



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 2.7 milestone 3

—DRAFT—

www.grgen.net



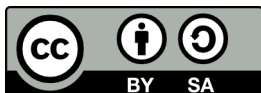
Jakob Blomer Rubino Geiß
Edgar Jakumeit

January 17, 2011

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 step-wise debugging. The Graph Rewrite Generator is a tool enabling elegant and convenient development of graph rewriting applications with comparable performance 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

Since the last version of this manual which was written for GRGEN.NET v1.4 a lot has happened, as can be seen quite easily in the fact that this manual describes GRGEN.NET v2.6. The porting of C to C# [Kro07] allowed for a faster pace of development, which yielded alternatives and subpatterns allowing for structural recursion [Jak08, HJG08], undirected edge support plus fine grain pattern conditions [Buc08], a data model that is more user friendly at the API, support for visited flags, and an prototypical implementation of an embedding of GRGEN.NET as a domain specific language into C# [Kro] – resulting in GRGEN.NET v2.0.

Then Dr. Rubino Geiß finished his dissertation [Gei08] and left; Prof. Goos retired. The succeeding Professor had no commitment to graph rewriting, so GRGEN.NET switched from a university project developed by students in their bachelor/masters thesis's to an open source project (which is still hosted at the IPD, reachable from www.grgen.net).

But development continued: With the introduction of generic set and map types in the model language to facilitate uses in computer linguistics and in the rule control language to allow for more concise rule combinations. With the 2+n pushdown automaton for matching patterns with structural recursion extended to handle pattern cardinality specifications and positive applications conditions [JBK10]. With massive API improvements, now featuring an interface of typed, named entities in addition to the old name string and object based interface. With the introduction of importers and exporters for GXL (the quasi standard graph exchange format in graph rewriting — but only in theory), and for GRS, a much easier and less chatty format.

Most of these features were introduced due to feedback from users and use cases: We want to thank the organizers of GraBaTs [RVG08], 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, for his “french”-courses and the ideas about how to handle program graphs. And thanks to several early users giving valuable feedback or even code (*code is of course the best contribution you can give to an open source project*), by name: Tom Gelhausen and Bugra Derre (you may have a look at <https://svn.ipd.uni-karlsruhe.de/trac/mx/wiki/Home> for their work at the other IPD), Paul Bedaride, Normen Müller, Pieter van Gorp and Nicholas Tung.

Regarding questions please contact the GRGEN.NET-Team via email to [grgen](mailto:grgen@ipd.info.uni-karlsruhe.de) at the host given by ipd.info.uni-karlsruhe.de.

We hope you enjoy using GRGEN.NET even more than we enjoyed developing it (it was fun but aging projects with code traces from many people are not always nice to work with ;).

Thank you for using GRGEN.NET.

Karlsruhe in January 2011, 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 and when not	1
1.3	Features of GRGEN.NET	2
1.4	System Overview	4
1.5	What is Graph Rewriting?	5
1.6	An Example	5
1.7	The Tools	7
1.7.1	GrGen.exe	7
1.7.2	GrShell.exe	8
1.7.3	LibGr.dll	9
1.7.4	yComp.jar	9
1.8	Development goals	9
2	Quickstart	13
2.1	Downloading & Installing	13
2.2	Creating a Graph Model	14
2.3	Creating Graphs	14
2.4	The Rewrite Rules	16
2.5	Debugging and Output	18
3	Graph Model Language	19
3.1	Building Blocks	19
3.1.1	Base Types	21
3.2	Type Declarations	21
3.2.1	Enumeration Types	22
3.2.2	Node and Edge Types	23
3.2.3	External Attribute Types	26
3.2.4	External Function Types	26
3.2.5	Attributes and Attribute Types	27
4	Rule Set Language	29
4.1	Building Blocks	29
4.1.1	Graphlets	30
4.2	Rules and Tests	33
4.3	Pattern Part	37
4.4	Replace/Modify Part	40
4.4.1	Implicit Definition of the Preservation Morphism r	40
4.4.2	Specification Modes for Graph Transformation	41
4.4.3	Syntax	42
4.5	Rule and Pattern Modifiers	44

4.6	Static Type Constraint and Exact Dynamic Type	45
4.6.1	Static Type Constraint	45
4.6.2	Exact Dynamic Type	46
4.7	Retrying and Copying	47
4.7.1	Retrying	47
4.7.2	Copy	48
4.8	Annotations	49
5	Types and Expressions	51
5.1	Built-In Types	51
5.2	Expressions	52
5.3	Boolean Expressions	53
5.4	Relational Expressions	53
5.5	Arithmetic and Bitwise Expressions	55
5.6	String Expressions	56
5.7	Set Expression	57
5.8	Map Expression	58
5.9	Type Expressions	59
5.10	Primary expressions	59
5.11	Operator Priorities	63
6	Nested and Subpatterns	65
6.1	Negative Application Condition (NAC)	66
6.2	Positive Application Condition (PAC)	68
6.3	Pattern Cardinality	68
6.4	Alternative Patterns	70
6.5	Subpattern Declaration and Subpattern Entity Declaration	71
6.5.1	Recursive Patterns	73
6.6	Nested Pattern Rewriting	76
6.7	Subpattern rewriting	78
6.7.1	Deletion and Preservation of Subpatterns	81
6.8	Regular Expression Syntax	83
7	Rule Application Control Language (XGRS)	85
7.1	Logical and sequential connectives	86
7.2	Loops	87
7.3	Rule application	88
7.4	Variable handling	90
7.5	Extended control	91
7.6	Quick reference table	92
8	Embedded Sequences and Storages	95
8.1	Imperative Statements	95
8.1.1	Exec and emit in rules	95
8.1.2	Deferred exec and emit here in nested and subpatterns	96
8.2	Reading, writing, and managing state	98
8.2.1	Storage and visited flag handling in the sequences	99
8.2.2	Storage and visited flag access in the rules	103
8.3	Merge, Split, and Node Replacement Grammars	105
8.4	Subgraph copying and Reachability via Flow Equations	111

9	GrShell Language	117
9.1	Building Blocks	117
9.2	GRSHELL Commands	119
9.2.1	Common Commands	119
9.2.2	Graph Commands	121
9.2.3	Validation Commands	122
9.2.4	Graph Input and Output Commands	124
9.2.5	Graph Change Recording and Replaying	126
9.2.6	Graph Manipulation Commands	126
9.2.7	Graph and Model Query Commands	129
9.2.8	Graph Visualization Commands (Nested Layout)	131
9.2.9	Action Commands (XGRS)	135
9.3	Graphical Debugger	136
9.3.1	Debugging Related Commands	136
9.3.2	Using the Debugger	136
9.4	Backend Commands	138
9.4.1	Backend selection and custom commands	139
9.4.2	LGSPBackend Custom Commands	139
10	Examples	141
10.1	Fractals	141
10.2	Busy Beaver	143
10.2.1	Graph Model	143
10.2.2	Rule Set	143
10.2.3	Rule Execution with GRSHELL	145
11	Application Programming Interface	149
11.1	Interface to the host graph	149
11.2	Interface to rules	150
11.3	Import/Export and miscellaneous stuff	152
11.4	External Class and Function Implementation	154
11.5	How to build.	155
11.6	A very short tour of the code	155
	Bibliography	157
	Index	161

CHAPTER 1

INTRODUCTION

1.1 What is GRGEN.NET?

GRGEN (Graph Rewrite GENERator) is a generative programming system for graph rewriting, which considerably eases the transformation of complex graph structured data, comparable to other programming tools like parser generators which ease the task of formal language recognition, or databases which ease the task of persistent data storage and querying.

It is combined from two groups of components: The first consists of the compiler **grgen** – transforming declarative graph rewrite rule specifications into highly efficient .NET-assemblies – and the execution environment **libGr**, which offer the basic functionality of the system. The second consists of the interactive command line **GrShell** and the graph viewer **yComp**, which offer a rapid prototyping environment supporting graphical and stepwise debugging of programmed rule applications.

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.net. Starting as a compiler construction helper tool it has grown into a software development tool for general purpose graph transformation, which offers the highest combined speed of development and execution for graph based algorithms through its declarative languages with automatic optimization.

1.2 When to use GRGEN.NET and when not

You may be interested in using GRGEN.NET if you have to tackle the task of transforming meshes of massively linked objects, i.e. graph-like data structures, as is the case in e.g. model transformation, computer linguistics, or modern compiler construction (any time there is more than one relation of interest in between the data entities your algorithm operates upon). 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 for you by GRGEN.NET. You specify your transformation task on a higher level of nodes connected by edges, and rewrite rules of patterns to be searched plus modifications to be carried out, and then let GRGEN.NET generate the algorithmic core of your application.

There is nothing to gain from GRGEN.NET if scalars, lists and trees are sufficient to *adequately* 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 graphs, but aren't because of the cost of maintaining graph structures by hand). 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 tool for software engineers. 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 (the editor generator DiaGen[Dia] may be of interest in this case).

1.3 Features of GRGEN.NET

The process of graph rewriting can be divided into four steps: Representing a graph according to a model (creating an instance graph), 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. We have organized the presentation of the features of the GRGEN.NET languages according to this breakdown of graph rewriting:

- The graph model (meta-model) language supports:
 - Typed nodes and edges, with multiple inheritance on types
 - Directed multigraphs (including multiple edges of the same type)
 - Undirected and arbitrarily directed edges
 - Node and edge types can be equipped with typed attributes (like structs) including powerful set and map types
 - Connection assertions to restrict the “shape” of graphs
 - Turing complete language for checking complex conditions
- The pattern language supports:
 - Plain isomorphic subgraph matching (injective mapping)
 - Homomorphic matching for a selectable set of nodes / edges, so that the matching is not injective
 - Attribute conditions (e.g. arithmetic-,boolean-,string- or set-expressions on the attributes)
 - Type conditions (including powerful instanceof-like type expressions)
 - Nested patterns, specifying negative and positive application conditions as well as iterated, optional, or alternative structures
 - Subpatterns for pattern reuse, allowing via recursion to match substructures of arbitrary depth (e.g. iterated paths) and breadth (e.g. multinodes)
 - Parameter passing to rules
- The rewrite language supports:
 - Keeping, adding and deleting graph elements according to the SPO approach
 - Choosing out of three additional rule application semantics: DPO or exact patterns only or induced subgraphs only
 - Attribute re-/calculation (assigning the result of e.g. arithmetic expressions to the attributes)
 - Retyping of nodes/edges (a more general version of casts known from common programming languages)
 - Creation of new nodes/edges of only dynamically known types or as exact copies of other nodes/edges

- Two modes of specification: A rule can either express changes to be made to the match or replace the whole match
 - A rewrite part for the nested patterns and subpatterns, so that complex structures can not only get matched, but also get rewritten
 - Embedded graph rewrite sequences capable of calling other rules (with access to the nodes/edges of the rule)
 - Emitting user-defined text to `stdout` or files during the rewrite steps
 - Visited flags and storages to communicate between rule applications using state
 - Parameter passing out of rules
- The rule application control language (grs: graph rewrite sequences) supports:
 - Logical and sequential connectives
 - Loops
 - Rule execution
 - Variable handling
 - Extended control (e.g decisions, transactions, backtracking, indeterministic choice)
 - Visited flags and storages management

These were the features of the core of GRGEN.NET-System, the generator `grgen.exe` and its languages plus its runtime environment `libGr`. In addition, the GRGEN.NET system offers a shell application, the GRSHELL, which features commands for

- graph management
- graph validation
- graph input and output
- graph change recording and replaying
- graph manipulation
- graph and model queries
- graph visualisation (including *nested* graphs keeping large graphs understandable)
- action execution
- *debugging*
- backend selection and usage

The debugging and graph visualisation commands are implemented in cooperation with the graph viewer YCOMP. Alternatively to GRSHELL, you can access the match and rewrite facility through LIBGR. This way you can build your own algorithmic rule applications in a .NET language of your choice.

1.4 System Overview

Figure 1.1 gives an overview of the GRGEN.NET system components.

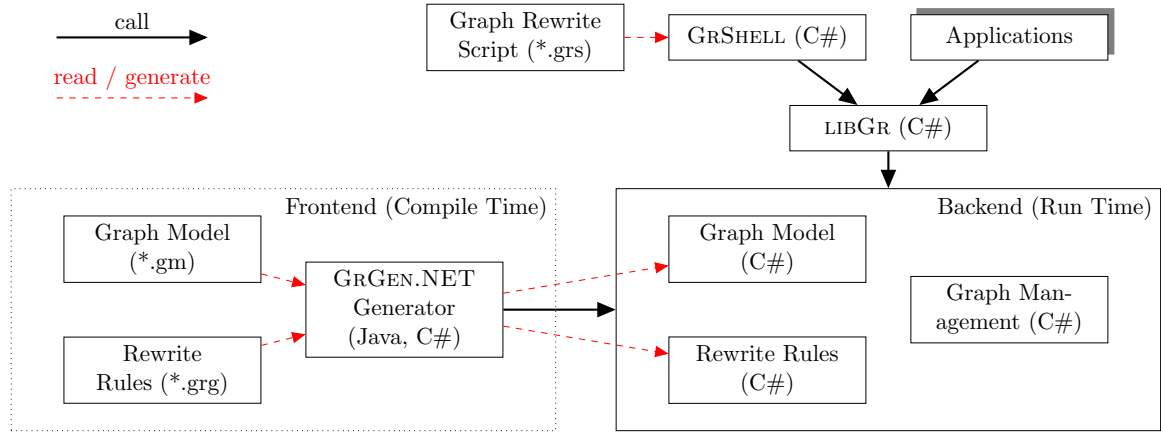


Figure 1.1: GRGEN.NET system components [Kro07]

A graph rewrite system¹ is defined by a rule set file (*.grg, which may include further rule set files) and zero or more graph model description files (*.gm). It is generated from these specifications by GrGen.exe and can be used by applications such as GRShell. Figure 1.2 shows the generation process.

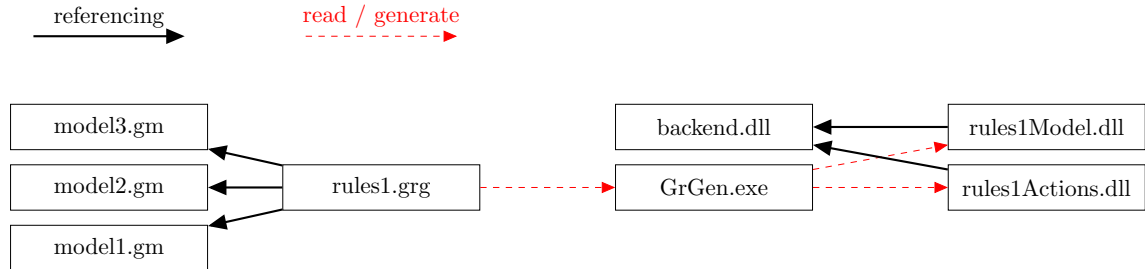


Figure 1.2: Generating a graph rewrite system

In general you have to distinguish carefully between a graph model (meta level), a host graph, a pattern graph and a rewrite rule. In GRGEN.NET pattern graphs are implicitly defined by rules, i.e. each rule defines its pattern. On the technical side, specification documents for a graph rewrite system can be available as source documents for graph models and rule sets (plain text *.gm and *.grg files) or as their translated .NET modules, either C# source files or their compiled assemblies (*.dll).

Generating a GRGEN.NET graph rewrite system may be considered a preliminary task. The actual process of rewriting as well as dealing with host graphs is performed by GRGEN.NET's backend. GRGEN.NET provides a backend API in two versions — the named and typed entities which get generated plus the name string and object based interface offered by the .NET library LIBGr. For most issues—in particular for experimental purposes—you might rather want to work with the GRShell because of its rapid prototyping support. However, GRShell does not provide the full power of the LIBGr; see also note 15 on page 38.

¹In this context, system is not a CH0-like grammar rewrite system, but rather a set of interacting software components.

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 the well-known term rewriting. 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 $L \rightarrow R$, whereas L —the *left hand side* of the rule—is called *pattern graph* and R —the *right hand side* of the rule—is the *replacement graph*.

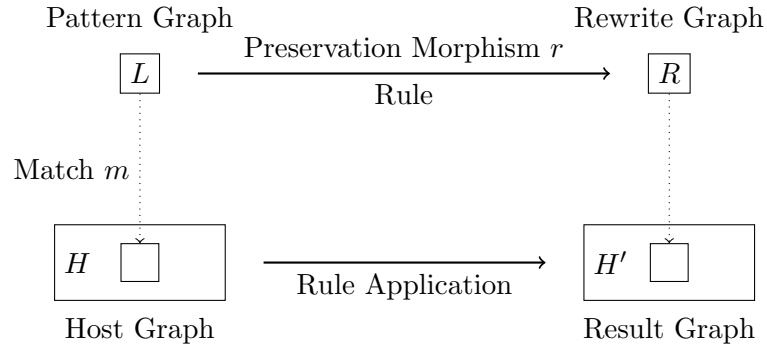


Figure 1.3: Basic Idea of Graph Rewriting

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 $p : L \xrightarrow{r} R$.

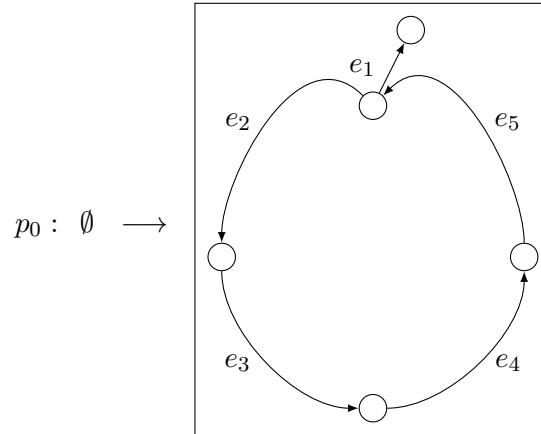
The transformation is done by *application* of a rule to a *host graph* H . To do so, we have to find an occurrence of the pattern graph in the host graph. Mathematically speaking, such a *match* m is an isomorphism from L to a subgraph of H . This morphism may not be unique, i.e. there may be several matches. Afterwards we change the matched spot $m(L)$ of the host graph, such that it becomes an isomorphic subgraph of the replacement 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: $H \xrightarrow{m,p} H'$.

1.6 An Example

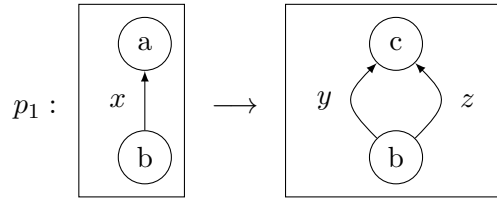
We'll have a look at a small example. Graph elements (nodes and edges) are labeled with and identifier. If a type is necessary then it is stated after a colon. We start using a special case to construct our host graph: an empty pattern always produces exactly one² match

²Because of the uniqueness of the total and totally undefined morphism.

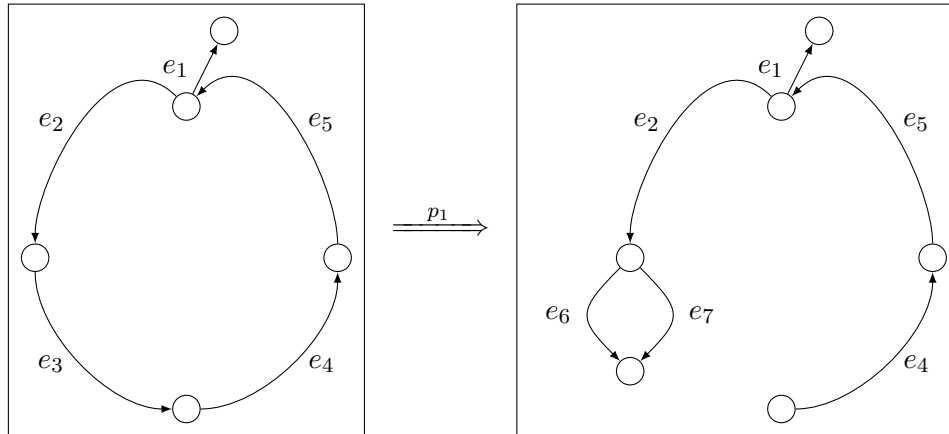
(independent of the host graph). So we construct an apple by applying



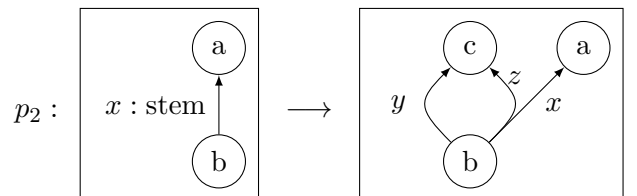
to the empty host graph. As the result we get an apple as new host graph H . Now we want to rewrite our apple with stem to an apple with a leaflet. So we apply



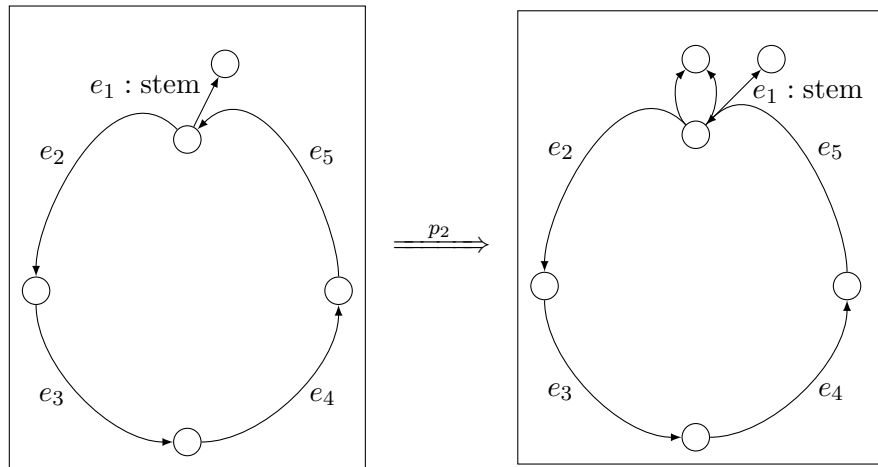
to H and get the new host graph H_1 , something like this:



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 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 The Tools

All the programs and libraries of GRGEN.NET are licensed under LGPL. Notice that the YCOMP graph viewer is not a part of GRGEN.NET; YCOMP ships with its own license. Although YCOMP is not free software, it's free for use in academic and non-commercial areas. You'll find the tools in the `bin` subdirectory of your GRGEN.NET installation.

1.7.1 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>] <rule-set>
```

rule-set is a file containing a rule set specification according to Chapter 4. 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 `tmpgrgen3` within the current directory will be created. The destination directory contains:
- `printOutput.txt`—a snapshot of `stdout` during program execution.
 - `NameModel.cs`—the C# source file(s) of the *rule-setModel1.dll* assembly.
 - `NameActions_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).
 - `NameActions.cs`—the C# source file of the *rule-setActions.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.
- b** Use the backend library *backend-dll* (default is *LGSPBackend*).
- o** Store generated assemblies in *output-dir*.

Requires

.NET 2.0 (or above) or Mono 1.2.3 (or above). Java Runtime Environment 1.5 (or above).

NOTE (1)

Regarding the column information in the error reports of the compiler please note that tabs count as one character.

NOTE (2)

The `grgen` compiler consists of a Java frontend used by the C# backend `grgen.exe`. The java frontend can be executed itself 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:

```
java -jar grgen.jar -i yourfile.grg
```

1.7.2 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 9.

Usage

```
[mono] grShell.exe [-N] [-C "<commands>"] <grshell-script>*
```

Opens the interactive shell. The `GRSHELL` will include and execute the commands in the optional list of *grshell-scripts* (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.

³*n* is an increasing number.

Options

- N Enables non-debug non-gui mode which exits on error with an error code instead of waiting for user input.
- C Execute the quoted GRShell commands immediately (before the first script file). Instead of a line break use a double semicolon ; ; to separate commands.

Requires

.NET 2.0 (or above) or Mono 1.2.3 (or above).

1.7.3 LibGr.dll

The LIBGR is a .NET assembly implementing GRGEN.NET's API. See the extracted HTML documentation for interface descriptions at http://www.grgen.net/doc/API_2_6/; a short introduction is given in chapter 11.

1.7.4 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. YCOMP implements several graph layout algorithms and has file format support for VCG, GML and YGF among others.

Usage

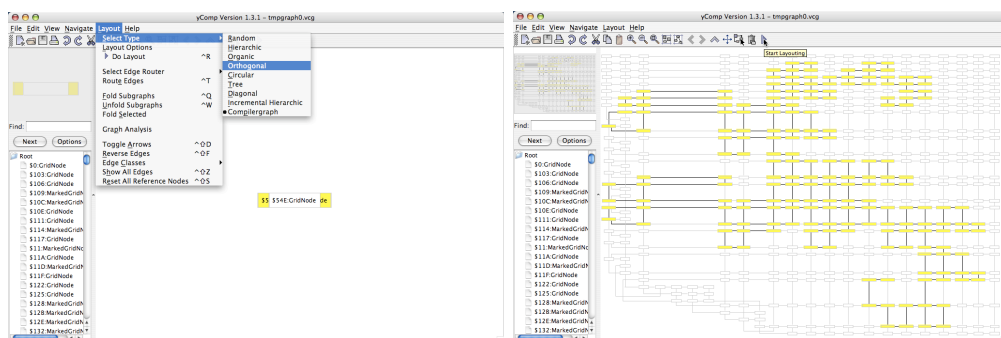
Usually YCOMP will be loaded by the GRShell. You might want to open YCOMP manually by typing

```
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 (hierarchical 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).

1.8 Development goals

The development goals of GRGEN.NET were

Expressiveness

is achieved by powerful and declarative specification languages for pattern matching and rewriting by rewrite rules, building upon a rich graph model language. In addition to the unmatched expressiveness of the basic actions, the rule language now offers nested and subpatterns which allow to handle substructures of arbitrary depth and arbitrary breadth declaratively within the rules, by now even surpassing the capabilities of the VIATRA2[VB07, VHV08] graph rewriting tool, the strongest competitor in rule expressiveness. The rules can be combined by graph rewrite sequences, a rule application control language with variables and logical as well as iterative control flow; it was recently extended by storages as pioneered by the VMTS[LLMC05] graph rewriting tool, allowing for more concise and faster solutions.

Performance

i.e. high speed at modest memory consumption, is needed to tackle real world problems. It is achieved by typing, easing the life of the programmer by eliminating large classes or errors as well as speeding up the pattern matcher, by the generative approach compiling the rules into executable code, and by the heuristic optimizations of search state space stepping and 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 as an optimization problem [BKG08, Bat06]. In contrast to systems like Fujaba[Fuj07, NNZ00], our strongest competitor regarding performance, our pattern matching algorithm is fully automatic and does not need to be tuned nor to be implemented in parts by hand. According to Varró's benchmark[VSV05], it is at least one order of magnitude faster than any other tool known to us.

Development Convenience

is gained by interactive and graphical debugging of the rule application, capable of visualizing the matched pattern, the rewrite which will be applied, and the currently active rule out of the rewrite sequence. 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). Another important factor regarding development convenience is currently read by you. There's one convenience not offered you might expect: a visual rule language and an editor. This brings a clear benefit – graph transformation specifications to be processed by GRGEN.NET can be easily generated – but especially is a good deal cheaper to implement. Given the limited resources of an university project this is an important point, as can be seen with the AGG[ERT99] tool, offering a nice graphical editor but delivering performance only up to simple toy examples (causing the wrong impression that graph rewriting is notoriously inefficient).

Well Founded Semantics

to ease formal, but especially human reasoning. The semantics of GRGEN.NET are specified in [Gei08], based upon graph homomorphisms, denotational evaluation functions and category theory. The GRGEN.NET-rewrite step is based by default on the *single-pushout approach* (SPO, for explanation see [EHK⁺99]), with the *double-pushout approach* (DPO, for explanation see [CMR⁺99]) available on request, too. The semantics of the recursive rules introduced in version 2.0 are given in [Jak08], utilizing pair star graph grammars on the meta level to assemble the rules of the object level. The formal semantics are not as complete as for the graph programming language GP[Plu09]

though, mainly due to the large amount of features – the convenience at using the language had priority over the convenience at reasoning formally about the language.

Platform Independence

is achieved by using languages compiled to byte code for virtual machines backed by large, standardized libraries, specifically: Java and C#. 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.

General Purpose Graph Transformation

in contrast to special purpose graph transformation. A lot of other graph based tools are geared towards special purpose applications, e.g. verification (GROOVE [Ren04]), or biology (XL [KK07], or model transformation (VIATRA2[VB07]). This means that design decisions were taken to ease uses in this application areas at the cost of rendering uses in other domains more difficult. And that features were added in a way which just satisfies the needs of the domain at hand instead of striving for a more general solution (which would have caused higher costs at designing and implementing this features). While the old GRGEN was built as a special purpose compiler construction tool for internal use (optimizations on the graph based compiler intermediate representation FIRM – see www.libfirm.org), the new GRGEN.NET was built from the beginning as a general purpose graph transformation tool for external use – to be used in areas as diverse as computer linguistics, software engineering, computational biology or sociology – for reasoning with semantic nets, transformation of natural language to UML models, model transformation, processing of program graphs, genome simulation, or pattern matching in social nets. Several of them are worked on, you may have a look at [BG09] or [GDG08] or [SGS09].

CHAPTER 2

QUICKSTART

In this chapter we'll build a GRGEN.NET system from scratch. You should already have read Chapter 1 to have a glimpse of the GRGEN.NET system and its components. We will use GRGEN.NET to construct non-deterministic state machines. We further show some actual graph rewriting by removing ε -transitions from our state machines. This chapter is mostly about the GRGEN.NET look and feel; please take a look at the succeeding chapters for comprehensive specifications.

2.1 Downloading & Installing

If you are reading this document, you probably did already download the GRGEN.NET software from our website (<http://www.grgen.net>). Make sure you have the following system requirements installed and available in the search path:

- Java 1.5 or above
- Mono 1.2.3 or above / Microsoft .NET 2.0 or above

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 (3)

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

2.2 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 machine¹. By graph meta model we mean a set of node types and edge types which are available for building state machine graphs (see Chapter 3). Figure 2.1 shows the meta model. What have we done? You can see two base types, `State` for

```

1 node class State {
2     id: int;
3 }
4
5 abstract node class SpecialState extends State;
6 node class StartState extends SpecialState;
7 node class FinalState extends SpecialState;
8 node class StartFinalState extends StartState, FinalState;
9
10 edge class Transition {
11     Trigger: string;
12 }
13
14 const edge class EpsilonTransition extends Transition;
```

Figure 2.1: Meta Model for State Machines

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. For the rest of the types we use inheritance (keyword `extends`) which works more or less like inheritance in object oriented languages. Accordingly the `abstract` modifier for `SpecialState` means that you cannot create a node of that precise type, but you might create nodes of non-abstract subtypes. As you can see GRGEN.NET supports multiple inheritance, and with `StartFinalState` we have constructed a “diamond” type hierarchy.

2.3 Creating Graphs

Let’s test our graph meta model by creating a state machine graph. We will use the GRShell (see Chapter 9) 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

```

1 using StateMachine;
```

Now, we could start by launching the GRShell and typing the commands interactively. This is, however, in most of the cases not the preferred way. We rather create a GRShell script, say `removeEpsilons.grs`, in the `/home/grgen` directory. Figure 2.2 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 “layout graph” button on the very right side of the button bar, you get a window similar to figure 2.3 (see also Section 1.7.4). The graph looks still a bit confusing. In fact it is quite normal that YCOMP’s automatic layout algorithm needs manual adjustments. Quit YCOMP and exit the GRShell by typing `exit`.

¹You’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 2.2: Constructing a state machine graph in GRSHELL

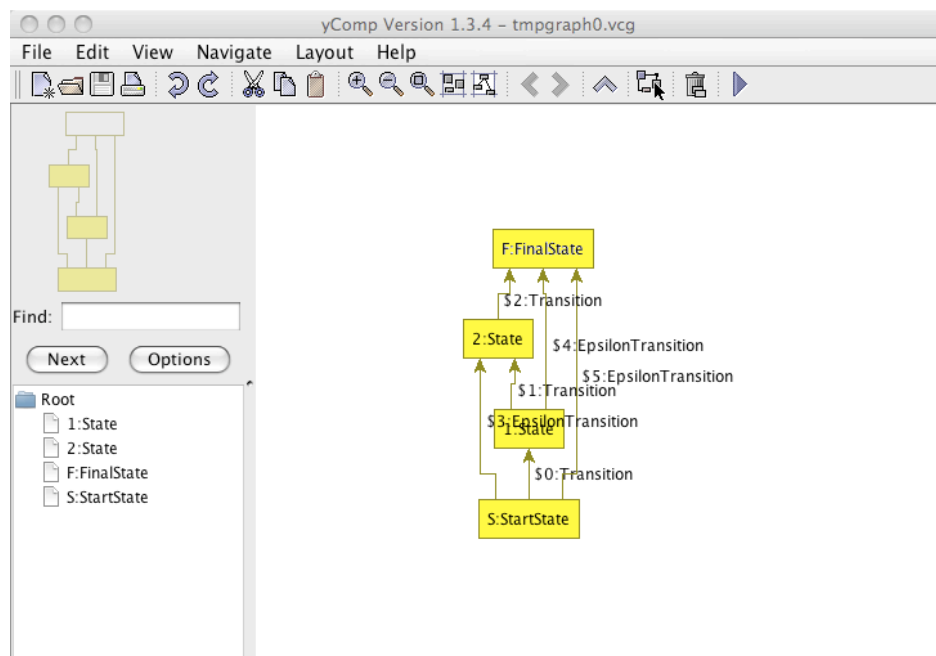


Figure 2.3: A first state machine

This script starts with creating an empty graph of the meta model `StateMachine` (which is referenced by the rule set `removeEpsilons.grg`) with the name `StateMachineGraph`. Thereafter we create nodes and edges. The colon notation indicates a node or edge type. Also note the inplace-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 `$` and `@` notation is due to the fact that we have two kinds of “names” in the GRShell. Namely we have *graph variables*—which we did not use, no graph variable is explicitly defined in this script—and *persistent names* that denote a specific graph element. Persistent names are set by `$=Identifier` on creation and they are accessed by `@(Identifier)`. The quote chars around “1” and “2” are used to type these characters as (identifier) strings rather than numbers.

2.4 The Rewrite Rules

We will now add the real rewrite rules to the rule set file `removeEpsilons.grg`. The idea is to “forward” the ε -transitions one after another, i.e. if we have a pattern like `a:State -EpsilonTransition-> b:State -e:Transition-> c:State` we forward to `a -e-> c`. After all such transitions are forwarded we can remove the ε -transitions altogether. The complete rule set is shown in figure 2.4. See Chapter 4 for the rule set language reference.

In detail: The rule set file consists of a number of rules and tests, each of them bearing a name, like `forwardTransition`. Rules contain a pattern expressed as several semicolon-separated pattern statements and a modify part or a rewrite part. Tests contain only a pattern; they are used to check for a certain pattern without doing any rewrite operation. If a rule is applied, GRGEN.NET tries to find the pattern within a host graph, for instance within the graph we created in Section 2.3. Of course there could be several matches for a pattern—GRGEN.NET will choose one of them arbitrarily.

Figure 2.4 also shows the syntax `x:NodeType` for nodes and `-e:EdgeType->` for Edges, which we have already seen in Section 2.3. There are also statements like `:FinalState` or `-:EpsilonTransition->`, i.e. we are searching for a node of type `FinalState` resp. an edge of type `EpsilonTransition`, but we are not assigning these graph elements to a name (like `x` or `e` above). Defining of names is a key concept of the GRGEN.NET rule sets: names work as connection points between several pattern statements and between the pattern and the replace / modify part. As a rule of thumb: If you want to do something with your found graph element, define a name; otherwise an anonymous graph element will do fine. Also have a look at example 7 on page 34 for additional pattern specifications. The difference between a replace part and a modify part is that a replace part deletes every graph element of the pattern which is not explicitly mentioned within its body. The modify part, in contrast, deletes nothing (by default), but just adds or adjusts graph elements. However, the modify part allows for *explicit* deletion of graph elements by using the `delete` statement.

What else can we do? We have negative application conditions (NACs), expressed by `negative {...}`; they prevent rules to be applied if the negative pattern is found. We also have attribute conditions, expressed by `if {...}`; a rule is only applicable if all such conditions yield true. Note, the dot notation to access attributes (as in `e.Trigger`). The `emit` statement prints a string to `stdout`. The `hom(x,y)` and `hom(x,y,z)` statements mean “match the embraced nodes homomorphically”, i.e. they can (but they don’t have to) actually be matched to the same node within the host graph. The `eval {...}` statement is used to recalculate attributes of graph elements. Have a look at the statement `y:StartFinalState<x>` in `addStartFinalState`: we *retype* the node `x`. That means the newly created node `y` is actually the node `x` (including its incident edges and attribute values) except for the node type which is changed to `StartFinalState`. Imagine retyping as a kind of a type cast.

The created rewrite rules might be considered as rewrite primitives. In order to implement more complex functionality, we will compose a sequence of rewrite rules out of them. For

```

1  using StateMachine;
2
3  test checkStartState {
4      x:StartState;
5      negative {
6          x;
7          y:StartState;
8      }
9  }
10
11 test checkDoublettes {
12     negative {
13         x:State -e:Transition-> y:State;
14         hom(x,y);
15         x -doublette:Transition-> y;
16         if {typeof(doublette) == typeof(e);}
17         if { ((typeof(e) == EpsilonTransition) || (e.Trigger == doublette.Trigger)); }
18     }
19 }
20
21 rule forwardTransition {
22     x:State -:EpsilonTransition-> y:State -e:Transition-> z:State;
23     hom(x,y,z);
24     negative {
25         x -exists:Transition-> z;
26         if {typeof(exists) == typeof(e);}
27         if { ((typeof(e) == EpsilonTransition) || (e.Trigger == exists.Trigger)); }
28     }
29     modify {
30         x -forward:typeof(e)-> z;
31         eval {forward.Trigger = e.Trigger;}
32     }
33 }
34
35 rule addStartFinalState {
36     x:StartState -:EpsilonTransition-> :FinalState;
37     modify {
38         y:StartFinalState<x>;
39         emit("Start_state_" + x.id + ")_mutated_into_a_start-and-final_state");
40     }
41 }
42
43 rule addFinalState {
44     x:State -:EpsilonTransition-> :FinalState;
45     if {typeof(x) < SpecialState;}
46     modify {
47         y:FinalState<x>;
48     }
49 }
50
51 rule removeEpsilonTransition {
52     -:EpsilonTransition->;
53     replace {}
54 }

```

Figure 2.4: Rule set for removing ε -transitions

instance we don't want to forward just one ε -transition as `forwardTransition` would do; we want to forward them all. Such a rule composing is called *graph rewrite sequence* (see Chapter 7). We add the following line to our shell script `removeEpsilons.grs`:

```
1 debug xgrs (checkStartState && checkDoublettes) && <forwardTransition* | addStartFinalState
  | addFinalState* | removeEpsilonTransition*>
```

This looks like a boolean expression and in fact it behaves similar. The whole expression is evaluated from left to right. A rule is successfully evaluated if a match could be found. 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-and (`&&`), i.e. if one of them fails the evaluation will be canceled. We continue by disjunctively connected rules (connected by `|`). The angle brackets (`<>`) around the transformation rules indicate transactional processing: If the enclosed sequence returns `false` for some reason, all the already performed graph operations will be rolled back. That means not all of the rules must find a match. 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.

2.5 Debugging and Output

If you execute the modified `GRSHELL` script, `GRGEN.NET` starts its debugger. This way you can follow the evaluation 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 rewrite operation in multiple steps; the `s` key jumps to the next rule; and the `r` key runs to the end of the graph rewrite sequence. Finally you should get a graph like the one in figure 2.5

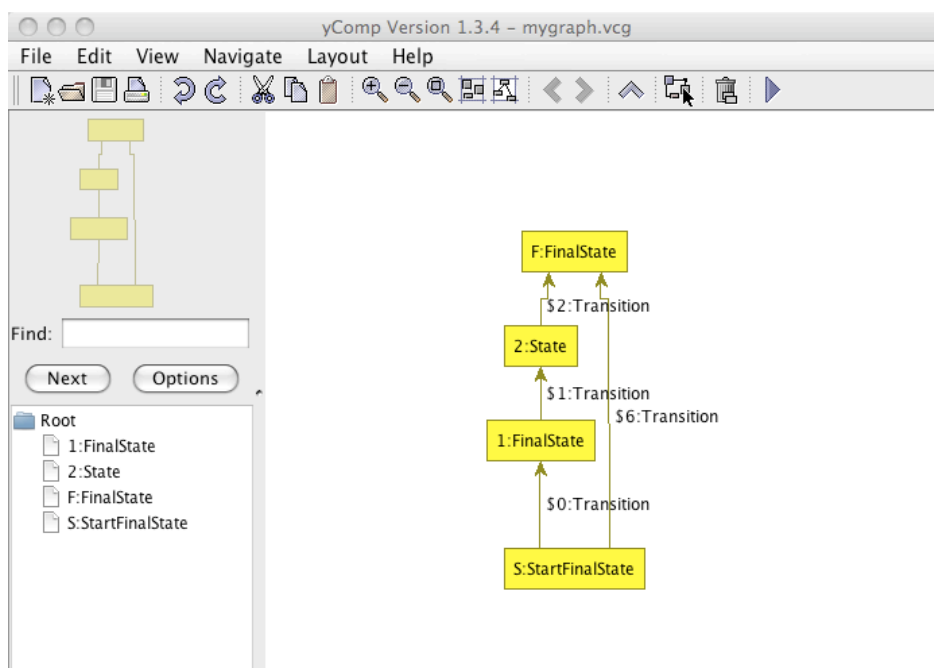


Figure 2.5: A state machine without ε -transitions

If everything is working fine you can delete the `debug` keyword in front of `xgrs`. Maybe you want to save a visualization of the resulting graph. This is possible by typing `dump graph mygraph.vcg` in `GRSHELL`, writing `mygraph.vcg` into the current directory in VCG format readable by `YCOMP`. Feel free to browse the `examples` folder shipped with `GRGEN.NET`; esp. the succession of examples in `FiniteStateMachine` and `ProgramGraphs` is recommended to have a look at, giving a nice overview of the capabilities of the software.

CHAPTER 3

GRAPH MODEL LANGUAGE

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 programming languages, except for the concept of methods that GRGEN.NET nodes and edges don't support. GRGEN.NET edge types can be directed and 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. Note that GRGEN.NET lacks a concept of overwriting. 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.

Connection Assertions

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

In this chapter as well as in Chapter 9 (GRSHELL) we use excerpts of Example 1 (the Map model) for illustration purposes.

3.1 Building Blocks

NOTE (4)

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 (or subsentence) of the language. Ellipses represent terminals whereas rectangles represent non-terminals. For further information on syntax diagrams see [MMJW91].

^aExtended Backus–Naur Form.

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 }
```

Basic elements of the GRGEN.NET graph model language are identifiers to denominate nodes, edges, 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 under-scores. 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* non-terminal may be annotated. See Section 4.8 for annotations of declarations.

NOTE (5)

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:

```

1  node class t_node;
2  node class t_node {
3      ...
4  }
```

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.

3.1.1 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 3.2), 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, its recommended to use the keywords **directed**, **undirected**, and **arbitrary** in order to specify inheritance (see Section 3.2). As soon as you decided for directed or undirected edge classes within your type hierarchy, you are not able to let ancestor classes inherited from a contradicting base type, of course. That is, no edge may be directed *and* undirected. This is an exception of the concept multi-inheritance. Figure 3.1 shows the edge type hierarchy.

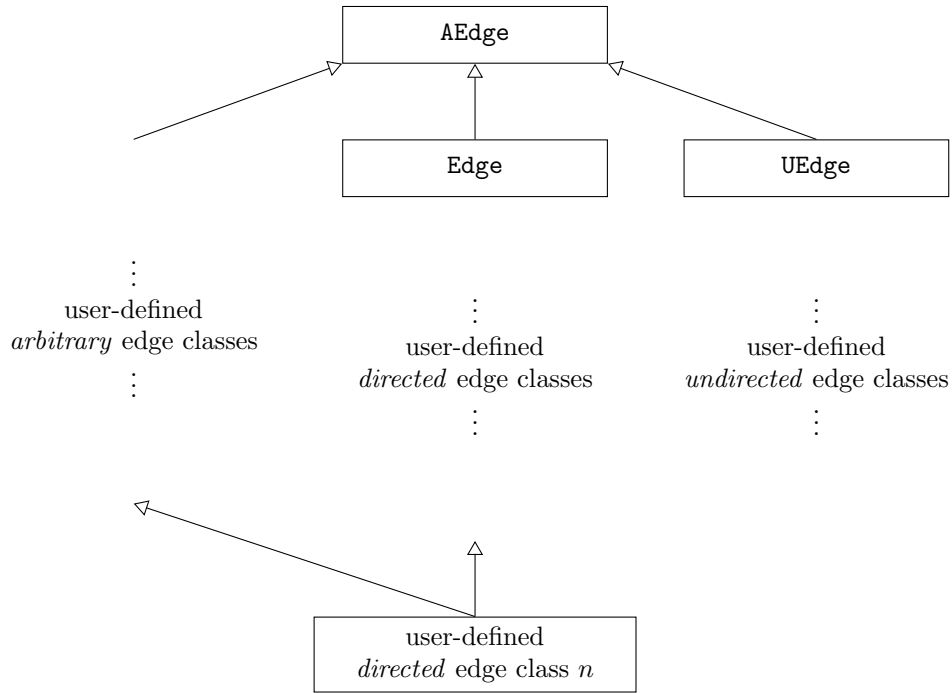
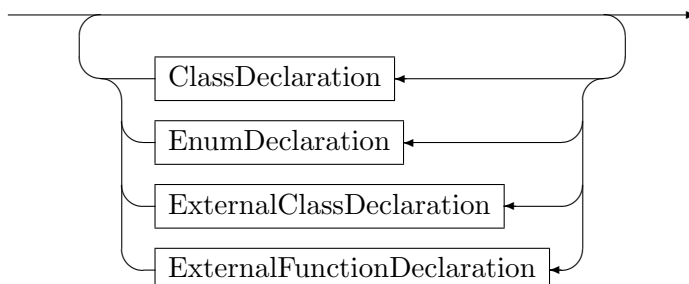


Figure 3.1: Type Hierarchy of GRGEN.NET Edges

3.2 Type Declarations

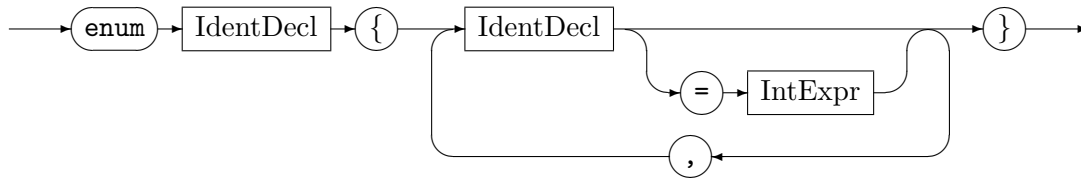
GraphModel



The graph model consists of zero or multiple type declarations. Whereas *ClassDeclaration* defines a node type or an edge type, *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. The *ExternalClassDeclaration* registers an external class, including its subtype hierarchy, excluding any attributes with GRGEN.NET which can subsequently be used in attribute computations with external function calls.

3.2.1 Enumeration Types

EnumDeclaration



Defines an enum type. An enum type is a collection of so called *enum items* that are associated with integral numbers, each. Accordingly, a GRGEN.NET enum is internally represented as `int` (see Section 5.1).

NOTE (6)

An enum type and an `int` are different things, but in expressions enum values are implicitly casted to `int` values (see Section 5.1).

NOTE (7)

Normally, assignments of `int` values to something that has an enum type are forbidden (see Section 5.1). Only inside a declaration of an enum type an int value may be assigned to the enum item that is currently declared. This also includes the usage of items taken from other enum types (because they are implicitly casted to `int`). However, items from other enum types must be written fully qualified in this case (which, e.g., looks like `MyEnum::a`, where `MyEnum` is the name of the other enum type).

EXAMPLE (2)

```

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}

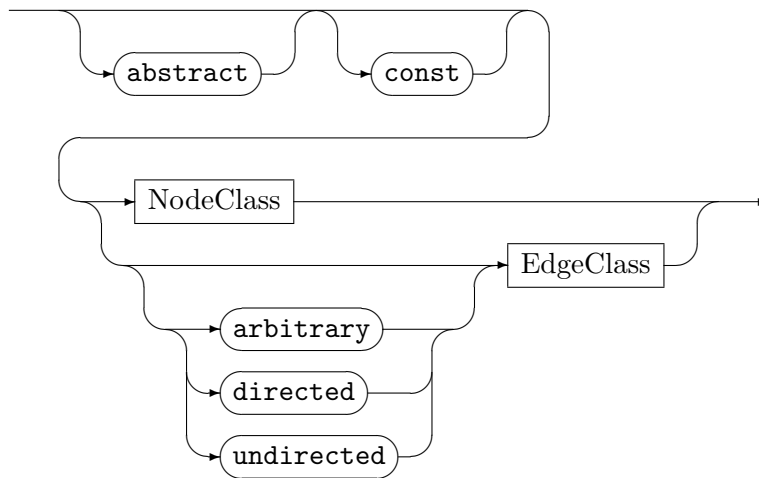
```

Consider, e.g., the declaration of the enum item `e`: By 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 (8)

The C-like semantics of enum item declarations implies, that multiple items of one enum type can be associated with the same `int` value. 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.

3.2.2 Node and Edge Types

ClassDeclaration

Defines a new node type or edge type. The keyword **abstract** indicates that you cannot instantiate graph elements of this type. Instead you have to derive non-abstract types to create graph elements. The abstract-property will not be inherited by subclasses, of course.

EXAMPLE (3)

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;

```

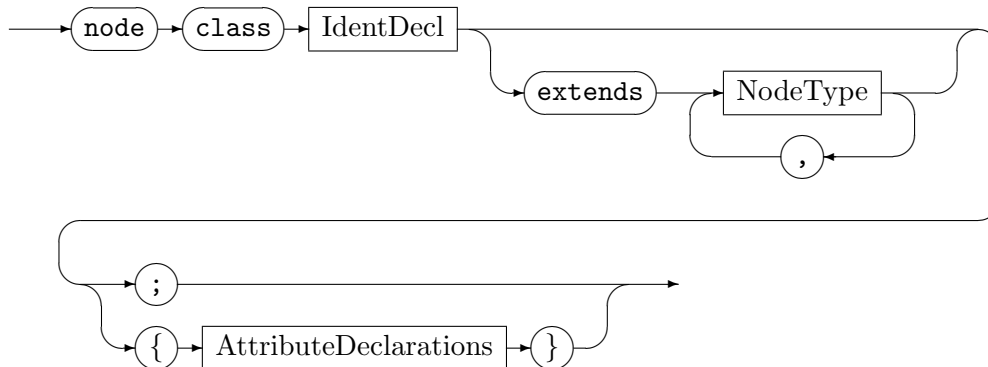
You will be able to create nodes of type `GhostTown`, but not of type `City` or `AbandonedCity`. However, nodes of type `GhostTown` are also of type `AbandonedCity` as well as of type `City` and they have the attribute `Size`, hence.

The keyword **const** indicates that rules may not write to attributes (see also Section 4.4, `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`.

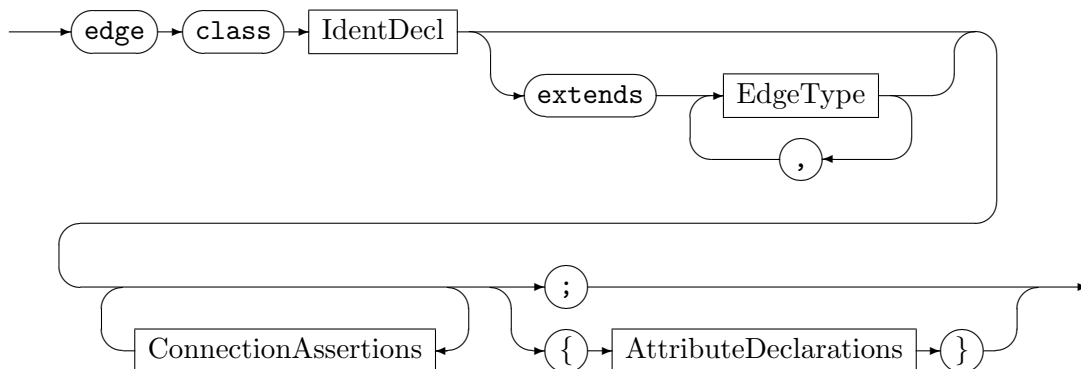
An **undirected** edge inherits from **UEdge**. If you do not specify any of those keywords, a **directed** edge is chosen by default. See also Section 3.1.1

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

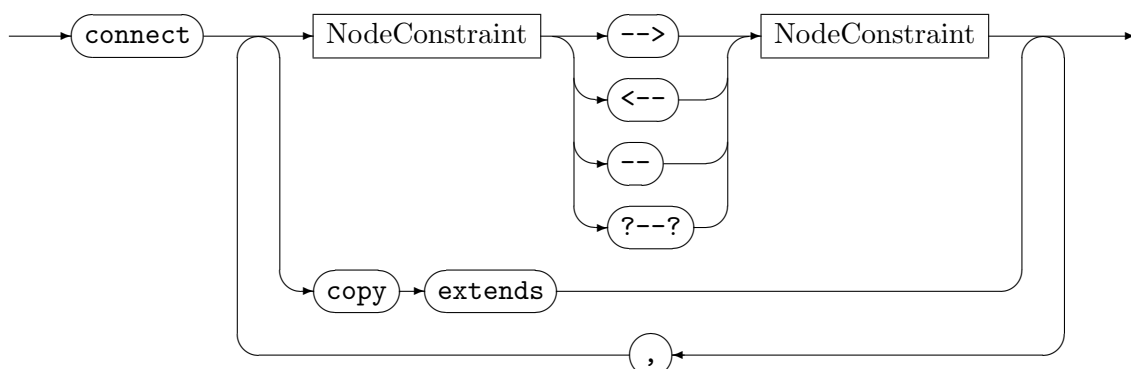


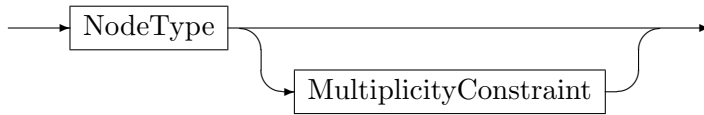
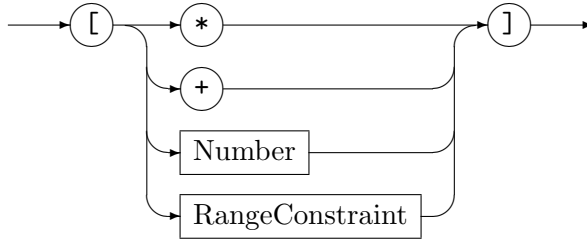
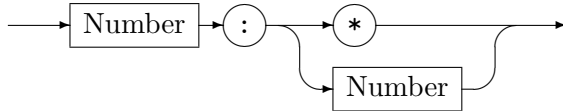
Defines a new edge type. Edge types can inherit from other edge types defined within the same file. If the **extends** clause is omitted, *EdgeType* will inherit from the built-in type *Edge*. Optionally edges can possess attributes. A *connection assertion* specifies that certain edge types should only connect specific nodes a given number of times. (see Section 3.1.1)

NOTE (9)

It is not forbidden to create graphs that are invalid according to connection assertions. GR-GEN.NET just enables you to check, whether a graph is valid or not. See also Section 9.2.2, **validate**.

ConnectionAssertions



NodeConstraint*MultiplicityConstraint**RangeConstraint*

A *connection assertion* is denoted as a pair of node types, optionally with their multiplicities. It allows you to specify the node types an edge may connect and the multiplicities with which such edges appear on the nodes. It is best understood as a simple pattern of the form (cf. 4.1.1) `: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. Furtheron there is the strict-only-specified mode, which only does strict validation of the edges for which connection assertions are given. See Section 9.2.2, **validate**, for an example.

The arrow syntax is based on the GRGEN.NET graphlet specification (see Section 4.1.1). 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 possible (fixed by the concrete type inheriting from it); in case of an directed edge the connection pattern gets matched in both directions. *Number* is an `int` constant as defined in Chapter 5. Table 3.1 describes the multiplicity definitions.

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,

$[n:*$]	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 3.1: GRGEN.NET node constraint multiplicities

i. e. the effect of using `copy extends` is the same as copying the connection assertions from the direct ancestors by hand.

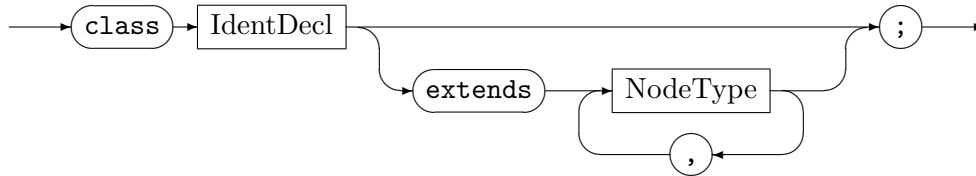
NOTE (10)

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.

3.2.3 External Attribute Types

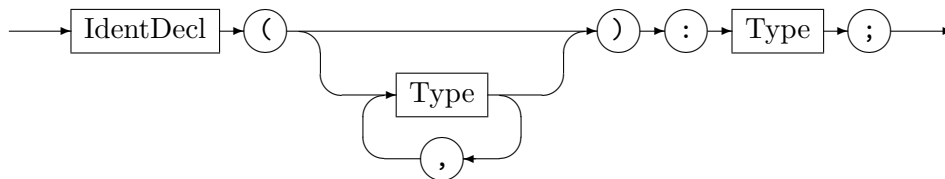
ExternalClassDeclaration



Registers a new attribute type with GRGEN.NET. You may declare the base types of the type, but not give attributes. The attribute type must be implemented externally, see 11.4; for GRGEN.NET the type is opaque, only external functions can do computations with it. You may extend GRGEN.NET with external attribute types if the built-in attribute types (cf. 5.1) are insufficient for your needs.

3.2.4 External Function Types

ExternalFunctionDeclaration

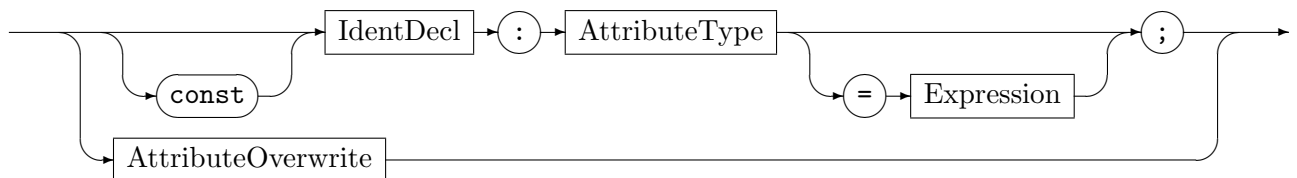


Registers an external function with GRGEN.NET to be used in attribute computation. An external function declaration specifies the expected input types and the output type. The function must be implemented externally, see 11.4. An external function call (cf. 5.10) may

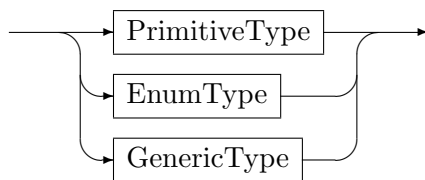
receive and return values of the built-in (attribute) types as well as of the external attribute types; the real arguments on the call sites are type-checked against the declared signature following the subtyping hierarchy of the built-in as well as of the external attribute types. You may extend GRGEN.NET with external functions if the built-in attribute computation capabilities (cf. 5.2) are insufficient for your needs.

3.2.5 Attributes and Attribute Types

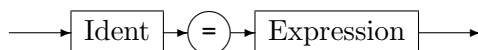
AttributeDeclaration



AttributeType



AttributeOverwrite



Defines a node or edge attribute. Possible types are enumeration types (**enum**) and primitive types or generic types. See Section 5.1 for a list of built-in primitive and generic types. Optionally attributes may be initialized with a *constant* expression. The expression has to be of a compatible type of the declared attribute. See Chapter 5 for the GRGEN.NET types and expressions reference. The *AttributeOverwrite* clause lets you overwrite initialization values for attributes of super classes. The initialization values are evaluated in the order as they appear in the rule set file.

NOTE (11)

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

```

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

```

NOTE (12)

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.

CHAPTER 4

RULE SET LANGUAGE

The rule set language forms the core of GRGEN.NET. Rule files refer to zero¹ or more graph models and specify a set of rewrite rules. The rule language covers the pattern specification and the replace/modify specification. Attributes of graph elements can be re-evaluated during an application of a rule. The following rewrite rule mentioned in Geiß et al. [GBG⁺06] gives a rough picture of the language:

EXAMPLE (4)

```
1  using SomeModel;
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 }
```

In this chapter we use excerpts of Example 4 (`SomeRule`) for illustration purposes. The nested negative which specify a pattern which must not be available in the host graph are described in the following Chapter 6

4.1 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.

¹Omitting 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.

Ident, IdentDecl

A non-empty character sequence of arbitrary length consisting of letters, digits, or under-scores. The first character must not be a digit. *Ident* may be an identifier defined in a graph model (see Section 3.1). *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 4.8 for annotations of declarations.

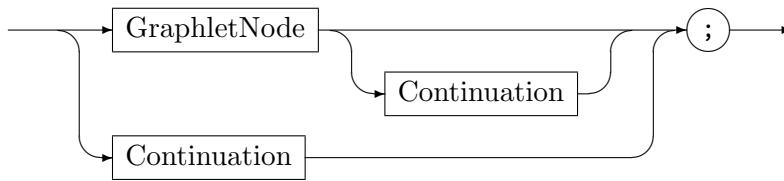
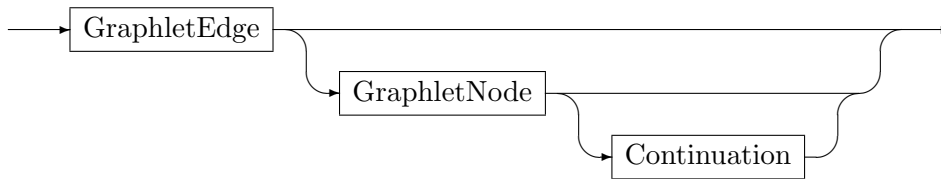
NOTE (13)

As in the GRGEN.NET model language (see note 5) 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 5.9 for the use of type expressions.

4.1.1 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 7, the statement `n1 --> n2` is the node identifier `n1` followed by the continuation graphlet `--> n2`.

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 `n1 --> n2` contains an anonymous edge; thus can be understood as `n1 -$1:Edge-> n2`. 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 pair wise distinct except for homomorphically matched graph elements (see Section 4.3). 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. 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)

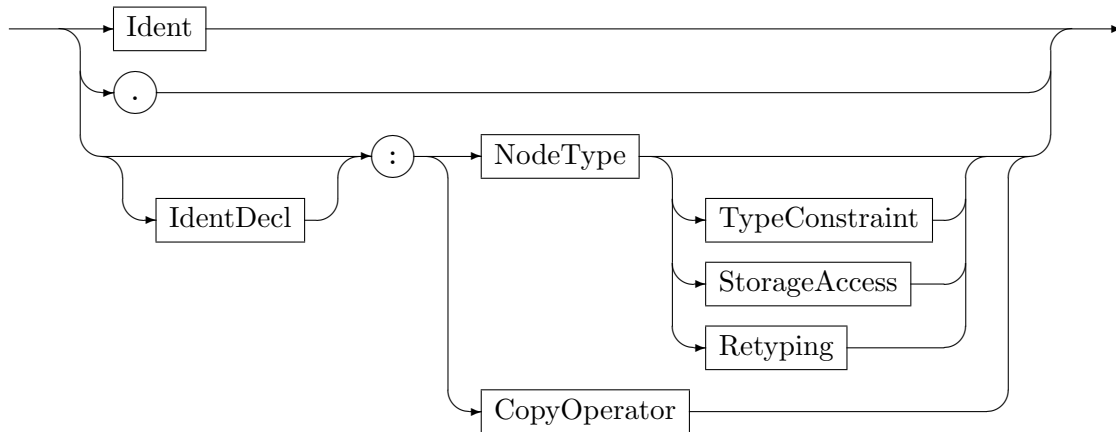
```
1 n1 --> n2 --> n3 <-- n1;
```

is equivalent to

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

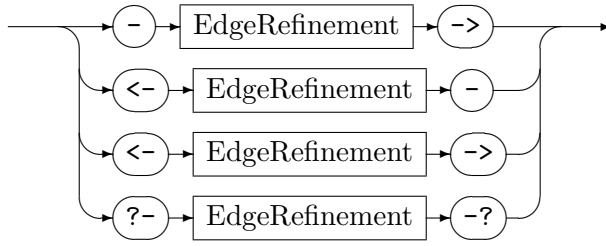
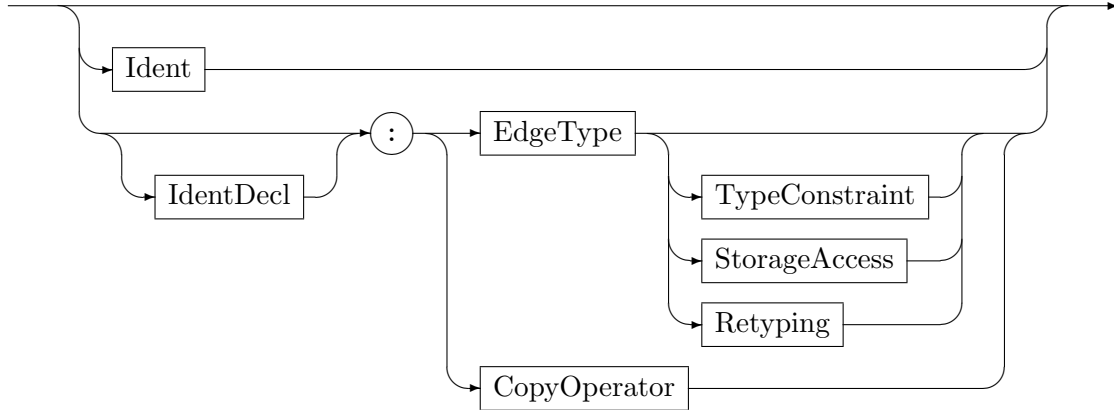
and `n1 --> n3` is equivalent to `n3 <-- n1`, of course.

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 17, the negative condition uses `n3` from the surrounding scope and defines `n4` and `e1`. We may safely reuse the variable name `e1` in the replace part.

GraphletNode

Specifies a node of type *NodeType*, maybe constrained in type with a *TypeConstraint* (see Section 4.6.1, *TypeConstraint*), maybe bound by a storage access (see 8.2.2, *StorageAccess*), maybe retyped with a *Retyping* (see Section 4.7.1, *Retyping*). Or a node having the same type and bearing the same attributes as another matched node (see Section 4.7.2, *CopyOperator*). The `.` is an anonymous node of the base type `Node`; remember that every node type has `Node` as super type. Type constraints are allowed in the pattern part only. The `CopyOperator` and the `Retyping` clause are allowed in the replace/modify part only.

Graphlet	Meaning
<code>x:NodeType;</code>	The name <code>x</code> is bound to a node of type <code>NodeType</code> or one of its subtypes.
<code>:NodeType;</code>	<code>\$1:NodeType</code>
<code>.;</code>	<code>\$1:Node</code>
<code>x;</code>	The node, <code>x</code> is bound to.

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. Type constraints are allowed in the pattern part only (see Section 4.6.1, *TypeConstraint*); the same holds for the storage access (see 8.2.2, *StorageAccess*). The *CopyOperator* and the *Retyping* clause are allowed in the replace/modify part only (see Section 4.7.2, *CopyOperator*, see Section 4.7.1, *Retyping*).

The different kind of arrow tips distinguish between directed, undirected, and arbitrary edges (see also Section 3.1.1). 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. Beware of accidentally “redirecting”² an edge: The graphlets

²You cannot directly express the redirection of edges. This is a direct consequence of the SPO approach. Redirection of edges can be “simulated” by either deleting and re-inserting an edge, or more indirectly by re-typing of nodes.

```
-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 replace part.

EXAMPLE (6)

Some attempts to specify a loop edge:

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

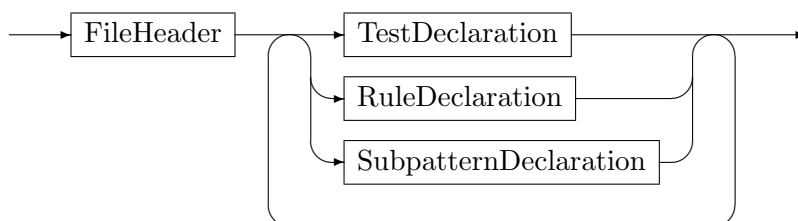
NOTE (14)

Although both, the pattern part and the replace/modify part use graphlets, there are subtle differences between them. These concern the *TypeConstraint* and *StorageAccess* clauses (only LHS), the *Retyping* clause and the *CopyOperator* (only RHS), and the allowed arrow tips for edges.

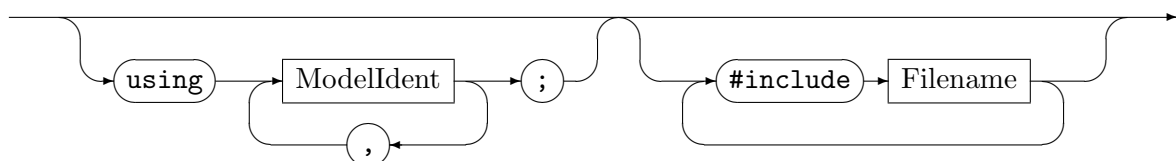
4.2 Rules and Tests

The structure of a rule set file is as follows:

RuleSet



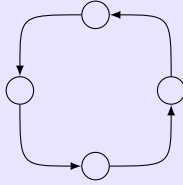
FileHeader



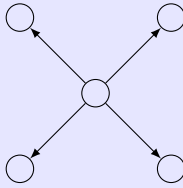
A rule set consists of the underlying graph models and several rewrite rules and tests (subpatterns will be introduced in 6.5). Additionally you may include further rule set files

EXAMPLE (7)

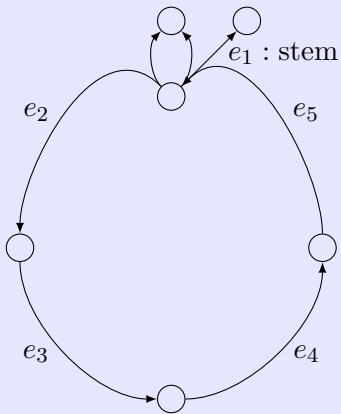
Some graphlets:



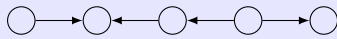
```
x:Node --> . --> . --> . --> x;
```



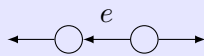
```
. <-- x:Node --> . ;  
. <-- x --> . ;
```



```
. <-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:

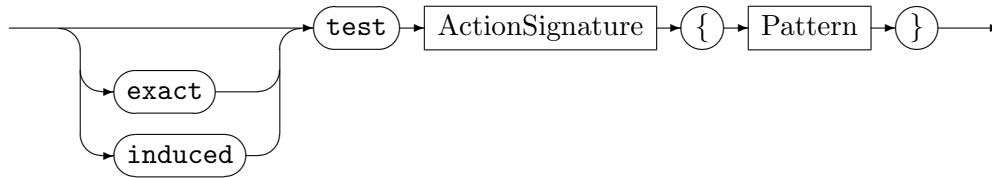
. -e:Edge-> . ; . -e-> . ; Would affect redirecting of edge e.

x -e:T-> y; x -e-> x; Would affect redirecting of edge e.

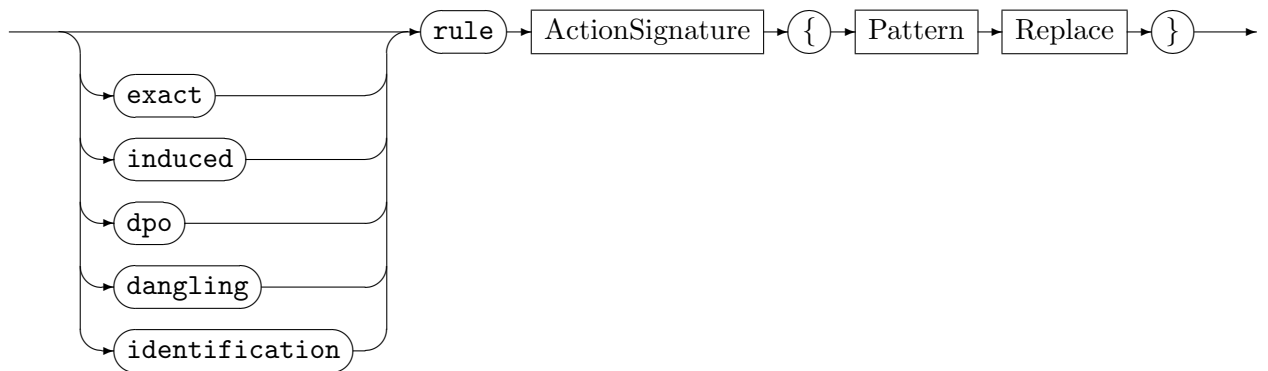
<-- --> ; There must be at least a node between the edges.

(without using directives, we prefer to suffix them with `.gri` in this case). In case of multiple graph models, GRGEN.NET uses the union of these models. In this case beware of conflicting declarations. There is no built-in conflict resolution mechanism like packages or namespaces for models. If necessary you can use prefixes as you might do in C.

TestDeclaration



RuleDeclaration



Declares a single rewrite rule such as `SomeRule`. It consists of a pattern part (see Section 4.3) in conjunction with its rewrite/modify part (see Section 4.4). 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 `exact`, `induced`, `dangling`, `identification`, and `dpo` pattern modifiers see Section 4.5.

EXAMPLE (8)

We define a test `SomeCond`

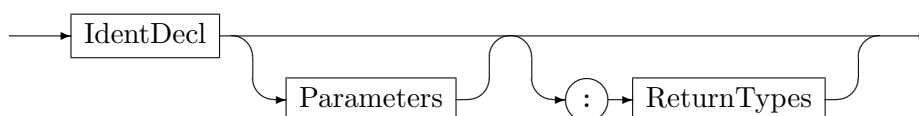
```
1 test SomeCond {
2   n:SeldomNodeType;
3 }
```

and execute in GRShell:

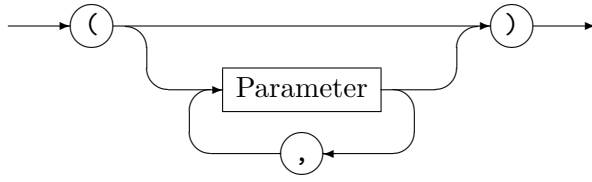
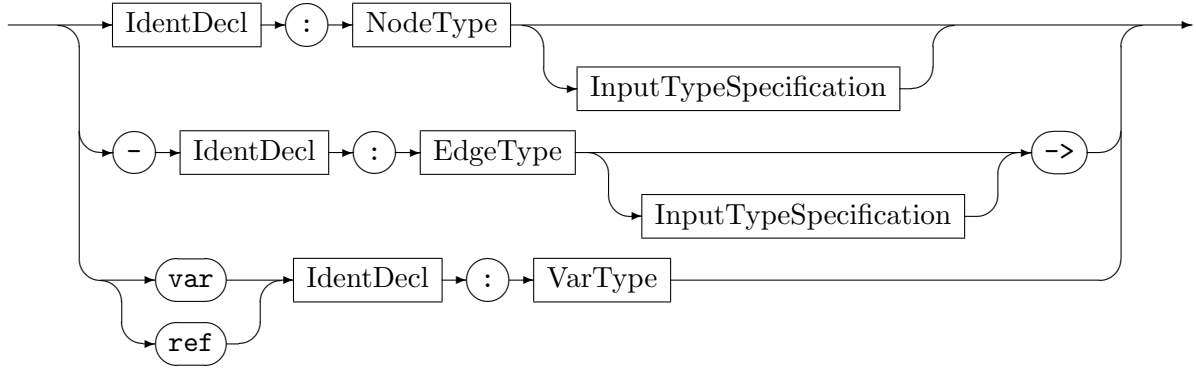
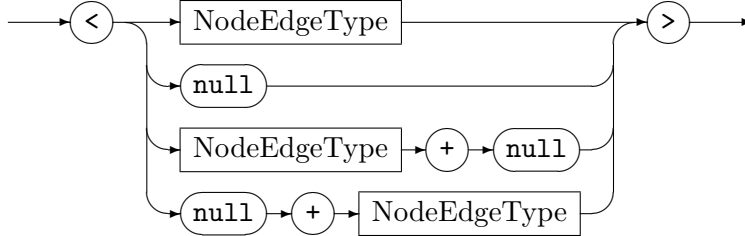
```
1 grs SomeCond & SomeRule
```

`SomeRule` will only be executed, if a node of type `SeldomNodeType` exists. For graph rewrite sequences in GRShell see Section 9.2.9.

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.

Parameters*Parameter**InputTypeSpecification*

Within a rule, graph element parameters are treated as graph elements of the pattern - just predefined. But in contrast to pervious versions it is the task of the user 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. If you need more flexibility and want to call the rule with parameters not fullfilling 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 (5.1). The primitive types require the **var** prefix and are handed in by-value; the generic 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 4.4) is undefined (concerning the parameters as well as the argument); they are only available for reading, the by-ref parameters additionally for set/map-addition and removal (cf. 8.1)

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)

```

1 test t(n1:Node<null>, n2:Node<null>) {
2   n1 ?--? n2;
3 }

```

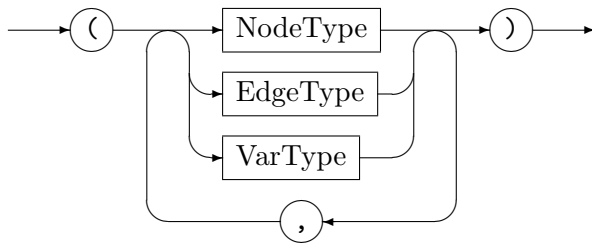
is equivalent to the tests `t1-t4` which are chosen dependent on what parameters are defined.

```

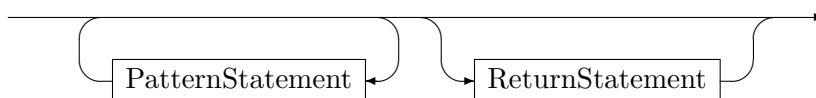
1 test t1(n1:Node, n2:Node) {
2   n1 ?--? n2;
3 }
4 test t2(n1:Node) {
5   n1 ?--? n2:Node;
6 }
7 test t3(n2:Node) {
8   n1:Node ?--? n2;
9 }
10 test t4 {
11   n1:Node ?--? n2:Node;
12 }

```

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.

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 must occur. See also Section 4.4, **return**.

4.3 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 explanation of the pattern modifiers **dpo**, **identification**, **dangling**, **induced**, and **exact** see Section 4.5.

EXAMPLE (10)

We extend `SomeRule` (Example 4) with a user defined node to match and we want it to return the rewritten graph elements `n5` and `e1`.

```

1  rule SomeRuleExt(varnode:Node):(Node, EdgeTypeB) {
2      n1:NodeTypeA;
3      ...
4
5      replace {
6          varnode;
7          ...
8          return(n5, e1);
9      eval {
10         ...

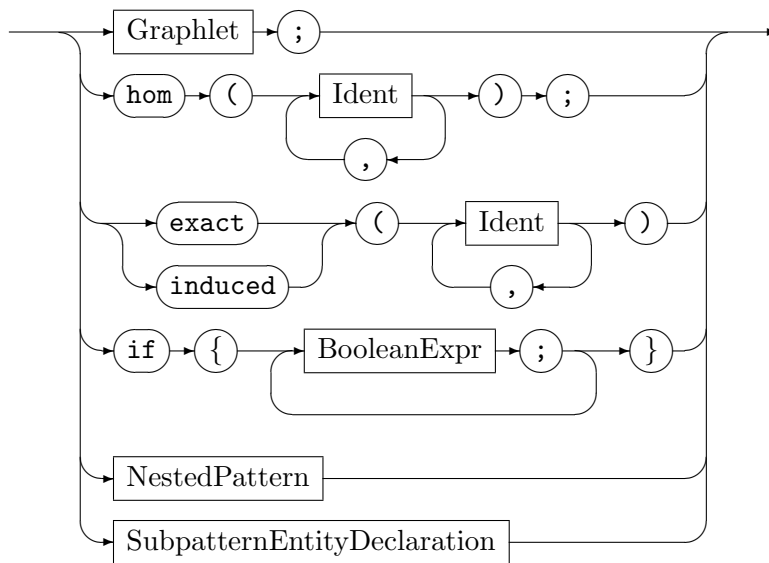
```

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

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 4.1 for a detailed explanation of names and graphlets.

NOTE (15)

The application of a rule is not deterministic (remember the example of the introduction in Section 1.6); in particular there may be more than one subgraph that matches the pattern. Whereas the `GRSHELL` selects one of them arbitrarily (without further abilities to control the selection), the underlying `LIBGR` provides a mechanism to deal with such ambiguities. Also notice that graph rewrite *sequences* introduce a further variant of non-determinism on rule application level: The `$<op>` flag marks the operator `<op>` as commutative, i.e. the execution order of its operands (rules) is non-deterministic. See Chapter 7 for further information on graph rewrite sequences.

PatternStatement

The semantics of the various pattern statements are given below:

Graphlet

Graphlets specify connected subgraphs. See Section 4.1 for a detailed explanation of graphlets.

Isomorphic/Homomorphic Matching

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 4 nodes **n1** and **n2** may be the same node. This is possible because they are of the same type (**NodeTypeA**). Inside a NAC the **hom** operator may only operate on graph elements that are either defined or used in the NAC. Nested **negative/independent** blocks inherit the **hom** declarations of their nesting pattern. In contrast to previous versions **hom** declarations are non-transitive, i.e **hom(a,b)** and **hom(b,c)** don't cause **hom(a,c)** unless specified.

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 5) 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 4.5 for an explanation of the **exact** and **induced** modifiers.

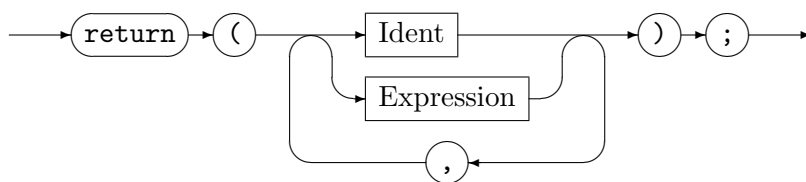
NestedPattern

will be explained in 6.1,6.2,6.3,6.4.

SubpatternEntityDeclaration

will be explained in 6.5.

Keep in mind that using type constraints or the **typeof** operator might be helpful. See Section 5.9 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 4.2, *ActionSignature*). Their types must be compatible to the return types specified.

NOTE (16)

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:

```

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 }
```

4.4 Replace/Modify Part

Besides specifying the pattern, a main task of a rule is to specify the transformation of a matched subgraph within the host graph. Such a transformation specification defines the transition from the left hand side (LHS) to the right hand side (RHS), i.e. which graph elements of a match will be kept, which of them will be deleted and which graph elements have to be newly created.

4.4.1 Implicit Definition of the Preservation Morphism r

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

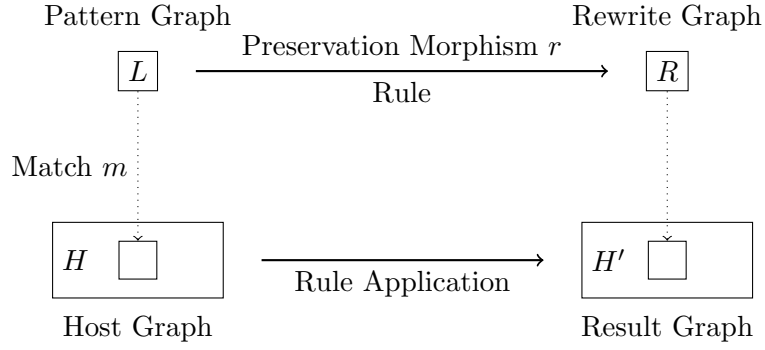


Figure 4.1: Process of Graph Transformation

Pattern (LHS)	Replace (RHS)	$r : L \rightarrow R$	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
$-e:T->$	$-e-> x:\text{Node};$	—	Illegal, redirection of e
$x:N \text{ } -e:E-> y:N;$	$x \text{ } -e-> ;$	$r : \{\text{lhs}.x\} \mapsto \{\text{rhs}.x\}$	Deletion of y . Hence deletion of e .

Table 4.1: Definition of the preservation morphism r

4.4.2 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 (occur) in the replace part, keep every graph element that is used, and create every additionally defined graph elements. “Using” means that the name of a graph element occurs in a replace graphlet. Attribute calculations are no 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 4.4.1 for a detailed explanation of the transformation semantics.

Modify mode

The modify mode can be regarded as a replace part in replace mode, where every pattern graph element is added (occurs) before the first replace statement. In particular all the anonymous graph elements are kept. Additionally this mode supports the `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 (17)

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 replace mode is advantageous, pinpointing what should stay. (Furthermore it might be simpler to generate code for, just dumping both patterns.)

EXAMPLE (11)

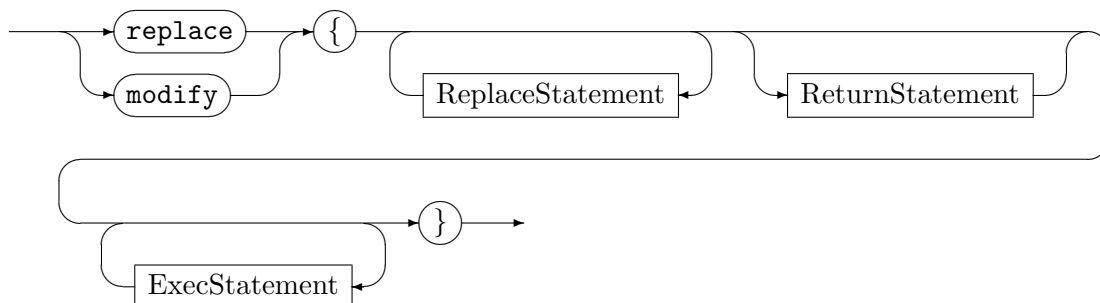
How might Example 10 look in modify mode? We have to denominate the anonymous edge between `n1` and `n2` in order to delete it. The node `varnode` should be kept and does not need to appear in the modify part. So we have

```

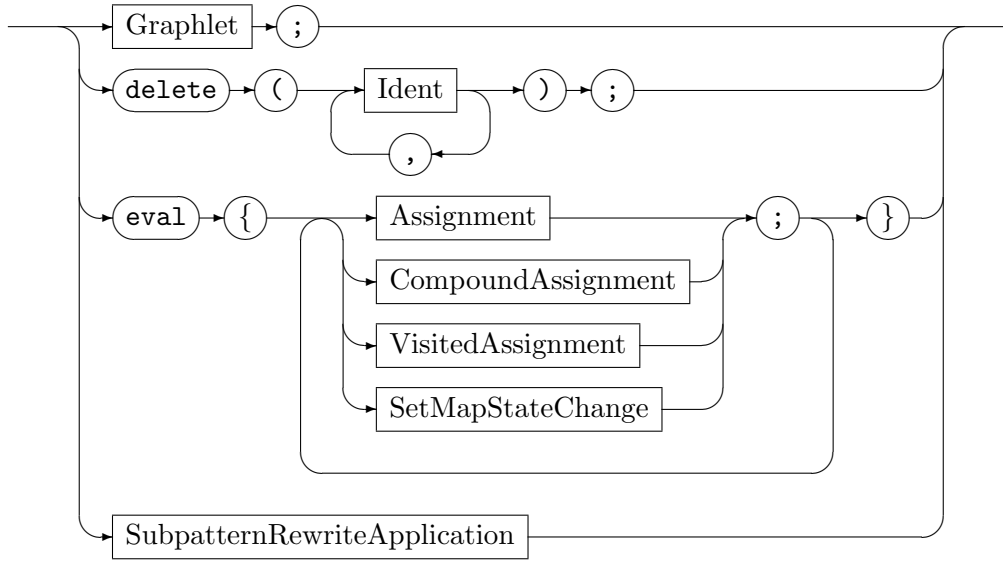
1 rule SomeRuleExtModify(varnode: Node): (Node, EdgeTypeB) {
2   ...
3   n1 -e0:Edge-> n2;
4   ...
5   modify {
6     n5:NodeTypeC<n1>;
7     n3 -e1:EdgeTypeB-> n5;
8     delete(e0);
9     eval {
10      ...

```

4.4.3 Syntax

Replace

Selects whether the replace mode or the modify mode is used. Several replace 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. Regarding rules it can only be given in the rewrite part. The *ExecStatement* will be introduced in chapter 8.

ReplaceStatement

The semantics of the various replace statements are given below:

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 replace graphlet. See Section 4.1 for a detailed explanation of graphlets.

Deletion

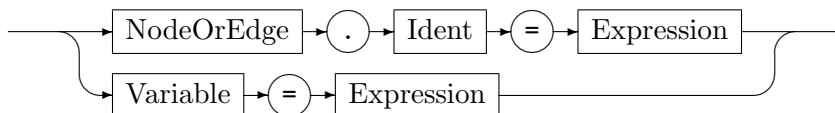
The **delete** operator is only available in the modify mode. It deletes the specified pattern graph elements. Multiple occurrences of **delete** statements are allowed. Deletion statements are executed after all other replace statements. Multiple deletion of the same graph element is allowed (but pointless)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

Attribute Evaluation

If a rule is applied, then the attributes of matched and inserted graph elements will be recalculated according to the **eval** statements. The *Assignment* will be explained below, the *CompoundAssignment*, *VisitedAssignment* and *SetMapStateChange* clauses will be introduced in chapter 8.

SubpatternRewriteApplication

will be explained in 6.7.

Assignment

Several evaluation parts are allowed within the replace part. Multiple evaluation statements will be internally concatenated, preserving their order. Evaluation statements have imperative semantics. In particular, GRGEN.NET does not care about data dependencies. Evaluation takes place before any graph element gets deleted and after all the new elements have been created. You may read (and write, although this is pointless) attributes of graph elements to be deleted. Assignment is carried out using value semantics, even for entities of **set**<K> and **map**<K,V> or **string** type. The only exception is the type **object**, there reference

semantics is used. The variable assignments are only useful in a few limited situations due to parameter passing being by-value. You can find out about the available *Expressions* in chapter 5.

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 $y.i = 42$, $y.j = 1$.

4.5 Rule and Pattern Modifiers

By default GRGEN.NET performs rewriting according to semantics as explained in Section 4.4.1. This behaviour can be changed with *pattern modifiers* and *rule modifiers*. 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 4.2 lists the pattern modifiers with their semantics, table 4.3 lists the rule only modifiers with their semantics. Example 13 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 4.2: Semantics of pattern modifiers

NOTE (18)

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.

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 by all means. 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 4.3: Semantics of rule only modifiers

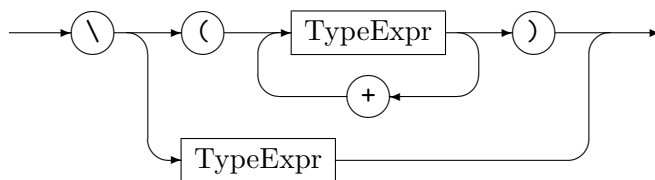
4.6 Static Type Constraint and Exact Dynamic Type

In the following section we'll have a look at a language construct to statically constrain the types to match by excluding forbidden types and 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.

4.6.1 Static Type Constraint

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

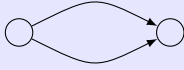
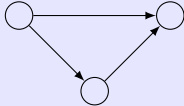

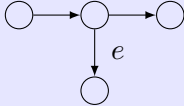
TypeConstraint

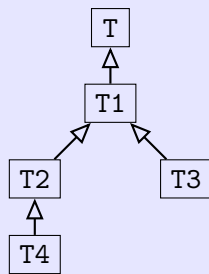


A type constraint is used to exclude parts of the type hierarchy. The operator **+** is used to create a union of its operand types. So the following pattern statements are identical:

<code>x:T \ (T1 + ... + Tn);</code>	<code>x:T; if {!(typeof(x) >= T1) && ... && !(typeof(x) >= Tn)}</code>
-------------------------------------	--

EXAMPLE (13)

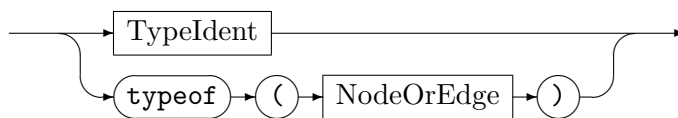
Host Graph	Pattern / Rule	Effect
	<code>{ . --> .; }</code>	Produces no match for exact nor induced
	<code>{ x:node --> y:node; }</code>	Produces three matches for induced(x,y) but no match for exact(x,y)
	<code>{ x:node; induced(x); }</code>	Produces no match
	<code>pattern{ --> x:node --> ; }</code>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

EXAMPLE (14)

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

4.6.2 Exact Dynamic Type

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 `e1: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 it 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 brother of this language construct, the **copy** operator, which is only applicable in the rewrite part.

EXAMPLE (15)

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.

4.7 Retyping and Copying

In the following section we'll have a look at a language construct which retypes an element to another type keeping incident elements (either to a type statically given or the runtime type of another element), and at the copy operator which allows to create a new element of the runtime type of another matched element, transferring the attribute values.

4.7.1 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)	Replace (RHS)	$r : L \longrightarrow R$	Meaning
$x:N1;$	$y:N2<x>;$	$r : lhs.x \mapsto rhs.x$	Preserve x , then retype x from N1 to N2 and bind name y to retyped version of x .
$e:E1;$	$f:E2<e>;$	$r : lhs.e \mapsto rhs.e$	Preserve e , then retype e from E1 to E2 and bind name f to the retyped version of e .

Table 4.4: Retyping of preserved 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 4.4 shows how retyping can be expressed both for nodes and edges). Retyping differs from a type cast: During replacement 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 4 node *n5* is a retyped node stemming from node *n1*. Note, that—conceptually—the retyping is performed *after* the SPO conform rewrite.

EXAMPLE (16)

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.

```

1 rule oneway {
2   x:City -y:Street->;
3
4   replace {
5     x_rt:Metropolies<x> -y_rt:Highway<y>->;
6   }
7 }

```

4.7.2 Copy

The copy operator allows to create a node or edge of the type of another node/edge, bearing the same attributes as that other node. It 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 it allows to simulate node replacement grammars or to copy entire structures, see 8.3 and 8.4.

CopyOperator



EXAMPLE (17)

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.

```

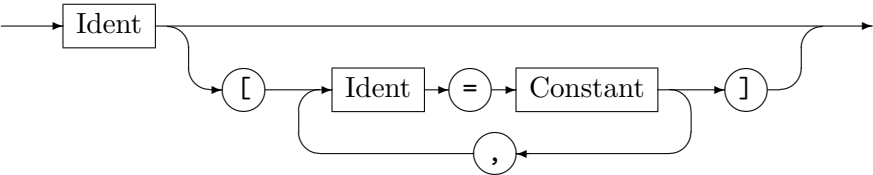
1 rule oneway {
2   a:Node -x:street-> y:Node;
3   negative {
4     y -:typeof(x)-> a;
5   }
6
7   replace {
8     a -x-> y;
9     y -:copy<x>-> a;
10  }
11 }

```

4.8 Annotations

Identifier definitions can be annotated by pragmas. Annotations are key-value pairs.

IdentDecl



Although you can use any key-value pairs between the brackets, only the identifier prio has an effect so far. You may use the annotations to transmit information from the specification files to API level where they can be enumerated.

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.

Table 4.5: Annotations

EXAMPLE (18)

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 like `v[prio=10000]:NodeTypeA -e[prio=5000]:EdgeType-> w:NodeTypeB`.

CHAPTER 5

TYPES AND EXPRESSIONS

5.1 Built-In Types

Besides user-defined node types, edge types, and enumeration types (as introduced in Chapter 3), GRGEN.NET supports the built-in primitive types in Table 5.1 and the built-in generic types in Table 5.2. The exact type format is backend specific. The LGSPBackend maps the GRGEN.NET primitive types to the corresponding C# primitive types, and the generic types to generic C#-Dictionaries of their corresponding primitive types (i.e. hashmaps), with `de.unika.ipd.grGen.libGr.SetValueType` as target type for sets, only used with the value `null`.

boolean	Covers the values true and false
int	A signed integer with at least 32 bits
float, double	A floating-point number with single precision or double precision respectively
string	A character sequence of arbitrary length
object	Contains a .NET object

Table 5.1: GRGEN.NET built-in primitive types

set<T>	A (mathematical) set of type T, where T may be an enumeration type or one of the primitive types from above; it may even be a node or edge 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 above; it may even be a node or edge type, then we speak of storages

Table 5.2: GRGEN.NET built-in generic types

Table 5.3 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.

According to table 5.3 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. 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.

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

Table 5.3: GRGEN.NET type casts

EXAMPLE (19)

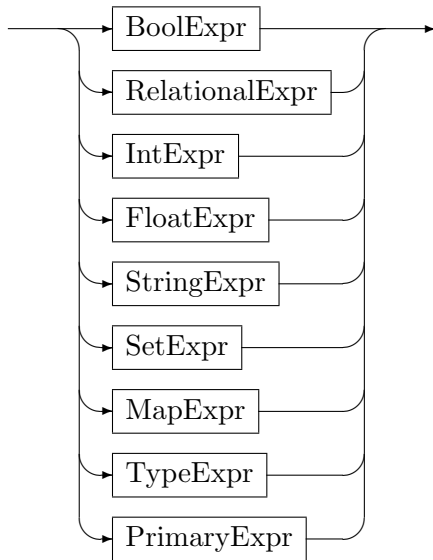
- Allowed:
`x.myfloat = x.myint; x.mydouble = (float) x.myint;`
`x.mystring = (string) x.mybool;`
- Forbidden:
`x.myfloat = x.mydouble; and x.myint = (int) x.mybool;`
`MyEnum1 = (MyEnum1Type) int; and MyEnum2 = (MyEnum2Type) MyEnum1;` where
`myenum1` and `myenum2` are different enum types.

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 3.2). The reason is, that the range of an enum type is just defined in that context.

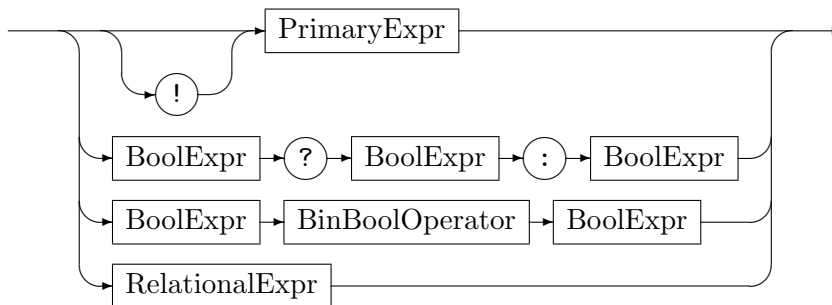
5.2 Expressions

GRGEN.NET supports numerous operations on the entities of the types introduced above, which are organized into left associative expressions. 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

5.3 Boolean Expressions

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

BoolExpr

The unary `!` operator negates a Boolean. The binary *BinBoolOperator* is one of the operators in Table 5.4. The ternary `?` operator is a simple if-then-else: If the first *BoolExpr* is evaluated

<code>^</code>	Logical XOR. True, iff either the first or the second Boolean expression is true.
<code>&&</code> <code> </code>	Logical AND and OR. Lazy evaluation.
<code>&</code> <code> </code>	Logical AND and OR. Strict evaluation.

Table 5.4: Binary Boolean operators, in ascending order of precedence

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

5.4 Relational Expressions

The relational expressions compare entities of different kinds, mapping them to the type boolean. 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:

`< <= == != >= >`

Their semantics are type dependent.

For arithmetic expressions on `int` and `float` or `double` types the semantics is given by Table 5.5 (by implicit casting they can also be used with all enum types).

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

Table 5.5: Compare operators on arithmetic expressions

`String` types, `boolean` types, and `object` types support only the `==` and the `!=` operators; for strings they denote whether the strings are the same or not, 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 set and map expressions, table 5.6 describes the semantics of the compare operators.

<code>A == B</code>	True, iff <i>A</i> and <i>B</i> are identical.
<code>A != B</code>	True, iff <i>A</i> and <i>B</i> are not identical.
<code>A < B</code>	True, iff <i>A</i> is a subset/map of <i>B</i> , but <i>A</i> and <i>B</i> are not identical.
<code>A > B</code>	True, iff <i>A</i> is a superset/map of <i>B</i> , but <i>A</i> and <i>B</i> are not identical.
<code>A <= B</code>	True, iff <i>A</i> is a subset/map of <i>B</i> or <i>A</i> and <i>B</i> are identical.
<code>A >= B</code>	True, iff <i>A</i> is a superset/map of <i>B</i> or <i>A</i> and <i>B</i> are identical.

Table 5.6: Compare operators on set/map expressions

For type expressions the semantics of compare operators are given by table 5.7, the rule to remember is: types grow larger with extension/refinement. An example is given in 5.9.

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

Table 5.7: Compare operators on type expressions

NOTE (20)

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

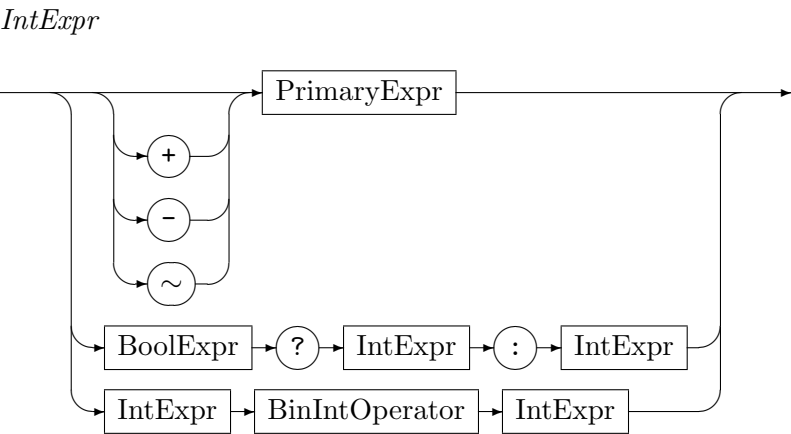
NOTE (21)

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

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

5.5 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).

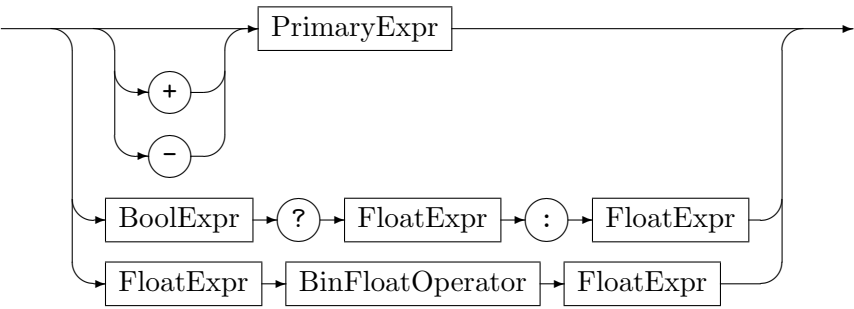


The \sim operator is the bitwise complement. That means every bit of an integer value will be flipped. The $?$ operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *IntExpr*, otherwise it returns the second *IntExpr*. The *BinIntOperator* is one of the operators in Table 5.8.

\wedge	Bitwise XOR, AND and OR
$\&$	
$ $	
\ll	Bitwise shift left, bitwise shift right and bitwise shift right preserving the sign
\gg	
\ggg	
$+$	Addition and subtraction
$-$	
$*$	Multiplication, integer division, and modulo
$/$	
$\%$	

Table 5.8: 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 5.9.

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

Table 5.9: Binary float operators, in ascending order of precedence

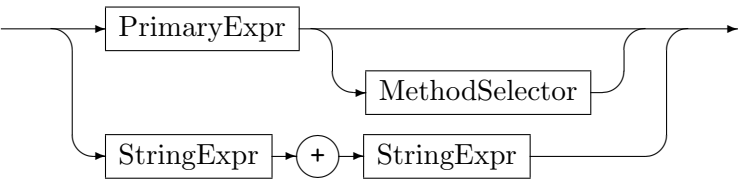
NOTE (22)

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

5.6 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



The operator **+** concatenates two strings. There are several operations on strings available in method call notation (**MethodSelector**), these are

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

NOTE (23)

The declarative rule language comes without the imperative set `s.add(x)` or `s.rem(x)` methods known from the XGRS, to add a value to a set use set union with a single valued set constructor, to remove a value from a set use set difference with a single valued set constructor (for set constructors cf. 5.10).

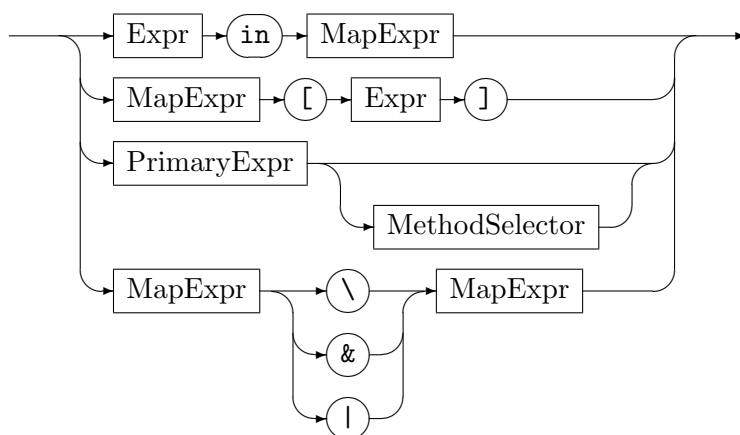
```
1 s | { "foo" }
2 s \ { n.a }
```

Used in this way they get internally optimized to set addition and removal.

5.8 Map Expression

Map expressions consist of the known mathematical set operations extended to maps, plus map value lookup and size counting.

MapExpr



	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 5.11: Binary map operators, in ascending order of precedence

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 s` denotes map domain membership $x \in \text{dom}(s)$, returning whether the domain of the set contains the given element, as `boolean`. 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. There are several operations on maps available in method call notation (`MethodSelector`), these are:

`.size()`

returns the number of elements in the map, as `int`

`.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>`

`.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

NOTE (24)

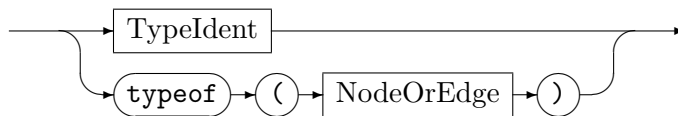
The declarative rule language comes without the imperative `m.add(x,y)` or `m.rem(x)` methods known from the XGRS, and *without* a mapping assignment operator `m[x]=y`, map lookup returns only a RHS value. To add a key,value-pair to a map use map union with a single valued map constructor, to remove a value from a map use map difference with a single valued set or map constructor (for map constructors cf. 5.10).

```
1 m | { "foo" -> 42 }
2 m \ { n.a -> n.b } or m \ { n.a }
```

Used in this way they get internally optimized to map addition and removal.

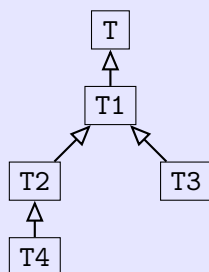
5.9 Type Expressions

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.

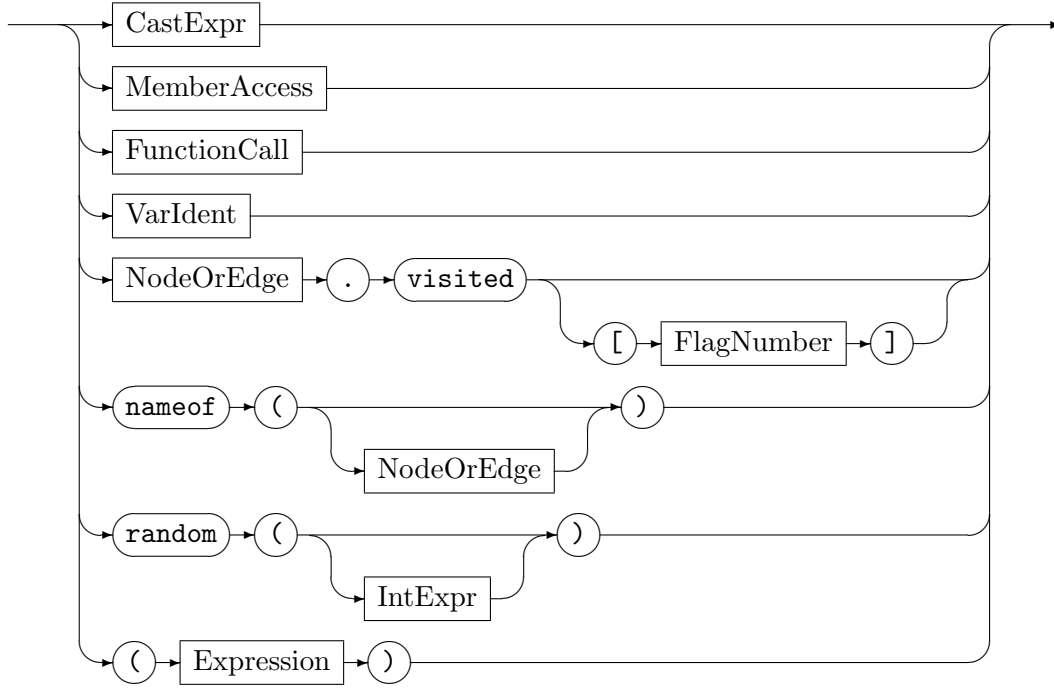
EXAMPLE (21)



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`.

5.10 Primary expressions

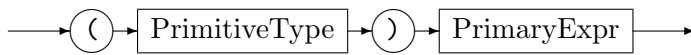
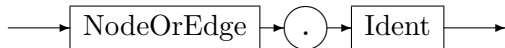
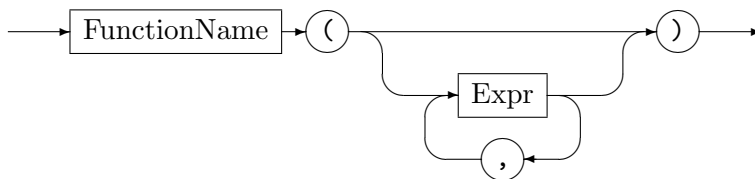
After we've seen the all the ways to combine expressions, finally we'll have a look at the atoms the expressions are built of.

PrimaryExpr

The **visited** query returns the status of the visited flag of the given number for the given graph element as **boolean**. If no flag number is given, the default number for the first visited flag of 0 is used. The visited flags are written in the assignments of the eval statements (see section 4.4). Make sure to allocate 8.2.1/11.3 visited flags before you try to use them (and deallocate them afterwards, as they are a sparse resource stored in some excess bits of the graph elements, or in some dictionary if the needed number of flags exceeds the number of available bits per graph element.)

The **nameof** query returns the name (persistent name, see example 69) of the given graph element as **string**; graphs elements at the API level bear no name, the operator can only (sensibly) be used with Shell-graphs (**NamedGraph**; unless you decide to use the **NamedGraph** on API level, too). If no graph element is given, the name of the 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 value given, excluding the value itself.

CastExpr*MemberAccess**FunctionCall*

The cast expression returns the original value casted to the new prefixed type. The member access **n.a** returns the value of the attribute **a** of the graph element **n**. A function

call employs an external (attribute evaluation) function (cf. 3.2.4) or one of the following built-in functions:

min(.,.)

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

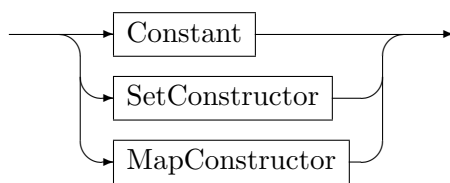
max(.,.)

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

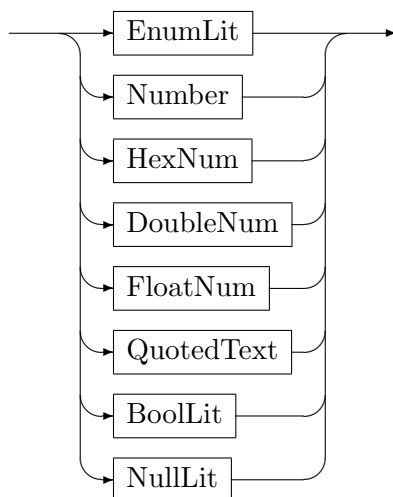
pow(.,.)

returns the first argument value to the power of the second value; both must be of type `double`

Literal



Constant



The Constants are:

EnumLit

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

Number

Is an `int` number in decimal notation without decimal point.

HexNum

Is an `int` number in hexadecimal notation starting with `0x`.

DoubleNum

Is a `double` number in decimal notation with decimal point, maybe postfixed with `d`.

FloatNum

Is a `float` number in decimal notation with decimal point, postfixed with `f`.

QuotedText

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

5.11 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 5.12: All operators, in ascending order of precedence

CHAPTER 6

NESTED AND SUBPATTERNS

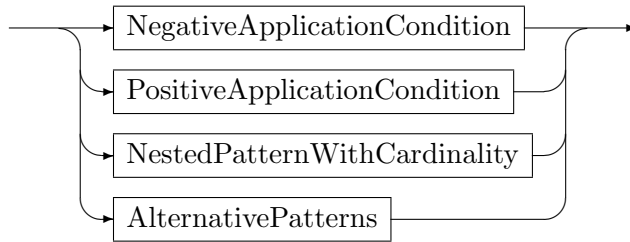
The extension of the rule set language described in the chapter 4 by nested patterns and subpatterns greatly enhances the flexibility and expressiveness of pattern matching and rewriting. The following patterns to match a simplified abstract syntax tree give a rough picture of the language of nested and subpatterns:

EXAMPLE (23)

```
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 expressions
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 }
```

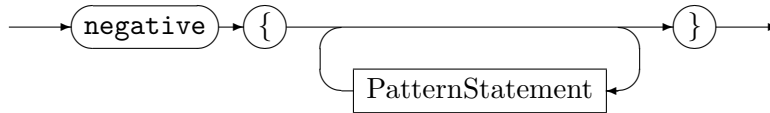
Until now we have seen rules and tests with one left hand side static pattern specification in a direct 1:1 correspondence with its dynamic match in the host graph on a successful application. From now on we will increase the expressiveness of the pattern language, and dependent on it the rewrite language, to describe much finer and more flexible 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 exists 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), we will start with the nested patterns:

NestedPattern



6.1 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 (24)

We specify a variable which is not badly typed, i.e. a node **x** of type **Variable** which must not be 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 (25)

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. 4.1.1).

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 (26)

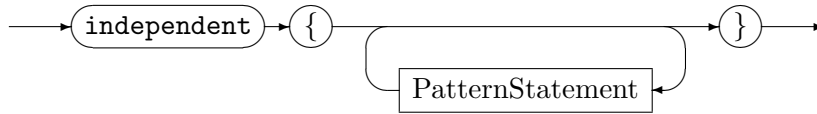
A fabricated example using parallel as well as nested negatives:

```

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 }
```

6.2 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 really needed if subpatterns want to match elements which were already matched during the subpattern derivation.

EXAMPLE (27)

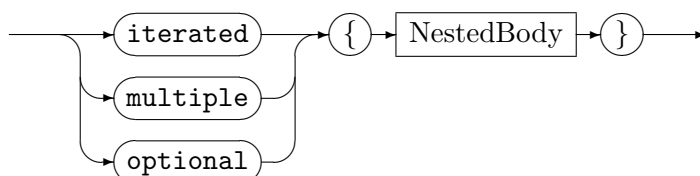
A further fabricated example rather giving the intention using **independent** patterns to check some conditions with only the main pattern available to rewriting:

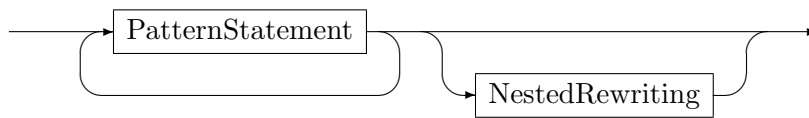
```

1 rule moveMethod
2 {
3   c:Class --> m:Method;
4   csub -:extending-> c;
5   csub:Class -e:Edge-> msub:Method;
6
7   independent {
8     // a complicated pattern to find out that m and msub have same signatures
9   }
10  independent {
11    // a complicated pattern to find out that msub is only using variables available in c
12  }
13  independent {
14    // a complicated pattern to find out that m is not used
15  }
16
17  modify { // move method upwards
18    delete(m);
19    delete(e);
20    c --> msub;
21  }
22 }
```

6.3 Pattern Cardinality

NestedPatternWithCardinality



NestedBody

The blocks allow to specify 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 nested body will be explained in Section 6.6.)

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 (28)

```

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

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 (29)

```

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

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 (30)

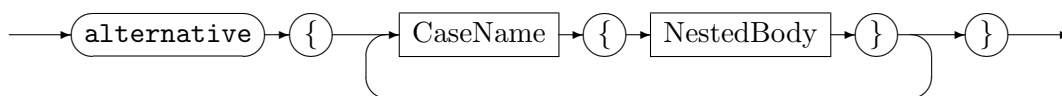
```

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 }

```

NOTE (25)

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).

6.4 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 (31)

```

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 (32)

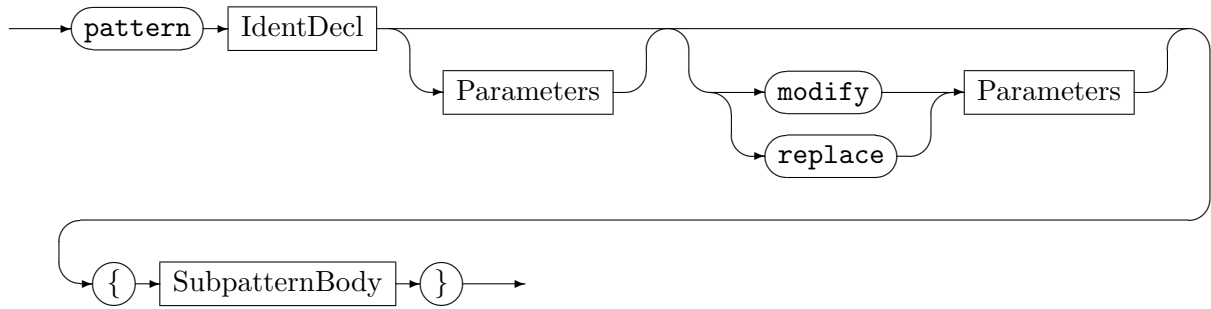
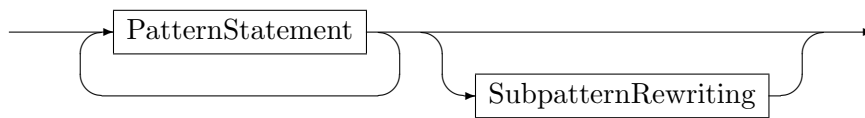
```

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 }

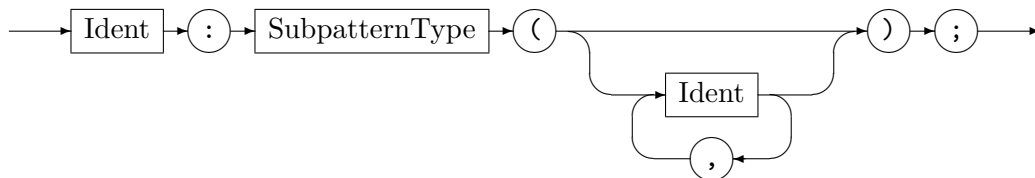
```

6.5 Subpattern Declaration and Subpattern Entity 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 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, due to the fact that a subpattern can't return anything (they are not actions, just a helper in constructing complex patterns). Subpatterns can receive additional rewrite parameters 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. (The nested body will be explained in Section 6.7.)

SubpatternEntityDeclaration

Subpattern entity declarations 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.

EXAMPLE (33)

```
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 }
```

6.5.1 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.

EXAMPLE (34)

```

1 test iteratedPath
2 {
3   root:Assign;
4   negative { --> root; }
5   :IteratedPath(root); // match iterated path = assignment list
6 }
7
8 pattern IteratedPath(prev:Node)
9 {
10  optional { // nothing or a linked assignment and again a list
11    prev --> a:Assign; // assignment node
12    :IteratedPath(a); // next one, plz
13  }
14 }

```

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)

```

1 pattern IteratedPath(prev:Node)
2 {
3   alternative {
4     Empty {
5       negative {
6         prev --> a:Assign;
7       }
8     }
9     Further {
10      prev --> a:Assign;
11      :IteratedPath(a);
12    }
13  }
14 }

```

EXAMPLE (35)

```

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.

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 better suited to this task). But combined with an iterated block, you may even match structures extending into breadth and depth, like e.g. a hierarchy of classes (i.e. match a spanning tree in the graph) giving 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 (26)

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 [9.2.8/41](#)).

EXAMPLE (36)

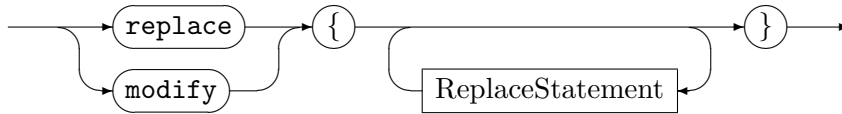
```

1 pattern SpanningTree(root:Class)
2 {
3   iterated {
4     root <:-:extending- next:Class;
5     :SpanningTree(next);
6   }
7 }

```

6.6 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. 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, which must be present even if the top level pattern is empty. 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 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 (37)

```

1 rule methods
2 {
3   c:Class;
4   iterated {
5     c --> m:Method;
6
7     replace {
8       c <-- m;
9     }
10  }
11
12  replace {
13    c;
14  }
15 }

```

EXAMPLE (38)

```

1 rule methodWithTwoOrThreeParameters(m:Method)
2 {
3   alternative {
4     Two {
5       m <-- n:Name;
6       m <-e1:Edge- v1:Variable;
7       m <-e2:Edge- v2:Variable;
8       negative {
9         v1; v2; m <-- :Variable;
10      }
11
12      modify {
13        delete(e1); m --> v1;
14        delete(e2); m --> v2;
15      }
16    }
17    Three {
18      m <-- n:Name;
19      m <-e1:Edge- v1:Variable;
20      m <-e2:Edge- v2:Variable;
21      m <-e3:Edge- v3:Variable;
22
23      modify {
24        delete(e1); m --> v1;
25        delete(e2); m --> v2;
26        delete(e3); m --> v3;
27      }
28    }
29
30    modify {
31    }
32 }

```

EXAMPLE (39)

This is an example which shows how to decide with an alternative on the target type of a retyping depending on the context.

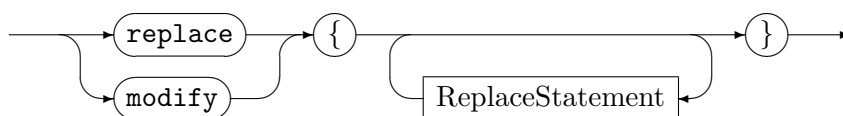
```

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
24  modify {
25  }
26 }
```

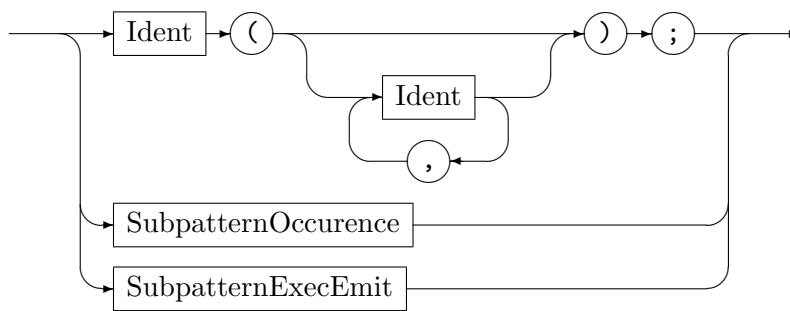
6.7 Subpattern rewriting

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 looks like a nested rewriting specification, but additionally there may be rewrite parameters given in the subpattern header (cf. 6.5) 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 *ReplaceStatement* already introduced (cf. 4.4.3). 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 *SubpatternOccurence* is explained in the next subsection 6.7.1. The *SubpatternExecEmit* is explained in chapter 8.

EXAMPLE (40)

This is an example for a subpattern rewrite application.

```

1 pattern TwoParametersAddDelete(mp:Method)
2 {
3   mp <-- v1:Variable;
4   mp <-- :Variable;
5
6   modify {
7     delete(v1);
8     mp <-- :Variable;
9   }
10 }
11 rule methodAndFurtherAddDelete
12 {
13   m:Method <-- n:Name;
14   tp:TwoParametersAddDelete(m);
15
16   modify {
17     tp(); // trigger rewriting of the TwoParametersAddDelete instance
18   }
19 }

```

EXAMPLE (41)

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

```
1 pattern IteratedPathReverse(prev:Node)
2 {
3   optional {
4     prev --> next:Node;
5     ipr:IteratedPathReverse(next);
6
7     replace {
8       prev <-- next;
9       ipr();
10    }
11  }
12
13  replace {
14  }
15 }
```

EXAMPLE (42)

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.

```

1 pattern ChainFromToReverseToCommon(from:Node, to:Node) replace(common:Node)
2 {
3   alternative {
4     rec {
5       from --> intermediate:Node;
6       cftrtc:ChainFromToReverseToCommon(intermediate, to);
7
8       replace {
9         from <-- intermediate;
10        from --> common;
11        cftrtc(common);
12      }
13    }
14    base {
15      from --> to;
16
17      replace {
18        from <-- to;
19        from --> common;
20        to --> common;
21      }
22    }
23  }
24
25  replace {
26    from; to;
27  }
28 }

```

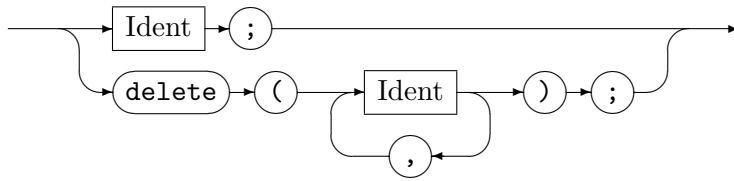
```

1 rule chainFromToReverseToCommon()
2 {
3   from:Node; to:Node;
4   cftrtc:ChainFromToReverseToCommon(from, to);
5
6   modify {
7     common:Node;
8     cftrtc(common);
9   }
10 }

```

6.7.1 Deletion and Preservation of Subpatterns

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

SubpatternOccurence

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 (43)

```

1 rule R {
2   m1:Method; m2:Method;
3   tp1:TwoParameters(m1);
4   tp2:TwoParameters(m2);
5
6   replace {
7     tp1; // is kept
8     // tp2 not included here - will be deleted
9     // tp1(); or tp2(); -- would apply dependent replacement
10    m1; m2;
11  }
12 }
```

NOTE (27)

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?).

```

1 pattern ForCreationOnly(mp:Method)
2 {
3   // some complex pattern you want to instantiate several times
4   // connecting it to the mp handed in
5 }
6 rule createSubpattern
7 {
8   m:Method;
9
10  modify {
11    :ForCreationOnly(m); // instantiate pattern ForCreationOnly
12  }
13 }
```

6.8 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; using it together with the subpatterns gives graph rewrite specifications which look like EBNF-grammars with embedded actions. Exceeding the more verbose syntax they offer constructs for matching the pattern a bounded number of times (same notation as the one for the bounded iteration in the xgrs).

<code>iterated { P }</code>	<code>(P)*</code>
<code>multiple { P }</code>	<code>(P)+</code>
<code>optional { P }</code>	<code>(P)?</code>
<code>alternative { l1 { P1 } .. lk { Pk } }</code>	<code>(P1 .. Pk)</code>
<code>negative { P }</code>	<code>~(P)</code>
<code>independent { P }</code>	<code>&(P)</code>
<code>modify { R }</code>	<code>{+ R }</code>
<code>replace { R }</code>	<code>{- R }</code>
<code>-</code>	<code>(P)[k] / (P)[k:1] / (P)[k:~]</code>

Table 6.1: Map of nested patterns in keyword syntax to regular expression syntax

EXAMPLE (44)

```

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 expressions
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 }
```


CHAPTER 7

RULE APPLICATION CONTROL LANGUAGE (XGRS)

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

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

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

Graph rewrite sequences are written down with a syntax similar to boolean and regular expressions. They are a means of composing complex graph operations out of single graph rewrite rules; they determine the control flow 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 assignment to their name, sequence local variables are explicitly declared with a typed variable declaration of the form `name:type`. 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 – if you save the graph in GRShell (or export to `.grs` using the `withvariables` option) 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 persistency 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 [45](#)).

EXAMPLE (45)

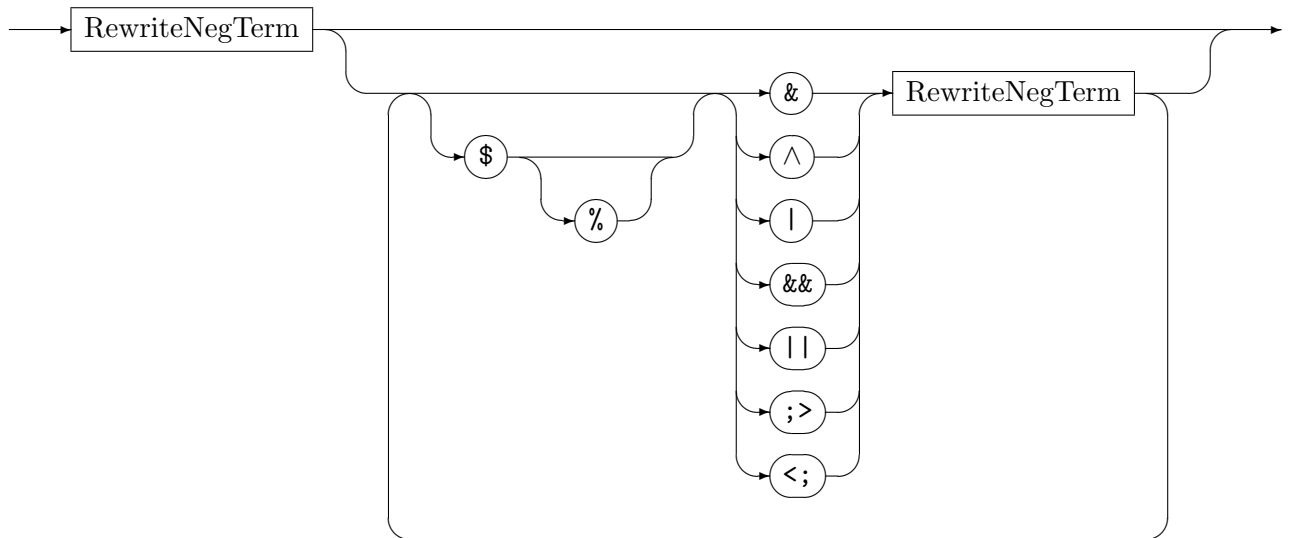
The graph rewrite sequence

1 $a = ((b, c) = R(x, y, z))$

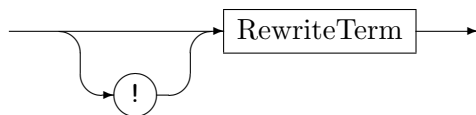
is valid. It assigns returned graph elements from rule R to variables b and c and the information whether R matched or not to variable a .

7.1 Logical and sequential connectives

RewriteSequence



RewriteNegTerm



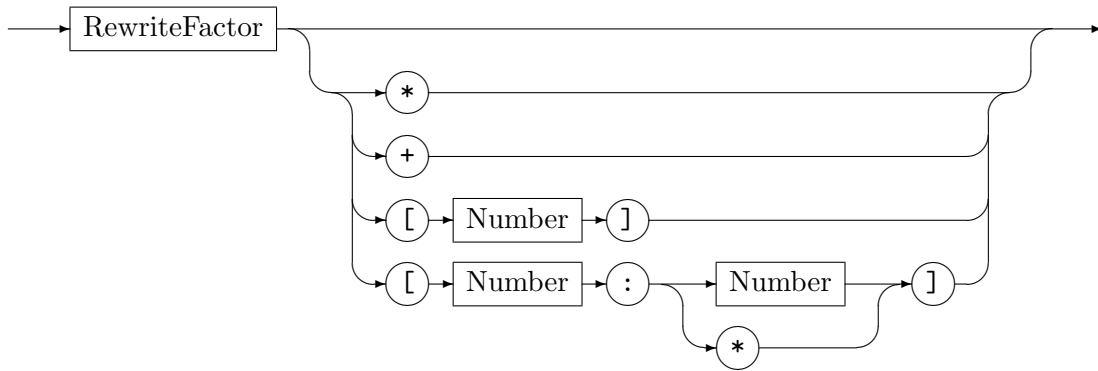
A graph rewrite sequence consists of several rewrite terms linked by operators. Table 7.1 gives the priorities and semantics of the operators, priorities in ascending order. The modifier $\$$ changes the semantics of the following operator to randomly execute the left or the right operand first (i.e. 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 $s \$\langle op \rangle t \$\langle op \rangle u$ are executed / evaluated in arbitrary order. The modifier $\%$ appended to the $\$$ overrides the random selection by a user selection (cf. see Section 9.3, choice points).

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
$s1 \wedge s2$	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 7.1: Semantics and priorities of rewrite sequence operators

7.2 Loops

RewriteTerm



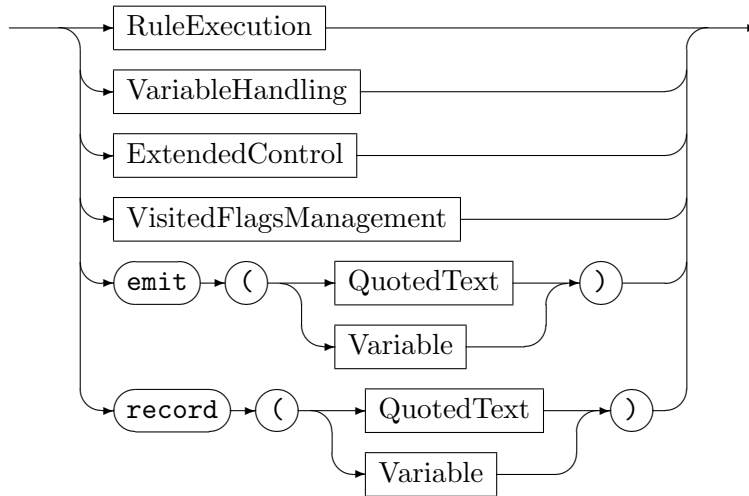
A rewrite term consists of a rewrite factor which can be executed multiple times. The star (*) executes a sequence repeatedly as long as its execution does not fail. Such a sequence always returns **true**. A sequence s^+ is equivalent to $s \&\& s^*$. The brackets ($[m]$) execute a sequence repeatedly 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.

NOTE (28)

Consider all-bracketing introduced in the next section for rewriting all matches of a rule instead of iteration if they are independent.

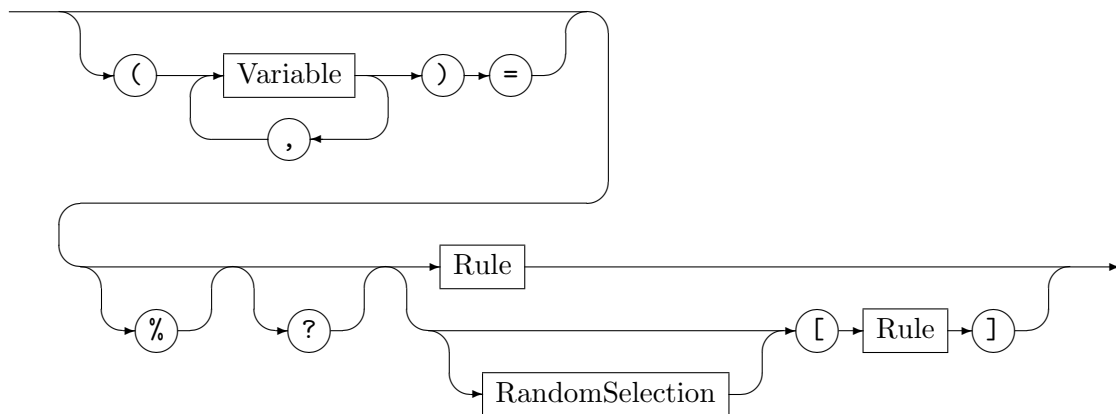
7.3 Rule application

RewriteFactor

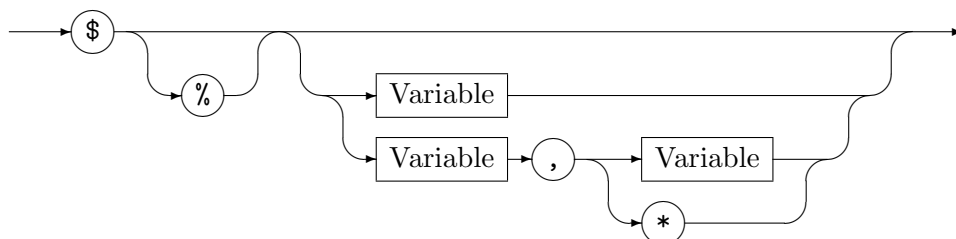


Rewrite factors are the building blocks of graph rewrite sequences. They split into four major areas: rule application, variable assignment, extended control, and visited flags management (plus the special area of storages to be described separately). Further on, there are the **emit** and **record** statements: the **emit** statement writes a double quoted string or the value of a variable to the emit target (stdout as default, or a file specified with the shell command **redirect emit**). The **record** statement writes a double quoted string or the value of a variable to the currently ongoing recordings (see 9.2.5). This feature allows to mark states reached during the transformation process in order to replay only interesting parts of an recording. It is recommended to write only comment label lines, i.e. **"#"**, some label, and **"\n"**.

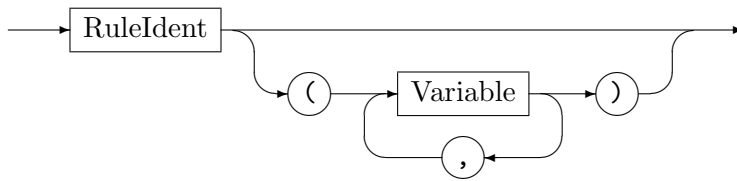
RuleExecution



RandomSelection



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 operator `?` 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 operator `%` is a multi-purpose flag. In the GRShell (see Chapter 9) it dumps the matched graph elements to `stdout`; in `debug`-mode (see Section 9.3) 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 1.7.3).

The square braces `[]` introduce a special kind of multiple rule application: Every pattern match produced by the rule will be rewritten; if at least one was found, rule application will succeed, otherwise it will fail. Attention: This 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). If *Rule* returns values, the values of *one* of the executed rules will be returned.

The random match selector `$v` searches for all matches and then randomly selects `v` of them to be rewritten (but at most as much as are available), with `$(r1)` being equivalent to `anonymousTempVar=1 & $anonymousTempVar[r1]`. Rule application will fail in case no match was found and succeed otherwise. You may change the lower bound for success by giving a variable containing the value to apply before the comma-separated upper bound variable. In case a lower bound is given the upper bound may be set to unlimited with the `*`. An `%` appended to the `$` denotes a choice point allowing the user to choose the match to be applied from the available ones in the debugger (see Section 9.3).

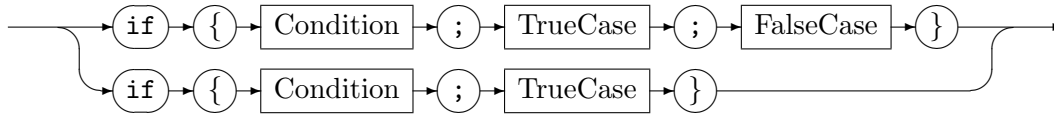
EXAMPLE (46)

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.

7.5 Extended control

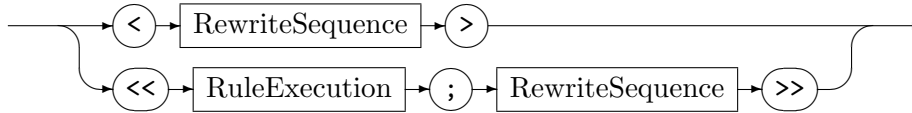
The extended control constructs offer further rule application control in the form of conditional sequences, transactions and backtracking, parenthesis, sequence constants, and indeterministic choice from a sets of rules or sequences.

ExtendedControl



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

ExtendedControl

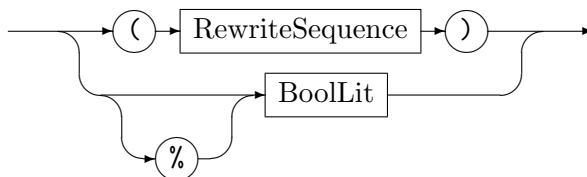


Graph rewrite sequences can be processed transactionally by using angle brackets (`<>`), i.e. if the return value is `false`, all the related operations on the host graph will be rolled back. Nested transactions are supported, i.e. a transaction which was committed is rolled back again if an enclosing transaction fails. Transactions 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: execute the rewrite of the match, then execute `s`. If `s` failed then rollback and continue with the loop. If `s` succeeded then commit and break from the loop.

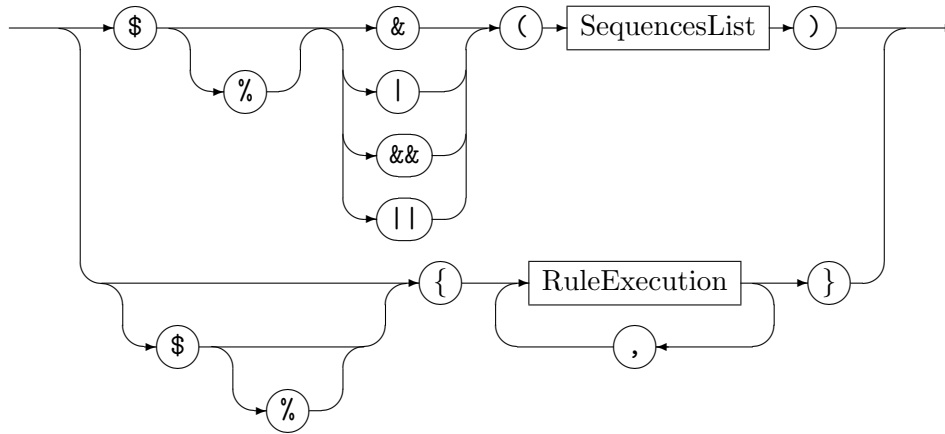
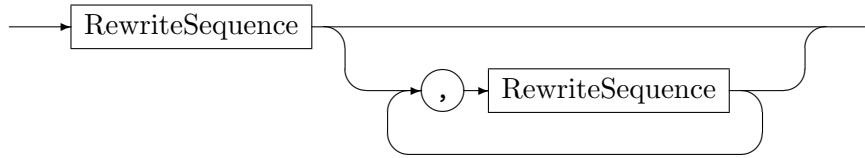
NOTE (29)

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.

ExtendedControl



Forcing execution order can be achieved by parentheses. Boolean literals `true/false` come in handy if a sequence is to be evaluated but its result must be a predefined value; furtheron a break point may be attached to them.

ExtendedControl*SequencesList*

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.

7.6 Quick reference table

Table 7.2 lists most of the operations of the graph rewrite expressions at a glance (the ones below the horizontal line are defined in the following chapter).

<code>s ;> t</code>	Execute <code>s</code> then <code>t</code> . Success if <code>t</code> succeeded.
<code>s <; t</code>	Execute <code>s</code> then <code>t</code> . Success if <code>s</code> succeeded.
<code>s t</code>	Execute <code>s</code> then <code>t</code> . Success if <code>s</code> or <code>t</code> succeeded.
<code>s t</code>	The same as <code>s t</code> but with lazy evaluation, i.e. if <code>s</code> is successful, <code>t</code> will not be executed.
<code>s & t</code>	Execute <code>s</code> then <code>t</code> . Success if <code>s</code> and <code>t</code> succeeded.
<code>s && t</code>	The same as <code>s & t</code> but with lazy evaluation, i.e. if <code>s</code> fails, <code>t</code> will not be executed.
<code>s ^ t</code>	Execute <code>s</code> then <code>t</code> . Success if <code>s</code> or <code>t</code> succeeded, but not both.
<code>if{r;s;t}</code>	Execute <code>r</code> . If <code>r</code> succeeded, execute <code>s</code> and return the result of <code>s</code> . Otherwise execute <code>t</code> and return the result of <code>t</code> .
<code>if{r;s}</code>	Same as <code>if{r;s;true}</code>
<code><s></code>	Execute <code>s</code> transactionally (rollback on failure).
<code><<r;s>></code>	Backtracking: try the matches of rule <code>r</code> until <code>s</code> succeeds.
<code>!s</code>	Switch the result of <code>s</code> from successful to fail and vice versa.
<code>\$<op></code>	Use random instead of left-associative execution order for <code><op></code> .
<code>s*</code>	Execute <code>s</code> repeatedly as long as its execution does not fail.
<code>s+</code>	Same as <code>s && s*</code> .
<code>s[n]</code>	Execute <code>s</code> repeatedly as long as its execution does not fail but <code>n</code> times at most.
<code>s[m:n]</code>	Same as <code>s[n]</code> but fails if executed less than <code>m</code> times.
<code>s[m:~]</code>	Same as <code>s*</code> but fails if executed less than <code>m</code> times.
<code>?Rule</code>	Switches <code>Rule</code> to a test.
<code>%Rule</code>	This is the multi-purpose flag when accessed from LIBGR. Also used for graph dumping and break points.
<code>[Rule]</code>	Rewrite every pattern match produced by the action <code>Rule</code> .
<code>def(Parameters)</code>	Check if all the variables are defined.
<code>true</code>	A constant acting as a successful match.
<code>false</code>	A constant acting as a failed match.
<code>#{r1,[r2],#[r3]}</code>	Tries to match all contained rules, then rewrites indeterministically one of the rules which matched. True if at least one matched.
<code>id=valloc()</code>	Allocates a visited-flag, returns its id.
<code>vfree(id)</code>	Frees the visited-flag given.
<code>vreset(id)</code>	Resets the visited-flag given in all graph elements.
<code>u=set<Node>{}</code>	Create storage set and assign to <code>u</code> .
<code>u.add(v)</code>	Add <code>v</code> to storage set <code>u</code> .
<code>u.rem(v)</code>	Remove <code>v</code> from storage set <code>u</code> .
<code>for{v in u; t}</code>	Execute <code>t</code> for every <code>v</code> in storage set <code>u</code> . One <code>t</code> failing pins the execution result to failure.
<code>u.clear()</code>	Clears the storage set <code>u</code> .
<code>v in u</code>	Membership query: succeeds if <code>v</code> is element of <code>u</code> , otherwise fails.
<code>u=map<N,Edge>{}</code>	Create storage map and assign to <code>u</code> . Operations are the same or similar to the operations of storage sets.
<code>v=u[w]</code>	Assign target value of <code>w</code> in <code>u</code> to <code>v</code> . Fails if <code>!(w in u)</code> .

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

Table 7.2: GRS expressions at a glance

CHAPTER 8

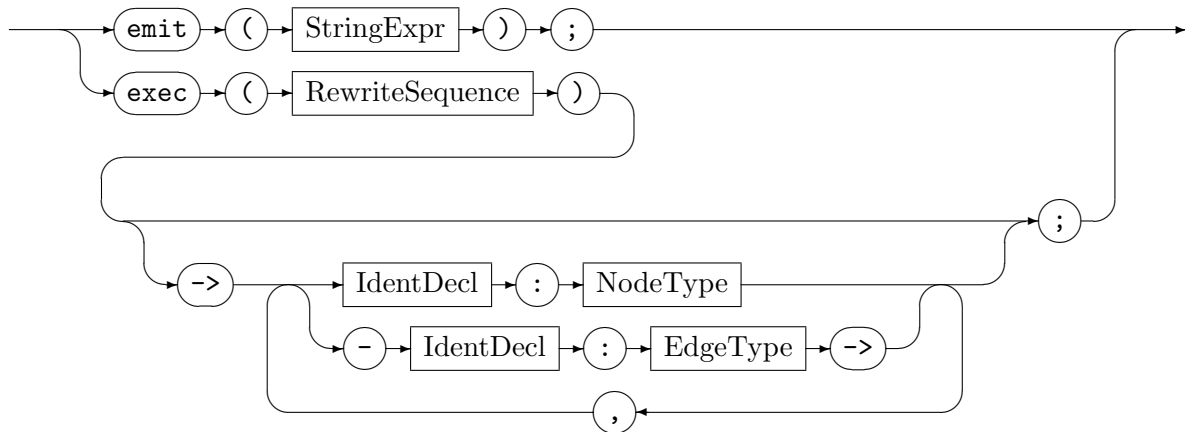
EMBEDDED SEQUENCES AND STORAGES

In this chapter we'll have a look at language constructs which allow to emit text from rules/-subpatterns and which allow to execute a graph rewrite sequence at the end of a rule invocation. The ability to execute a sequence at the end of a rule invocation allows to combine rules and build complex rules. Furthermore we'll have a look at language constructs which allow to build transformations by introducing state as a communications means between several rule applications. Finally we apply these techniques to accomplish certain common transformation tasks.

8.1 Imperative Statements

8.1.1 Exec and emit in rules

ExecStatement



The statements **emit** and **exec** enhance the declarative rewrite part by imperative clauses. That means, i) these statements are executed in the same order as they appear within the rule, and ii) they are executed after all the rewrite operations are done, i.e. they operate on the modified host graph. However, *attribute* values of deleted graph elements are still available for reading. The **eval** statements are executed before the execution statements, i.e. the execution statements work on the recalculated attributes.

XGRS Execution

The **exec** statement executes a graph rewrite sequence, which is a composition of graph rewrite rules. Yielded graph elements may be included into the RHS pattern, but as of now they are only accessible from the **return** statement. See Chapter 7 for a description of graph rewrite sequences. The **exec** statement is one of the means available in GRGEN.NET to build complex rules and split work into several parts, see 8.3 for a discussion of this topic.

Text Output

The **emit** statement prints a string to the currently associated output stream (default is **stdout**). See Chapter 5 for a description of string expressions. For emitting in between the emits from subpatterns, there is an additional **emithere** statement available.

EXAMPLE (47)

The following example works on a hypothetical network flow. We don't define all the rules nor the graph meta model. It's just about the look and feel of the **exec** and **emit** statements

```

1 rule AddRedundancy
2 {
3   s: SourceNode;
4   t: DestinationNode;
5   modify {
6     emit ("Source_node_is_" + s.name + ".Destination_node_is_" + t.name + ".");
7     exec ( (x:SourceNode) = DuplicateNode(s) & ConnectNeighbors(s, x)* );
8     exec ( [DuplicateCriticalEdge] );
9     exec ( MaxCapacityIncidentEdge(t)* );
10    emit ("Redundancy_added");
11  }
12 }
```

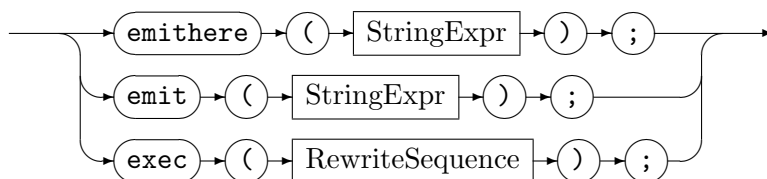
EXAMPLE (48)

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** statement is similar to a return but does not jump out. The anonymous output variables written in the yield statement are assigned in the given order to the pattern variables specified after the assign to operator **->**.

```

1 rule foo : (A,B)
2 {
3   modify {
4     exec( (a,b)=bar && yield(a,b) ) -> u:A, v:B;
5     return(u,v)
6   }
7 }
```

8.1.2 Deferred exec and emithere in nested and subpatterns

SubpatternExecEmit

The statements **emit**, **emithere** and **exec** enhance the declarative nested rewrite part by imperative clauses. The **emit** and **emithere** statements 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. The `exec` statements are executed i) after the rule which used their patterns was executed and ii) in the order as they appear within the rule. They are a slightly different version of the `exec`-statements from the *ExecStatement* introduced in 4.4.3, only available in the rewrite parts of subpatterns or nested alternatives/iterateds (but not in the rewrite part of rules as the original embedded sequences). They are executed after the original rule calling them was executed, so they can't get extended by `yields`, as the containing rule is not available any more when they get executed.

NOTE (30)

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 8.3 for a discussion on this topic.

NOTE (31)

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 sequences 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).

EXAMPLE (49)

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

```

1 rule caller
2 {
3   n:Node;
4   sub:Subpattern();
5
6   modify {
7     sub();
8     exec(r(n));
9   }
10 }
11 pattern Subpattern
12 {
13   n:Node;
14   modify {
15     exec(s(n));
16   }
17 }

```

EXAMPLE (50)

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

```

1 pattern BinaryExpression(root:Expr)
2 {
3   root --> l:Expr; le:Expression(l);
4   root --> r:Expr; re:Expression(r);
5   root <-- binOp:Operator;
6
7   modify {
8     le(); // rewrites and emits the left expression
9     emitthere(binOp.name); // emits the operator symbol in between the left tree and the
      right tree
10    re(); // rewrites and emits the right expression
11  }
12 }

```

8.2 Reading, writing, and managing state

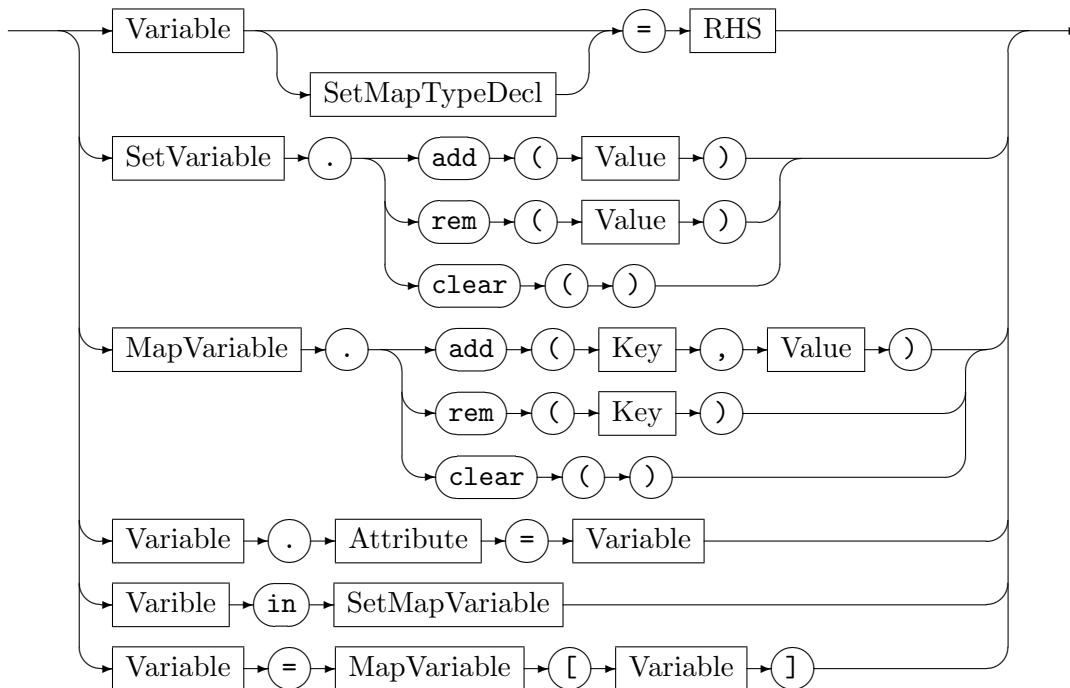
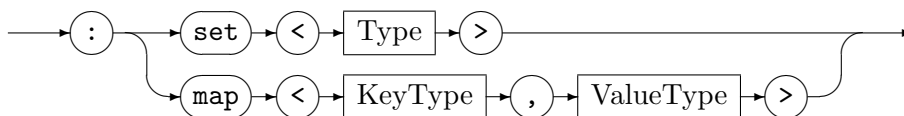
Besides the normal scalar variables from the rule application control language there are sets and map valued storages available for communicating some information between rule applications; additionally there are several visited flags per graph element available for this means. Both allow to split a transformation into passes mediated by their state.

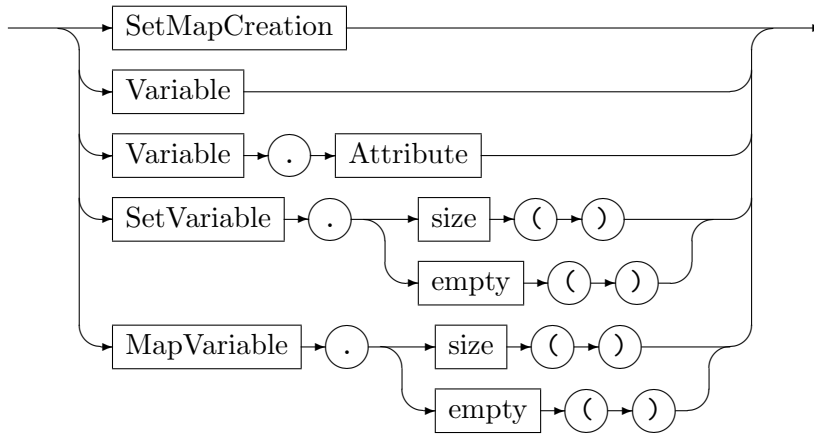
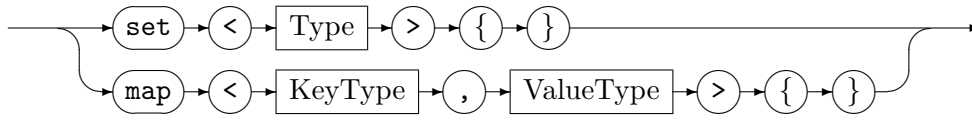
NOTE (32)

The storage sets are more efficient in case the count of elements of interest is a good deal smaller than the number 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 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).

8.2.1 Storage and visited flag handling in the sequences

Storages are variables of set or map type (cf. 5.1) storing nodes or edges. Primarily used in the sequences, from where they are handed in to the rules via parameters; but additionally set/map member attributes may be used as storages, esp. for doing data flow analyses, cf. 8.4. 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 set. A difference to the sets and maps in the rewrite rules is that they only offer imperative addition and removal instead of union, intersection, difference and construction. The splitting of transformations into passes mediated by set/map valued global variables allows for subgraph copying without model pollution, cf. 8.4; please have a look at 8.3 and 8.4 regarding a discussion on when to use which transformation combinators and for storage examples.

VariableHandling*SetMapTypeDecl*

RHS*SetMapCreation*

A set/map must be created and assigned to a variable before it can be used.

EXAMPLE (51)

```

1 x=set<NodeTypeA>{}
2 y:map<Node,Edge> = map<Node,Edge>{}

```

The first line declares or references a variable **x** (without static type) and assigns the newly created, empty set of type **set<NodeTypeA>** to it as value. The second line declares a variable **y** of type **map<Node,Edge>** and assigns the newly created, empty map of the same type to it as value.

There are several operations on set variables available in method call notation, these are:

Set addition:

s.add(v) adds the value **v** to the set **s**, succeeds always.

Set removal:

s.rem(v) removes the value **v** from the set **s**, succeeds always.

Set clearing:

s.clear() removes all values from the set **s**, succeeds always.

Very similar operations are available on map variables:

Map addition:

m.add(k,v) adds the pair **(k,v)** to the map **m**, succeeds always.

Map removal:

m.rem(k) removes the pair **(k,unknown)** from the map **m**, succeeds always.

Map clearing:

m.clear() removes all key-value-pairs from the map **m**, succeeds always.

There are further operations which are only available in variable assignments:

Size assignment:

`v=sm.size()` writes the number of entries in the set or map `sm` to the variable `v`, succeeds always.

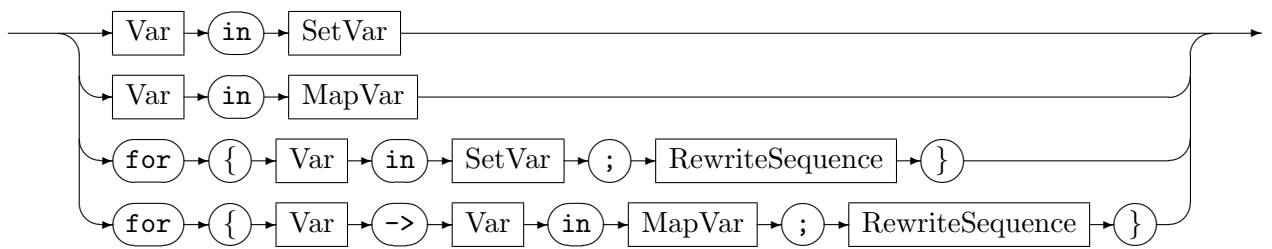
Emptiness assignment:

`v=sm.empty()` writes to the variable `v` whether the set or map `sm` is empty, succeeds always.

Map lookup assignemt:

`v=m[e1]` assigns the result of the map lookup to the variable `v`, succeeds iff `e1` is contained in `m`, fails otherwise, not touching the variable `v`.

RewriteFactor



Handling of the storages is completed by the rewrite factors for membership query and storage iteration. The binary operator `e1 in sm` checks for set/map membership; it returns true if `e1` is contained in the set or the domain of the map, otherwise false. The `for` command iterates over all elements in the set or all key-value pairs in the map and executes for each element / key-value pair the nested graph rewrite sequence; it completes successfully iff all sequences were executed successfully (an empty set/map causes immediate successful completion).

EXAMPLE (52)

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 iteratedly 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.

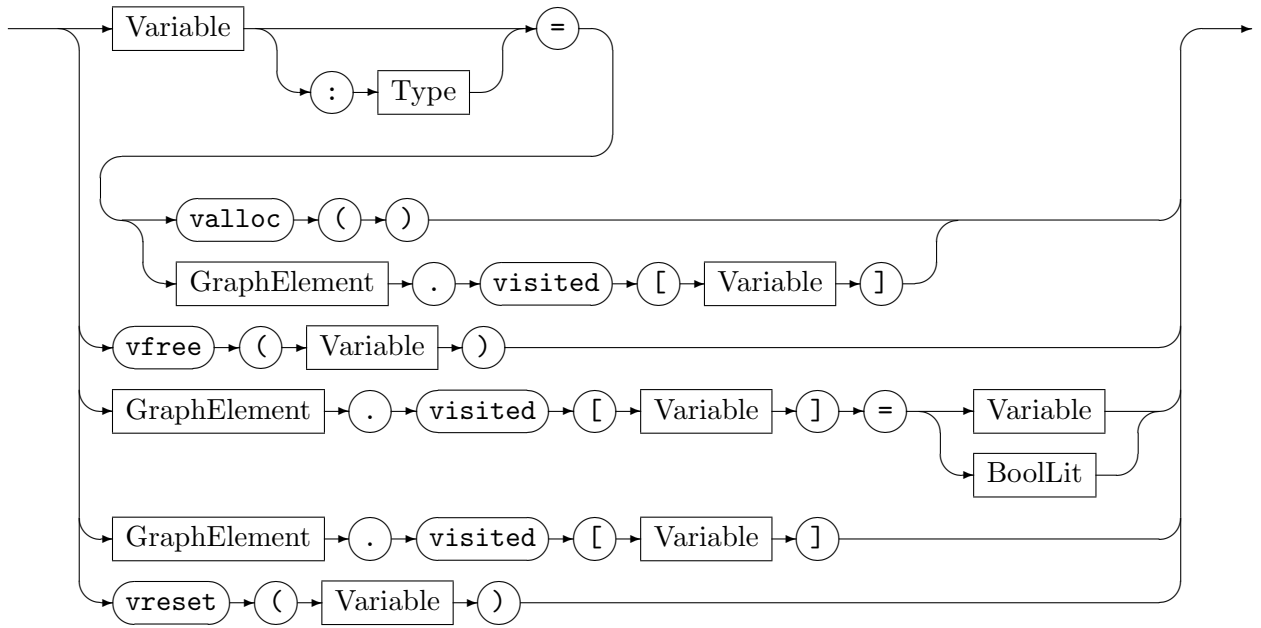
```
x=set<Node>{} ;> ( (v)=t() && x.add(v) )+ && for{v in x; r(v)} <; x.clear()
```

Handling in the storage to the rule, and using the set `add` method as introduced down below in 8.1 within the rule to fill the storage, allows to shorten the sequence to:

```
x=set<Node>{} ;> ( t(x) )+ && for{v in x; r(v)} <; x.clear()
```

NOTE (33)

The set/map over which the `for` loop iterates must stay untouched during iteration.

VisitedFlagsManagement

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. Due to this 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). The operations always return true as sequence results (with exception of the operation reading the flag, it fails iff the visited flag is not set for the graph element); if you try to access a not previously allocated visited flag, an exception is thrown. The operations managing the visited flags 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 within the rules by the **visited**-expression and the **visited**-assignment, as well as by the **visited** flag reading and writing rewrite factors. 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 stored in the variable must be of integer type, stemming from a previous allocation.

Flag writing:

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**; visited flags are normally written by rules of the rule language.

Flag reading:

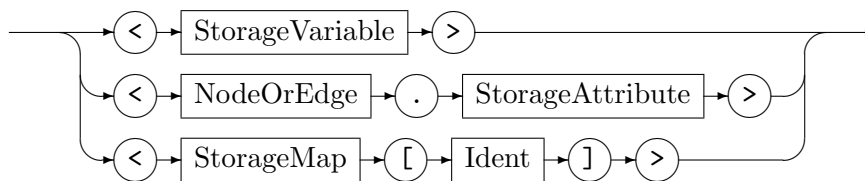
By **e.visited[f]** – returns the visited status of the flag given by the flag id variable **f** of the graph element **e**; visited flags are normally read by tests and rules of the rule language.

Flag resetting:

By **vreset** – resets the visitor flag given by the flag id variable in all graph elements.

8.2.2 Storage and visited flag access in the rules

Storages can be used in the rule control language as introduced above 8.2.1, they can get filled or emptied in the rules as defined here 4.4.3, a discussion about their usage and examples are given here 8.3 and here 8.4. 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 angles. 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 angles (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.

StorageAccess**EXAMPLE (53)**

Queries the graph for the neighbouring cities to the cities contained in the storageset.

```

1 test neighbour(ref startCities:set<City>) : City
2 {
3   :City<startCities> -:Street-> n:City;
4   return(n);
5 }

```

EXAMPLE (54)

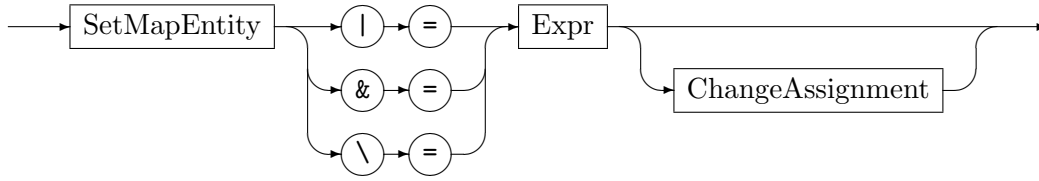
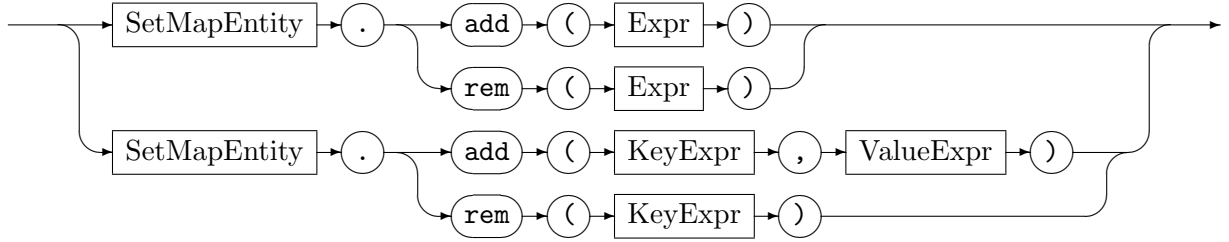
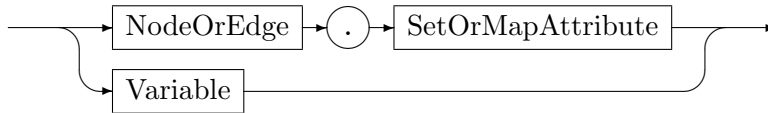
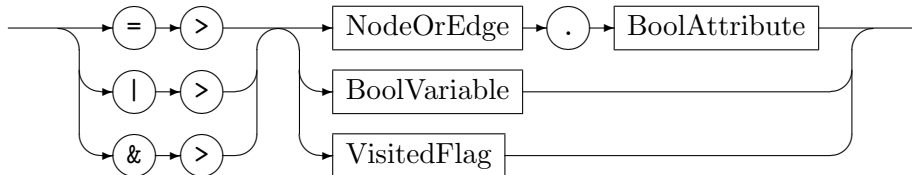
Queries for the neighbour of the neighbour of a city matched. With the first neighbouring relation queried from the storagemap assumed to contain the neighbouring relation of some cities of interest, and the second neighbouring relation queried from the graph.

```

1 test neighbourneighbour(ref neighbours:map<City, City>) : City
2 {
3   someCity:City;
4   nc:City<neighbours[someCity]> -:Street-> nnc:City;
5   return(nnc);
6 }

```

These were the storage queries available in the pattern part; additionally you can query the storage sets and maps in the **if** attribute evaluation clause, with the set and map expressions as given in the Types and Expressions chapter 5. Furthermore you can add and remove elements from the storages in the **eval** clause of the rewrite part.

CompoundAssignment*SetMapStateChange**SetMapEntity**ChangeAssignment*

The by-ref set/map parameters or set/map attributes can be operated upon by the set/map state change methods, which allow to only partially change the set/map by adding and removing elements resp. pairs of elements; they are especially useful for sets or maps containing nodes or edges.

The same holds for the compound assignments, which 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 set/map valued entities it is offered due to performance reasons. The compound assignment statements are set/map union $|=$, intersection $\&=$ and difference $\backslash=$ assignment.

The compound assignments 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 8.4.

EXAMPLE (55)

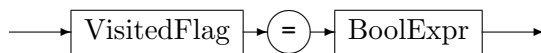
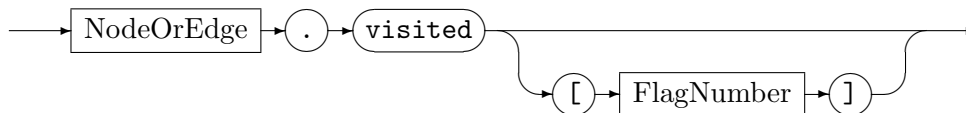
The set/map state change methods `add` and `rem` allow to add graph elements to storages or remove graph elements from storages, i.e. sets or maps holding nodes and edges used for rewrite in the calling sequence (cf. 8.2.1). This way you can write transformations consisting of several passes with one 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 }

```

The visited flag queries available in the `if` attribute evaluation clause of the pattern part are given in the Types and Expressions chapter 5. Additionally you can set them in the `eval` clause of the rewrite part.

VisitedAssignment*VisitedFlag*

The `visited` flag assignment sets the `boolean` status of the visited flag of the given number for the given graph element. If no flag number is given, the default number for the first visited flag of 0 is used. Make sure to allocate 8.2.1/11.3 visited flags before you try to use them (and deallocate them afterwards, as they are a sparse resource stored in some excess bits of the graph elements, or in some dictionary if the needed number of flags exceeds the number of available bits per graph element.)

8.3 Merge, Split, and Node Replacement Grammars

In this and the following section we'll have a look at several common graph rewriting tasks and how to accomplish them in GRGEN.NET. 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.

There are two-and-a-half means available in the GRGEN.NET-rule language to build combined, complex rules:

nested and subpatterns

which 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.

embedded graph rewrite sequences

for deferred rule execution 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); or for splitting work into several parts, reusing already available functionality.

storages and storagemaps

to store elements collected in one run and reuse them as input in another run, i.e. using multi-value variables to break up the transformation into passes. (This is the half point, as they are not a means to build a complex rule, but a means to communicate between rules building complex transformations.)

In the following we'll employ them to merge and split nodes, emulate node replacement grammars, copy substructures and compute flow equations over the graph structure with sets in the nodes.

NOTE (34)

You should use the means to build complex transformations in the order given, first try to solve the problem with nested and subpatterns, if they don't get the job done (often because of the isomorphy constraint locking of already matched elements) or feel awkward then embedded graph rewrite sequences, and if these are still inadequate then storages and storagemaps; i.e. from declarative-local to imperative-global. This helps in keeping the code readable and easily adaptable. And don't forget the last resort if you must solve a task so complex that rules, control and storages are not sufficient: adding helper nodes, edges, or attributes to the graph model and the graph itself (with the visited flags being a light version of this tactic). Note: storages and storagemaps might be useful when you are experiencing performance problems, replacing a search in the graph by a lookup in a dictionary. But beware of elements already deleted from the graph still hanging out in your storage because you forgot to remove them.

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, 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.

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 (56)

```

1 rule T1
2 {
3   n:BB -:cf-> n;
4
5   replace {
6     n; // delete relexive edges
7   }
8 }
9
10 rule T3
11 {
12   pred:BB -first:cf-> succ:BB;
13   pred   -other:cf-> succ;
14
15   modify { // kill multiedges
16     delete(other);
17   }
18 }
```

The interesting part is T2, this is the first version using an iterated statement:

EXAMPLE (57)

```

1 rule T2
2 {
3   pred:BB -e:cf-> succ:BB;
4   negative {
5     -e->;
6     -:cf-> succ; // if succ has only one predecessor
7   }
8   iterated {
9     succ -ee:cf-> n:BB;
10
11     modify { // then merge succ into this predecessor
12       pred -:copy<ee>-> n; // copying the succ edges to pred
13     }
14   }
15
16   modify { // then merge succ into this predecessor
17     delete(succ);
18   }
19 }
```

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 (58)

```

1 rule T2exec
2 {
3   pred:BB -e:cf-> succ:BB;
4   negative {
5     -e->;
6     -:cf-> succ; // if succ has only one predecessor
7   }
8
9   modify { // then merge succ into this predecessor
10    exec([copyToPred(pred, succ)] ;> delSucc(succ));
11  }
12 }
13
14 rule copyToPred(pred:BB, succ:BB)
15 {
16   succ -e:cf-> n:BB;
17
18   modify {
19     pred -:copy<e>-> n;
20   }
21 }
22
23 rule delSucc(succ:BB)
24 {
25   modify {
26     delete(succ);
27   }
28 }

```

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 (59)

We do the splitting by keeping the indeterministically chosen first cf edge, splitting off only further cf edges, replicating their common target.

```

1 rule split(succ:BB)
2 {
3   pred:BB -first:cf-> succ;
4   multiple {
5     otherpred:BB -other:cf-> succ;
6
7     modify {
8       otherpred -newe:cf-> newsucc:copy<succ>;
9       delete(other);
10      exec(copyCfSuccFromTo(succ, newsucc));
11    }
12  }
13
14  modify {
15  }
16 }
17
18 rule copyCfSuccFromTo(pred:BB, newpred:BB)
19 {
20   iterated {
21     pred -e:cf-> succ:BB;
22
23     modify {
24       newpred -:copy<e>-> succ;
25     }
26   }
27
28   modify {
29   }
30 }

```

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 `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.

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 (60)

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.

```

1 rule example
2 {
3   n:NT;
4
5   iterated {
6     n -:E1-> m:T;
7
8     modify {
9       x <-:E2- m;
10    }
11  }
12
13  iterated {
14    n <-e2:E2- m:T;
15
16    modify {
17      y -:E2-> m;
18      z <-:copy<e2>- m;
19    }
20  }
21
22  modify {
23    delete(n);
24    x:T -- y:T -- z:T -- x;
25  }
26 }
```

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 (61)

```

1 rule cliqueStep
2 {
3   nt:NT;
4
5   iterated {
6     nt -- neighbour:T;
7
8     modify {
9       t -- neighbour;
10      nnt -- neighbour;
11    }
12  }
13
14  modify {
15    delete(nt);
16    t:T -- nnt:NT;
17  }
18 }
19
20 rule cliqueTerminal
21 {
22   nt:NT;
23
24   modify {
25     :T<nt>;
26   }
27 }

```

The examples can be found in the `tests/nodeReplacementGrammar` directory.

8.4 Subgraph copying and Reachability via Flow Equations

Copy structures

Structures are copied in two passes, the first copying and collecting all nodes of interest, the second 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, 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 simple 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.

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.

EXAMPLE (62)

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.

```

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    exec( [CopyOutgoingEdge(root, oldToNew)] ); // deferred copy edges
16  }
17 }
18
19 rule CopyOutgoingEdge(n:Node, ref oldToNew:map<Node, Node>)
20 {
21   n -e:Edge-> m:Node;
22   hom(n,m); // reflexive edges
23   nn:Node<oldToNew[n]>; nm:Node<oldToNew[m]>;
24   hom(nn,nm); // reflexive edges
25
26   modify {
27     nn -ee:copy<e>-> nm;
28   }
29 }

```

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.

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://www2.imm.dtu.dk/~riis/PPA/slides2.pdf> for some reading on this topic; especially as this is a general method to compute non-local informations 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 one. 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 (63)

```

1 node class N
2 {
3   reachable:set<N>;
4 }
```

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 (64)

```

1 rule directReachability
2 {
3   hom(n,m);
4   n:N --> m:N;
5
6   modify {
7     eval { n.reachable.add(m); }
8   }
9 }
```

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 (65)

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.

```

1 rule propagateBackwards(n:N, s:N, var changed:boolean) : (boolean)
2 {
3   modify {
4     eval { n.reachable |= s.reachable |> changed; }
5     return(changed);
6   }
7 }

```

The example can be found in the `tests/dataFlowAnalysis` directory, just add `debug` before the `xgrs` in `dataFlowAnalysisForReachability.grs` and watch it run. A sample situation showing a propagation step is given in 8.1. The subgraph at the top-left is already handled as you can see by the reachable set displayed in each node.

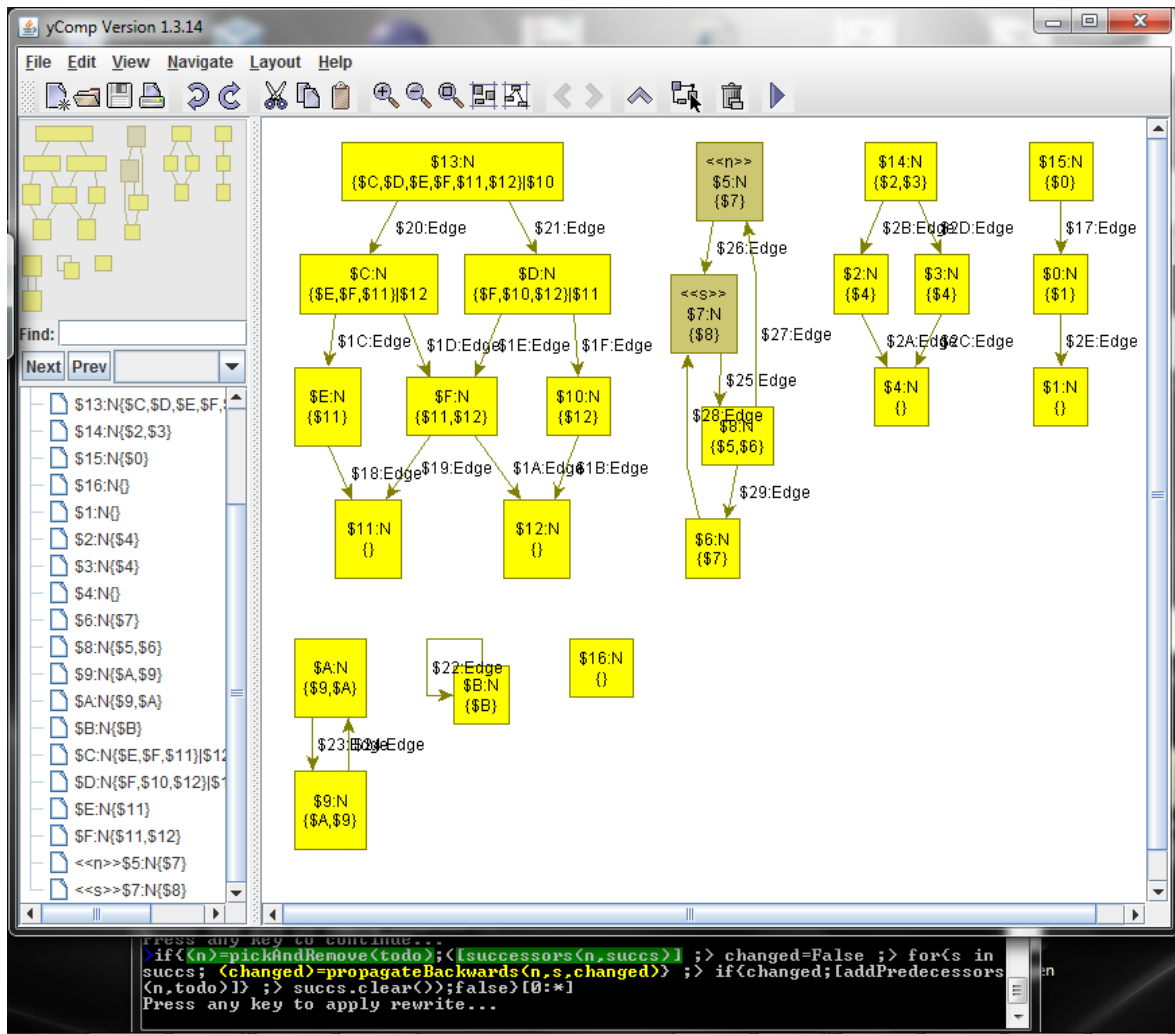


Figure 8.1: Situation from dataflow analysis

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 (66)

```
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 (67)

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 (68)

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 `xgrs` in `dataFlowAnalysisForReachabilityWorklist.grs` and watch it run. A sample situation showing a worklist building step is given in 8.2. The subgraph at the top-left is already handled as you can see by the reachable set displayed in each node.

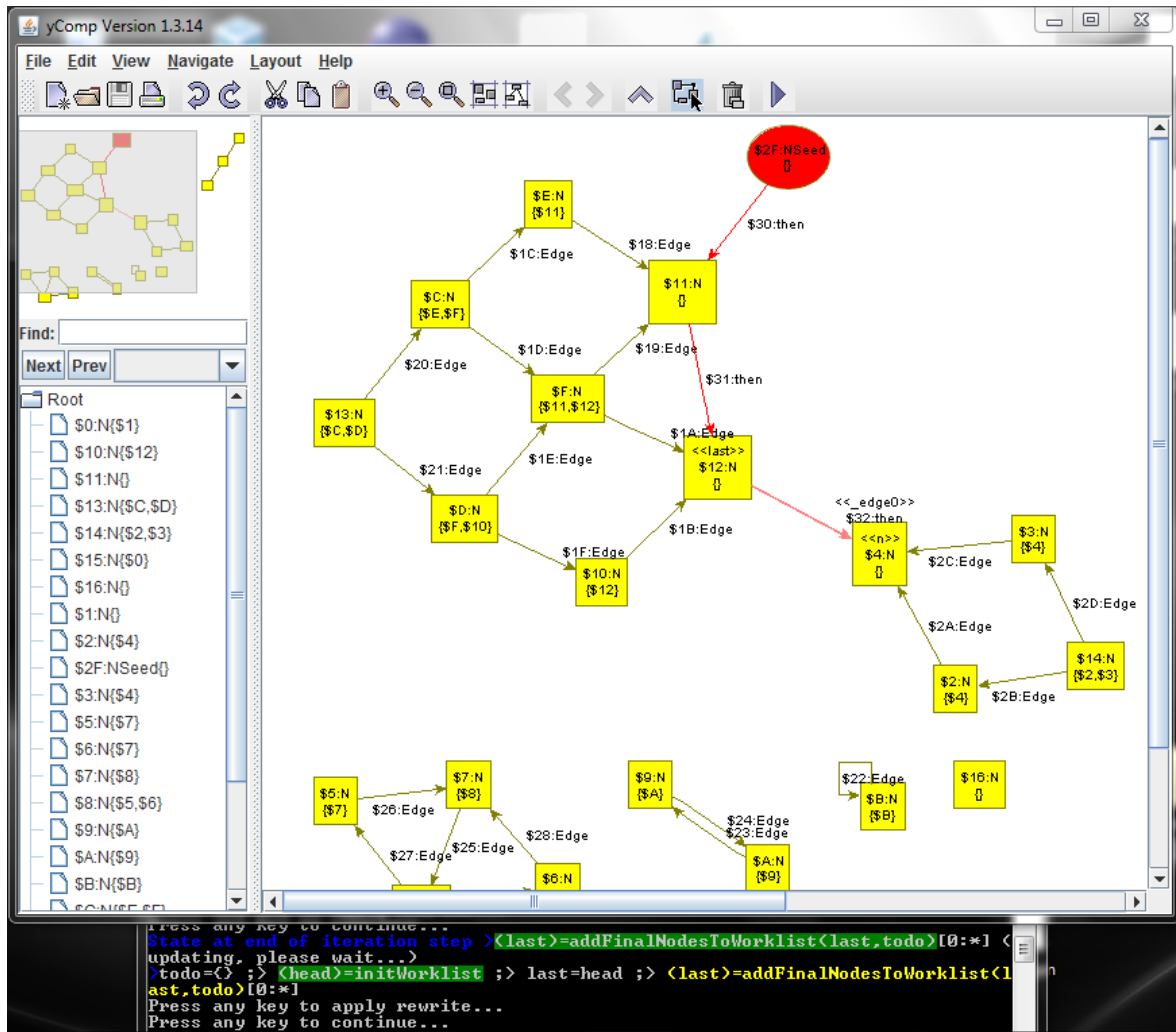


Figure 8.2: Situation from worklist building

CHAPTER 9

GRSHELL LANGUAGE

GRSHELL is a shell application built on top of LIBGR. It belongs to GRGEN.NET's standard equipment. GRSHELL is capable of creating, manipulating, and dumping graphs as well as performing and debugging graph rewriting. The GRSHELL provides a line oriented scripting language. GRSHELL scripts are structured by simple statements separated by line breaks.

9.1 Building Blocks

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. The following items are required for representing text, numbers, and rule parameters.

Text

May be one of the following:

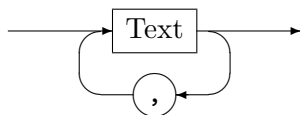
- A non-empty character sequence consisting of letters, digits, and underscores. The first character must not be a digit.
- Arbitrary text enclosed by double quotes (`"`).
- Arbitrary text enclosed by single quotes (`'`).

Due to the chosen parser generator shell keywords are not allowed for type names, attribute values and other entities (even if they are legal in the rule language). 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.

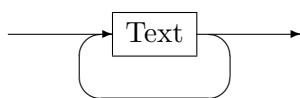
Number

Is an `int` or `float` constant in decimal notation (see also Section 5.1).

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 a graph element or a value.

NodeType, EdgeType

Identifier of a node type resp. edge type defined in the model of the current graph.

AttributeName

Identifier of an attribute.

Graph

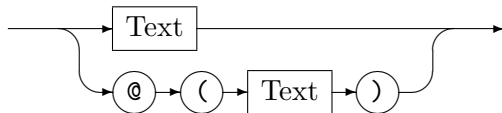
Identifies a graph by its name.

Action

Identifies a rule by its name.

Color

One of the following color identifiers: Black, Blue, Green, Cyan, Red, Purple, Brown, Grey, LightGrey, LightBlue, LightGreen, LightCyan, LightRed, LightPurple, Yellow, White, DarkBlue, DarkRed, DarkGreen, DarkYellow, DarkMagenta, DarkCyan, Gold, Lilac, Turquoise, Aquamarine, Khaki, Pink, Orange, Orchid. These are the same color identifiers as in VCG/YCOMP files (for a VCG definition see [San95]).

GraphElement

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 9.2.6). The specializations *Node* and *Edge* of *GraphElement* require the corresponding graph element to be a node or an edge respectively.

EXAMPLE (69)

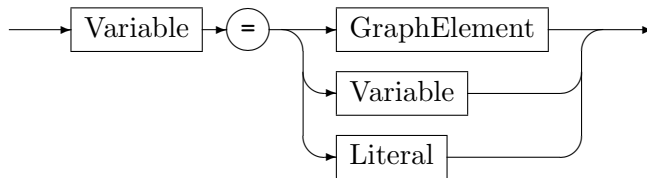
We insert a node, anonymously and with a constructor (see also Section 9.2.6):

```

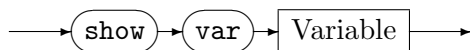
1 > new graph "../lib/lgsp-TuringModel.dll" G
2 New graph "G" of model "Turing" created.
3
4 # insert an anonymous node...
5 # it will get a persistent pseudo name
6 > new :State
7 New node "$0" of type "State" has been created.
8 > delete node @("$0")
9
10 # and now with constructor
11 > new v:State($=start)
12 new node "start" of type "State" has been created.
13 # Now we have a node named "start" and a variable v assigned to "start"
```

NOTE (35)

Persistent names will be saved (`save graph...`, see Section 9.2.4) and exported, and, if you visualize a graph (`dump graph...`, see Section 9.2.4), graph elements will be labeled with their persistent names. Persistent names have to be unique for a graph (the graph they belong to).



Assigns the variable or persistent name *GraphElement* or literal to *Variable*. 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 variables known to GRShell are the graph global variables (see 7 for the distinction between graph global and sequence local variables).

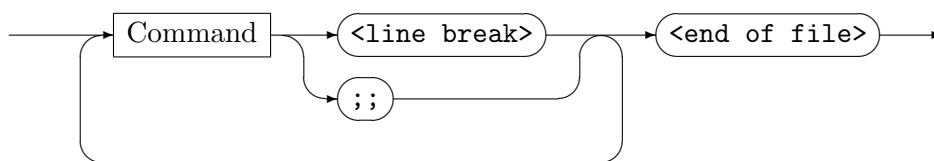


Prints the content of the specified variable.

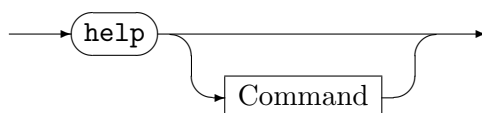
9.2 GRShell Commands

This section describes the GRShell commands. Commands are assembled from basic elements. 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.

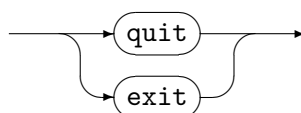
Script



9.2.1 Common Commands



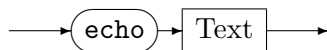
Displays an information message describing all the supported commands. A command `Command` displayed with `...` has further help available, which can be displayed with `help Command`.



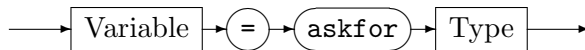
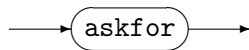
Quits GRShell. If GRShell is opened in debug mode, a currently active graph viewer (such as yCOMP) will be closed as well.



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.



Prints *Text* onto the GRShell command prompt.



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 xgrs Variable=%(Type).`)

EXAMPLE (70)

```
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.

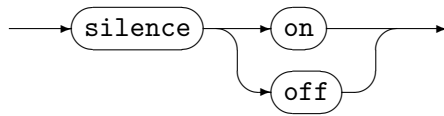


CommandLine is an arbitrary text, the operating system attempts to execute.

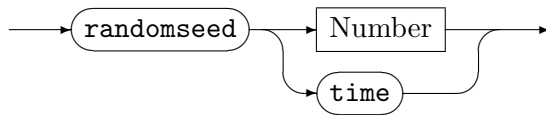
EXAMPLE (71)

On a Linux machine you might execute

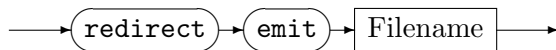
```
1 !sh -c "ls |_grep_stuff"
```



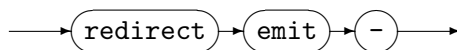
Switches the new node / edge created / deleted messages on(default) or off. Switching them off allows for much faster execution of scripts containing a lot of creation commands.



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.

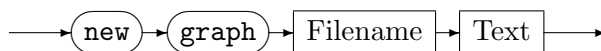


Redirects the output of the emit-statements in the rules from stdout to the given file.

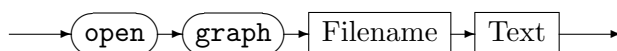


Redirects the output of the emit-statements in the rules to stdout (again).

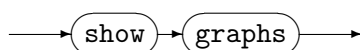
9.2.2 Graph Commands



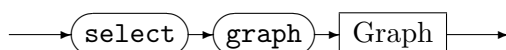
Creates a new graph with the model specified in *Filename*. Its name is set to *Text*. The model file can be either source code (e.g. `turing_machineModel.cs`) or a .NET assembly (e.g. `lgsp-turing_machineModel.dll`). It's also possible to specify a rule set file as *Filename*. In this case the necessary assemblies will be created on the fly.



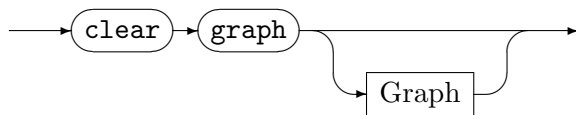
Opens the graph *Text* stored in the backend. However, the *LGSPBackend* doesn't support persistent graphs, and as the *LGSPBackend* is the only backend available at the moment, this command is currently useless. You may achieve persistence by using import/export or save/include instead.



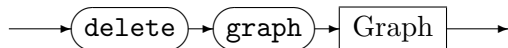
Displays a list of currently available graphs.



Selects the current working graph. This graph acts as *host graph* for graph rewrite sequences (see also Sections 1.5 and 9.2.9). Though you can define multiple graphs, only one graph can be the active “working graph”.



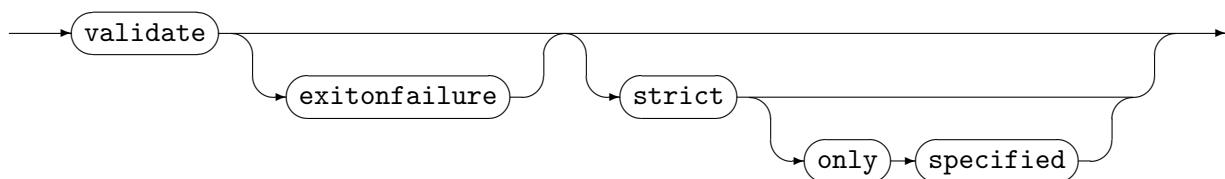
Deletes all graph elements of the current working graph resp. the graph *Graph*.



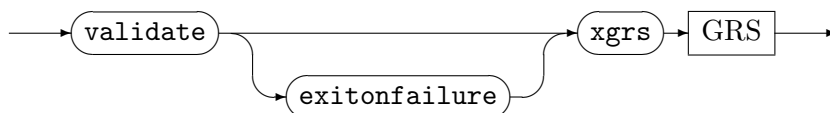
Deletes the graph *Graph* from the backend storage.

9.2.3 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. 3.2.2). 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 a too big 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 xgrs 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 (72)

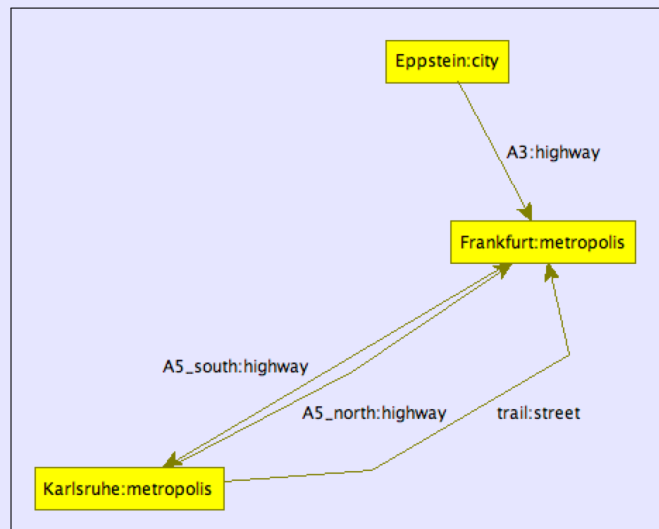
We reuse a simplified version of the road map model from Chapter 3:

```

1 model Map;
2
3 node class city;
4 node class metropolis;
5
6 edge class street;
7 edge class highway
8     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:



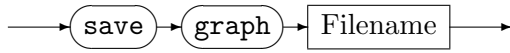
This graph is valid but not strict valid.

```

1 > validate
2 The graph is valid.
3 > validate strict only specified
4 The graph is NOT valid:
5   CAE: city "Eppstein" -- highway "A3" --> metropolis "Frankfurt" not specified
6 > validate strict
7 The graph is NOT valid:
8   CAE: city "Eppstein" -- highway "A3" --> metropolis "Frankfurt" not specified
9   CAE: metropolis "Karlsruhe" -- street "trail" --> metropolis "Frankfurt" not specified
10 >

```

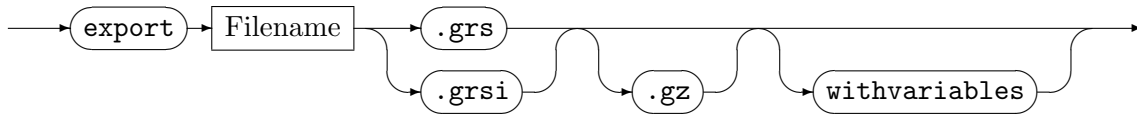
9.2.4 Graph Input and Output Commands



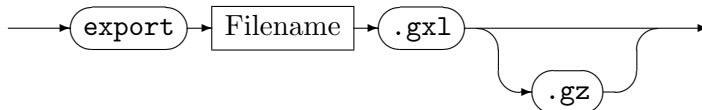
Dumps the current graph as GRShell script into *Filename*. The created script includes

- 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

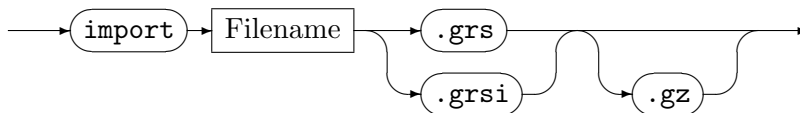
but not necessarily using the same commands you typed in during construction. Such a script can be loaded and executed by the `include` command (see Section 9.2.1).



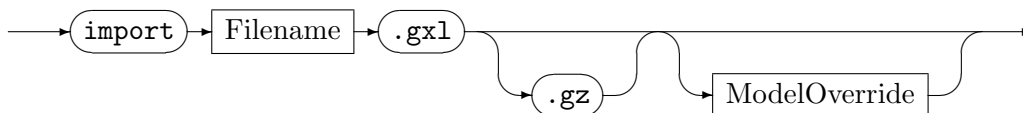
Exports an instance graph in GRS (.grs/.grsi) format, which is a reduced GRShell script (it can get imported and exported on API level 11.3 without using the GRShell). It contains the `new graph` command, followed by `new node` commands, followed by `new edge` commands. If the `.gz` suffix is given the graph is saved zipped. If `withvariables` is specified, the (graph global) variables are exported, too. 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. The `save` command is for saving about a GRShell session including graph global variables and visualization commands, the goal of the `export` command is basic graph rewrite system interoperability. The persistent names are saved in contrast to the following format GXL.



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 bloated. Exporting fails if the graph model contains attributes of `set<S>`-, `map<S,T>`-, or `object`-type. 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` (but an exported graph can be imported by `include`, too)). The referenced graph model must be available as `.gm`-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. The `.gxl-graph` must be compatible to the `.gm-model`. If the `.gz` suffix is given the graph is expected to be zipped.

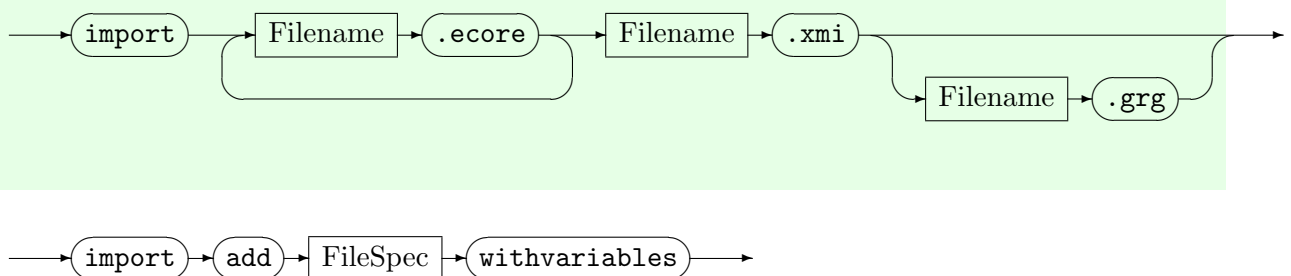
NOTE (36)

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. 9.2.9).
4. Now you are ready to use the actions.

NOTE (37)

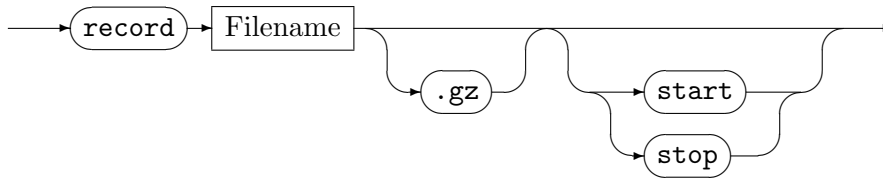
Further formats available for import are `.ecore` plus `.xmi`. These are formats common to the model transformation community which are not directly geared towards graphs, so they can't be imported directly. Instead during the import process an intermediate `.gm` is written which is equivalent to the `.ecore` given – you may inspect it to see how the content gets mapped (the importer maps 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; class names are prefixed by the names of the packages they are contained in to prevent name clashes). 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 give specify a `.grg` containing the rules to apply (using further rule and model files). The importer was added for a GraBaTs challenge and is available as-is – it may or may not work for you, if you need more it's on you to improve it. An export is not available – we coded the export we needed with `emit` statements.



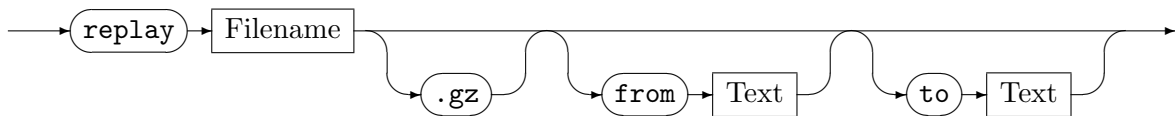
Imports the graph in the specified file and adds it to the current graph (instead of overwriting the old graph with the new graph). The `FileSpec` is of the same format as the file specification

in the other two imports. The **withvariables** argument only yields an effect if the file to import contains variable specifications (the content of old variables of the same name is overwritten).

9.2.5 Graph Change Recording and Replaying



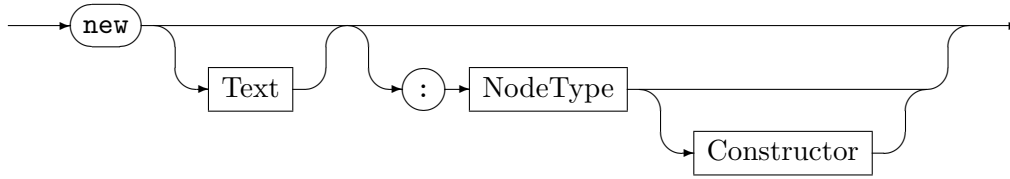
The **record** command starts or stops recording of graph changes to the specified file. If neither **start** nor **stop** are given, recording to the specified file is toggled (i.e. started if no recording to the file is underway or stopped if the file is already recorded to). Recording starts with an export (cf. 9.2.4) 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. 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. 9.2.6) but also comments telling about the rewrites and transaction events which occurred (whose effects were recorded).



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. 7.3) in the sequences, marking relevant states during a transformation process. An example for **record** and **replay** is given in **tests/recordreplay**.

9.2.6 Graph Manipulation Commands

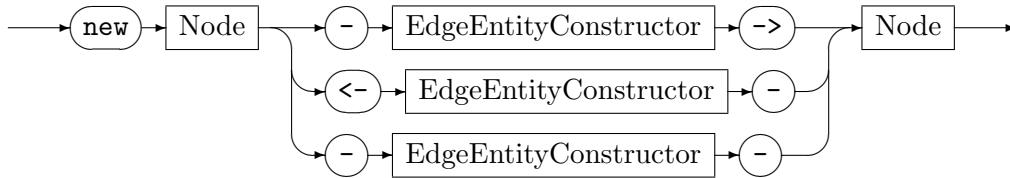
Graph manipulation commands alter existing graphs; they allow to create, retype and delete graph elements and change attributes. These are tasks which are or at least should be carried out by the rules of the rule language in the first place. On shell level they are available and mainly used as elementary instructions in creating an initial graph, in exporting and importing a graph, as well as in change recording and replaying.



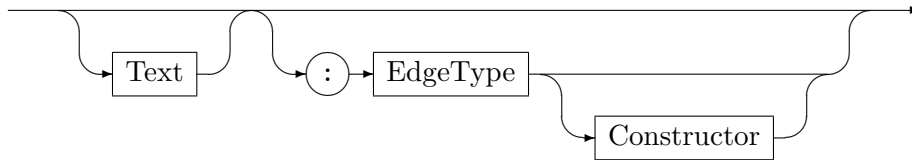
Creates a new node within the current graph. Optionally a variable *Text* is assigned to the new node. If *NodeType* is supplied, the new node will be of type *NodeType* and attributes can be initialized by a constructor. Otherwise the node will be of the base node class type *Node*.

NOTE (38)

The GRShell can reassign variables. This is in contrast to the rule language (Chapter 4), where we use *names*.

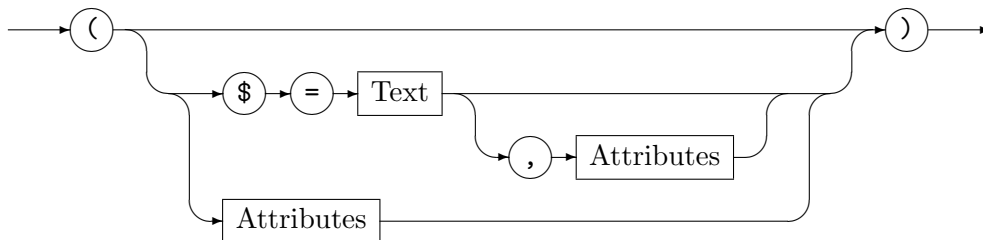


EdgeEntityConstructor

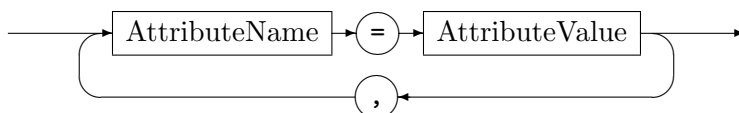


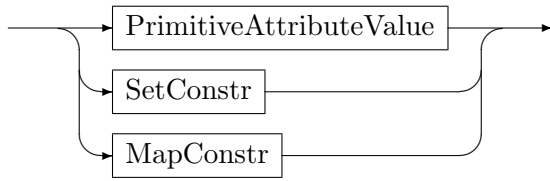
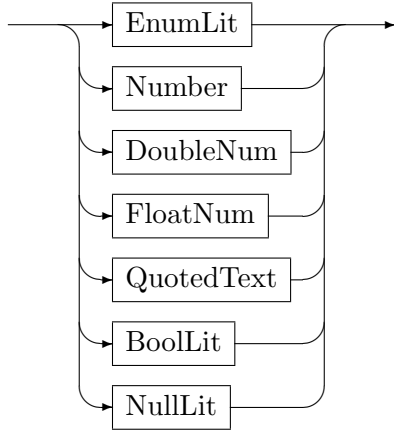
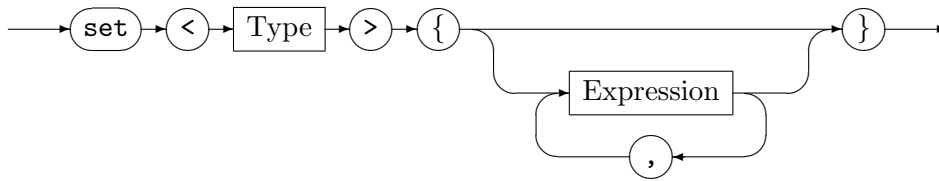
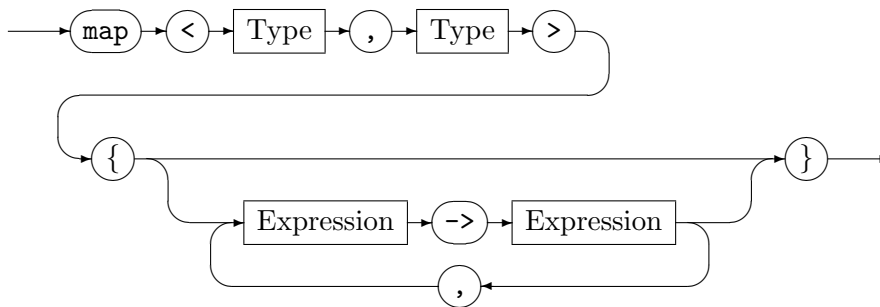
Creates a new edge within the current graph between the specified nodes, directed from the first to the second *Node* in the case of *-->*, directed from the second to the first *Node* in the case of *<--*, or undirected in the case of *--*. Optionally a variable *Text* is assigned to the new edge. If *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



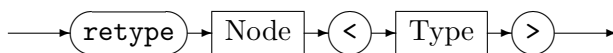
Attributes



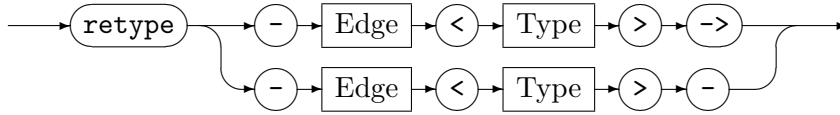
Attribute Value*PrimitiveAttribute Value**SetConstr**MapConstr*

A constructor is used to initialize a new graph element (see **new** ... below). A comma separated list of attribute declarations 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` \leftarrow 0; `boolean` \leftarrow `false`; `float` \leftarrow 0.0f; `double` \leftarrow 0.0; `string` \leftarrow ""; `set`<T> \leftarrow `set`<T>{}; `map`<S,T> \leftarrow `map`<S,T>{}; `enum` \leftarrow unspecified;).

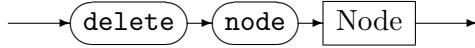
The \$ is a special attribute name: a unique identifier of the new graph element. This identifier is also called *persistent name* (see Example 69). This name can be specified by a constructor only.



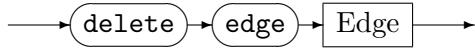
Retypes the node *Node* from its current type to the new type *Type*. Attributes common to initial and final type are kept. Incident edges are kept as well.



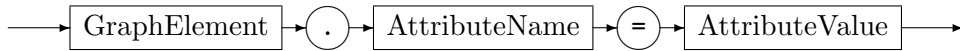
Retypes the edge *Edge* from its current type to the new type *Type*. Attributes common to initial and final type are kept. Incident nodes are kept as well.



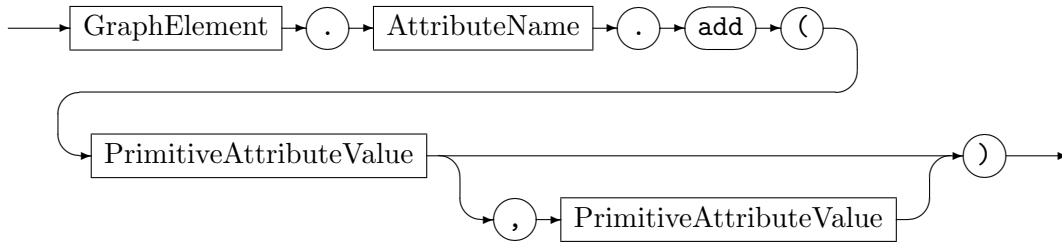
Deletes the node *Node* from the current graph. Incident edges will be deleted as well.



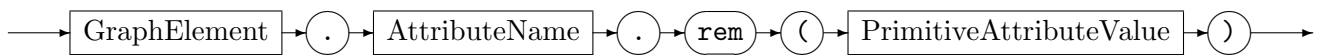
Deletes the edge *Edge* from the current graph.



Set the attribute *AttributeName* of the graph element *GraphElement* to the value *AttributeValue* (for the different possible attribute values see above).

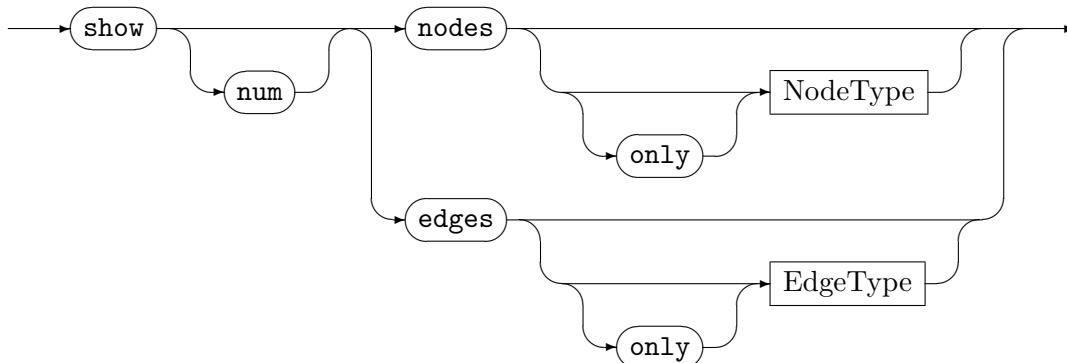


Add the value *PrimitiveAttributeValue* (for the different possible primitive attribute values see above) to the set valued attribute *AttributeName* of the graph element *GraphElement* or add the key-value pair consisting of the two *PrimitiveAttributeValue*s to the map valued attribute *AttributeName* of the graph element *GraphElement* in the two parameter case.

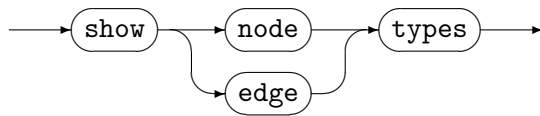


Remove the value (/key) *PrimitiveAttributeValue* from the set (/map) valued attribute *AttributeName* of the graph element *GraphElement*.

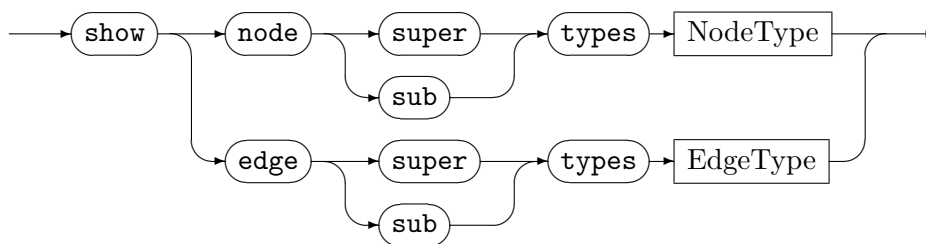
9.2.7 Graph and Model Query Commands



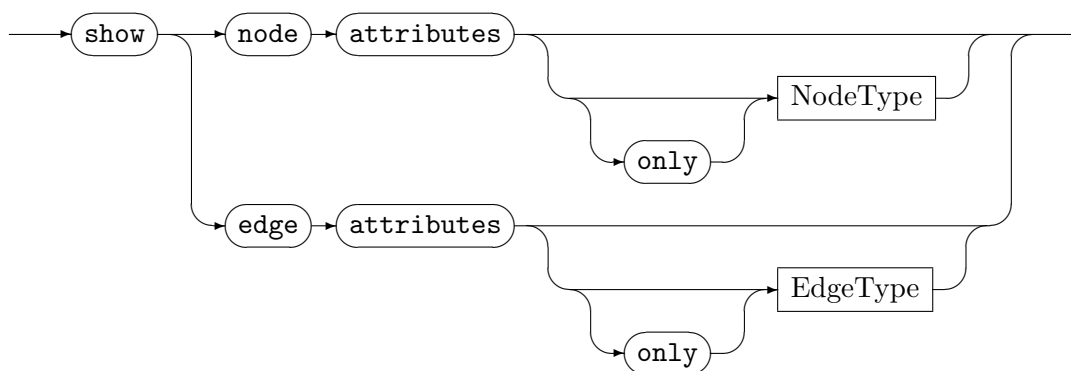
Gets the persistent names and the types of all the nodes/edges of the current graph. If a node type or edge type is supplied, only elements compatible to this type are considered. The **only** keyword excludes subtypes. Nodes/edges without persistent names are shown with a pseudo-name. If the command is specified with **num**, only the number of nodes/edges will be displayed.



Gets the node/edge types of the current graph model.



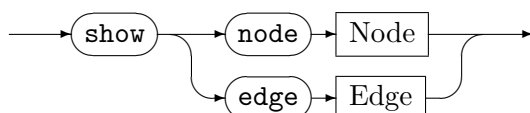
Gets the inherited/descendant types of *NodeType*/*EdgeType*.



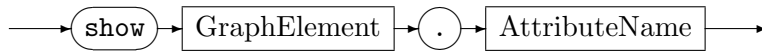
Gets the available node/edge attribute types. If *NodeType*/*EdgeType* is supplied, only attributes defined in *NodeType*/*EdgeType* are displayed. The **only** keyword excludes inherited attributes.

NOTE (39)

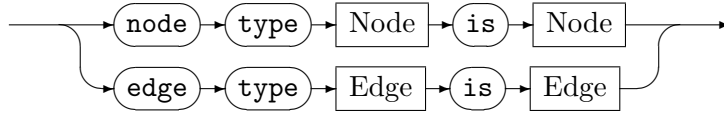
The **show nodes/edges attributes...** command covers types and *inherited* types. This is in contrast to the other **show...** commands where types and *subtypes* are specified or the direction in the type hierarchy is specified explicitly, respectively.



Gets the attribute types and values of a specific graph element.

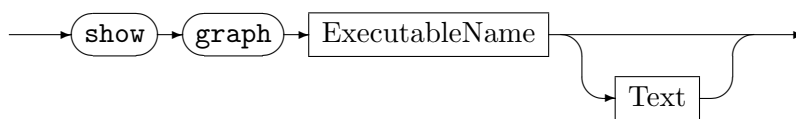


Displays the value of the specified attribute.

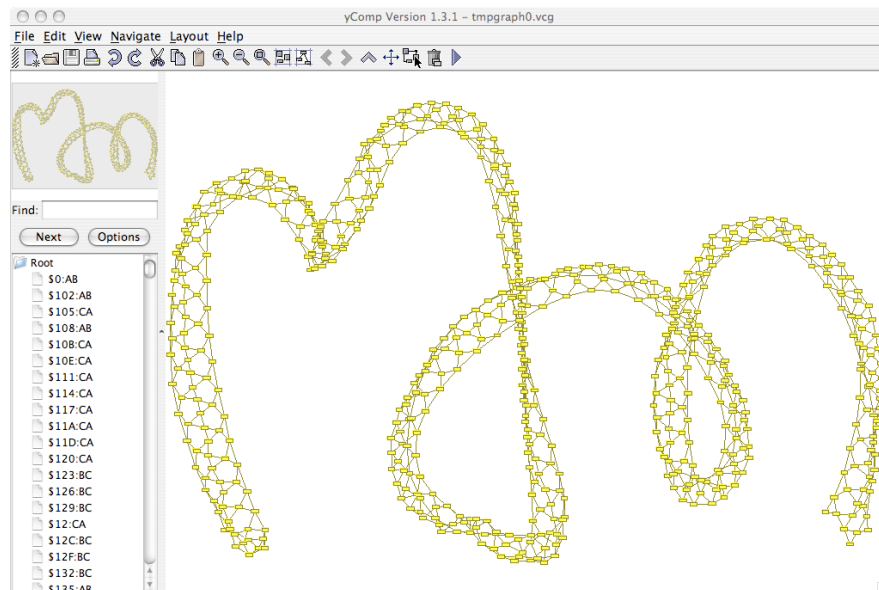


Gets the information whether the first element is type-compatible to the second element.

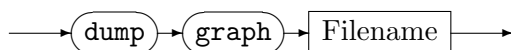
9.2.8 Graph Visualization Commands (Nested Layout)



Dumps the current graph in VCG format into a temporary file. The temporary VCG 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 YCOMP¹ as executable (**show graph ycomp**), this may look like



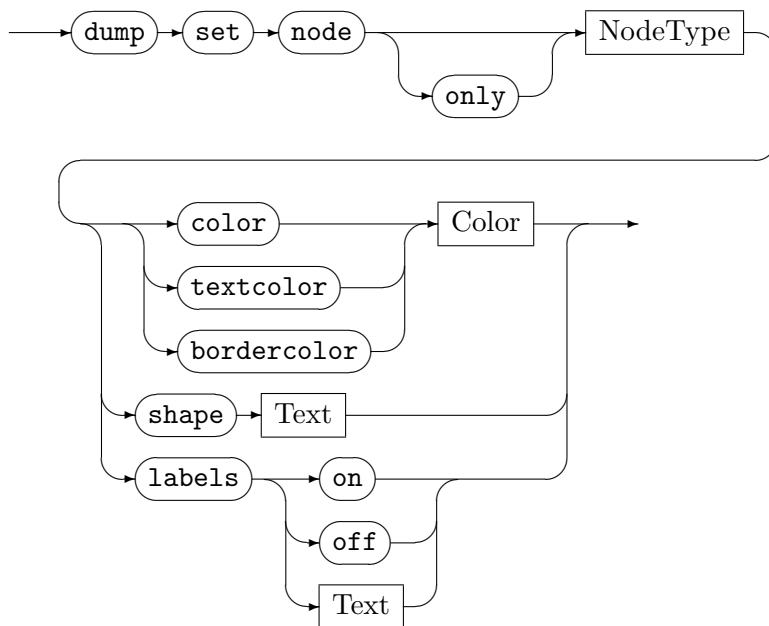
The temporary file will be deleted, when the application *Filename* is terminated if GRShell is still running at this time.



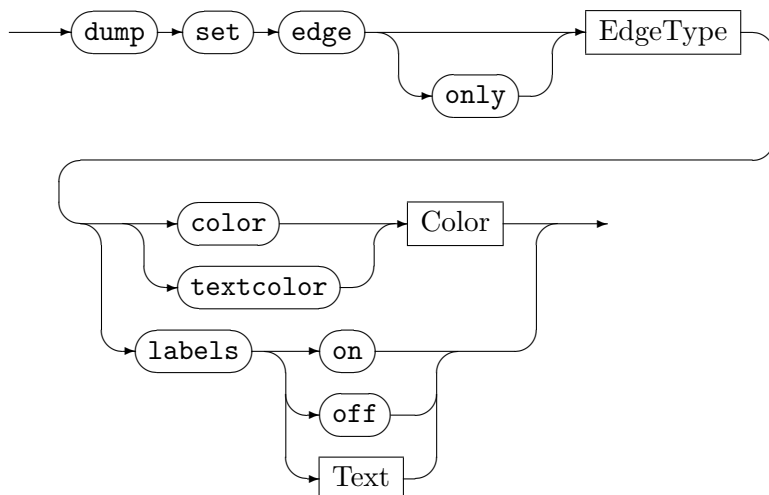
Dumps the current graph in VCG format into the file *Filename*.

The following commands control the style of the VCG output. This affects **dump graph**, **show graph**, and **enable debug**.

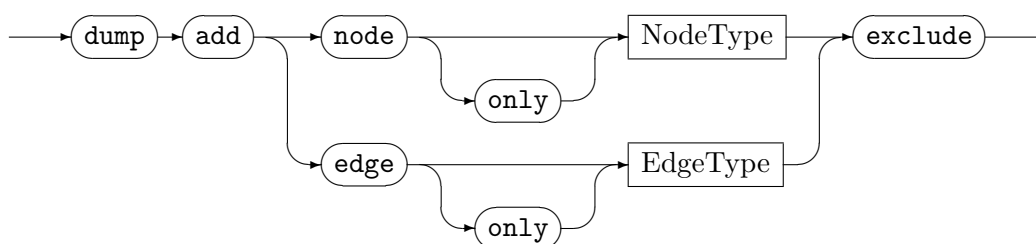
¹See Section 1.7.4.



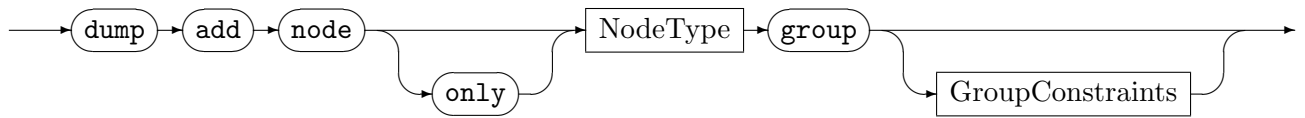
Sets the color, text color, border color, the shape or the label of the nodes of type *NodeType* and all of its subtypes. The keyword **only** excludes the subtypes. The available colors are specified at the begin of this chapter. The following shapes are supported: **box**, **triangle**, **circle**, **ellipse**, **rhomb**, **hexagon**, **trapeze**, **utrapeze**, **lparallelogram**, **rparallelogram**. Those are shape names of YCOMP (for a VCG definition see [San95]). The default labeling is set to **on** with **Name:Type**, it can be overwritten by an specified label string (e.g. the source code line originating the node in a program graph) or switched off.



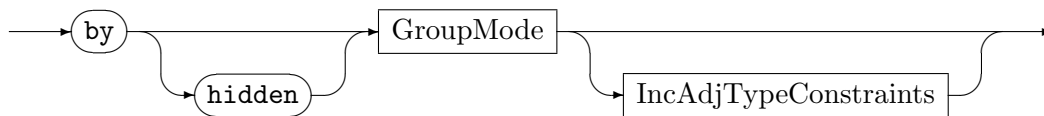
Sets the color, text color or label of the edges of type *EdgeType* and all of its subtypes. The keyword **only** excludes the subtypes. The available colors are specified at the begin of this chapter. The default labeling is set to **on** with **Name:Type**, it can be overwritten by an specified label string or switched off.



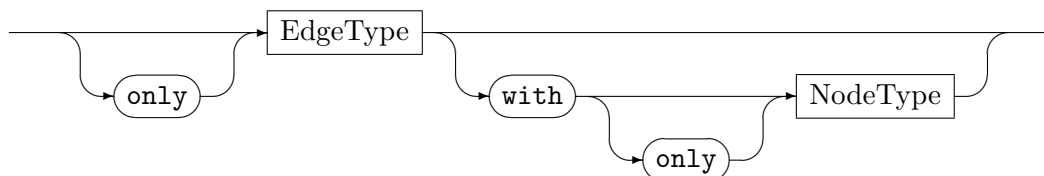
Excludes nodes/edges of type *NodeType*/*EdgeType* and all of its subtypes from output, for a node it also excludes its incident edges. The keyword **only** excludes the subtypes from exclusion, i.e. subtypes of *NodeType*/*EdgeType* are dumped.



GroupConstraints



IncAdjTypeConstraints

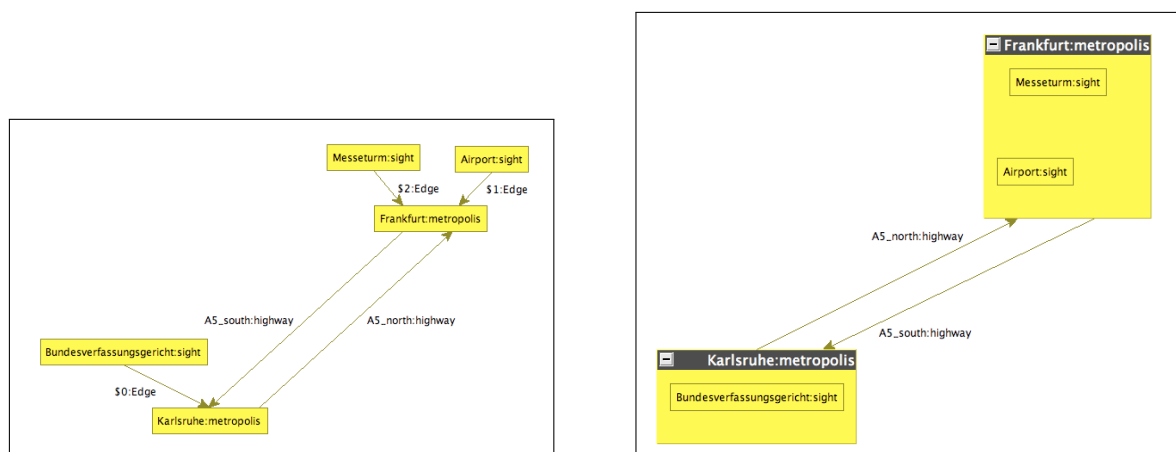


Declares *NodeType* and subtypes of *NodeType* as group node type. All the differently typed nodes that point to a node of type *NodeType* (i.e. there is a directed edge between such nodes) will be grouped and visibly enclosed by the *NodeType*-node (nested graph). **GroupMode** is one of **no**,**incoming**,**outgoing**,**any**; **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.

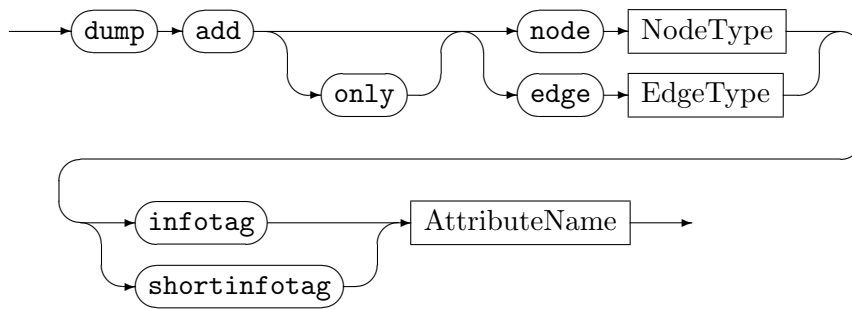
NOTE (40)

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.)

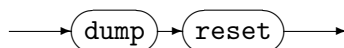
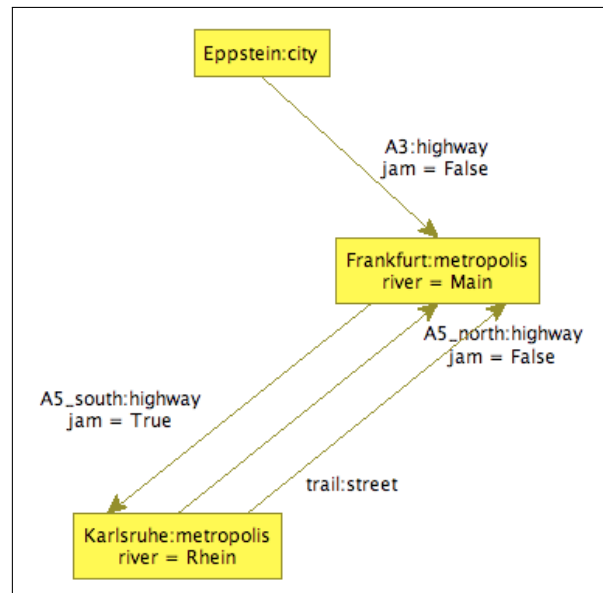
The following example shows *metropolis* ungrouped and grouped:



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:

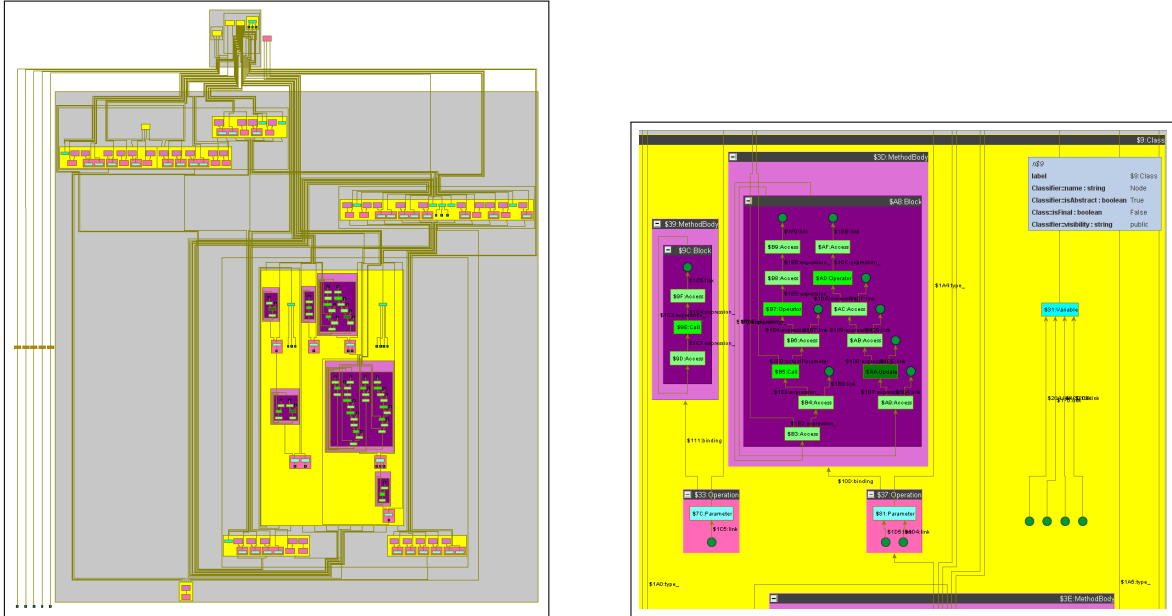


Resets all style options (`dump set...`) and (`dump add...`) to their default values.

NOTE (41)

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 keep 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 the 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/info tags/shortinfo tags 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.

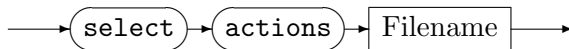
The following example shows several of the layout options employed to considerably increase the readability of a program graph (as given in `examples/JavaProgramGraphs-GraBaTs08`):



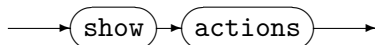
Overview of the initial program graph and some details of the “Node” class

9.2.9 Action Commands (XGRS)

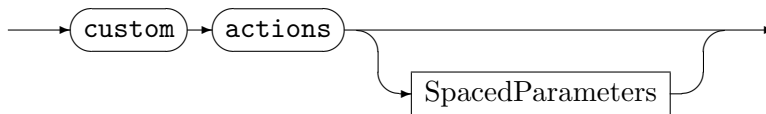
An *action* denotes a graph rewrite rule.



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 9.4.2).

GraphRewriteSequence



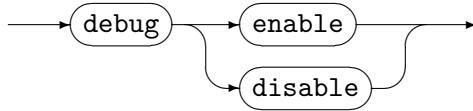
This executes the graph rewrite sequence *SimpleRewriteSequence*. See Chapter 7 for graph rewrite sequences. Additionally to the variable assignment in rule-embedded graph rewrite sequences, you are also able to assign *persistent names* to parameters via **Variable = (Text)**.

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 4.2, *Parameters*.

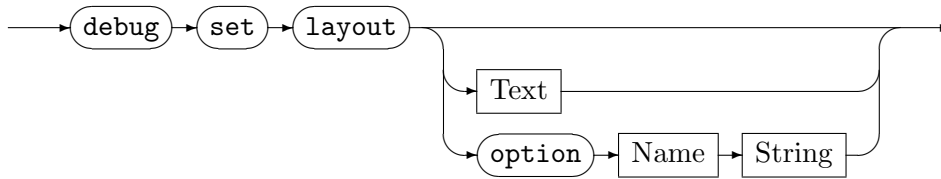
9.3 Graphical Debugger

The GRShell together with YCOMP build GRGEN.NET's graphical debugger.

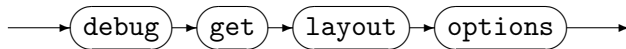
9.3.1 Debugging Related Commands



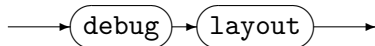
Enables and disables the debug mode. The debug mode shows the current working graph in a YCOMP window. All changes to the working graph are tracked by YCOMP immediately.



Sets the default graph layout algorithm to *Text*. If *Text* is omitted, a list of available layout algorithms is displayed. See Section 1.7.4 on YCOMP layouters. The **option** version allows to specify layout options by name value pairs. The available layout options can be listed by the following command.



Prints a list of the available layout options of the layout algorithm.



Forces re-layout of the graph shown in yComp (same as pressing the play button within yComp).

GraphRewriteSequence



This executes the graph rewrite sequence *SimpleRewriteSequence* in the debugger. Same as **xgrs SimpleRewriteSequence** in the previous section, but allows tracing the rewriting process step-by-step.

9.3.2 Using the Debugger

The debugging process follows of a series of debug situations, which result from a user selection of the underlying execution situations according to interest. During each debugging step 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 commands chosen by the user; 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 which matched during sequence execution are shown on dark green background, the rules which failed during sequence execution are shown on dark red background; at the begin of a new loop iteration the highlighting state of the contained rules is reset. During execution

YCOMP² will display the changes to the graph from every single step. Besides deciding on what is shown from the execution of the current rule, the user determines with the debug commands where to continue the execution (the rule focused next; but again this depends on success/failure of the currently active rule). The debug commands are given in Table 9.1. A run is shown in the following example 73.

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 are only processed when they are reached, but on the other hand are also processed if a user command would skip them. 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 in 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 works 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 works as a choice point if it is appended to the **\$** randomize modifier switching a random decision into a user decision. This holds for the binary operators, the random match selector of all bracketed rules, the random-all-of operators and the one-of-set braces. The concept behind this is: you need some randomization for simulation purposes — then use the randomize modifier **\$**. You want to force a certain decision overriding the random decision to try out another execution path while debugging the simulation flow — 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 xgrs** 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 can't be toggled off.

s (step)	Execute the current rewrite rule (match, and rewrite 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 doing the rewrite showing the resulting graph. In addition, afterwards the execution of subrules from embedded xgrs (exec) is shown step by step.
(step) o (ut)	Continue execution until the end of the current loop. If the execution is not in a loop at this moment, 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 73).
n (ext)	Go to the next rewrite rule which 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.
r (un)	Continue execution (until the end or a breakpoint).
a (bort)	Cancel the execution immediately.

Table 9.1: GRSHELL debug commands

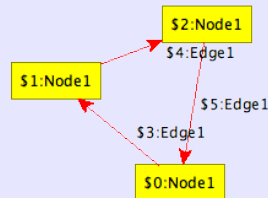
²Make sure, that the path to your **yComp.jar** package is set correctly in the **ycomp** shell script within GRGEN.NET's **/bin** directory.

EXAMPLE (73)

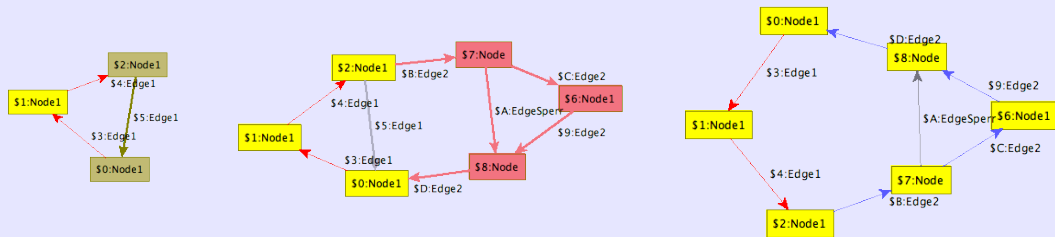
We demonstrate the debug commands with a slightly adjusted script for the Koch snowflake from GRGEN.NET's examples (see also Section 10.1). The graph rewriting sequence is

```
1 debug xgrs (makeFlake1* & (beautify & doNothing)* & makeFlake2* & beautify*)[1]
```

YCOMP will be opened with an initial graph (resulting from `grs init`):



We type `detailed step` to apply `makeFlake1` step by step resulting in the following graphs:



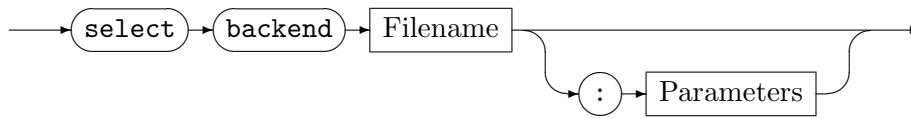
The following table shows the “break points” of further debug commands, entered one after another:

Command	Active rule
s	makeFlake1
o	beautify
s	doNothing
s	beautify
u	beautify
o	makeFlake2
r	—

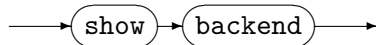
9.4 Backend Commands

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.

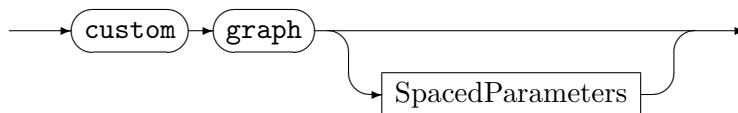
9.4.1 Backend selection and custom commands



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.



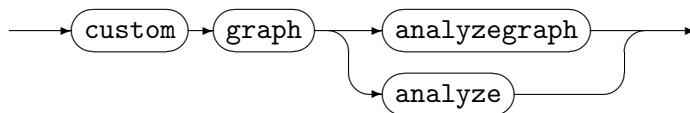
List all the parameters supported by the currently selected backend. The parameters can be provided to the `select backend` command.



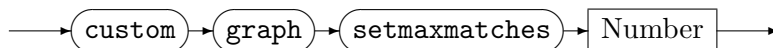
Executes a command specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSP backend see Sections 9.4.2).

9.4.2 LGSPBackend Custom Commands

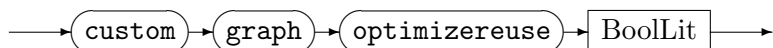
The LGSPBackend supports the following custom commands:



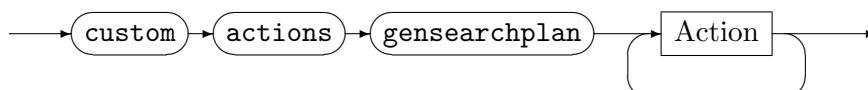
Analyzes the current working graph. The analysis data provides vital information for efficient search plans. Analysis data is available as long as GRShell is running, i.e. when the working graph changes, the analysis data is still available but maybe obsolete.



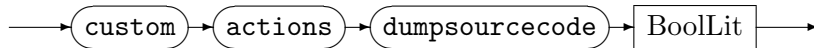
Sets the maximum amount of possible pattern matches to *Number*. This command affects the expression `[Rule]`. If *Number* is less or equal to zero, the constraint is reset.



If set to false it prevents deleted elements from getting reused in a rewrite (i.e. it disables a performance optimization). If set to true, new elements may not be discriminable anymore from already deleted elements using object equality, hash maps, etc.



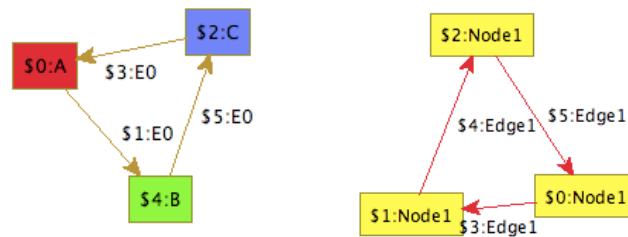
Creates a search plan (and executable code from it) for each rewrite rule *Action* using the data from analyzing the graph (`custom graph analyze`). Otherwise a default search plan is used. Analyzing and search plan/code generation themselves take some time, but they can lead to massively faster pattern matching, thus overall execution times (the less uniform the type distribution and edge wiring between the nodes is, the higher are the improvements to be expected). 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 instead of using multiple commands for each rule to improve the search plan generation performance.



If set to true, C# files will be dumped for the newly generated searchplans (similar to the `-keep` option of the generator).

10.1 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 with the initial host graphs



for the Sierpinski triangle resp. the Koch snowflake. First of all we have to compile the model and rule set files. So execute in GRGEN.NET's `bin` directory

```
GrGen.exe ..\specs\sierpinski.grg
GrGen.exe ..\specs\snowflake.grg
```

or

```
mono GrGen.exe ../specs/sierpinski.grg
mono GrGen.exe ../specs/snowflake.grg
```

respectively. If you are on a Unix-like system you have to adjust the path separators of the GRShell scripts. Just edit the first three lines of `/test/Sierpinski.grs` and `/test/Snowflake.grs`. And as we have the file `Sierpinski.grs` already opened, we can increase the number of iterations to get even more beautiful graphs¹. Just follow the comments. Be careful when increasing the number of iterations of Koch's snowflake—YCOMP's layout algorithm might need some time and attempts to layout it nicely. We execute the Sierpinski script by

```
GrShell.exe ..\test\Sierpinski.grs
```

or

```
mono GrShell.exe ../test/Sierpinski.grs
```

respectively. Because both of the scripts are using the debug mode, we complete execution by typing `r(un)`. See Section 9.2.9 for further information. The resulting graphs should look like Figures 10.1 and 10.2.

¹Be careful: The running time increases exponentially in the number of iterations.

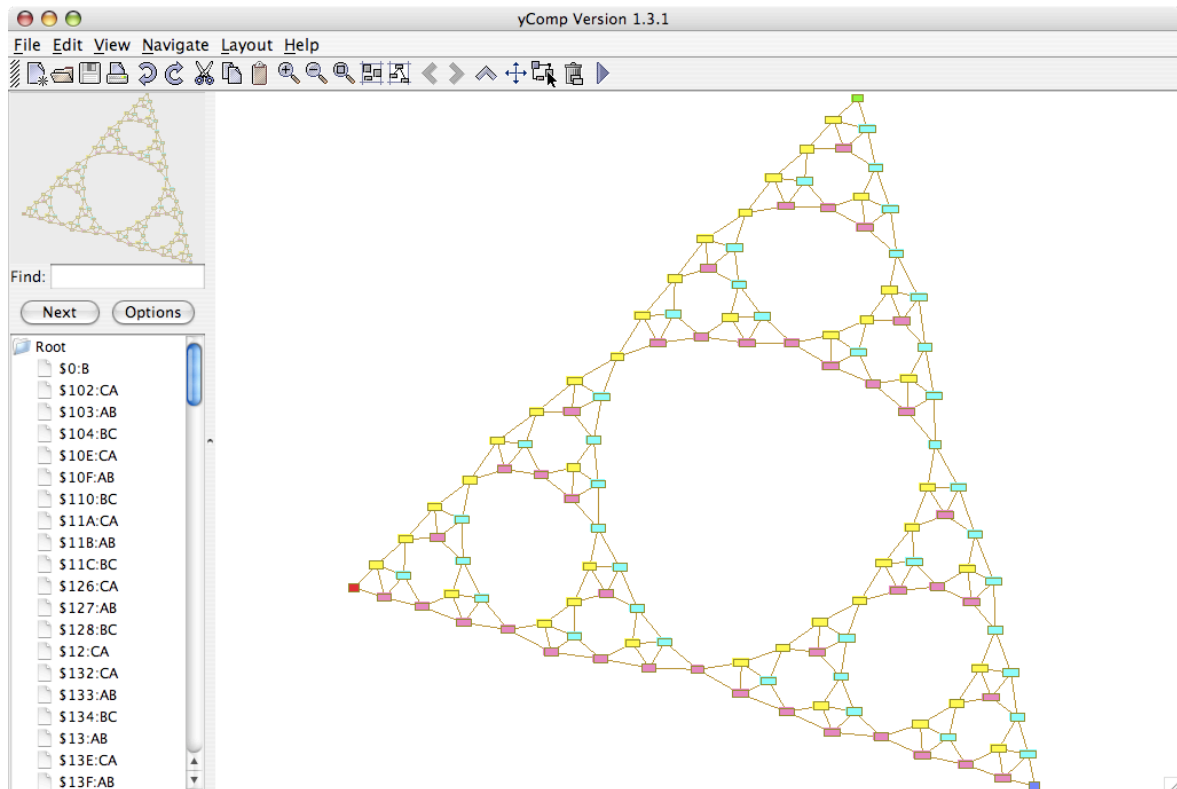


Figure 10.1: Sierpinski triangle

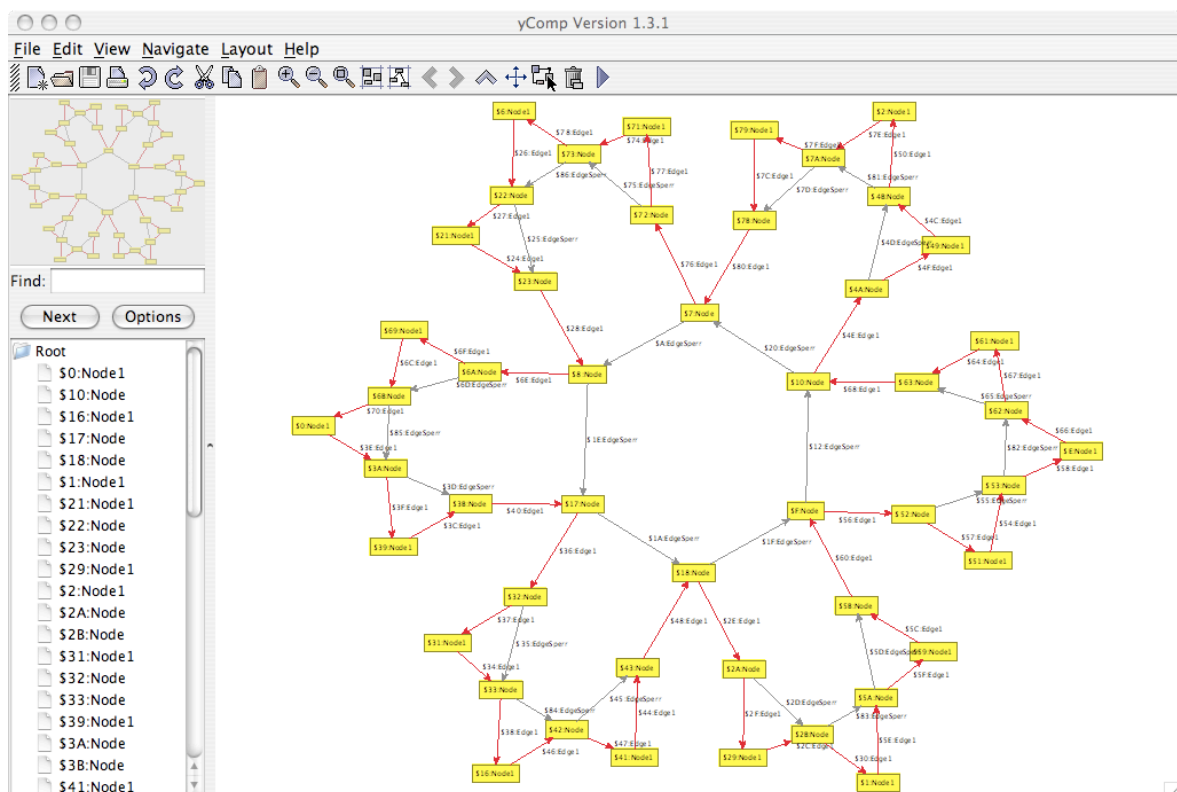


Figure 10.2: Koch snowflake

10.2 Busy Beaver

We want GRGEN.NET to work as hard as a busy beaver [Kro07, Dew84]. Our busy beaver is a Turing machine that has got five states plus a “halt”-state; it writes 1,471 bars onto the tape and terminates [MB00]. So first of all we design a Turing machine as graph model. Besides, this example shows that GRGEN.NET is Turing complete.

We use the graph model and the rewrite rules to define a general Turing machine. Our approach is to basically draw the machine as a graph. The busy beaver logic is implemented by rule applications in GRShell.

10.2.1 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 $[0 : 1]$.

```

2 node class TapePosition;
3 edge class right
4   connect TapePosition[0:1] --> TapePosition[0:1];
5
6 edge class value
7   connect TapePosition[1] --> TapePosition[1];
8 edge class zero extends value;
9 edge class one extends value;
10 edge class empty extends value;
```

Furthermore we need states and transitions. The machine’s current configuration is modeled with a `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 147).

```

12 node class State;
13
14 edge class RWHead;
15
16 node class WriteValue;
17 node class WriteZero extends WriteValue;
18 node class WriteOne extends WriteValue;
19 node class WriteEmpty extends WriteValue;
20
21 edge class moveLeft;
22 edge class moveRight;
23 edge class dontMove;
```

10.2.2 Rule Set

Now the rule set: We begin the rule set file `Turing.grg` with

```

1 using TuringModel;
```

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:

```

12 rule readOneRule {
13   s:State -h:RWHead-> tp:TapePosition -:one-> tp;
14   s -:one-> wv:WriteValue;
15   modify {
16     delete(h);
17     wv -:RWHead-> tp;
18   }
19 }
20
21 rule readEmptyRule {
22   s:State -h:RWHead-> tp:TapePosition -:empty-> tp;
23   s -:empty-> wv:WriteValue;
24   modify {
25     delete(h);
26     wv -:RWHead-> tp;
27   }
28 }
29
30 rule writeZeroRule {
31   wv:WriteZero -rw:RWHead-> tp:TapePosition -:value-> tp;
32   replace {
33     wv -rw-> tp -:zero-> tp;
34   }
35 }
36
37 rule writeOneRule {
38   wv:WriteOne -rw:RWHead-> tp:TapePosition -:value-> tp;
39   replace {
40     wv -rw-> tp -:one-> tp;
41   }
42 }
43
44 rule writeEmptyRule {
45   wv:WriteEmpty -rw:RWHead-> tp:TapePosition -:value-> tp;
46   replace {
47     wv -rw-> tp -:empty-> tp;
48   }
49 }
50
51 rule moveLeftRule {
52   wv:WriteValue -:moveLeft-> s:State;

```

```

53   ww -h:RWHead-> tp:TapePosition <-r:right- ltp:TapePosition;
54   modify {
55       delete(h);
56       s -:RWHead-> ltp;
57   }
58 }
59
60 rule moveRightRule {
61   ww:WriteValue -:moveRight-> s:State;
62   ww -h:RWHead-> tp:TapePosition -r:right-> rtp:TapePosition;
63   modify {
64       delete(h);
65       s -:RWHead-> rtp;
66   }
67 }
68
69 rule dontMoveRule {
70   ww:WriteValue -:dontMove-> s:State;
71   ww -h:RWHead-> tp:TapePosition;
72   modify {
73       delete(h);
74       s -:RWHead-> tp;
75   }
76 }
77
78 rule ensureMoveLeftValidRule {
79   ww:WriteValue -:moveLeft-> :State;
80   ww -:RWHead-> tp:TapePosition;
81   negative {
82       tp <-:right-;
83   }
84   modify {
85       tp <-:right- ltp:TapePosition -:empty-> ltp;
86   }
87 }
88
89 rule ensureMoveRightValidRule {
90   ww:WriteValue -:moveRight-> :State;
91   ww -:RWHead-> tp:TapePosition;
92   negative {
93       tp -:right->;
94   }
95   modify {
96       tp -:right-> rtp:TapePosition -:empty-> rtp;
97   }
98 }

```

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 10.1).

10.2.3 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:

```

1 select backend "../bin/lgspBackend.dll"

```

```

2 new graph "../lib/lgsp-TuringModel.dll" "Busy_Beaver"
3 select actions "../lib/lgsp-TuringActions.dll"
4
5 # Initialize tape
6 new tp:TapePosition($="Startposition")
7 new tp -:empty-> tp
8
9 # States
10 new sA:State($="A")
11 new sB:State($="B")
12 new sC:State($="C")
13 new sD:State($="D")
14 new sE:State($="E")
15 new sH:State($ = "Halt")
16
17 new sA -:RWHead-> tp
18
19 # Transitions: three lines per state and input symbol for
20 #   - updating cell value
21 #   - moving read-write-head
22 # respectively
23
24 new sA_0: WriteOne
25 new sA -:empty-> sA_0
26 new sA_0 -:moveLeft-> sB
27
28 new sA_1: WriteOne
29 new sA -:one-> sA_1
30 new sA_1 -:moveLeft-> sD
31
32 new sB_0: WriteOne
33 new sB -:empty-> sB_0
34 new sB_0 -:moveRight-> sC
35
36 new sB_1: WriteEmpty
37 new sB -:one-> sB_1
38 new sB_1 -:moveRight-> sE
39
40 new sC_0: WriteEmpty
41 new sC -:empty-> sC_0
42 new sC_0 -:moveLeft-> sA
43
44 new sC_1: WriteEmpty
45 new sC -:one-> sC_1
46 new sC_1 -:moveRight-> sB
47
48 new sD_0: WriteOne
49 new sD -:empty-> sD_0
50 new sD_0 -:moveLeft->sE
51
52 new sD_1: WriteOne
53 new sD -:one-> sD_1
54 new sD_1 -:moveLeft-> sH
55
56 new sE_0: WriteOne
57 new sE -:empty-> sE_0
58 new sE_0 -:moveRight-> sC
59
60 new sE_1: WriteOne

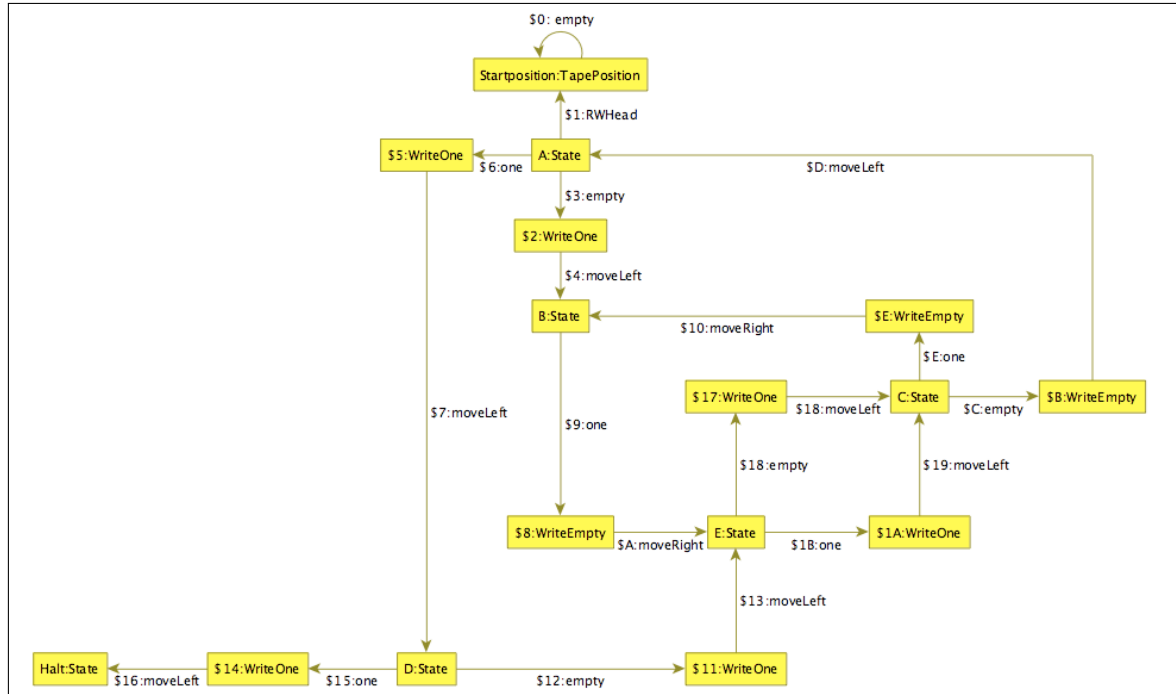
```

```

61 new sE -:one-> sE_1
62 new sE_1 -:moveLeft-> sC

```

Our busy beaver looks like this:



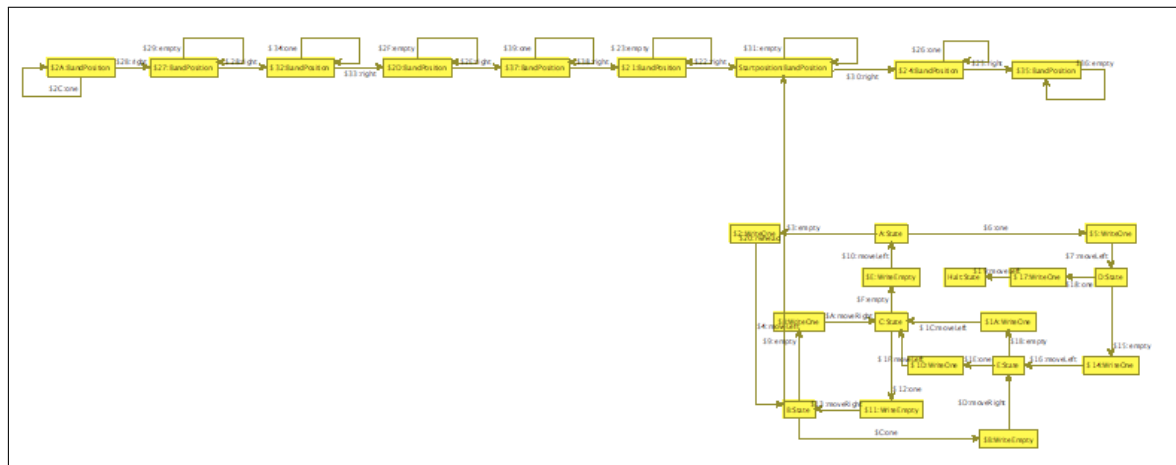
We have an initial host graph now. The graph rewrite sequence is quite straight forward and generic to the Turing graph model. Note that for each state the “...Empty... — ...One...” selection is unambiguous.

```

64 xgrs ((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 he takes about 8.5 seconds².

²On a Pentium 4, 3.2Ghz, with 2GiB RAM.

```
66 custom graph analyze_graph  
67 custom actions gen_searchplan readOneRule readEmptyRule writeOneRule writeEmptyRule  
    ensureMoveLeftValidRule ensureMoveRightValidRule moveLeftRule moveRightRule
```

Let the beaver run:

```
69 xgrs ((readOneRule | readEmptyRule) & (writeOneRule | writeEmptyRule) &  
    (ensureMoveLeftValidRule | ensureMoveRightValidRule) & (moveLeftRule |  
    moveRightRule))*
```


CHAPTER 11

APPLICATION PROGRAMMING INTERFACE

This chapter describes the Application Programming Interface of GRGEN.NET, i.e. of the system runtime - the LibGr - and of the assemblies generated from the model and rule specifications. We'll have a look at

- the interface to the graph and model
- the interface to the rules and matches
- the porter module for importing and exporting of graphs and miscellaneous stuff
- implementing external class and function declarations

From the input file `Foo.grg` `grgen.exe` generates 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.

NOTE (42)

If you work on the API level it is helpful to keep the generated source code which normally is thrown away after it was compiled into the assemblies `lgsp-FooModel.dll` and `lgsp-FooActions.dll`. Use the `-keep` option when you call `grgen.exe` to do so.

11.1 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, it is implemented by a sealed class `Bar` with generated code and with code from the lgsp backend. Furtheron 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(i.e. inexact) interfaces from `libGr`. Finally, the namespace contains a class `FooGraph` which defines an lgsp-graph of a model equivalent to `FooGraphModel`; it contains convenience functions to easily create nodes and edges of exact type in the graph.

NOTE (43)

If you want to use the type-safe interface, use the interface `IBar`, and the `CreateNodeBar`-methods of `FooGraph` or the `CreateNode`-method of `Bar`. If you want to use the generic interface, your entry point is the `IGraphModel`, with `INodeModel.GetType("Bar")` returning a `NodeType`, used in `IGraph.AddNode(NodeType)` returning an `INode`.

11.2 Interface to 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 defines how a match of the rule looks like, extending the generic rule-unspecific `IMatch` interface. Furtheron there 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, and the modify methods doing 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 there 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 `de.unika.ipd.grGen.libGr.BaseActions` (in fact `de.unika.ipd.grGen.lgsp.LGSPActions`), which supports iteration over the entities defined in the actions using further, generic(i.e. inexact) interfaces from `libGr`. Additionally, at runtime it contains the instances of the actions singletons, as member `bar` of the exact type `IAction_bar`.

NOTE (44)

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 (45)

The old generic interface of string names and entities of node-, edge-, and object-type is implemented with the new interface of exactly typed, named entities. Thus you will receive runtime exceptions when doing operations which are not type-safe with the generic interface, in contrast to GRGEN.NET < v2.5. If you need the flexibility of the old input parameters semantics of silently failing rule application on 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 (74)

Normally you want to use the type-safe interface of the generated code as it is much more convenient. Only if your application must get along with models and actions unknown before it is compiled 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. Here we'll show a short example on how to use GRGEN.NET with the type-safe API; further examples are given in the examples-api folder of the GRGEN.NET-distribution.

We'll start with including the namespaces of the libGr and the lgsp backend shipped with GRGEN.NET plus the namespaces of our actions and models, generated from `Foo.grg`.

```
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;
```

Then we create a graph with model bound at generation time and create actions to operate on this graph. Afterwards we create a single node of type `Bar` in the graph and save it to the variable `b`. Finally we apply the action `bar(Bar x) : (Bar)` to the graph with `b` as input receiving the output as well. The rule is taken from the actions via the member named as the action.

```
FooGraph graph = new FooGraph();  
FooActions actions = new FooActions(graph);  
Bar b = graph.CreateNodeBar();  
actions.bar.Apply(graph, b, ref b); // input of type Bar, output of type Bar
```

EXAMPLE (75)

This is an example doing mostly the same as the previous example 74, in a slightly more complicated way allowing for more control. Here we create the model separate from the graph, then the graph with a model not bound at generation time. We create the actions to apply on the graph, and a single node of type `Bar` in the graph, which we assign again to a variable `b`. 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` – which is different from the first version – and remember the found matches in the matches variable with its exact type. Finally we take the first match from the matches and execute the rewrite with 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).

```
IGraphModel model = new FooGraphModel();
LGSPGraph graph = new LGSPGraph(model);
FooActions actions = new FooActions(graph);
Bar b = Bar.CreateNode(graph);
IAction_bar bar = Action_bar.Instance;
IMatchesExact<Rule_bar.IMatch_bar> matches = bar.Match(graph, 0, b);
bar.Modify(graph, matches.First);
```

11.3 Import/Export and miscellaneous stuff

GrGen natively supports the following formats:

GRS/GRSI

Reduced GrShell script files (graph only, model from `.gm`; a very limited version of the normal `.grs`)

GXL

Graph eXchange Language (`.gxl`-files, see <http://www.gupro.de/GXL/>)

ECORE/XMI

Ecore(`.ecore`) model files and XMI(`.xmi`) graph file. Import only, export must be programmed with `emit`-statements. In an intermediate step, a `.gm` file is generated for the model.

While both GRS and GXL importers expect one file (the GXL importer allows to specify a model override, see GrShell import, Note 36), the EMF/ECORE importer expects first one or more `.ecore` files and following optionally a `.xmi` files and/or a `.grg` file (cf. Note 37). To use additional custom graph models you should supply an own `.grg` file which may be based on the automatically generated `.grg` file, if none was supplied (see the Program-Comprehension example in `examples/ProgramComprehension-GraBaTs09`).

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/JavaProgramGraphs-Example.cs`

The GRS(I) importer returns a `NamedGraph`, which is a wrapper for the `LGSPGraph` behind it (exposed via the `IGraph` interface of libGr). If you don't need the persistent names, get rid of them by casting to the named graph, getting the wrapped graph, and forgetting any

references to the named graph. If you want to keep it you should get the wrapped graph (casting it to `LGSPGraph`) and set its property `NamedGraph` to the named graph; this will cause the `nameof` operator to work as known, i.e. as if using `GrShell` which employs named graphs. Please be aware that naming is rather expensive. A named graph supplying the name to element and element to name mappings normally uses up about the same amount of memory as the wrapped `LGSPGraph` defining the real graph.

There are further examples available in the `examples-api` folder of the `GRGEN.NET`-distribution:

- How to use the graph rewrite sequences offered by the `libGr` on API level is shown in `examples-api/BusyBeaverExample/BusyBeaverExample.cs`.
But normally you want to use your favourite .NET programming language for control together with the type-safe interface when working on API level.
- 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 (hopefully) better performing matchers based on this information 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 the 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`.
- How to communicate with `yComp` on the API level (from your own code) is shown in `examples-api/YCompExample/YCompExample.cs`.

NOTE (46)

`LIBGR` allows for splitting a rule application into two steps: Find all the subgraphs of the host graph that match the pattern first, then rewrite one of these matches. By returning a collection of all matches, the `LIBGR` retains the complete graph rewrite process under control. As a `LIBGR` user have a look at the following methods of the `IAction` interface:

```
IMatches Match(IGraph graph, int maxMatches, object[] parameters);
object[] Modify(IGraph graph, IMatch match);
```

In C#, this might look like:

```
IMatches myMatches = myAction.Match(myGraph, -1, null); /* -1: get all the matches */
for(int i=0; i<myMatches.NumMatches; ++i)
{
    if(inspectCarefully(myMatches.GetMatch(i))
    {
        myAction.Modify(myGraph, myMatches.GetMatch(i));
        break;
    }
}
```

NOTE (47)

While C# allows input arguments values to be of a subtype of the declared interface parameter type (OO), it requires that the argument variables for the **out** parameters are 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 `rules Foo.grg` we can't 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;
```

11.4 External Class and Function Implementation

For a model `Foo` which contains external functions and/or classes (cf. [3.2.4](#) and [3.2.3](#)), GRGEN.NET generates a file `FooModelExternalFunctions.cs` located besides 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.
- within the `de.unika.ipd.grGen.expression` namespace a public partial class named `ExternalFunctions` with a body of comments giving the expected function prototypes.

The partial classes are empty, you must implement them in a file named `FooModelExternalFunctionsImpl.cs` located in the folder of the `FooModelExternalFunctions.cs` file by

- fleshing out the partial classes skeletons with attributes containing data of interest and maybe helper methods
- fleshing out 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 now know attributes or methods of the external classes, or do complicated custom computations with the values you receive from a function 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 generated (and overwritten) for you. In `examples/ExternalAttributeEvaluationExample` and `examples-api/ExternalAttributeEvaluationExample` you find a fabricated example showing how to use the external classes and functions.

11.5 How to build

In case you want to build GRGEN.NET on your own you should recall the system layout 1.1. The graph rewrite generator consists of a frontend written in Java and a backend written in C#. The frontend was extended and changed since its first version, but not replaced. 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 capable of matching nested and subpatterns. The frontend is available in the `frontend` subdirectory of the source distribution or the public subversion repository at <https://pp.info.uni-karlsruhe.de/svn/grgen-public/trunk/grgen>. It can be build with the included `Makefile` on Linux or the `Makefile.Cygwin` on Windows with the cygwin environment yielding a `grgen.jar`. Alternatively you may add the `de` subfolder and the jars in the `jars` subfolder to your favourite IDE, but then you must take care of the ANTLR parser generation pre-build-step on your own. The backend is available in the `engine-net-2` subdirectory. It contains a VisualStudio 2005 solution file containing projects for the `libGr`, the `lgspBackend` (`libGr-Search-Plan-Backend`) and the `GrShell`. Before you can build the solution you must execute `./src/libGr/genparser.bat`, `./src/libGr/GRSImporter/genparser.bat` and `./src/GrShell/genparser.bat` to get the CSharpCC parsers for the rewrite sequences, the GRS importer and the shell generated. Under LINUX you may use `make_linux.sh` to get a complete build. To get the API examples generated you must execute `./genlibs.bat`. The `doc` subdirectory contains the sources of the user manual, for building say `./build_cygwin.sh grgen` on Windows in `cygwin-bash` or `./build grgen` on Linux in `bash`. The `syntaxhighlighting` subdirectory contains syntax highlighting specifications for the GrGen-files for Notepad++ and vim.

You can check the result of your build with the test suite we use to check against regressions. It is split into syntactic tests in `frontend/test` checking that example model and rule files can get compiled by `grgen` (or not compiled, or only compiled emitting warnings) and the resulting code can get compiled by `csc`. The tests get executed by calling `./test.sh` or `./testpar.sh` from `bash` or `cygwin-bash` (`testpar.sh` executes them in parallel speeding up execution on multi core systems considerably at the price of potential false positive reports); deviations from a gold standard are reported. And semantic tests in `engine-net-2/tests` checking that executing example shell scripts on example models and rules yields the expected results. They get executed by calling `./test.sh`. You must check that all tests are reported as `Success`.

11.6 A very short tour of the code

The frontend is spread over the directories `parser`, `ast`, `ir`, `be` and `util`, with their code being used from `Main.java`. 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 file `GrGen.g`. The semantic actions of the parser build an abstract syntax tree consisting of instances of a multiple classes as 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 in an (largely) preorder walk. Afterwards the AST is checked by `check` and `checkLocal` in an (largely) postorder walk for type and other semantic constraints. Finally an intermediate representation is built from the abstract syntax tree by the `getIR` and `constructIR` methods. 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 IR classes are the input to the two backends of the frontend, as given in the folders `be/C` and `be/Csharp`. The directory `be/C` contains the code generator for the C based backend 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.) The directory `be/Csharp` contains the code generator for the C# based backend of GRGEN.NET. It generates the model source code `FooModel.cs` with the node and edge classes for a rule file named `Foo.grg`, and the intermediate rules source code `FooActions_intermediate` with a description of the patterns to match, a description of the embedded graph rewrite sequences, and the rewriting code. It does *not* generate the complete rules source code with the matcher code or the code for the embedded rewrite sequences, this is done by `grgen.exe` which only calls the `grgen.jar` of the frontend. You may call the Java archive on your own to get a visualization of the model and rewrite rules from a `.vcg-dump` of the IR, cf. Note2.

The real matcher code is generated by the backend given in `engine-net-2`, in the `src/lgspBackend` subdirectory. The processing is done in several passes in `lgspMatcher-Generator.cs`; 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 `Matching-Pattern`-objects of the subpatterns. First a `PlanGraph` is created from the `PatternGraph` and data from analysing the host graph (for generating the initial matcher some default data given from the frontend is used). A minimum spanning arborescent is searched defining a hopefully optimal set of operations for matching the pattern (the hopes are founded, see [BKG08]). A `SearchPlanGraph` is built from the arborescent marked in the `PlanGraph` and used thereafter for scheduling the operations into a `ScheduledSearchPlan`. Out of the `ScheduledSearchPlan` a `SearchProgram` is built by the `SearchProgramBuilder`; in a further pass the `SearchProgram` is completed by the `SearchProgramCompleter`, before the C# code gets finally generated by calling the `emit` methods of the `SearchProgram`. The compiled graph rewrite sequences are handled by the `lgspSequenceChecker` and `lgspSequenceGenerator` (together with the sequence parser from the libGr). The `src/GrGen` subdirectory contains the `grgen.exe` compiler driver procedure. The `src/libGr` subdirectory contains the libGr, offering the base interfaces a user of GRGEN.NET sees on the API level for the model, the actions, the pattern graphs and the host graph. They get implemented in code from the lgsp backend and in the generated code. The libGr further offers a generic, name string and object based interface to access the named entities of the generated code. In addition it offers the rewrite sequence parser which gets generated out of `SequenceParser.csc`, building the rewrite sequence AST from the classes in `Sequence.cs` further utilizing `SymbolTable.cs`. The rewrite sequence classes contain a method `ApplyImpl(IGraph graph)` which executes them. Finally the libGr offers several importers and exporters in the `src/libGr/IO` and `src/libGr/GRSImporter` subfolders. The `src/GrShell` subdirectory contains the GrShell application, which builds upon the generic interface (as it must be capable of coping with arbitrary used defined models and actions at runtime) and the sequence interpretation facilities offered by the libGr. The command line parser of GrShell gets generated out of `GrShell.csc`, the shell implementation is given in `GrShellImpl.cs`. Graphical debugging is offered by the `Debugger.cs` together with the `YCompClient.cs`. The `examples` subdirectory of `engine-net-2` contains a bunch of examples for using GRGEN.NET with GrShell. The `examples-api` subdirectory contains several examples of how to use GRGEN.NET from the API. 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 `ipd.info.uni-karlsruhe.de`).

BIBLIOGRAPHY

- [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.
- [Bat05b] Veit Batz. Generierung von Graphersetzungen mit programmierbarem Suchalgorithmus. Studienarbeit, 2005.
- [Bat06] Gernot Veit Batz. An Optimization Technique for Subgraph Matching Strategies. Technical Report 2006-7, Universität Karlsruhe, Fakultät für Informatik, April 2006.
- [BG09] Paul Bédaride and Claire Gardent. Semantic Normalisation: a Framework and an Experiment. In *Eighth International Conference on Computational Semantics*, 2009.
- [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. preliminary version, submitted to AGTIVE 2007.
- [Buc08] Sebastian Buchwald. Erweiterung von GrGen.NET um DPO-Semantik und ungerichtete Kanten, 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 *[Roz99]*, volume 1, pages 163–245. 1999.
- [Dew84] A. K. Dewdney. A computer trap for the busy beaver, the hardest-working turing machine. *Scientific 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 *[Roz99]*, volume 1, pages 247–312. 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 *[Roz99]*, volume 2, pages 551–603. 1999.

- [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*. PhD thesis, Universität Karlsruhe, Nov 2008.
- [Hac03] Sebastian Hack. Graphersetzung für Optimierungen in der Codeerzeugung. Master’s thesis, IPD Goos, 12 2003.
- [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>".
- [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.
- [Jak08] Edgar Jakumeit. Mit GrGen.NET zu den Sternen – Erweiterung der Regelsprache eines Graphersetzungswerkzeugs um rekursive Regeln mittels Sterngraphgrammatiken und Paargraphgrammatiken. Master’s thesis, Universität Karlsruhe, jul 2008.
- [JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen.NET. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:263–271, 2010. 10.1007/s10009-010-0148-8.
- [KBG⁺07] Moritz Kroll, Michael Beck, Rubino Geiß, Sebastian Hack, and Philipp Leiß. yComp. <http://www.info.uni-karlsruhe.de/software.php?id=6>, 2007.
- [KG07] Moritz Kroll and Rubino Geiß. Developing Graph Transformations with GrGen.NET. In *Applications of Graph Transformation with Industrial relevance - AGTIVE 2007*, 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.
- [Kro] Moritz Kroll. G#: GrGen.NET in C#. Master’s thesis, Universität Karlsruhe (TH).
- [Kro07] Moritz Kroll. GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen, 5 2007. Studienarbeit, Universität Karlsruhe.
- [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.
- [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.
- [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.
- [Plu09] Detlef Plump. The Graph Programming Language GP. In *Proc. Algebraic Informatics (CAI 2009)*, pages 99–122, 2009.
- [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 Language C: ANSI/ISO 9899-1990*. 1990.
- [San95] Georg Sander. VCG Visualization of Compiler Graphs—User Documentation v.1.30. Technical report, Universität des Saarlandes, 1995.
- [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.
- [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 suchprogramm-basierte Backend von GrGen, 2005. Studienarbeit, Universität Karlsruhe.
- [Tea07] The Mono Team. Mono. <http://www.mono-project.com/>, 2007.

- [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 *Applications of Graph Transformations with Industrial Relevance*, pages 456–470. 2008.
- [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.
- [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

SubpatternOccurence, 82

abstract, 23

actions, 135, 140

add, 125, 133, 134

alternative, 70

analyze, 139

analyzegraph, 139

arbitrary, 23

askfor, 120

attributes, 130

backend, 139

bordercolor, 132

by, 133

class, 24, 26

clear, 122

color, 132

connect, 25

const, 23, 27

copy, 25

custom, 135, 139, 140

dangling, 35

debug, 136

def, 90

delete, 43, 82, 122, 129

directed, 23

disable, 136

dpo, 35

dump, 131–134

dumpsourcocode, 140

echo, 120

edge, 24, 129–134

edges, 130

emit, 88, 95, 121

emithere, 96

enable, 136

enum, 22

eval, 43

exact, 35, 39

exclude, 133

exec, 95

exit, 120

exitonfailure, 122

export, 124

extends, 24–26

for, 101

from, 126

gensearchplan, 140

get, 136

graph, 121, 122, 124, 131, 139

group, 133

help, 119

hidden, 133

hom, 39

identification, 35

if, 39, 91

import, 124, 125

in, 57, 58, 99, 101

include, 33, 120

independent, 68

induced, 35, 39

infotag, 134

is, 131

iterated, 69

labels, 132

layout, 136

modify, 42, 72, 76, 78

multiple, 69

nameof, 60

negative, 66

new, 121, 127

node, 24, 129–134

nodes, 130

null, 36

num, 130

off, 121, 132

on, 121, 132

only, 122, 130, 132–134

open, 121

optimizereuse, 139

option, 136

optional, 69

options, 136

pattern, 72

quit, 120

[randomseed](#), 121
[record](#), 88, 126
[redirect](#), 121
[ref](#), 36
[replace](#), 42, 72, 76, 78
[replay](#), 126
[reset](#), 134
[return](#), 40
[retype](#), 128, 129
[rule](#), 35
[save](#), 124
[select](#), 121, 135, 139
[set](#), 132, 136
[setmaxmatches](#), 139
[shape](#), 132
[shortinfotag](#), 134
[show](#), 119, 121, 130, 131, 135, 139
[silence](#), 121
[specified](#), 122
[start](#), 126
[stop](#), 126
[strict](#), 122
[sub](#), 130
[super](#), 130
[test](#), 35
[textcolor](#), 132
[time](#), 121
[to](#), 126
[type](#), 131
[typeof](#), 46, 59
[types](#), 130
[undirected](#), 23
[using](#), 33
[validate](#), 25, 122
[valloc](#), 102
[var](#), 36, 119
[vfree](#), 102
[visited](#), 60, 102, 105
[vreset](#), 102
[with](#), 133
[withvariables](#), 124
[xgrs](#), 122, 135, 136

Non-Terminals

[ActionSignature](#), 35
[AlternativePattern](#), 70
[Assignment](#), 43
[AttributeDeclaration](#), 27
[AttributeOverwrite](#), 27
[AttributeType](#), 27
[Attribute Value](#), 128
[Attributes](#), 128

[BoolExpr](#), 53
[CastExpr](#), 60
[ClassDeclaration](#), 23
[Command](#), 119
[CompoundAssignment](#), 104
[ConnectionAssertion](#), 25
[Constant](#), 61
[Constructor](#), 128
[Continuation](#), 30
[CopyOperator](#), 48
[EdgeClass](#), 24
[EdgeRefinement](#), 32
[EnumDeclaration](#), 22
[ExecStatement](#), 95
[Expression](#), 53
[ExtendedControl](#), 91
[ExternalClassDeclaration](#), 26
[ExternalFunctionDeclaration](#), 26
[FileHeader](#), 33
[FloatExpr](#), 56
[FunctionCall](#), 60
[GraphElement](#), 118
[GraphModel](#), 22
[GraphRewriteSequence](#), 135, 136
[Graphlet](#), 30
[GraphletEdge](#), 32
[GraphletNode](#), 31
[GroupConstraints](#), 133
[IdentDecl](#), 49
[IncAdjTypeConstraints](#), 133
[InputTypeSpecification](#), 36
[IntExpr](#), 55
[Literal](#), 61
[MapConst](#), 128
[MapConstructor](#), 62
[MapExpr](#), 58
[MemberAccess](#), 60
[MultiplicityConstraint](#), 25
[NegativeApplicationCondition](#), 66
[NestedPattern](#), 66
[NestedPatternWithCardinality](#), 69
[NestedRewriting](#), 76
[NodeClass](#), 24
[NodeConstraint](#), 25
[Parameter](#), 36
[Parameters](#), 117
[Pattern](#), 37
[PatternStatement](#), 39
[PositiveApplicationCondition](#), 68
[PrimaryExpr](#), 60
[PrimitiveAttribute Value](#), 128
[RHS](#), 100

RandomSelection, 89
RangeConstraint, 25
RelationalExpr, 54
Replace, 42
ReplaceStatement, 43
ReturnStatement, 40
ReturnTypes, 37
Retyping, 47
RewriteFactor, 88, 101
RewriteNegTerm, 86
RewriteSequence, 86
RewriteTerm, 87
Rule, 89
RuleDeclaration, 35
RuleExecution, 89
RuleSet, 33
Script, 119
SequencesList, 92
SetConstr, 128
SetConstructor, 62
SetExpr, 57
SetMapCreation, 100
SetMapStateChange, 104
SetMapTypeDecl, 99
SpacedParameters, 117
StorageAccess, 103
StringExpr, 56
SubpatternBody, 72
SubpatternDeclaration, 72
SubpatternEntityDeclaration, 72
SubpatternExecEmit, 96
SubpatternRewriteApplication, 79
SubpatternRewriting, 78
TestDeclaration, 35
Type, 46
TypeConstraint, 45
TypeExpr, 59
VariableHandling, 90, 99
VisitedAssignment, 105
VisitedFlagsManagement, 102

General Index

.gm, 4
 .grg, 4, 7
 .grs, 4, 8
 !, 93, 120
 (), 93
 *, 93
 +, 93
 ;;, 119
 >, 93
 <;, 93

<<;>>, 93
 <>, 93
 @, 118
 [], 93
 #, 117
 \$<number>, 30
 \$<op>, 38, 93
 \$, 128
 &&, 93
 &, 93
 ^, 93
 ||, 93
 |, 93

 action, *see* graph rewrite sequence
 action command, 135
 adjacent, 37
 AEdge, 21
 all bracketing, 89
 alternative patterns, 70
 analyzing graph, 139
 annotation, 20, 30, 49
 anonymous, 30, 40, 41, 118
 API, 4, 9, 149
 application, 5, 38
 application programming interface, *see* API
 arbitrary, 21, 32
 attribute, 27, 128–130, 134
 attribute condition, 39
 attribute evaluation, 43

 backend, 4, 51, 135, 139
 backslash, 119
 backtracking, 91
 binding of names, 30
 boolean, 51
 break point, 89–91, 137
 building GrGen, 155
 built-in generic types, *see* generic types
 built-in types, *see* primitive types
 busy beaver, 143
 by-reference, 36
 by-value, 36

 cardinality, *see* pattern cardinality
 case sensitive, 20, 29, 117
 choice point, 86, 89, 90, 92, 137
 color, 118, 132
 command line, 120
 comment, 117
 compatible types, *see* type-compatible
 compiler graph, *see* layout algorithm
 conditional sequence, 91

- connection assertion, [24](#), [25](#), [122](#)
- constructor, [118](#), [128](#)
- continuation, *see* graphlet
- copy, [48](#)
- copy structure, *see* subgraph copying

- dangling condition, [45](#)
- data flow analysis, [112](#)
- debug mode, [136](#)
- debugger, [136](#)
- declaration, [20](#), [22](#), [38](#)
- default graph model, [29](#)
- default search plan, [140](#)
- default value, [128](#)
- definition, [20](#), [49](#)
- deletion, [40](#), [43](#)
- determinism, *see* non-determinism
- directed, [32](#)
- double, [51](#)
- double-pushout approach, [10](#)
- DPO, *see* double-pushout approach
- dumping graph, [124](#)
- dynamic type, [46](#)

- EBNF, *see* rail diagram, *see* regular expression syntax
- Edge, [21](#), [24](#)
- edge (graphlet), [32](#)
- edge type, [24](#)
- edNCE, *see* node replacement grammar
- emit, [88](#)
- empty pattern, [5](#), [37](#)
- enum item, [22](#), [52](#)
- enum type, [22](#), [51](#)
- evaluation, *see* attribute evaluation
- exact dynamic type, *see* dynamic type
- example, [5](#), [13](#), [141](#), [143](#)
- export, [124](#), [152](#)
- expression, [52](#)
- expression variable, [60](#), *see* name, [127](#)
- external class, [22](#)
- external class implementation, [154](#)
- external function, [22](#), [26](#)
- external function implementation, [154](#)

- features, [2](#)
- float, [51](#)
- flow equations, [112](#)

- generator, [4](#)
- generic types, [51](#)
- graph global variable, [118](#)
- graph global variables, [85](#)
- graph model, [4](#), [19](#), [22](#), [29](#), [33](#), [121](#)
- graph model language, [19](#)
- graph rewrite rules, [5](#)
- graph rewrite script, [4](#), [8](#), [120](#), [124](#)
- graph rewrite sequence, [38](#), [85](#), [122](#), [135](#), [136](#)
- graph rewriting, [5](#)
- graphlet, [30](#), [39](#), [40](#), [43](#)
- GrGen.exe, [7](#)
- group node, [133](#)
- GRS, *see* graph rewrite sequence
- GrShell, [4](#), [8](#), [23](#), [117](#)
- GrShell script, *see* graph rewrite script
- GrShell variable, *see* graph global variable
- GrShell.exe, [8](#)

- hierarchic, *see* layout algorithm, *see* group node
- homomorphic matching, [30](#), [39](#)
- host graph, [5](#), [40](#), [121](#)

- identification condition, [45](#)
- identifier, [20](#), [30](#)
- imperative, *see* attribute evaluation
- imperative statements, [95](#)
- imperativeandstate, [95](#)
- import, [124](#), [152](#)
- indeterministic choice, [91](#)
- info tag, [134](#)
- inheritance, [19](#), [24](#), [130](#)
- int, [51](#)
- isomorphic matching, [39](#)

- Kantorowitsch tree, [137](#)
- Koch snowflake, [141](#)

- label, [119](#), [134](#)
- layout, *see* layout algorithm, *see* visualization
- layout algorithm, [9](#), [136](#), [141](#)
- left hand side, [5](#), [40](#)
- LGPL, [7](#)
- LGSPBackend, [51](#), [139](#)
- LHS, *see* left hand side
- libGr, [4](#), [9](#), [23](#), [38](#), [117](#)
- loop, [87](#)

- map, [51](#), [99](#)
- match, [5](#)
- matching strategies, [10](#)
- merge node, [106](#)
- modifier, [44](#)
- modify mode, [41](#)
- multiplicity, *see* connection assertion

- NAC, *see* negative application condition
- name, [30](#), [40](#), [127](#)

- negative application condition, 66
- nested graph, *see* group node, 131
- nested layout, *see* group node, 131
- nested pattern rewrite, 76
- nested transaction, *see* transaction
- Node, 24
- node (graphlet), 31
- node replacement grammar, 109
- node type, 24
- non-determinism, 38
- object, 51
- one-of-set braces, 92
- order of precedence, 53, 55–58, 63
- organic, *see* layout algorithm
- orthogonal, *see* layout algorithm
- overview, system, 4
- PAC, *see* positive application condition
- parameter, 35, 135, 139
- pattern, 37
- pattern cardinality, 68, 69
- pattern graph, 5, 30
- pattern modifiers, 44
- persistent graph, 121
- persistent name, 90, 118, 128, 130
- positive application condition, 68
- pragma, *see* annotation
- precedence, *see* order of precedence
- preservation, 40
- preservation morphism, 5, 40
- primitive types, 51
- prio, 49
- quickstart, 13
- rail diagram, 19
- random match selector, 89
- random-all-of operators, 92
- re-evaluation, *see* attribute evaluation
- reachability, 112
- record, 88, 126
- recursive pattern, 73
- redefine, 30
- redirecting, 32
- regular expression syntax, 83, 87
- replace mode, 41
- replacement graph, 5, 30, 40
- replay, 126
- return type, 35
- return value, 40
- retype, 31, 128
- retyping, 47
- rewrite rule, 4, 35
- RHS, *see* right hand side
- right hand side, 5, 40
- rule application, *see* application
- rule application language, 85
- rule modifiers, 44
- rule set, 29, 33, 135
- rule set language, 29
- rule set language nested and subpatterns, 65
- scope, 31, 66, 68
- script, *see* graph rewrite script
- search plan, 49, 139, 147
- search plans, 10
- sequence constant, 91
- sequence local variables, 85
- set, 51, 99
- shell, *see* GrShell
- short info tag, 134
- Sierpinski triangle, 141
- signature, 35
- simulation, 137
- single-pushout approach, 10, 32, 40
- some-of-set braces, 92
- spanning tree, 75
- split node, 106
- SPO, *see* single-pushout approach, 44
- spot, 5, 30
- storage, 98
- storage access, 103
- string, 51
- structural recursion, 73
- subgraph copying, 111
- subpattern, 71
- subpattern declaration, 72
- subpattern entity declaration, 72
- subrule, 78
- syntax diagram, *see* rail diagram
- test, 35, 37
- transaction, 91
- transformation specification, 40
- Turing complete, 143
- type cast, *see* retyping, 51
- type constraint, 31, 32, 45
- type expression, 30, 54, 59
- type hierarchy, 19, 21, 45, 48
- type-compatible, 131
- UEdge, 21
- UML class diagram, 54
- undefined variables, 135
- undirected, 32
- user input assignment, 90

- validate, [122](#)
- variable, *see* expression variable, [60](#), *see* graph
 - global variables, *see* sequence local
 - variables, [118](#), [127](#), [135](#)
- variable predicate, [90](#)
- Varró's benchmark, [10](#)
- VCG, [9](#), [118](#), [131](#)
- visited flag, [60](#), [98](#), [105](#)
- visualization, *see* group node, [131](#)

- working graph, [121](#)
- worklist, [115](#)

- yComp, [9](#), [131](#), [137](#)
- yComp.jar, [137](#)