr/IO` subfolder, as well as dumpers into file formats meant for visualization.

For graph comparison, a pattern may be created out of one of the graphs, and searched in the other one, by utilizing an `InterpretationPlanBuilder` to build an `InterpretationPlan`, and by calling `Execute` on it (an interpretation plan is a simplified version of a search plan, that is executed at runtime instead of getting emitted into an actions file – when it is utilized a certain number of times, it gets emitted and compiled (at runtime) in order to speed up pattern matching and thus graph comparison. Take a look at `lgspBackend/GraphComparison` for the code and at Section 16.5) for further explanations.

The filter implementations of the auto-generated filters are generated by the `FilterGenerator.cs`, the auto filter for filtering symmetric matches from automorphic patterns is further utilizing the `SymmetryChecker.cs`. The auto-supplied filters are available pre-implemented in the `libGr/matchesHelper.cs` and the `lgspBackend/Actions/lgspMatches.cs` files.

GrShell and Debugger

The `src/GrShell` subdirectory contains the GrShell application. The bulk of the implementation is contained in the libGrShell in `src/libGrShell`, though. Its command line parser is generated by CSharpCC out of the `GrShell.csc` grammar file, the shell implementation/functionality is given in `GrShellImpl.cs`, execution starts in the `Main` method in `GrShellMain.cs`. The library and accordingly the shell builds upon the sequence interpretation facilities offered by the libGr and by the `GrShellSequenceApplierAndDebugger.cs`, and the "generic" interface of the libGr, because it has to be capable of coping with arbitrary user defined models and actions at runtime. Another shell application on top of the libGrShell is the GGrShell, the graphical GrShell, which is a Windows Forms application based on a `GuiConsoleControl` (controlled by a GUI message loop).

Stepwise and graphical debugging is offered by the `Debugger.cs` and several helper classes, esp. including the `GraphViewerClient`, which contains higher-level graph viewer functionality (with parts of the visualization configuration and the implementation of the hierarchical node grouping specification), delegating to a lower-level `basicClient` implementing the `IBasicGraphViewerClient` interface for real graph rendering. Further notable is the `GraphViewer` class, an auxiliary class with helper methods for viewing and dumping graph (which in turn delegates most of the tasks to the `GraphViewerClient`). One of the usable basic clients implementing `IBasicGraphViewerClient` is the `YCompClient.cs` (inheriting from the `GraphViewerApplicationProxy`), it communicates with the yComp application with a custom protocol over a TCP connection to localhost. They are contained in the graphViewerAndSequenceDebugger library.

Another (internal-only) graph viewer is offered by the `MSAGLClient.cs` and its extended version, the `MSAGLExtClient.cs`, they also implement the `IBasicGraphViewerClient` interface. They are contained in the graphViewerAndSequenceDebuggerWindowsForms library, together with the `GuiDebuggerHost.cs`, which offers a GUI debugger (based on the same debugger application logic used by the console debugger, just embedded into a different hosting environment).

The dlls are loaded on demand, and can be used from API level; you could implement a debugger for another GUI framework and OS with the same mechanisms.

The graph viewer extMSAGLExt is an external application (like yComp) that is based on the `MSAGLExtClient.cs` and available as source code (in contrast to yComp). It additionally implements VCG import, and support for nearly the same server mode/custom protocol that is offered by yComp, on the client side implemented by the `ExtMSAGLExtClient.cs` (also inheriting from the `GraphViewerApplicationProxy`).

Further Libraries and Applications

The libGrPersistenceProviderSQLite offers an implementation of the persistent graph based on the SQLite database.

The libConsoleAndOS is a console and OS abstraction library defining console interfaces and a mapping to the normal stdout/stdin console.

The libConsoleAndOSWindowsForms library offers a Windows Forms implementation of the console interface (based on a `RichTextBox`).

You could implement a persistence provider for another database or a console implementation for another GUI framework and OS.

You find more detailed information (on them) in the `CLAUDE.md` files in the project directories.

The `src/GrGen` subdirectory contains the driver procedure of the `grgen.exe` compiler.

The `examples-api` subdirectory contains several examples of how to use GrGen.NET from the API.

The `examples` subdirectory contains a bunch of examples for using GrGen.NET with GrShell.

In case you want to contribute and got further questions don't hesitate to contact us (via email to `grgen` at the host given by `grgen.de`).

CHAPTER 36

DEVELOPMENT GOALS AND DESIGN DECISIONS

GrGen.NET is the successor of the GrGen tool presented at ICGT 2006 [GBG+06]. The ".NET" postfix of the new name indicates that GrGen has been reimplemented in C# for the Microsoft .NET or Mono environment [Mic07, Tea07]; it is open source licensed under LGPL3(www.gnu.org/licenses/lgpl.html) and available for download at www.grgen.de.

Over the course of time, it moved from high-performance fixed-shaped patterns and limited sequences for control to flexibly-shaped patterns (with nested and subpatterns) and very high programmability (for one with the sequences and for the other with an integrated imperative programming language), while retaining its performance.

GrGen.NET offers development at the abstraction level of graph representations. It offers the convenience of linguistic abstraction, with dedicated syntax for graph processing, and static type checking. In contrast to limited libraries or the very leaky abstractions often offered by common programming frameworks. The abstraction is not halting at the languages, but carried through to development support. With a debugger showing the data and the processing effects on that very level, removing a major shortcoming of many domain-specific languages that are linguistically better but are undebuggable at their level. And with performance optimization helpers operating at this level, in the form of search plan explanation of the pattern matchers, as well as profiling instrumentation and statistics printing for the search steps.

The main characteristics and development goals of GrGen.NET have been:

- productivity through declarative pattern-based rules of very high expressiveness
- high-performance pattern matching and graph rewriting, based on a model implementation tailored for it, and an optimizing compiler
- general-purpose graph-representation processing enabled by its very high programmability, and customizability
- debugability: the processing state and esp. the graph can be inspected, esp. visually

Whenever a new feature was found to be needed, we asked ourselves: can this be made more *general*, still keeping it *understandable* and *efficiently* implementable.

In the following, we'll take a deeper look at the development goals of GrGen.NET (comparing it to competitor tools on the way).

36.1 Expressiveness

is achieved by *declarative* specification languages for *pattern matching* and *rewriting* that are bursting with features, building upon a rich graph model language.

In addition to the unmatched expressiveness of the single-element operations offered by the pattern and the rewrite parts, are *nested* and *subpatterns* available for combining them flexibly. They allow processing substructures of arbitrary depth and breadth with a single rule application. This surpasses the capabilities of the VIATRA2[VB07, VHV08] and GROOVE

[Ren04] tools, our strongest competitors regarding rule expressiveness. You may have a look at the GrGen.NET solution of the program understanding case [JB11] of the TTC 2011 highlighting how *concise* and *elegant* solutions become due to the expressiveness of the language constructs.

While the patterns can be combined in a functional way with nested and subpatterns to build complex rules, the rules can be combined in an imperative way with graph rewrite sequences, executing one graph state change after the other.

The sequences used for orchestrating the rules are at the time of writing the most advanced *strategy language* offered by any graph rewriting tool, featuring variables of elementary as well as container types (called storages when containing nodes, as pioneered by the VMTS[LLMC05] tool), and sequential, logical, conditional, and iterative control flow as the base operations, plus sequence definitions and calls. Small graph-changes and computations can be executed with sequence computations without the need to enter and exit again the rule layer. The most notable feature of the sequences are their *transactional* and *backtracking constructs*, which support search space crawling and state space enumeration with their ability to roll changes to the graph back to an initial state, saving you a considerable amount of change reversal and bookkeeping code you'd need for implementing this on your own.

Expressiveness is achieved by language elements that were designed to be *general* and *orthogonal*. GrGen.NET offers the more general iterated patterns subsuming the multinodes as they are common for graph rewriting tools. GrGen.NET offers the more general recursive patterns subsuming the regular path expressions which are the more common choice. You get means to *combine patterns*, instead of dedicated solutions for single nodes or edges. GrGen.NET offers generic container types, and not only storages, as is more common (non-container-variables are even more common), and it offers them as graph element attributes, too and not only for processing variables. GrGen.NET offers not only node and edge types, but also a graph type allowing to store subgraphs; and especially does it allow that graph element attributes are typed with it. You can work with a type system as you know it from object-oriented-languages (where types can be used freely in defining other types).

Expressiveness is not for free, its price is *increased learning effort*. GrGen.NET offers the biggest toolbox for graph rewriting that you can find, with many tools and high-powered tools. For nearly any graph-representation-based task do you find a collection of tools that allow you to handle it in an adequate way, leading to a *concise solution* and *short development* and *maintenance durations* and thus costs. But you have to learn these tools and how to use them beforehand. For small tasks that effort is likely higher than the net gains in the end.

For subtasks for which the declarative pattern-based rules do not work well (this happens occasionally), are you free to revert to *imperative* and *object-oriented programming* as known from the C++, Java, C# language family, utilizing the imperative parts of the GrGen.NET-languages. You lose a good amount of the benefits of employing the tool then (we consider model transformation tools offering own – but only imperative – languages pointless (and OCL-expressions are only a small improvement in this regard)), but neither are you stuck, nor are you forced to convoluted workarounds, as it may happen with tools that offer only pattern-based rules, e.g. Henshin[ABJ+10] or AGG[ERT99]. The *multi-paradigm languages* of GrGen.NET offer you all the constructs you need for transforming graph-representations.

36.2 General-Purpose Graph Rewriting

in contrast to *special-purpose* graph rewriting requires high *programmability*, *genericity* and *customizability*.

Some graph based tools are geared towards special application domains, e.g. biology (XL [KK07]) or verification (GROOVE [Ren04]). This means that design decisions were taken that ease uses in these application areas, at the cost of rendering uses in other domains more difficult. And that features were added in a way that just satisfies the needs of the domain at

hand and its tasks, instead of striving for a more general solution (which would have caused higher costs at designing and implementing these features).

This can be seen well in the approach towards state space enumeration. Instead of offering a built-in fixed-function state space enumerator like GROOVE and Henshin do, are you given the language devices needed to program one with a few lines of code, with the backtracking abilities of the sequences, and subgraph extraction, comparison, and storing in variables/graph element attributes. This highlights the costs of being general-purpose: you *must program* what you need. But it also highlights the benefits: you *can program exactly* what is needed, being very *flexible* in how state-space enumeration is carried out. The former tools expect a rule set which they apply exhaustively to create a state space, one state per rule application. But state space enumeration requires a lot of memory due to the state explosion phenomenon – for practical applications you need control over which changes are to be seen as states and which not, and single rules often are too low-level regarding this classification, materializing many superfluous states. GrGen.NET gives you that kind of control (backtracking rule application and state materialization are separate things), allowing you to unfold deeper given the same amount of memory. When it comes to non-toy-examples, *programmability wins* over coarse-grain *fixed-function* units. But of course only as long as you don't need to reinvent the wheel, when you can program based on a *suitable level of abstraction*.

The backtracking constructs are useful in crawling a search space, too, where the reached states don't need to be materialized into the graph, and doing so would be inadequate. When it comes to usability in many contexts, for many tasks, again programmability wins over coarse-grain fixed-function units. This should not hinder you to choose the latter when your task does not require more, they are easier to use, but beware of the day when the requirements change and you hit a wall. The approach of GrGen.NET is to offer you *languages* to achieve your goals, not *pre-coded solutions* that may or may not fit to your task at hand. This holds not only for the languages used to describe the data or to specify the computations, but also for the visualization and the debugger. They can be customized easily to the characteristics of your graph representation.

Programmability always must come with means to abstract and encapsulate the programs into own units and parametrize them, so you are not exposed to details where you don't need them (but can access them when you need to do so, in contrast to opaque fixed-function units). They are available with the rules and tests, the subpatterns, the procedures and functions including method procedures and method functions, the filter functions, and sequence definitions; and the input and output parameters available for them.

While the old GrGen started as a special-purpose compiler construction tool for internal use with fixed-shape patterns (optimizations on the graph based compiler intermediate representation FIRM – see `www.libfirm.org`), was the new GrGen.NET built from the beginning as a general-purpose graph transformation tool for external use, somewhat towards the vision presented in [BFG96] – to be employed in areas as diverse as computer linguistics, engineering, computational biology or software engineering – for reasoning with semantic nets or knowledge graphs, transformation of natural language to UML models, model transformation, processing of program graphs, genome simulation, or pattern matching in social nets (or RDF graphs). Have a look at [BG09] (describing a usage in computer linguistics; but the dedicated tool grew [BGP18] also supporting feature structures is maybe a better match in that field) or [GDG08] or [SGS09] for some of the results. More recent uses are from the field of computational design synthesis, with booggie [Hel13] as an early work, and [KVAB22] as a more current overview paper.

Or at the results of GrGen.NET for the diverse tasks posed in the transformation tool contests. It was always amongst the best rated tools, a record that is only achievable for a general-purpose tool with expressive and universally usable languages.

36.3 Performance

i.e. high speed at modest memory consumption, is needed to tackle real world problems. It is achieved by an *optimized model implementation* that is tailored for efficient and scalable pattern matching and rewriting, while not growing memory needs out of proportion – as it happens for incremental match engines storing and maintaining the matches of all rules at any time (as offered by VIATRA2[VB07]). The nodes and edges are organized in a system of ringlists, giving immediate access to the elements of a type, and to the incident elements of an element. When needed, attribute indices or incidence count indices can be applied, giving fast access to graph elements based on attribute values, or incidence counts.

The generative approach and its *compilation* of the rules into executable code is helping tremendously regarding performance, employing nested loops for matching the pattern elements (and a recursive descent based pushdown machine for combining the patterns, which gets optimized by one of the whole-program i.e. whole-rule-set optimizations with subpattern inlining; whole-world knowledge is brought into play also regarding other compiler functionalities). But it comes at a cost: the test-debug-cycle is slowed down compared to an interpreter, and you are less flexible regarding runtime changes.

A further help regarding performance are the *types*, which are speeding up the pattern matcher – besides being a help in modeling the domain, and besides easing your life by eliminating large classes of errors at compile time.

Performance is further fostered by *explicit control* with sequences, and rule applications from *preset parameters* on, defining where to match the rules (giving rooted pattern matching), and which rule to match when (getting faster to a rule that matches, based on domain knowledge), in contrast to approaches based on implicit control, most notably the graph-grammar approach with its parameterless rules (only controlled by an optional layering). Another rich control language is offered by the port-graph rewriting tool Porgy [FKP+20], a less rich but visual one by the story diagrams [FNTZ00] of Fujaba (similar to UML activity diagrams). (Explicit control can be also exercised from a programming language of your choice, e.g. C# [HWG03] or a .NET version of Python[VRDJ95], utilizing the rules at .NET-API level, with the benefit that you don't have to learn a dedicated strategy language.)

Performance is gained by the *host graph sensitive search plans*. In order to accelerate the matching step, we internally introduce *search plans* to represent different *matching strategies* and equip these search plans with a cost model, taking the present host graph into account. The task of selecting a good search plan is then considered an optimization problem [BKG08, Bat06]. In contrast to systems like Fujaba[Fuj07, NNZ00], our strongest competitor regarding performance, is our pattern matching algorithm fully automatic and neither needs to be tuned nor partly be implemented by hand.

Furthermore, several strength reduction and inlining *optimizations* are employed by the compiler to eliminate the overhead of the higher level of abstraction where it is not needed. In addition, there is the *search state space stepping* optimization available, which boosts rule applications executed from inside a loop (under normal circumstances), by starting matching of the next iteration where the previous iteration stopped. It plays an important role in being at least one order of magnitude faster than any other tool known to us according to Varró's benchmark[VSV05]. Finally, search intensive tasks can be *parallelized*, reaping benefits from our-days multicore machines; have a look at our solution for the Movie Database case of the TTC14[Jak14] to see what can be achieved this way.

While striving for performance, we still balance with simplicity. This can be seen in the choice of the AA-tree[And93] as balanced binary search tree to implement the indices, and in the choice of the (also more general) multi-index queries instead of dedicated spatial indices.

36.4 Understandability and Learnability

was taken care of by evaluating for each language construct several options, preferring constructs already known from imperative and object-oriented programming languages as well as parser generators — the ones which seemed most clean and intuitive while satisfying the other constraints were chosen (satisfying other constraints also includes fitting to the already available constructs to achieve conceptual integrity). This can be noted in comparison with the languages of the GReTL [HE11] tool, esp. with its powerful graph query language GReQL, which may be pleasing to someone from the realm of formal specification, but which are not to the *mind of a programmer*. This can be noted in comparison with functional programming languages, which throw out the baby with the bath in their attempt to control modifyability (the concept of state is more adequate to describe the world we live in, makes things better understandable for our brains, and performs better on our machines), and hamper learnability with their (over-)use of higher order programming. You may have a look at the GrGen.NET solution of the Hello World! case [BJ11b] to judge on your own (as of today, the solution would look even better because of the improved query support). As we know that even the best designed language is not self explaining we put an emphasis on the *user manual* currently read by you. Especially since the consequences of the development goals expressiveness and programmability are inevitably increasing the learning effort.

GrGen.NET is a pretty heavyweight beast — you need to learn some languages in order to use and control it (but easily learnable ones, as explained above). But thereafter, you are able to develop at the abstraction level of graph representations, and at a speed outpacing by far any developer working with a traditional programming language (for a graph-representation processing task). A specification change carried out in a day in GrGen.NET easily amounts to a man-week or even man-month(!) change in a traditional programming language.

36.5 Development Convenience

is gained especially by the offered *interactive* and *graphical debugging* of rule applications. The debugger visualizes the matched pattern and the changes carried out by rewriting in the graph where they apply, for the currently active rule in the sequence (which is highlighted).

While rewrite rule application is indeterministic per se, GrGen.NET is implemented so that its behaviour is as deterministic as possible, so that you can attack issues multiple times until they can be solved.

A further point easing development is the *application programming interface* of the generated code, which offers access to named, statically typed entities, catching most errors before runtime and allowing the code completion mechanisms of modern IDEs to excel. In addition, a generic interface operating on name strings and .NET objects is available for applications where the rules may change at runtime (as e.g. the GrShell).

The API allows you to lookup graph elements by type, and to fetch all incident elements of an element, even in reverse direction, laying the foundation for graph-oriented programming. In contrast to traditional programming that operates in passes over graph-representations from some root objects on, without direct access to elements inside the data structure. And with only the ability to follow an outgoing edge from its source node on, which means to dereference a pointer that points to the target. Every node supports an unbounded number of incident edges, increasing flexibility and regularity compared to some statically fixed pointer fields as utilized normally.

There's one convenience not offered you might expect: a visual rule language and an editor. We're following a first things first policy here, and graphical debugging is a must-have, while a graphical rule editor is only nice-to-have (we consider tools offering the other way round mere toys, created by people without own experience in graph-representation processing). Besides, textual languages bring some benefits on their own: graph transformation

specifications to be processed by GrGen.NET can be easily *generated*, maybe by agentic coding tools like Claude Code[Ant25], stored in a source code management system, and *textually diff'ed*. A graphical editor can be implemented on top of the GrGen.NET languages, reading and writing their textual serialization format — volunteers are welcome. Textual languages are also a good deal cheaper to implement. Given the limited resources of an university or open-source project this is an important point, as can be seen with the AGG[ERT99] tool, offering a graphical editor but delivering performance only up to simple toy examples — unfortunately causing the wrong impression that graph rewriting is notoriously inefficient.

36.6 Well Founded Semantics

to ease formal, but especially human reasoning. The semantics of GrGen.NET, or better a subset of the current GrGen.NET are specified formally in [Gei08], based upon graph homomorphisms, denotational evaluation functions and category theory. The GrGen.NET-rewrite step is based on the *single-pushout approach* (SPO, for explanation see [EHK+99]). The semantics of the recursive rules introduced in version 2.0 are sketched in [Jak08], utilizing pair star graph grammars on the meta level to assemble the rules of the object level. The formal semantics were outpaced by development, though, if you want to reason formally you have to restrain yourself to a subset of the current GrGen.NET.

But you may just prefer the simple graph programming language GP[Plu09] in this case, which was developed with formal reasoning in mind. For us, the convenience at using the language had priority over the convenience at reasoning formally about the language. In order to achieve simple reasoning, you need to (over-)simplify the languages to a level that makes using them much harder, as solutions need to be expressed with more, and more convoluted code. You see this struggle in the programming languages community in theory people asking for the ban of the break and continue control flow constructs, although code using them is clearer and better readable. They do so because the constructs make proving propositions about the code much harder. You see this struggle in the graph rewriting community in the preference of the *double-pushout* approach (DPO, for explanation see [CMR+99]) by the theory people, although rules using them require convoluted coding for a simple node deletion. We optimized our languages for people that need to get real work done in them, not for people who want to prove that the program is up to a specification coded in predicate or some other kind of logic.

And another remark about design: errors should be caught early. GrGen.NET supports you in this regard with its well-founded semantics (you could use for formal verification), its static type checking, and the means for validation/testing described in Chapter 28.

36.7 Platform Independence

is achieved by using languages compiled to byte code for virtual machines backed by large, standardized libraries, specifically: C# and to a small degree Java. This should prevent the fate of the grandfather of all graph rewrite systems, PROGRES[SWZ99], which achieved a high level of sophistication, but is difficult to run by now, or simply: outdated.

Java is only needed to build GrGen.NET itself (to generate some of its parsers) and run yComp, the code generator as well as the generated code and its supporting libraries are entirely .NET based. The .NET framework allows to use native code, so if you really need to, you can integrate via .NET - native interop a graph rewrite system into some native application.

There's another trait to platform independence: GrGen.NET is built on its own model layer, in contrast to the Java model transformation world, where EMF has emerged as a quasi-standard. Don't expect this to change, the performance of declarative pattern matching and rewriting is our top concern, and that can be best achieved with a model optimized for this

task (the current type and neighbourhood ringlist implementation is efficient and scalable, and supports the search space stepping optimization; directed edges are always navigable in both directions, giving the planner more room for choosing a good search plan; and some fields are included that render isomorphy checking cheap and supply some cost-free visited flags).

EMF is not that important for us for another reason: we consider the MDA vision an illusion. The costs of developing a domain-specific language, a model as a high-level representation of a domain, and a code generator emitting executable code from it (maybe with an intermediate platform dependent model in the chain) are considerable and can only be amortized over many use cases or for large program families, with a high degree of commonalities (otherwise there's nothing domain-specific to exploit) and a high degree of variation (otherwise there would be no real benefit compared to direct coding). Model-based development of single applications is economical nonsense, no matter what the claimed benefits are, the costs will never pay off. So only large program families are left, which are few, or languages for certain infrastructure tasks that are common among multiple applications, which requires a general-purpose language for that specific domain so it can be adapted to the exact context, leading to further increased costs (as can be seen in the feature development of the GrGen language, a domain-specific language for graph-representation processing...). We assume there are a good deal more use cases existing for graph representation processing than for model-based development. Nowadays, a new twist comes in due to AI code generators, that require only natural language specifications.

Many graph rewriting and model transformation tools are from the Java technical space, a few are from the .NET technical space, e.g. GraphSynth [NC07], some are native application e.g. written in C++ [Str97], but there you find more graph libraries and graph algorithm libraries than graph rewrite engines. In contrast to many of the graph based tools, GrGen.NET has a strong background in formal language processing (a realm in which parser generator tools like ANTLR [Par07] are used – we use it to implement our domain specific languages in the frontend).

Platform independence is also achieved by not using advanced language-specific features (e.g. LINQ) - this allows for easy/reliable automatic porting (e.g. by compiler technology as offered by the converters by Tangible Software Solutions) - or single-platform frameworks (Windows Forms is also supported on Linux, WPF is not).

36.8 General-Purpose Graph Transformation

in contrast to *model* transformation. Several model transformation tools offer graph pattern matching as a means of transformation and graph-based tools have been successfully used to model many domains; a clear separation is not possible then [JBW+14]. But pattern matching is typically limited in dedicated model transformation tools, as is their performance regarding it, esp. their ability for achieving automatically high-performance solutions — the EMF poses an obstacle to efficient automatic-declarative pattern matching. Often, model transformation tools offer just *mapping single elements* of a source model (inspecting some local context via OCL-expressions) to some target elements, which is only sufficient for simple tasks. Those tools – e.g. ATL[JABK08], to a lesser degree Epsilon[PKR+09] – typically offer only *batch-wise offline* mapping of one graph-like representation to another one. GrGen.NET works well for model transformation task, esp. due to its compiler construction roots; so feel free to use it if you are one of the few that indeed benefit directly from the model driven architecture approach or are interested in supplying corresponding models and languages to the benefit of others.

Furthermore, graph *transformation* in contrast to graph *databases*. The focus is on efficient *pattern-based* matching, and rewritings that are strongly coupled to the query results, capable of executing fine-grain spot-wise changes (with transaction support for crawling search

spaces). Executed on an *in-memory* graph-structure, by a *single user* at a time. In contrast to e.g. the graph database Neo4J[The17], that offers non-pattern queries that are feeding loosely coupled simple updates via a query-result container in between, which is accumulating the matches but renders fine-grain changes based on the exact matches/spots difficult. Executed on a *backing-store*, by *multiple users* that are working concurrently at the same time, isolated by transactions.

So GrGen.NET is not suited for scenarios where many users need to access data at the same time. It is very fast, so you can handle multiple-user scenarios simply by serializing the accesses — this could be well sufficient for your tasks, but won't scale towards really large numbers of users. You may use it as an application-embedded database, as GrGen.NET *does* offer *online* modification of graph-structures, persistence of online modifications is offered by writing to an embedded SQLite, or by recording to a change log that needs to be replayed at next program start (besides writing a full dump from memory at session end, as it is common to batch-wise offline processing).

GrGen.NET is not suited for scenarios where the limits of main memory are exceeded. A well equipped single machine with 128GB of main memory is capable of storing somewhat under about 500 Million nodes plus 1 Billion edges (naked, without attributes and without names), see Chapter 31 for the maths. This should be sufficient for many tasks, but still there are task that grow beyond it. Besides, other things come into play at those sizes: importing such a graph will take some time, maybe longer than you're willing to wait – the GRS importer is munching at some ten thousand elements per second. Databases that work on a backing store and only shuffle data visited in are at an advantage here (amortizing import cost over the course of processing). GrGen.NET scales very well (much better than the tools of the authors from the model transformation community that write papers about scalability) and can easily handle millions of elements, matching a pattern in such a graph within the blink of an eye, but it does not contain dedicated optimizations for very huge datasets.

GrGen.NET is a tool that was built with achieving goals in practice in mind. Despite a shared pair graph grammar inspiration is it unrelated to triple graph grammars and their implicit but unrealistic assumption that typically one graph representation is to be transformed to another, basically equivalent, just somehow a bit differently formulated graph representation, without information loss or enrichment in either direction (bidirectional transformation tasks are very seldom). GrGen.NET can be used for meta-programming (esp. due to its compiler construction roots), but otherwise you work with it at the object and class level, not at the meta, meta-meta, or meta-meta-meta tralala-ding-ding-dong level (the abstraction levels of the true model-based software engineers). When you don't care about expressiveness and performance for real world tasks, but are just interested in pimping your thesis, pick AGG and do some critical pair analysis. GrGen.NET is the most sophisticated implementation of algebraic graph transformation, but it was not created in order to showcase that the theory can be made executable. Neither was it created to prove a visual programming point. GrGen.NET was created because we know from our own experience that a great graph processing language and a supporting environment with visual debugging are of real help for graph-representation-based tasks.

BIBLIOGRAPHY

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In DorinaC. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin Heidelberg, 2010.

[And93] Arne Andersson. Balanced search trees made simple. In *In Proc. 3rd Workshop on Algorithms and Data Structures*, pages 60–71. Springer, 1993.

[Ant25] Anthropic. Claude code. https://code.claude.com, 2025.

[Ass00] Uwe Assmann. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):583–637, 2000.

[Bat05a] Gernot Veit Batz. Graphersetzung für eine Zwischendarstellung im Übersetzerbau. Master's thesis, Universität Karlsruhe, 2005. http://www.info.uni-karlsruhe.de/papers/da_batz.pdf.

[Bat05b] Veit Batz. Generierung von Graphersetzungen mit programmierbarem Suchalgorithmus. Studienarbeit, 2005. http://www.info.uni-karlsruhe.de/papers/sa_batz.pdf.

[Bat06] Gernot Veit Batz. An Optimization Technique for Subgraph Matching Strategies. Technical Report 2006-7, Universität Karlsruhe, Fakultät für Informatik, April 2006. http://www.info.uni-karlsruhe.de/papers/TR_2006_7.pdf.

[BBH+13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and efficient construction of static single assignment form. In Ranjit Jhala and Koen Bosschere, editors, *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin Heidelberg, 2013.

[BFG96] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Issues in the practical use of graph rewriting. In Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, pages 38–55. Springer Berlin Heidelberg, 1996.

[BG09] Paul Bédaride and Claire Gardent. Semantic Normalisation: a Framework and an Experiment. In *Eighth International Conference on Computational Semantics*, 2009.

[BGP18] Guillaume Bonfante, Bruno Guillaume, and Guy Perrier. *Application of Graph Rewriting to Natural Language Processing*. Wiley, 2018.

[BJ11a] Sebastian Buchwald and Edgar Jakumeit. Compiler Optimization: A Case for the Transformation Tool contest. In Gorp et al. [GMR11], pages 6–16. https://arxiv.org/pdf/1111.4737v1.pdf.

[BJ11b] Sebastian Buchwald and Edgar Jakumeit. Saying Hello World with GrGen.NET - A Solution to the TTC 2011 Instructive Case. In Gorp et al. [GMR11], pages 281–294. https://arxiv.org/pdf/1111.4757.pdf.

[BJ11c] Sebastian Buchwald and Edgar Jakumeit. Solving the TTC 2011 Compiler Optimization Case with GrGen.NET. In Gorp et al. [GMR11], pages 42–53. https://arxiv.org/pdf/1111.4742.pdf.

[BKG08] Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In *Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) Proceedings*, 2008. http://www.info.uni-karlsruhe.de/papers/agtive_2007_search_plan.pdf.

[Buc08] Sebastian Buchwald. Erweiterung von GrGen.NET um DPO-Semantik und ungerichtete Kanten. http://www.info.uni-karlsruhe.de/papers/sa_buchwald.pdf, 6 2008. Studienarbeit.

[CMR+99] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *[Roz99]*, volume 1, pages 163–245. World Scientific Publishing Co., Inc., 1999.

[Dew84] A. K. Dewdney. A computer trap for the busy beaver, the hardest-working turing machine. *Scientic American*, 251(2):10–12, 16, 17, 8 1984.

[Dia] DiaGen Developer Team. The Diagram Editor Generator. http://www.unibw.de/inf2/DiaGen/.

[Dör95] Heiko Dörr. *Efficient Graph Rewriting and its Implementation*, volume 922 of *LNCS*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[EHK+99] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout A. and Comparison with Double Pushout A. In Grzegorz Rozenberg, editor, *[Roz99]*, volume 1, pages 247–312. World Scientific Publishing Co., Inc., 1999.

[ER97] Engelfriet and Rozenberg. Node Replacement Graph Grammars. Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, 1997.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer. The AGG Approach: Language and Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *[Roz99]*, volume 2, pages 551–603. World Scientific Publishing Co., Inc., 1999.

[FKP+20] Maribel Fernández, Hélène Kirchner, Bruno Pinaud, Jason Vallet, and Janos Varga. Porgy Strategy Language: User Manual. Research report, Université de Bordeaux, LaBRI ; Inria Bordeaux Sud-Ouest ; King's College London, April 2020.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph grammar language based on the unified modelling language and java. In *Theory and Application of Graph Transformations*, pages 157–167, 2000.

[Fou07] Free Software Foundation. Gnu lesser general public license, 2007.

[Fuj07] Fujaba Developer Team. Fujaba-Homepage. http://www.fujaba.de/, 2007.

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2006.

[GDG08] Tom Gelhausen, Bugra Derre, and Rubino Geiß. Customizing GrGen.NET for Model Transformation. In *GraMoT*, pages 17–24, 2008.

[Gei08] Rubino Geiß. Graphersetzung mit anwendungen im übersetzerbau. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000009876, Nov 2008. Dissertation.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java™Language Specification*. Addison-Wesley Professional, 3 edition, 2005.

[GMR11] Pieter Van Gorp, Steffen Mazanek, and Louis M. Rose, editors. *Proceedings Fifth Transformation Tool Contest*, volume 74 of *EPTCS*, 2011. https://arxiv.org/abs/1111.4407.

[GPB+22] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. Sqlite: Past, present, and future. *Proc. VLDB Endow.*, 15(12):3535–3547, 2022.

[Hac03] Sebastian Hack. Graphersetzung für Optimierungen in der Codeerzeugung. http://www.info.uni-karlsruhe.de/papers/da_hack.pdf, 12 2003. Diplomarbeit.

[HE11] Tassilo Horn and Jürgen Ebert. The GReTL Transformation Language. In *ICMT 2011*, volume 6707 of *LNCS*, pages 183–198, 2011.

[Hel13] Bergen Helms. *Object-Oriented Graph Grammars for Computational Design Synthesis*. PhD thesis, Technische Universität München, 2013.

[HJG08] Berthold Hoffmann, Edgar Jakumeit, and Rubino Geiß. Graph Rewrite Rules with Structural Recursion. 2nd Intl. Workshop on Graph Computational Models (GCM 2008), 2008. http://www.info.uni-karlsruhe.de/papers/GCM2008.pdf.

[HKT14] Tassilo Horn, Christian Krause, and Matthias Tichy. The TTC 2014 Movie Database Case. http://ceur-ws.org/Vol-1305/paper2.pdf, 2014.

[HSESW05] Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. GXL: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 2005.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[Jak08] Edgar Jakumeit. Mit GrGen.NET zu den Sternen – Erweiterung der Regelsprache eines Graphersetzungswerkzeugs um rekursive Regeln mittels Sterngraphgrammatiken und Paargraphgrammatiken. http://www.info.uni-karlsruhe.de/papers/da_jakumeit.pdf, jul 2008. Diplomarbeit.

[Jak11] Edgar Jakumeit. EBNF and SDT for GrGen.NET. http://www.info.uni-karlsruhe.de/software/grgen/EBNFandSDT.pdf, 2011. Presented at AGTIVE 2011.

[Jak14] Edgar Jakumeit. Solving the TTC 2014 Movie Database Case with GrGen.NET. http://ceur-ws.org/Vol-1305/paper17.pdf, 2014.

[JB11] Edgar Jakumeit and Sebastian Buchwald. Solving the TTC 2011 Reengineering Case with GrGen.NET. In Gorp et al. [GMR11], pages 168–180. https://arxiv.org/pdf/1111.4751.pdf.

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen.NET – The expressive, convenient and fast graph rewrite system. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:263–271, 2010. http://dx.doi.org/10.1007/s10009-010-0148-8.

[JBW+14] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Markus Lepper, Arend Rensink, Louis Rose, Sebastian Wätzoldt, and Steffen Mazanek. A survey and comparison of transformation tools based on the transformation tool contest. *Science of Computer Programming*, 85:41 – 99, 2014. http://www.sciencedirect.com/science/article/pii/S0167642313002803.

[KBG+07] Moritz Kroll, Michael Beck, Rubino Geiß, Sebastian Hack, and Philipp Leiß. yComp. http://www.info.uni-karlsruhe.de/software/ycomp, 2007.

[KG07] Moritz Kroll and Rubino Geiß. Developing Graph Transformations with GrGen.NET. http://www.info.uni-karlsruhe.de/papers/agtive_2007_grgennet.pdf, 2007. preliminary version, submitted to AGTIVE 2007.

[KK07] Ole Kniemeyer and Winfried Kurth. The Modelling Platform GroIMP and the Programming Language XL. Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) Proceedings, 2007.

[Kro07] Moritz Kroll. GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen. http://www.info.uni-karlsruhe.de/papers/sa_kroll.pdf, 5 2007. Studienarbeit, Universität Karlsruhe.

[KVAB22] Lothar Kolbeck, Simon Vilgertshofer, Jimmy Abualdenien, and André Borrmann. Graph rewriting techniques in engineering design. *Frontiers in Built Environment*, Volume 7 - 2021, 2022.

[Lin02] Götz Lindenmaier. libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Technical Report 2002-5, Universität Karlsruhe, Fakultät für Informatik, Sep 2002.

[LLMC05] Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. In *Electronic Notes in Theoretical Computer Science*, pages 65–75, 2005.

[LMS99] Litovsky, Métivier, and Sopena. Graph Relabelling Systems and Distributed Algorithms. Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3, 1999.

[MB00] H. Marxen and J. Buntrock. Old list of record TMs. http://www.drb.insel.de/~heiner/BB/index.html, 8 2000.

[Mic02] Microsoft. Windows Forms. https://learn.microsoft.com/en-us/dotnet/desktop/winforms/?view=netframeworkdesktop-4.8&redirectedfrom=MSDN, 2002.

[Mic07] Microsoft. .NET. http://msdn2.microsoft.com/de-de/netframework/aa497336.aspx, 2007.

[MMJW91] Andrew B. Mickel, James F. Miner, Kathleen Jensen, and Niklaus Wirth. *Pascal user manual and report (4th ed.): ISO Pascal standard*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[NC07] Sandeep Nair and Matthew Campbell. A graph topology optimization method for the synthesis of robust and efficient routes. *Proceedings of ICED 2007, the 16th International Conference on Engineering Design*, 01 2007.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Data Flow Analysis*, pages 35–139. Springer Berlin Heidelberg, 1999.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE '00 Proceedings of the 22nd international conference on Software engineering*, pages 742–745, 2000.

[NPDH07] Lev Nachmanson, Sergey Pupyrev, Tim Dwyer, and Ted Hart. MSAGL. https://www.microsoft.com/en-us/research/project/microsoft-automatic-graph-layout/, 2007.

[Ora96] Oracle. JAVA. https://www.java.com/en/, 1996.

[oT87] Massachusetts Institute of Technology. Mit license, 1987.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[PKR+09] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fiona A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 162–171, Washington, DC, USA, 2009. IEEE Computer Society.

[Plu09] Detlef Plump. The Graph Programming Language GP. In *Proc. Algebraic Informatics (CAI 2009)*, pages 99–122, 2009.

[PT90] Frances Paulisch and Walter Tichy. Edge: An extendible graph editor. *Softw., Pract. Exper.*, 20:S1/63–S1/88, 06 1990.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. Applications of Graph Transformation with Industrial Relevance (AGTIVE '03) Proceedings, 2004.

[Roz99] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999.

[RVG08] Arend Rensink and Pieter Van Gorp. Graph-based tools: The contest. Proceedings 4th Int. Conf. on Graph Transformation (ICGT '08), 2008.

[SAI+90] Herbert Schildt, American National Standards Institute, International Organization for Standardization, International Electrotechnical Commission, and ISO/IEC JTC 1. *The annotated ANSI C standard: American National Standard for Programming Languages C: ANSI/ISO 9899-1990*. McGraw-Hill, Inc., 1990.

[San95] Georg Sander. VCG Visualization of Compiler Graphs—User Documentation v.1.30. Technical report, Universität des Saarlandes, 1995.

[Sch08] Jochen Schimmel. Parallelisierung von Graphersetzungssystemen. https://ps.ipd.kit.edu/downloads/da_2008_jochen_schimmel.pdf, November 2008. Diplomarbeit.

[SGS09] Jochen Schimmel, Tom Gelhausen, and Christoph A. Schaefer. Gene Expression with General Purpose Graph Rewriting Systems. In *Proceedings of the 8th GT-VMT Workshop*, 2009.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[SWZ99] A. Schürr, A. Winter, and A. Zündorf. Progres: Language and environment. In G. Rozenberg, editor, *Handbook on Graph Grammars*, volume Applications, Vol. 2, pages 487–550. World Scientific, 1999.

[Sza05] Adam M. Szalkowski. Negative Anwendungsbedingungen für das suchprogrammbasierte Backend von GrGen. http://www.info.uni-karlsruhe.de/papers/sa_szalkowski.pdf, 2005. Studienarbeit, Universität Karlsruhe.

[Tea07] The Mono Team. Mono. http://www.mono-project.com/, 2007.

[The17] The Neo4j Team. The neo4j manual. http://www.neo4j.org/, 2017.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the intermediate representation firm. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik, Dec 1999.

[VB07] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming*, 68(3):214–234, 2007.

[VHV08] Gergely Varró, Ákos Horváth, and Dániel Varró. Recursive Graph Pattern Matching. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 456–470. Springer Berlin Heidelberg, 2008.

[VRDJ95] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[VSV05] G. Varró, A. Schürr, and D. Varró. Benchmarking for Graph Transformation. Technical report, Department of Computer Science and Information Theory, Budapest University of Technology and Economics, March 2005.

[VVF06] Gergely Varró, Dániel Varró, and Katalin Friedl. Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In G. Karsai and G. Taentzer, editors, *GraMot 2005, International Workshop on Graph and Model Transformations*, volume 152 of *ENTCS*, pages 191–205. Elsevier, 2006.

[Wei88] David Weininger. Smiles, a chemical language and information system. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

[WKR02] Winter, Kullbach, and Riediger. An Overview of the GXL Graph Exchange Language. Software Visualization - International Seminar Dagstuhl Castle, 2002.

[yWo07] yWorks. yFiles. http://www.yworks.com, 2007.

INDEX

Keywords

IteratedFiltering, 103
SubpatternOccurrence, 95
abstract, 28
actions, 320–322
adaptvariables, 322
add, 312, 318, 330, 338–340, 357, 451
alternative, 75
analyze, 320
arbitrary, 28
askfor, 314
attributes, 358, 362
auto, 203, 206, 258
backend, 319
bordercolor, 337
break, 144, 357, 358
by, 339
case, 143
cd, 317
class, 29, 212, 213, 264, 447, 450
clear, 316
color, 337, 338
connect, 30
const, 28, 32
contains, 357
continue, 144, 357, 358
copy, 30, 450
custom, 320–322
dangling, 43, 125
debug, 346, 349–351, 357, 358
def, 237, 313
delete, 52, 95, 316, 319, 358
directed, 28
disable, 350
do, 143
dpo, 43, 125
dump, 337–340
dumpsourcecode, 322
echo, 308
edge, 29, 308, 316, 338, 339
edges, 362
else, 143, 316
emit, 133, 318, 357, 450
emitdebug, 133
emithere, 135
emitheredebug, 135
enable, 350
enableassertions, 322
endif, 316
endsWith, 357
enum, 33
equals, 357
eval, 52, 313, 351
exact, 43, 47, 125
exclude, 338, 340
exec, 133, 312, 317, 351, 363
exit, 308
exitonfailure, 363
explain, 321
export, 328, 329
extends, 29, 30, 212, 213, 447
external, 447–450
filter, 202, 449
for, 144, 160, 182, 241, 260, 274, 382
from, 331
function, 146, 448
gensearchplan, 321
get, 349, 350
graph, 309, 312, 316, 319–321, 327, 335, 337, 340, 365
group, 339
gshow, 365
gui, 346
halt, 357
help, 308
hidden, 339
highlight, 357
hom, 47, 49
identification, 43, 125
if, 47, 119, 143, 316, 358
implements, 44
import, 329, 330
in, 144, 160, 161, 164, 167, 171, 182, 228, 241, 312, 382
include, 43, 316

independent, 49, 72
induced, 43, 47, 125
infotag, 339
is, 363
iterated, 73
keepdebug, 318, 451
labels, 337, 338
layout, 350
lazynic, 318
linestyle, 338
ls, 317
match, 264, 358
mode, 350, 351
model, 365
modify, 51, 81, 89, 92
multiple, 73
nameof, 395
negative, 70
new, 215, 309, 310, 312, 318, 319, 358, 451
node, 29, 308, 316, 337–339
nodebugevents, 318
nodes, 362
noevents, 318
noinline, 318
null, 45
num, 362
object, 308
off, 317, 318, 337, 338, 451
on, 317, 318, 337, 338, 357, 358, 451
only, 337–339, 358, 362, 363
optimizereuse, 321
optimizereusepoolsize, 321
option, 340, 346, 350
optional, 73
options, 349, 350
pattern, 81
patterns, 365
persist, 323
procedure, 147, 448
profile, 313, 318
pwd, 317
quit, 308
randomseed, 317
record, 331
recordflush, 331
redirect, 316, 318
ref, 45, 146, 147
reference, 318, 451
rem, 357
replace, 51, 81, 89, 92
replay, 331
reset, 340
return, 48, 146, 147
retype, 315, 358
rule, 43
save, 321, 327
scan, 137
select, 319, 320
set, 318, 319, 337, 338, 346, 350, 351, 358, 451
setmaxmatches, 322
shape, 337
shortinfotag, 339
show, 313, 314, 319, 320, 335, 362, 363
silence, 317
specified, 363
start, 331
startsWith, 357
statistics, 319, 321
stop, 331
strict, 363
sub, 362
super, 362
switch, 143
test, 43
textcolor, 337, 338
thickness, 338
this, 189
time, 317
to, 323, 331
transient, 308
tryscan, 137
twopane, 346
type, 363
typeof, 62, 127
types, 362
undirected, 28
unique, 213, 397
using, 43
validate, 31, 363
valloc, 196
var, 45, 146, 147, 314
vfree, 196
vfreenonreset, 196
visited, 195, 234, 238
vreset, 196
while, 143
with, 323, 339, 346
xgrs, 312, 351, 363
yield, 102, 234

Non-Terminals

ActionDeclaration, 43
ActionSignature, 44

AdvancedEdgeTypeConstructs, 124
AdvancedModelDeclarations, 212
AdvancedNodeTypeConstructs, 124
AlternativePattern, 75
Arguments, 146
ArithmeticSequenceExpression, 236
ArrayConstr, 311
ArrayConstructor, 167
ArrayCopyConstructor, 167
ArrayExpr, 167
ArrayOperator, 167
Assignment, 140
AssignmentTarget, 151, 234
AttributeDeclaration, 32
AttributeIndexDecl, 379
AttributeInitialization, 132
AttributeInitializationList, 132
AttributeOverwrite, 32
AttributeType, 32
AttributeValue, 311
Attributes, 311
AutoFunctionBody, 258
AutoMatchClassBody, 258
BasicSequenceExpression, 237
BitwiseSequenceExpression, 237
BoolExpr, 57
BooleanSequenceExpression, 235
CastExpr, 68
ChangeAssignment, 162
ClassDeclaration, 28
CloneOperator, 130
Command, 308
CompoundAssignment, 161, 164, 167, 171
Computation, 232
ConditionalSequenceExpression, 235
ConnectionAssertion, 30
Constant, 64
Constructor, 311
ContainerSequenceExpression, 240
Continuation, 38
CopyCompareDeclaration, 450
CopyOperator, 130
Decision, 143
DefVariableDecl, 98
DequeConstr, 311
DequeConstructor, 171
DequeCopyConstructor, 171
DequeExpr, 171
DequeOperator, 171
DoWhileLoop, 143
DumpEdgeContinuation, 338
DumpNodeContinuation, 337
EarlyExit, 144
EdgeClass, 29
EdgeRefinement, 40
EmitParseDeclaration, 450
Entity, 307
EntityKind, 308
EntityType, 308
EnumDeclaration, 33
EqualsAnyParallelization, 280
ExecStatement, 133
Expression, 56
ExtendedControl, 119
ExternalClassDeclaration, 447
ExternalFunctionDeclaration, 448
ExternalProcedureDeclaration, 448
ExternalSequenceDeclaration, 449
FileHeader, 41
FilterCall, 206, 207
FilterCalls, 206
FilterDecl, 203
FilterFunctionDefinition, 202, 449
FiltersDecl, 203
FloatExpr, 60
ForContainerHeader, 160
ForContainerWithIndexVarHeader, 160
ForFunctionHeader, 182
ForIndexFunctionHeader, 382
ForLoop, 144
FunctionCall, 146
FunctionDefinition, 146
FunctionMethodCall, 219
FunctionMethodDefinition, 219
FunctionParallelization, 279
GlobalVarDecl, 43
GraphElement, 307
GraphModel, 28
GraphRewriteSequence, 312, 351
GraphRewriteSequenceExpression, 313
Graphlet, 38
GraphletEdge, 40
GraphletNode, 39
GroupConstraints, 339
HomomorphySpecification, 49
IdentDecl, 451
IfElse, 143
Implements, 44
IncAdjTypeConstraints, 339
IndexAccess, 380
IndexAccessLoopSequence, 383
IndexArguments, 380
IndexFunctionCall, 380, 385
IndexedAssignemt, 164, 167, 171

InputTypeSpecification, 45
IntExpr, 59
IsoConstraint, 49
IteratedQuery, 67
Literal, 64
LocalVariableDecl, 141
LockStatement, 282
MapConstr, 311
MapConstructor, 164
MapCopyConstructor, 164
MapExpr, 164
MapOperator, 164
MappingClause, 275
MatchClassConstructor, 264
MatchClassDeclaration, 264
MatchClassFilterCall, 206, 207, 265
MatchClassFilterCalls, 206, 265
MatchClassSpec, 255
MemberAccess, 65, 201
Merging, 130
MessageFilter, 357
MethodCall, 161, 164, 167, 171
MethodSelector, 61, 250
MethodTarget, 222
ModelUsage, 43
MultiBacktracking, 273
MultiRuleExecution, 262
MultiRulePrefixedSequence, 262
MultiRulePrefixedSequenceBacktracking, 273
MultiRuleQuery, 255
MultipleIndexAccess, 385
MultipleIndexArguments, 385
MultiplicityConstraint, 30
NamedIndexAccessExpr, 395
NameofAssignment, 395
NegativeApplicationCondition, 70
NestedPattern, 70
NestedPatternWithCardinality, 73
NestedRewriting, 89
NewOperator, 215
NodeClass, 29
NodeConstraint, 30
Object, 307
ObjectAttributeInitialization, 215
ObjectAttributeInitializationList, 215
ObjectClass, 212
ObjectConstructor, 312
OutputAssignment, 151
PackageDefinitionAction, 226
PackageDefinitionModel, 225
PackageUsageActions, 226
PackageUsageModel, 225
Parameter, 45, 146, 147
Parameters, 306
Pattern, 45
PatternStatement, 47
PerElementClause, 207, 250
PersistenceSuffix, 323
PersistentGraphParameters, 327
PositiveApplicationCondition, 72
PrimaryExpr, 63
PrimarySequenceExpression, 237
PrimitiveAttributeValue, 311
ProcedureCall, 151
ProcedureDefinition, 147
ProcedureMethodCall, 222
ProcedureMethodDefinition, 222
RHSPatternDefVariableDecl, 104
RandomSel, 115
RangeConstraint, 30
Redirect, 131
ReferenceToContainingGraphDecl, 228
RelationalExpr, 57
RelationalSequenceExpression, 236, 240
Repetition, 143
ReturnAssignment, 115
ReturnStatement, 48, 146, 147
ReturnTypes, 45
Retyping, 128
Rewrite, 51
RewriteFactor, 114, 231, 241
RewriteNegTerm, 116
RewriteSequence, 116
RewriteSequenceDefinition, 269
RewriteSequenceSignature, 269
RewriteStatement, 52
RewriteTerm, 119
Rule, 115, 228
RuleDeclaration, 43
RuleExecution, 115, 260
RuleExecutionAtom, 262
RuleModifier, 115
RulePrefixedSequence, 261
RulePrefixedSequenceAtom, 261
RuleQuery, 251
RuleSet, 41
RulesInclusion, 43
Script, 308
SequenceExpression, 235
SequenceParallelization, 281
SequencesList, 263
SetConstr, 311
SetConstructor, 161
SetCopyConstructor, 161

SetExpr, 161
SetOperator, 161
SimpleVariableHandling, 118
SpacedParameters, 306
SpecialSequenceExpression, 238
Statement, 140
StorageAccess, 175
StringExpr, 61
StringSequenceExpression, 237
SubpatternBody, 81
SubpatternDeclaration, 81
SubpatternEntityDeclaration, 82
SubpatternExecEmit, 135
SubpatternRewriteApplication, 92
SubpatternRewriting, 92
SwitchCase, 143
TestDeclaration, 43
TransientObject, 307
TransientObjectClass, 213
Type, 127
TypeConstraint, 127
TypeExpr, 62
TypeNameSpec, 358
UniqueConstraintDecl, 397
UniqueIndexAccessExpr, 397
UniqueIndexDecl, 397
UniqueObjectConstraintDecl, 213
VisitedAssignment, 195
VisitedFlag, 195
WhileLoop, 143
YieldExpression, 102

General Index

.grg, 368
.grs, 369
`!`, 121, 317, 450
`*`, 121, 307
`+`, 121
`;;`, 308
`;>`, 121
`<;`, 121
`<<;>>`, 267, 277
`<>`, 267, 277
`@`, 307
`[]`, 121
`#§`, 369
`#`, 306
`$<number>`, 38
`$<op>`, 121
`$`, 311
`%`, 312
`&&`, 121
`&`, 121
`^`, 121
`clone`, 201
`||`, 121
`|`, 121

accumulation, 248
action, *see* graph rewrite sequence
action command, 319
adjacent, 46
advanced control, 269
advanced matching and rewriting, 123
AEdge, 27
all bracketing, 120
alternative patterns, 75
analyzing graph, 320
annotation, 26, 38, 451
anonymous, 38, 49, 50, 305
API, 10, 371, 427
application, 4, 47
application programming interface, *see* API
arbitrary, 27, 40
arborescence, 468
array, 159
assertion, 198, 322
assignments, 140
attribute, 32, 311, 312, 314, 315, 339, 362
attribute condition, 47
attribute evaluation, *see* computation, 139, 140
attribute initialization, 132
auto-generated filters, 203
automorphic, 204
automorphic pattern, 272

backend, 55, 159, 319, 320
backslash, 308
backtracking, 270
binding of names, 38
block nesting, 51
boolean, 55
bounded iterated path, 83, 186
breadth-first search, 139, 187
break, 75
break point, 115, 118, 232, 353
building GrGen, 453
built-in generic types, *see* generic types
built-in types, *see* primitive types
busy beaver, 421
by-reference, 45
by-value, 45
byte, short, int, long, 55

cardinality, *see* pattern cardinality

case sensitive, 26, 38, 306
choice point, 115, 117, 118, 263, 264, 353
clone, 129, 201, 217
code generator, 471
color, 337
command line, 317
comment, 306
compare structure, *see* subgraph comparison
compatible types, *see* type-compatible
compiler graph, *see* layout algorithm
computation, 52
computation definition, 145
computations, 139
conditional sequence, 119
connection assertion, 29, 30, 363
constructor, 305, 311
containment, 451
continuation, *see* graphlet
controlflow, 141
copy, 129, 217
copy structure, *see* subgraph copying
count, 101, 260

dangling condition, 126
data flow analysis, 297
database transactions, 275
debug mode, 350
Debugger, 335
debugger, 351, 352
declaration, 26, 28, 47
def, 104
default graph model, 37
default search plan, 321
default value, 311
definition, 26, 451
deletion, 50, 52
depth-first search, 139, 187
deque, 159
determinism, *see* non-determinism
Developing GrGen.NET, 453
directed, 40
DOT, 335, 336
double, 55
double-pushout, 488
DPO, *see* double-pushout approach
dumping graph, 327
dynamic type, 127

EBNF, *see* rail diagram, *see* regular expression syntax
Edge, 27, 29
edge (graphlet), 40
edge type, 29
edgeByName, 395
edgeByUnique, 397
edNCE, *see* node replacement grammar
emit, 198
emitdebug, 198
empty pattern, 5, 45
enum item, 33, 56
enum type, 33, 55
evalhere, 104, *see* ordered evaluation
evaluation, *see* attribute evaluation
exact dynamic type, *see* dynamic type
example, 5, 17, 419, 421
export, 328, 329, 434
expression, 56
expression variable, 63, *see* name, 310
extensions, 447
external, 450
external class, 212
external class implementation, 440
external filter function, 449
external function, 212, 448
external function implementation, 440
external match filter implementation, 441
external procedure, 448
external procedure implementation, 440
external sequence, 449
external sequence implementation, 441
extMSAGLExt, 349, 372

features, 12
filter function definition, 202
filters, 201
float, 55
flow equations, 297
functions, 139

generated code, 455
generic types, 159
GGrShell, 370
GGrShell.exe, 370
graph, 179
graph functions, 179
graph global variable, 43, 113, 305
graph isomorphy checking, 192
graph model, 25, 28, 37, 43, 211, 309
graph model language, 25
graph procedures, 179
graph rewrite rules, 4
graph rewrite script, 316, 327, 369
graph rewrite sequence, 47, 113, 312, 351, 363
graph rewrite sequence definition, 313
graph rewrite sequence expression, 313, 351
graph rewriting, 4

graph-oriented programming, 211
graphlet, 38, 47, 49, 52
graphof, 228
graphviz, 336
GrGen.exe, 367
group node, 339
GRS, *see* graph rewrite sequence, *see* graph rewrite sequence
GrShell, 11, 28, 305, 361, 369
GrShell variable, *see* graph global variable, *see* graph global variable
GrShell.exe, 369

hierarchic, *see* group node, *see* layout algorithm
homomorphic matching, 39, 48
host graph, 4, 49, 319
host graph sensitive search plan driven graph pattern matching, 468

identification condition, 126
identifier, 26, 38
imperative, *see* attribute evaluation
imperative statements, 133
imperativeandstate, 133, 285
import, 434
indeterministic choice, 263
indices, 377
info tag, 339
inheritance, 26, 29, 212, 213, 362
inlining, 404, 476
internal graph representation, 455
internals, 453
iso constraint, 49
isomorphic matching, 48
isomorphy checking, 192
iterated path, 83, 186
iterated query, 67, 102
iterative deepening, 139, 187

Kantorowitsch tree, 354
Koch snowflake, 419

label, 305, 339
layout, *see* visualization, *see* layout algorithm
layout algorithm, 350, 373, 419
left-hand side, 4, 49
LGPL, 10, 367
LGSPBackend, 55, 159, 319, 320
lgspBackend, 371
LHS, *see* left-hand side
libGr, 28, 305, 371
local variable, 104
loop, 119

map, 159
match, 5
match type, 201
matching strategies, 486
maybeDeleted, 451
merge, 124, 130
merge node, 286
methods, 211
MIT license, 345
modifier, 124
modify mode, 50
MSAGL, 345
multi rule application language, 261
multiplicity, *see* connection assertion

NAC, *see* negative application condition
name, 38, 49, 310
name initialization, 132
named nested pattern, 79
nameof, 395
negative application condition, 70
nested graph, *see* group node, 335
nested layout, *see* group node, 335
nested pattern rewrite, 89
nested patterns, 69
nested transaction, *see* transaction
Node, 29
node (graphlet), 39
node replacement grammar, 290
node type, 29
nodeByName, 395
nodeByUnique, 397
non-determinism, 47

Object, 212
object, 55
object attribute initialization, 215
object constructor, 312
object type, 212
object-oriented programming, 211
one-of-set braces, 263
optimization, 399
order of precedence, 57, 60, 68, 161, 165, 169, 172
organic, *see* layout algorithm
orthogonal, *see* layout algorithm
overview, system, 9

PAC, *see* positive application condition
package definition, 225, 226
parallelization, 279
parallelize, 451
parameter, 44, 312, 319

path, 83
pattern, 45
pattern cardinality, 73
pattern graph, 4, 38, 45
pattern matching implementation, 456
pattern modifiers, 125
pattern part, 45
pause insertion, 272
pause insertions, 270
performance, 399
persistence provider transaction, 275
persistent graph, 309, 323
persistent name, 237, 305, 312, 362
positive application condition, 72
pragma, *see* annotation
precedence, *see* order of precedence
preservation, 50
preservation morphism, 4, 49
primitive types, 55
prio, 451
procedures, 139
pushdown machine, 459

quickstart, 17

rail diagram, 26
random match selector, 115
random-all-of operators, 263
re-evaluation, *see* attribute evaluation
reachability, 297
record, 198, 331
recursive pattern, 82
redefine, 38
redirect, 131
redirecting, 40
regular expression syntax, 86, 119
replace mode, 50
replay, 331
return type, 44
return value, 48
retype, 124, 315
retyping, 128
rewrite, 49
rewrite graph, 4, 38, 49
rewrite part, 49
rewrite rule, 43
RHS, *see* right-hand side
right-hand side, 4, 49
ringlists, 455
rule application, *see* application
rule application language, 113
rule language, 37
rule modifiers, 125
rule set, 37, 41, 320
rule set language, 37

schedule, 468
scope, 39, 70, 72
search plan, 320, 425, 452, 468, 486
search program, 456, 468
search space, 270
search state space stepping, 459
sequence call, 269
sequence computations, 231
sequence constant, 118
sequence definition, 269, 313
sequence local variable, 113
set, 159
shell, *see* GrShell
short info tag, 339
Sierpinski triangle, 419
signature, 44
simulation, 353
single-pushout approach, 50, 488
some-of-set braces, 263
spanning tree, 84, 294, 296
split node, 286
SPO, 125
spot, 5, 38
stack machine, 459
state space, 270
state space enumeration, 302
storage, 159, 239
storage access, 174
string, 55
structural recursion, 82
subgraph comparison, 294
subgraph copying, 295
subpattern, 81
subpattern declaration, 81
subpattern entity declaration, 82
subpatterns, 81
subrule, 91
subsequences, 269
symmetric matches, 204
symmetry reduction, 205, 272
syntax diagram, *see* rail diagram

test, 43, 46
this, 189
traceability, 292, 296
transaction, 270
transformation in a narrow sense, 292
transformation specification, 49
transient object type, 213
TransientObject, 213

Turing complete, 421
type cast, 55, *see* retyping
type constraint, 124, 127
type expression, 38, 57, 62
type hierarchy, 27, 32, 127, 128, 214
type-compatible, 363

UEdge, 27
UML class diagram, 59
undefined variables, 312
undirected, 40
uniqueof, 397
user input assignment, 118

V-Structure, 192, 470
validate, 363
validityCheck, 451
variable, *see* expression variable, 63, *see* graph
    global variable, *see* sequence local variable, 305, 310, 312
Varró's benchmark, 486
VCG, 11, 335, 337, 372
visited access, 195
visited flag, 179
visualization, *see* group node, 335

weighted one operator, 264
working graph, 319
worklist, 300

yComp, 11, 335, 345, 352, 372
yComp.jar, 352
yield, 98, 105
yielding, 97
yielding outwards, 104