# The GRGEN.NET User Manual

## Refers to GRGEN.NET Release 2.6 RC3

## —DRAFT—

www.grgen.net

Jakob Blomer      Rubino Geiß

Edgar Jakumeit

May 24, 2010

# ABSTRACT

GRGEN.NET is a graph rewrite tool enabling elegant and convenient development of graph rewriting applications with comparable performance to conventionally developed ones. GRGEN.NET uses attributed, typed, and (un)directed multigraphs with multiple inheritance on node and edge types. Extensive graphical debugging integrated into an interactive shell complements the feature highlights of GRGEN.NET. 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.

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 s 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. 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), 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 some interesting results of this work at IPD Tichy), Paul Bedaride, Normen Müller, and Nicholas Tung.

Regarding questions please contact the GRGEN.NET-Team via email to grgen 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 enjoyable ;).

Thank you for using GRGEN.NET.

Karlsruhe in May 2010, Edgar Jakumeit on behalf of the GRGEN.NET-Team

# 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

# CONTENTS

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 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 arbitraty 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
  - Two modes of specification: A rule can either express changes to be made to the match or replace the whole match
  - 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
  - A rewrite part for the nested patterns and subpatterns, so that complex structures can not only get matched, but also get rewritten

  - Visited flags (which can be written here, in addition to reading them in the pattern language)

  - Parameter passing out of rules

- The rule application language (grs: graph rewrite sequences) supports:

  - Composing several rules with sequential, logical, and iterative control

  - Variables to store graph elements handed into rules and returned from rules

  - Visited flags management

  - Storages, i.e. sets and maps capable of storing graph elements and allowing element-wise iteration

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 manipulation, |
| – graph and model queries, | – graph visualisation, |
| – 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.
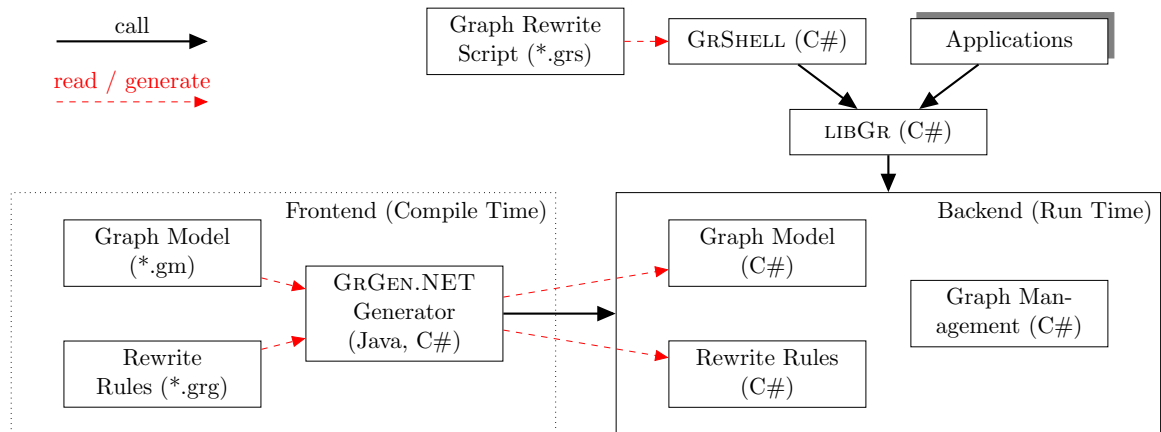


Figure 1.1: GRGEN.NET system components [Kro07]

A graph rewrite system[1] 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.

---

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

Figure 1.2: Generating a graph rewrite system

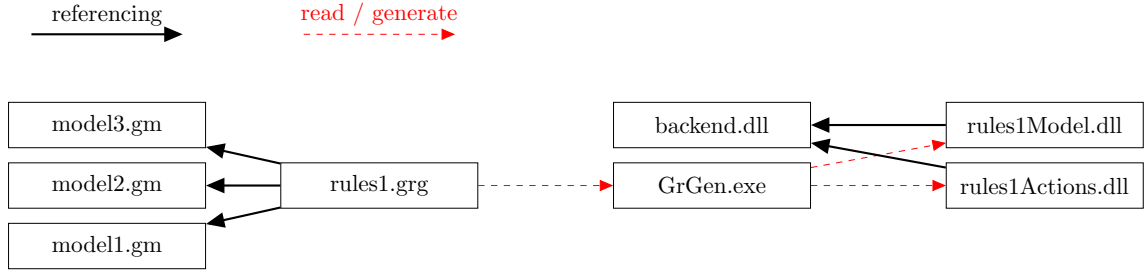In general you have to distinguish carefully between a graph model (meta level), a host graph, a pattern graph and a rewrite rule. In GRGEN.NET pattern graphs are implicitly defined by rules, i.e. each rule defines its pattern. On the technical side, specification documents for a graph rewrite system 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 GR-GEN.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 13 on page 34.

## 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*.
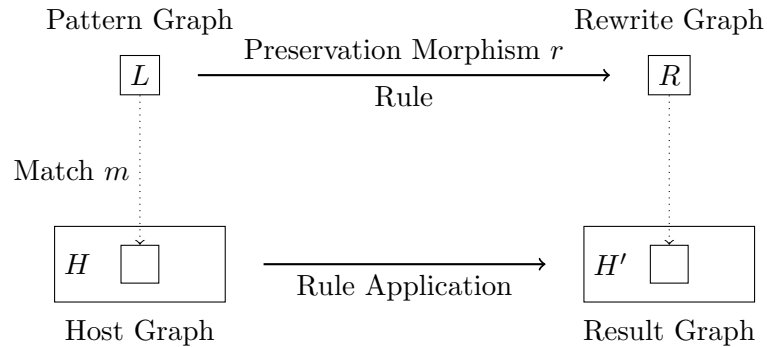


Figure 1.3: Basic Idea of Graph Rewriting

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 one[2] match (independent of the host graph). So we construct an apple by applying
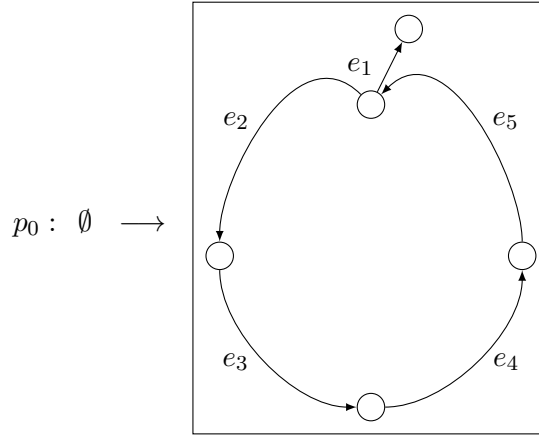


to the empty host graph. As the result we get an apple as new host graph $H$. Now we want to rewrite our apple with stem to an apple with a leaflet. So we apply
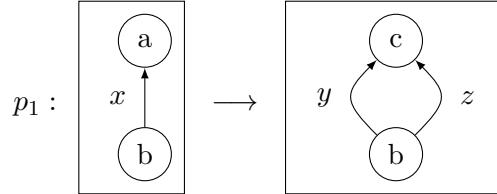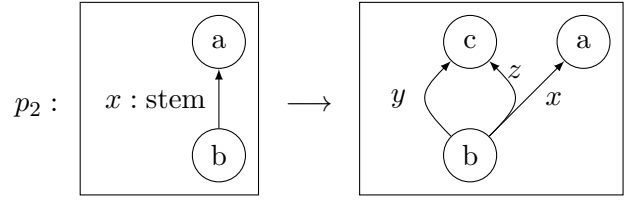


to $H$ and get the new host graph $H_1$, something like this:
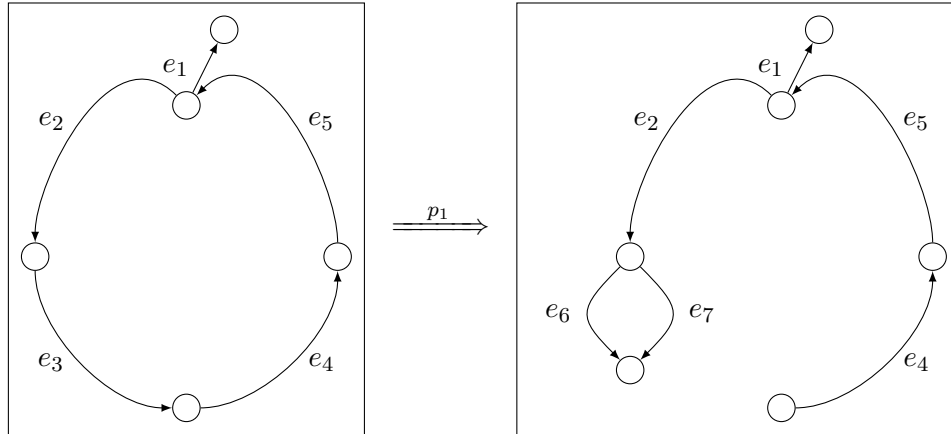


What happened? GRGEN.NET has arbitrarily chosen one match out of the set of possible matches, because $x$ matches edge $e_3$ as well as $e_1$. A correct solution could make use of edge

---

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

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

$$p_2 : \qquad \boxed{\begin{array}{c} \text{a} \\ \uparrow \\ x : \text{stem} \\ \text{b} \end{array}} \quad \longrightarrow \quad \boxed{\begin{array}{c} \text{c} \qquad \text{a} \\ y \qquad z \quad x \\ \text{b} \end{array}} \quad .$$

If we apply $p_2$ to the modified $H_1$ this leads to
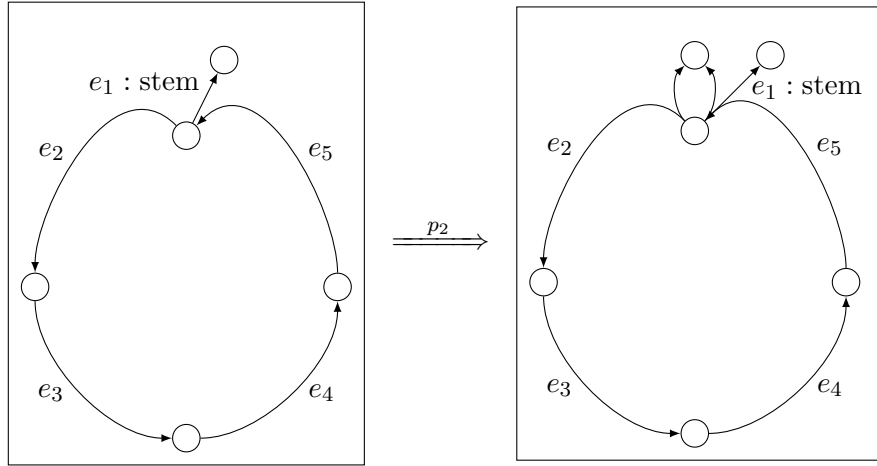


## 1.7   The Tools

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 GR-GEN.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 `tmpgrgen`$n$[3] within the current directory will be created. The destination directory contains:

-   • `printOutput.txt`—a snapshot of `stdout` during program execution.
-   • *Name*`Model.cs`—the C# source file(s) of the *rule-set*`Modell.dll` assembly.
-   • *Name*`Actions_intermediate.cs`—a preliminary version of the C# source file of the *rule-set*'s actions assembly. This file is for internal debug purposes only (it contains the frontend actions output).
-   • *Name*`Actions.cs`—the C# source file of the *rule-set*`Actions.dll` assembly.

-use    Don't re-generate C# source files. Instead use the files in *existing-dir* to build the assemblies.

-debug    Compile the assemblies with debug information.

-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 8.

*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-script*s (usually `*.grs` files) in the given order. The `grs` suffixes may be omitted. GRSHELL returns 0 on successful execution, or in non-interactive mode -1 if the specified shell script could not be executed, or -2 if a `validate` with `exitonfailure` failed.

---

[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/libGr/; a short introduction is given in chaper 10.

### 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 (`hierarchic` optimized for graph based compiler intermediate representations) may not be a good choice for your graph at hand. Instead `organic` or `orthogonal` might be worth trying). Use the rightmost blue play button to start the layout process. Depending on the graph size this may take a while.
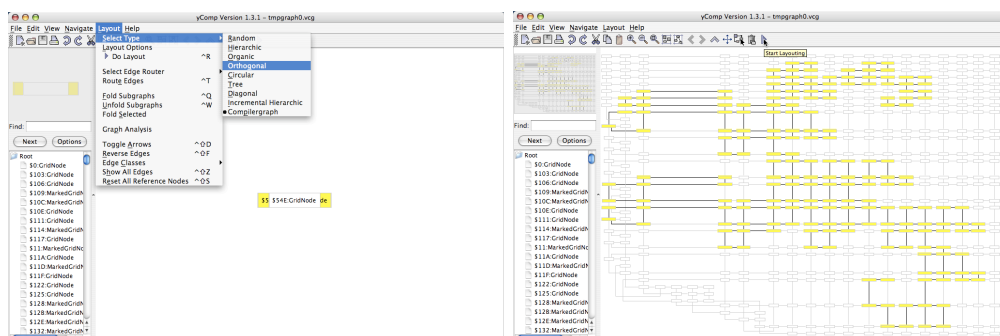


*Requires*

Java Runtime Environment 1.5 (or above).

## 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, builing 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, easening 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 easening 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 may 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.

**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 achived 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 tools are geared towards special purpose applications, e.g. model transformation or biology (corresponding examples are VIATRA2[VB07] and XL [KK07]). This means that design decisions were taken to ease uses in this applications at the cost of rendering uses in other domains more difficult. While the old GRGEN was built as a special purpose compiler construction tool, the new GRGEN.NET was built from the beginning as a general purpose graph transformation tool, 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 $\varepsilon$-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 machine[1]. By graph meta model we mean a set of node types and
edge types which are available for building state machine graphs (see Chapter 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 8) 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 similiar 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`.

---

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

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

Figure 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 `$` and `@` notation is due to the fact that we have two kinds of "names" in the GRSHELL. Namely we have *shell variables*—which we did not use, no shell variable is explicitly defined in this script—and *persistent names* that denote a specific graph element. Persistent names are set by `$=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 $\varepsilon$-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 $\varepsilon$-transitions alltogether. 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 `-:EpsilonTransistion->`, 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 30 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 boolean conditions, expressed by `if {...}`; a rule is only applicable if all such conditions hold 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 $\varepsilon$-transitions

instance we don't want to forward just one $\varepsilon$-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-ands (`&&`), 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
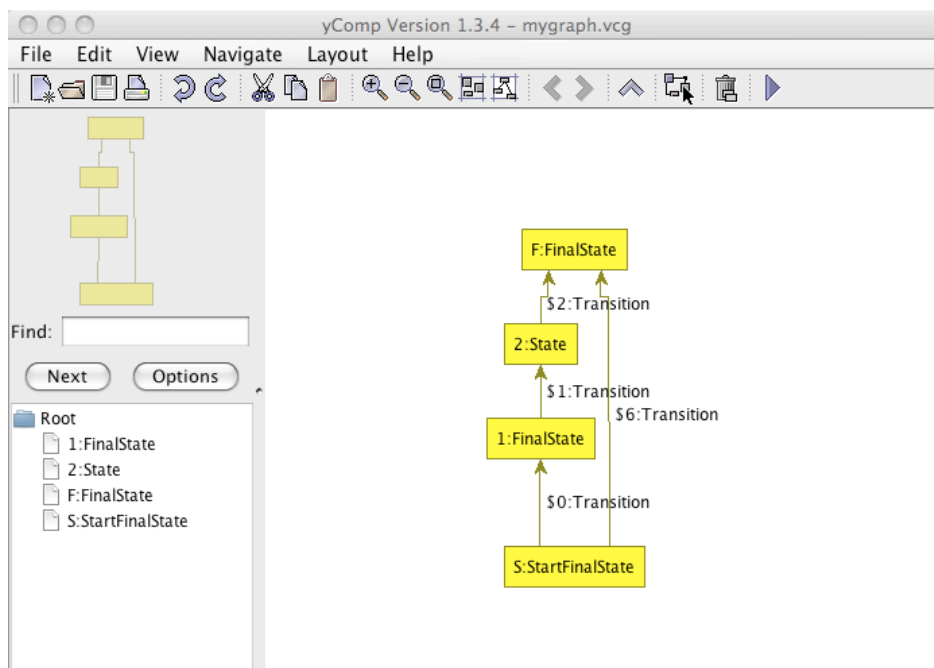


Figure 2.5: A state machine without $\varepsilon$-transitions

If everything is working fine you can delete the `debug` keyword in front of `xgrs`. Maybe you want to save the resulting graph. This is possible by typing `dump graph mygraph.vcg` in the GRSHELL. GRSHELL writes the graph in `mygraph.vcg` into the current directory. Files in VCG format are readable by YCOMP. Feel free to browse the `examples` folder shiped with GRGEN.NET and have a look at further 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 8 (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 diagram*s (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].

---

[a]Extended 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 GR-GEN.NET graph model language is case sensitive.

*Ident, IdentDecl*

A non-empty character sequence of arbitrary length consisting of letters, digits, or underscores. The first character must not be a digit. *Ident* and *IdentDecl* differ in their role: While *IdentDecl* is a *defining* occurrence of an identifier, *Ident* is a *using* occurrence. An *IdentDecl* non-terminal may be annotated. See Section 4.9 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 anchestors 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 hierarchie, you are not able to let anchestor 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.
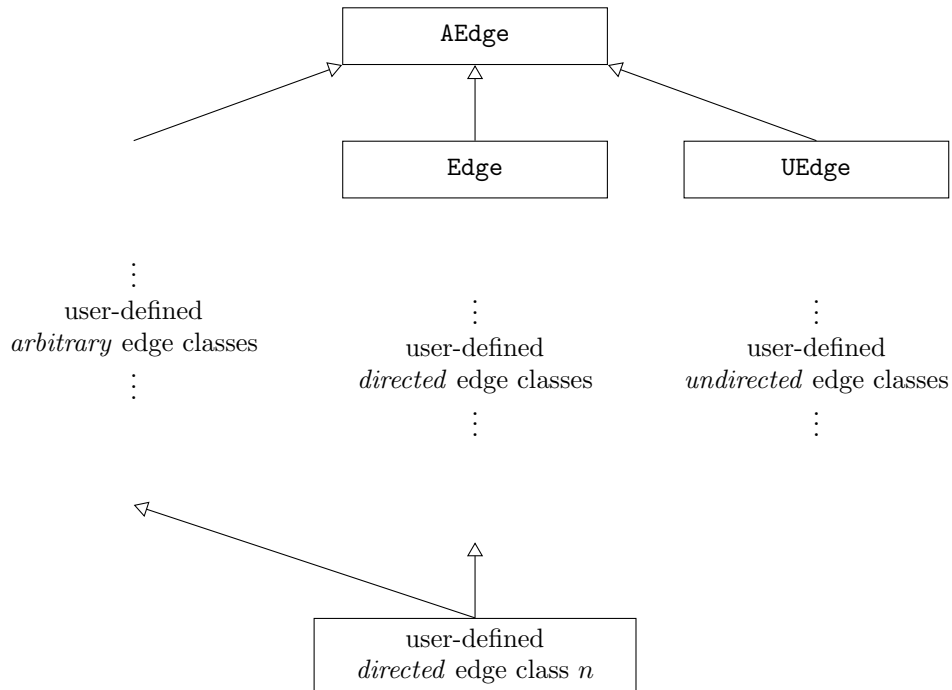


Figure 3.1: Type Hierarchy of GRGEN.NET Edges

## 3.2   Type Declarations

*GraphModel*
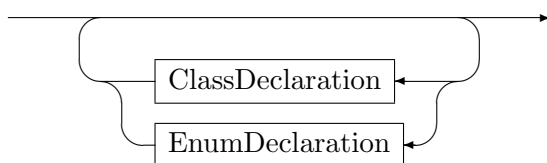


The graph model consists of zero or multiple type declarations. Whereas *ClassDeclaration* defines a node type or an edge type, *EnumDeclaration* defines an enum type to be used as a

type for a attributes of nodes or edges. Like all identifier definitions, types do not need to be declared before they are used.

*EnumDeclaration*
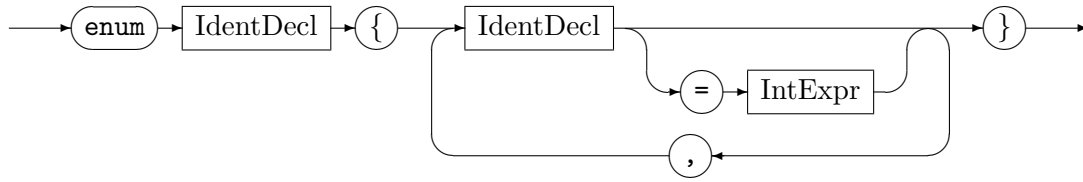


Defines an enum type. An enum type is a collection of so called *enum items* that are associated with integral numbers, each. Accordingly, a GRGEN.NET enum is internally represented as `int` (see Section 6.1).

---

**NOTE (6)**

An enum type and an `int` are different things, but in expressions enum values are implicitly casted to `int` values (see Section 6.1).

---

**NOTE (7)**

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

*ClassDeclaration*
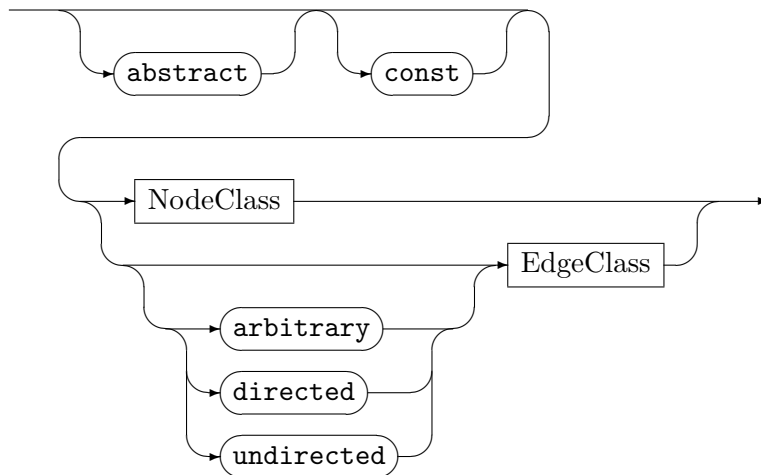


Defines a new node type or edge type. The keyword `abstract` indicates that you cannot instantiate graph elements of this type. Instead you have to derive non-abstract types to create graph elements. The abstract-property will not be inherited by subclasses, of course.

---

**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 choosen by default. See also Section 3.1.1

*NodeClass*
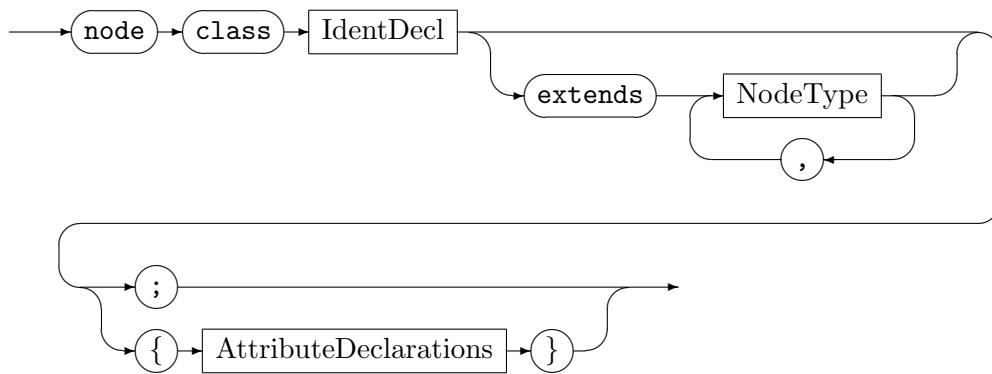


Defines a new node type. Node types can inherit from other node types defined within the same file. If the **extends** clause is omitted, *NodeType* will inherit from the built-in type Node. Optionally nodes can possess attributes.

*EdgeClass*
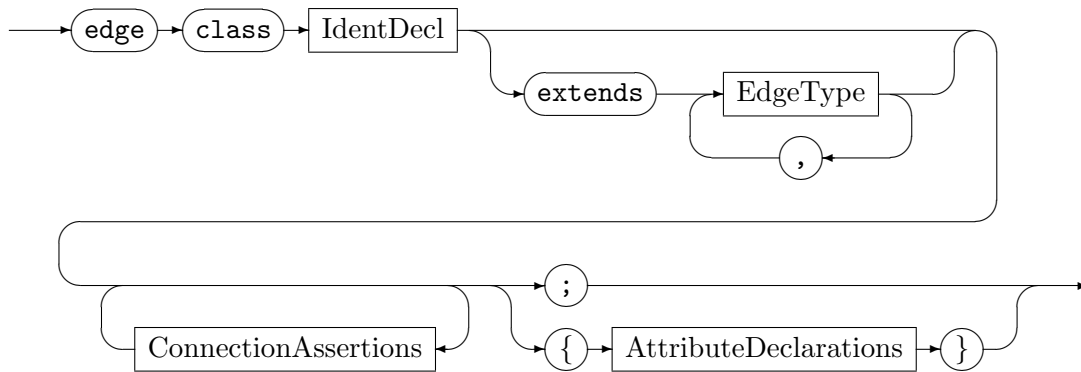


Defines a new edge type. Edge types can inherit from other edge types defined within the same file. If the **extends** clause is omitted, *EdgeType* will inherit from the built-in type Edge. Optionally edges can possess attributes. A *connection assertion* specifies that certain edge types should only 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 8.2.2, **validate**.

*ConnectionAssertions*
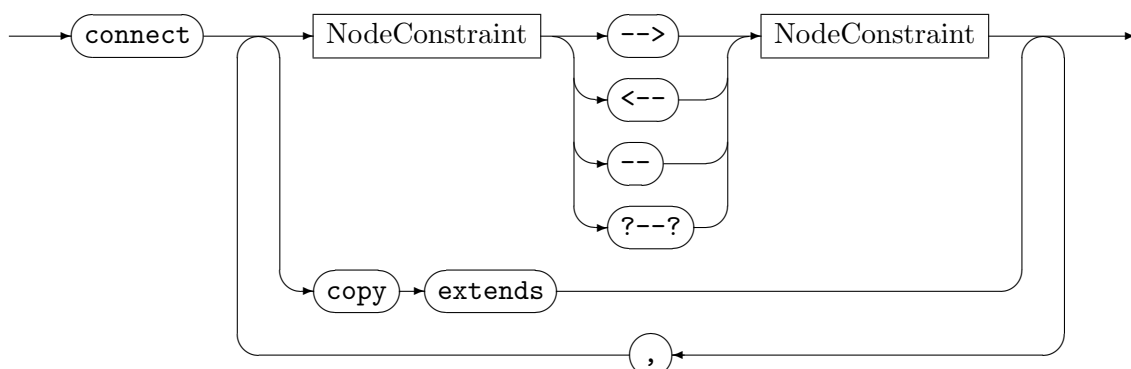
*NodeConstraint*
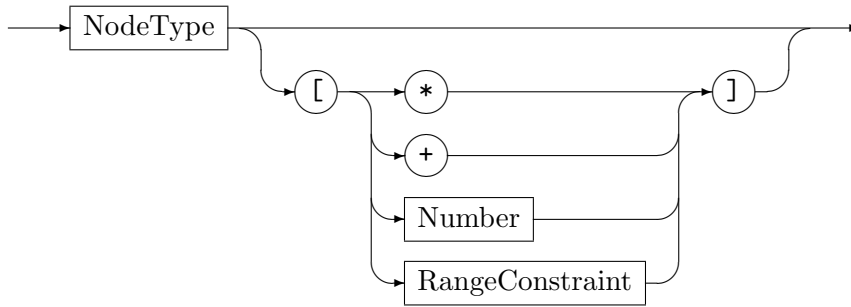


*RangeConstraint*
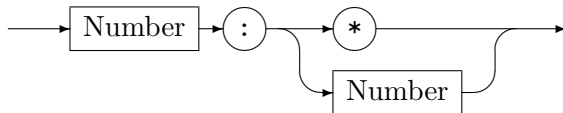


A *connection assertion* is denoted as a pair of node types in conjunction with their multiplicities. It is best understood as a simple pattern of the form(cf. 4.1.1) `:SourceNodeType -:EdgeType-> :TargetNodeType`, of which every occurrence in the graph is searched. In contrast to a real such pattern 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. Don't get surprised by matches of non-disjunct source/target types/subtypes in the case of undirected and especially arbitrary edges. Multiple connection assertions are applied by conjunction, i. e. all of them must be fulfilled. It might happen that none of the connection assertion of an `EdgeType` are matching an edge of this type in the graph. This is accepted in the case of normal validation; but as you normally want to see *only* the specified connections occuring in the graph, there is the additional mode of strict validation: if an edge is not covered by a single matching connection, validation fails. See Section 8.2.2, `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 6. Table 3.1 describes the multiplicity definitions.

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

Table 3.1: GRGEN.NET node constraint multiplicities

In order to apply the connection assertions of the supertypes to an `EdgeType`, you may use the keywords `copy extends`. The `copy extends` assertion "imports" the connection asser-

tions of the *direct* ancestors of the declaring edge. This is a purely syntactical simplification, i. e. the effect of using `copy extends` is the same as copying the connection assertions from the direct ancestors by hand.
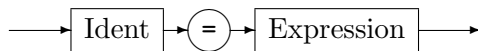
### 3.2.1 Attribute Types

*AttributeDeclaration*
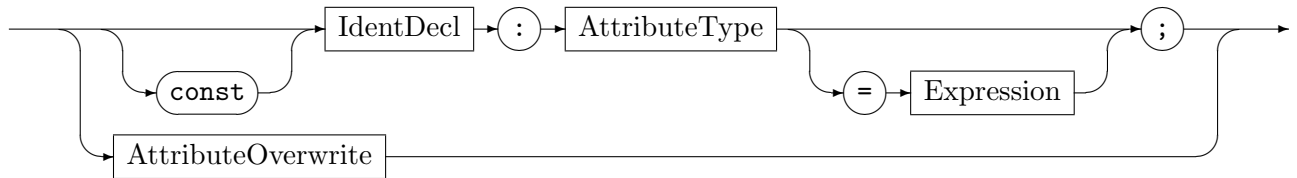


*AttributeType*
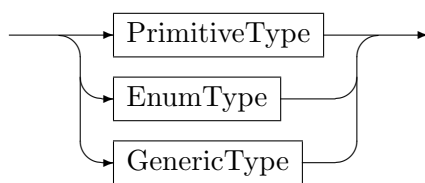


*AttributeOverwrite*



Defines a node or edge attribute. Possible types are enumeration types (`enum`) and primitive types or generic types. See Section 6.1 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 6 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 (10)**
>
> The following attribute declarations are *illegal* because of the order of evaluation of initialization values:
>
> ```
> 1 x:int = y;
> 2 y:int = 42;
> ```

# RULE SET LANGUAGE

The rule set language forms the core of GRGEN.NET. Rule files refer to zero[1] or more graph models and specify a set of rewrite rules. The rule language covers 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 5

## 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.

---

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

*Ident*, *IdentDecl*

A non-empty character sequence of arbitrary length consisting of letters, digits, or underscores. The first character must not be a digit. *Ident* may be an identifier defined in a graph model (see Section 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 4.9 for annotations of declarations.

---

**NOTE (11)**

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 6.9 for the use of type expressions.

### 4.1.1  Graphlets

*Graphlet*
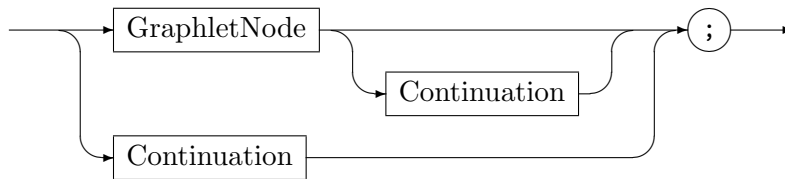


*Continuation*
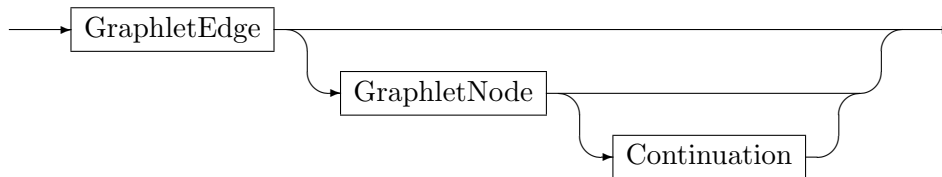


A graphlet specifies a connected subgraph. GRGEN.NET provides graphlets as a descriptive notation to define both, patterns to search for as well as the subgraphs that replace or modify matched spots in a host graph. Any graph can be specified piecewise by a set of graphlets. In Example 4, line 7, the statement `n1 --> n2` is the node identifier `n1` followed by the continuation graphlet `--> n2`.

All the graph elements of a graphlet have *name*s. 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*
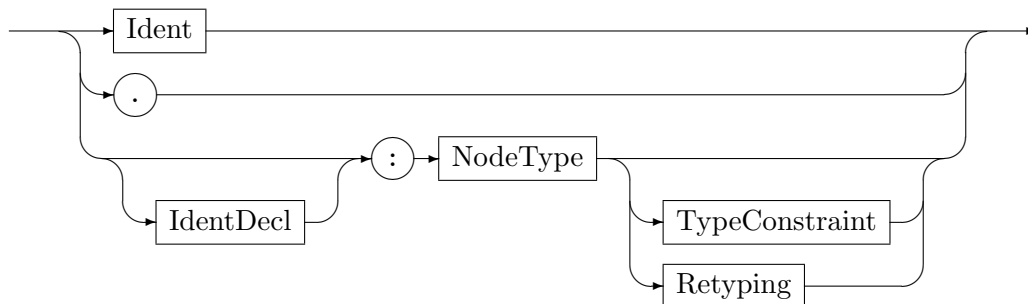


Specifies a node of type *NodeType*, maybe constrained in type with a *TypeConstraint* (see Section 4.6, *TypeConstraint*), maybe retyped with a *Reytping* (see Section 4.8, *Retyping*). The `.` is an anonymous node of the base type `Node`; remember that every node type has `Node` as super type. Type constraints are allowed in the pattern part only. Retyping is allowed in the replace/modify part only.

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

*GraphletEdge*
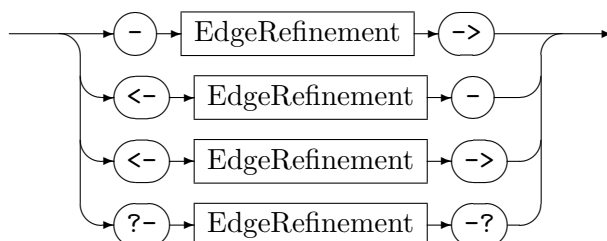
*EdgeRefinement*
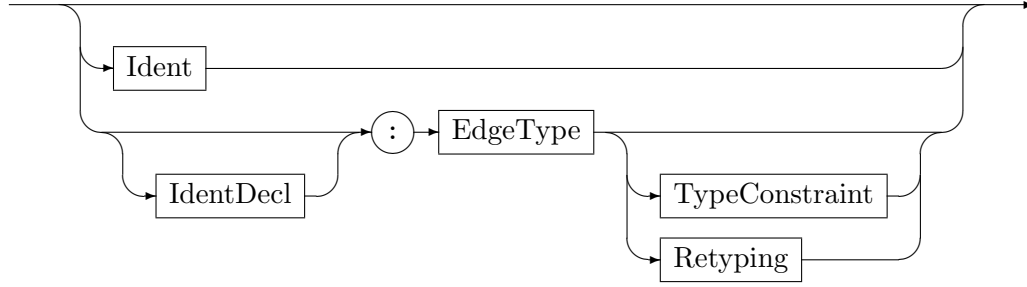


A *GraphletEdge* specifies an edge. Anonymous edges are specified by an empty *EdgeRefinement* clause, i. e. `-->`, `<--`, `<-->`, `--`, `?--?`  or `-:T->`, `<-:T-`, ... for an edge type `T`, respectively. A non-empty *EdgeRefinement* clause allows for detailed edge type specification. Type constraints are allowed in the pattern part only (see Section 4.6, *TypeConstraint*). Retyping is allowed in the replace/modify part only (see Section 4.8, *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

```
-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.

---

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

**EXAMPLE (6)**

Some attempts to specify a loop edge:

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

**NOTE (12)**

Although both, the pattern part and the replace/modify part use graphlets, there are subtle differences between them. These concern the *TypeConstraint* and *Retyping* clause plus the allowed arrow tips for edges.

## 4.2   Rules and Tests

The structure of a rule set file is as follows:

*RuleSet*
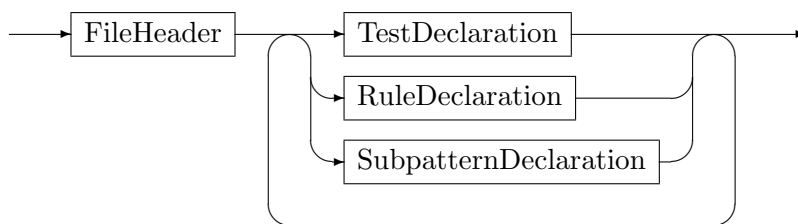


*FileHeader*
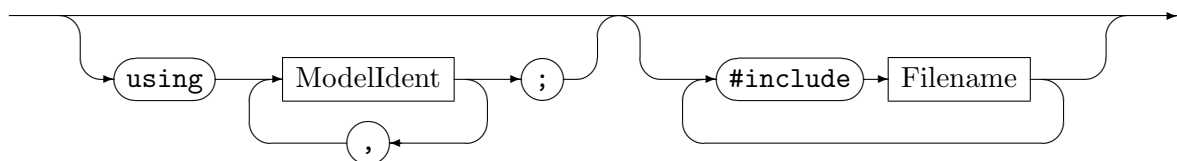


A rule set consists of the underlying graph models and several rewrite rules and tests (subpatterns will be introduced in 5.5). Additionally you may include further rule set files (without `using` directives, we prefer to suffix them with `.gri` in this case). In case of multiple graph models, GRGEN.NET 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*
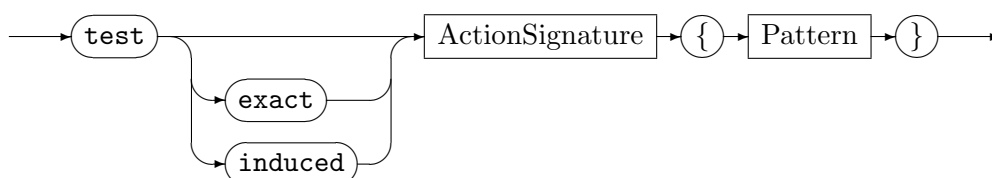
**EXAMPLE (7)**

Some graphlets:

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

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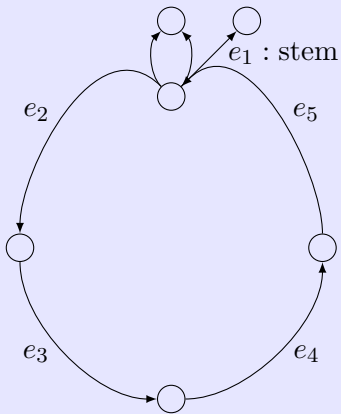
```
. <-e1:stem- n1:Node -e2:Edge-> . -e3:Edge-> .
  -e4:Edge-> . -e5:Edge-> n1;
n1 --> n2:Node;
n1 --> n2;
```

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

```
-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. |

*RuleDeclaration*
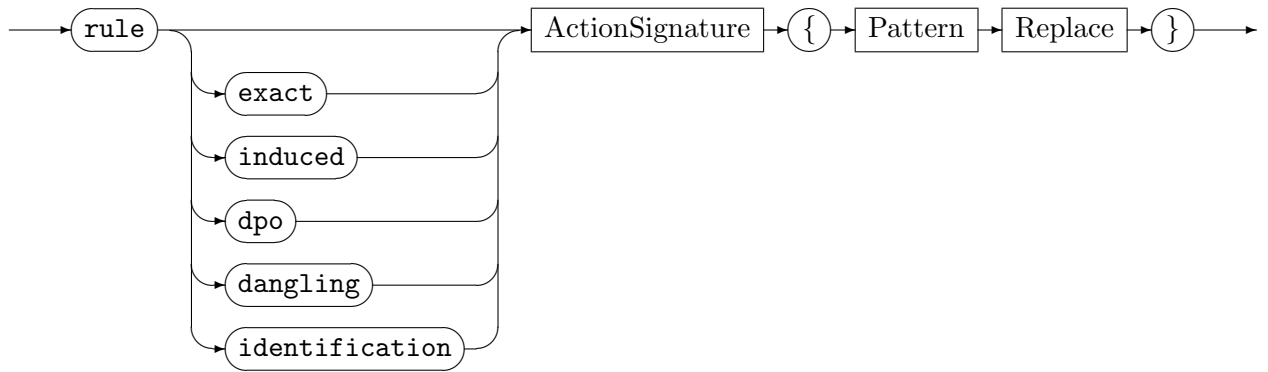


Declares a single rewrite rule such as `SomeRule`. It consists of a pattern part (see Section 4.3) in conjunction with its rewrite/modify part (see Section 4.4). A *test* has no rewrite specification. It's intended to check whether (and maybe how many times) a pattern occurs (see example 8). For an explanation 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 8.2.8.

---

*ActionSignature*
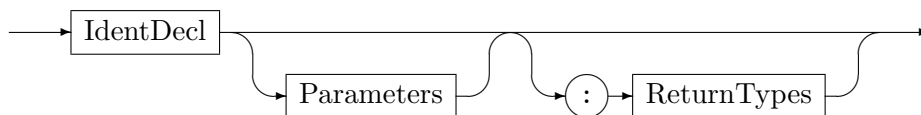


The signature sets the name of a rewrite rule to *IdentDecl* and optionally names and types of formal parameters as well as a list of return types. Parameters and return types provide users with the ability to exchange graph elements between rules, similar to parameters of procedural languages. This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

*Parameters*
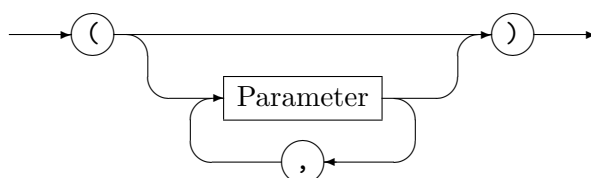
*Parameter*
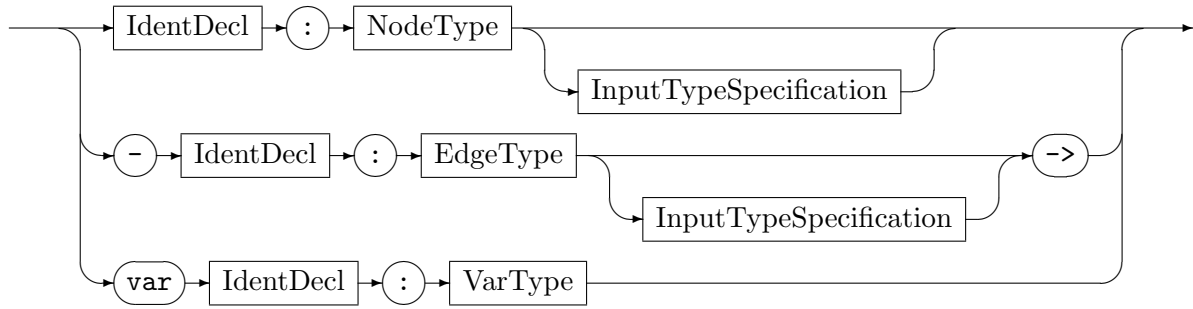


*InputTypeSpecification*
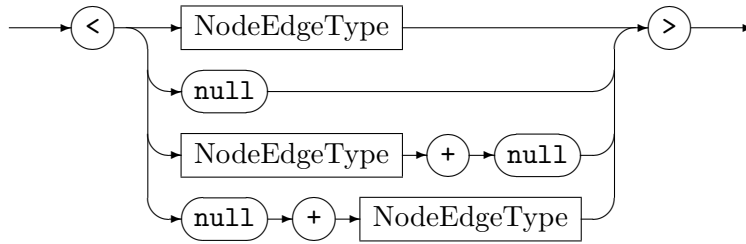


Within a rule, graph element parameters are treated as graph elements of the pattern - just predefined. But in contrast to pervious versions it is the task of the user to ensure the elements handed in satisfy the interface, i.e. parameters must not be null and must be of the type specified or a subtype 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`-keyword, VarType is one of the attribute types supported by GRGEN.NET(6.1). Please note that the effect of assigning to a var parameter in `eval` (see 4.4) is undefined (concerning the parameters as well as the argument).

*ReturnTypes*
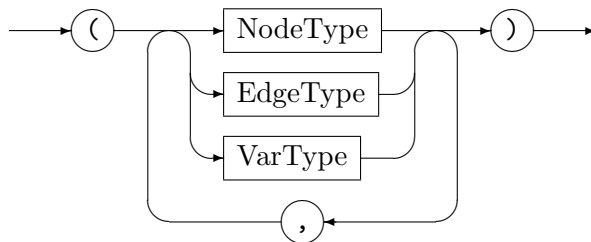


The return types specify edge and node types of graph elements that are returned by the replace/modify part. If return types are specified, the `return` statement is mandatory. Otherwise no `return` statement must occur. See also Section 4.4, `return`.

## 4.3 Pattern Part

*Pattern*
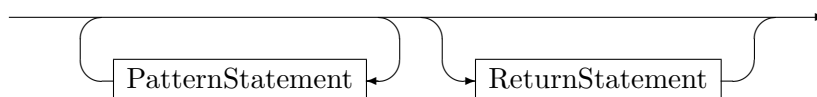
**EXAMPLE (9)**

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.

**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.

A pattern consists of zero or more pattern statements and, 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.

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 (13)**

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*
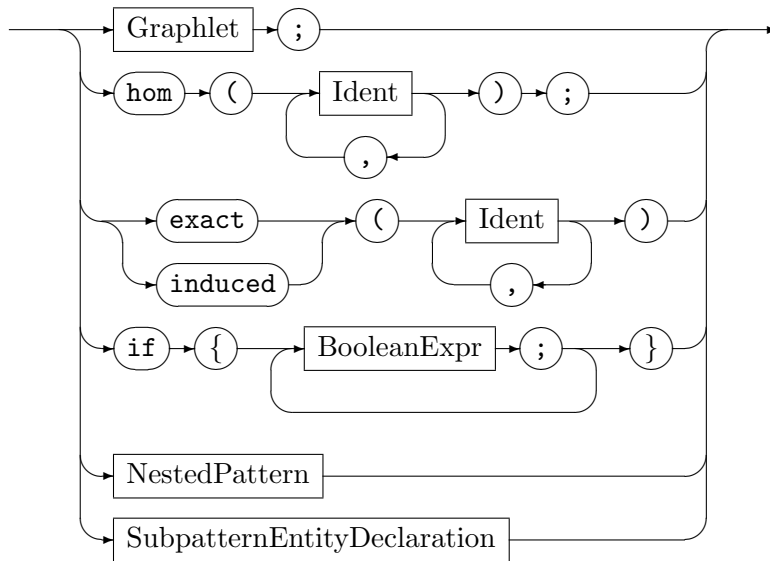


The semantics of the various pattern statements are given below:

*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` delarations 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.

*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 5.1,5.2,5.3,5.4.

*SubpatternEntityDeclaration*
> will be explained in 5.5.

Keep in mind that using type constraints or the `typeof` operator might be helpful. See Section 6.9 for further information.

*ReturnStatement*
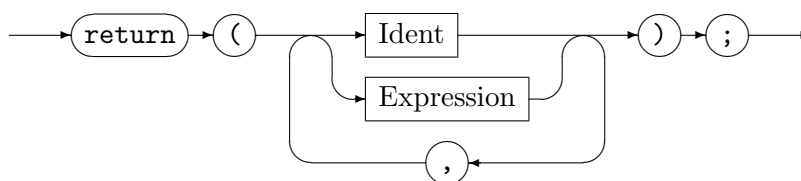


Returned graph elements (given by their name) and value entities (given by an expression computing them) must appear in the same order as defined by the return types in the signature (see Section 4.2, `ActionSignature`). Their types must be compatible to the return types specified.

> **NOTE (14)**
>
> 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:
>
> ```
> rule numberNode(var id:int) : (int)
> {
>   n:NodeWithIntId;
>   if { n.id == 0; }
>
>   modify {
>     eval {
>       n.id = id;
>     }
>     return (id + 1);
>   }
> }
> ```

## 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
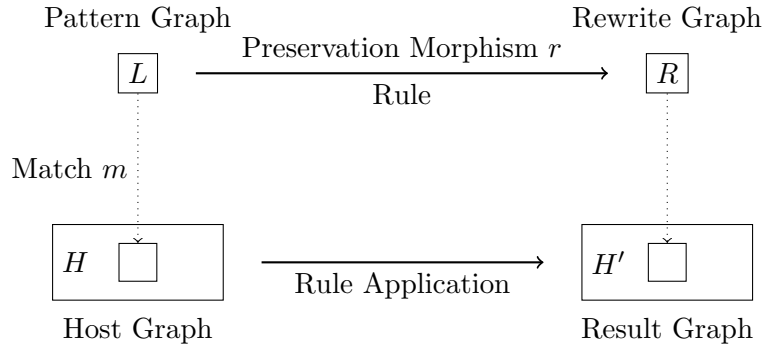


Figure 4.1: Process of Graph Transformation

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).

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

---

**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*
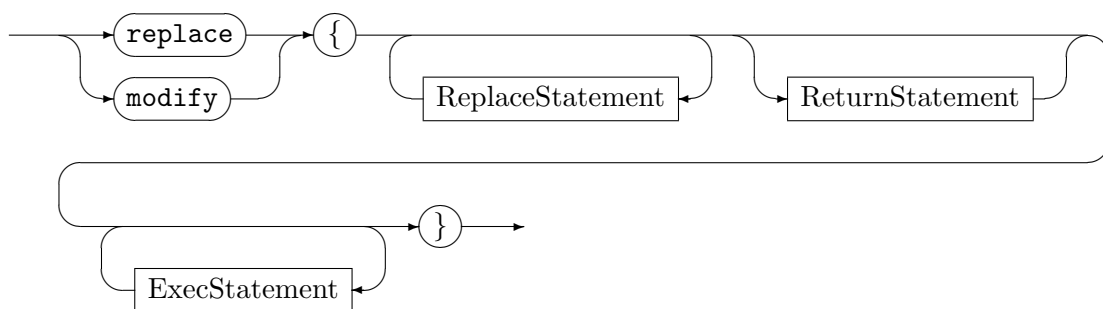


Selects whether the replace mode or the modify mode is used. Several replace statements describe the transformation from the pattern subgraph to the destination subgraph.

*ReplaceStatement*
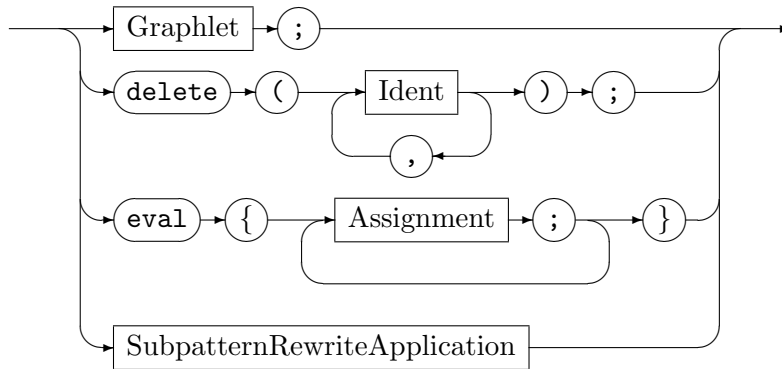


The semantics of the various replace statements are given below:

*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.

*SubpatternRewriteApplication*

> will be explained in 5.7.

*Assignment*
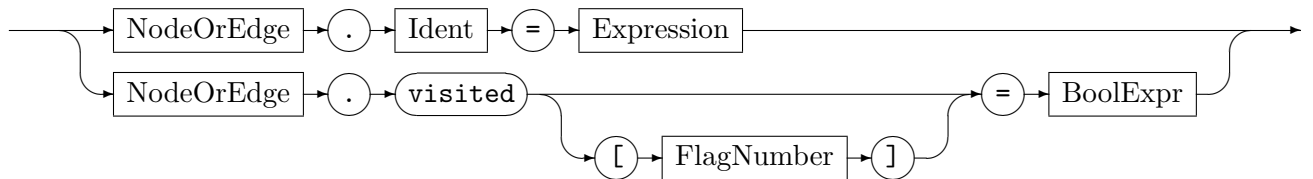


Several evaluation parts are allowed within the replace part. Multiple evaluation statements will be internally concatenated, preserving their order. Evaluation statements have imperative semantics. In particular, GRGEN.NET does not care about data dependencies. Evaluation takes place before any graph 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 , even for entities of `set<K>` and `map<K,V>` or `string` type. The only exception is the type `object`, there is used. 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 allocate7.4/10.3 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.)

**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 choosen 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 restricitive 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
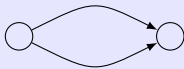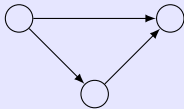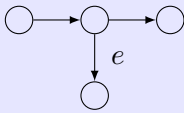
**NOTE (15)**

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

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

## 4.6 Static Type Constraint

Node/edge declaration with type constraint.

*TypeConstraint*
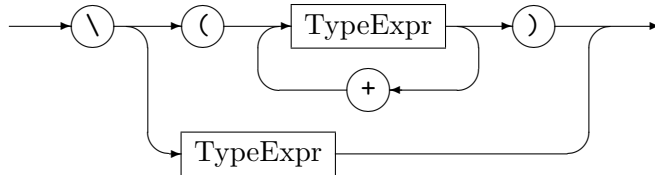


A type constraint is used to exclude parts of the type hierarchy. The operator + is used to create a union of its operand types. So the following pattern statements are identical:

| | |
|---|---|
| `x:T \ (T1 + ` $\cdots$ ` + T`$n$`);` | `x:T;`<br>`if {!(`*typeof*`(x) >= T1) && ` $\cdots$<br>`&& !(`*typeof*`(x) >= T`$n$`)}` |

---

**EXAMPLE (14)**
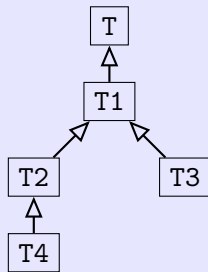


The type constraint `T\(T2+T3)` applied to the type hierarchy on the left side yields only the types `T` and `T1` as valid.

---

## 4.7 Exact Dynamic Type

*Type*
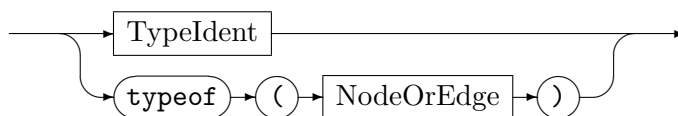


The type of a graph element may be given by a type identifier, or a typeof denoting the exact dynamic type of a matched graph element. The element declaration `el:typeof(x)` introduces a graph element of the type the host graph element `x` is actually bound to.

**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.

## 4.8 Retyping

In addition to graph rewriting, GRGEN.NETallows 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. Syntacically it is specified by giving the original node enclosed in left and right angles.

*Retyping*
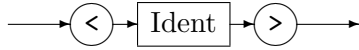


| Pattern (LHS) | Replace (RHS) | $r : L \longrightarrow R$ | Meaning |
|---|---|---|---|
| `x:N1;` | `y:N2<x>;` | $r : \text{lhs}.x \mapsto \text{rhs}.x$ | Preserve `x`, then retype `x` from `N1` to `N2` and bind name `y` to retyped version of `x`. |
| `e:E1;` | `f:E2<e>;` | $r : \text{lhs}.e \mapsto \text{rhs}.e$ | 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 ore 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      replace {
4        x_rt:Metropolies<x> -y_rt:Highway<y>->;
5      }
6  }
```

## 4.9 Annotations

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

*IdentDecl*
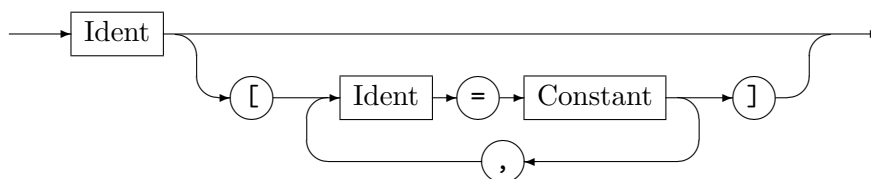


Although you can use any key-value pairs between the brackets, only the identifier has an effect so far.

| 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 (17)**

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`.

## 4.10  Imperative Statements

*ExecStatement*
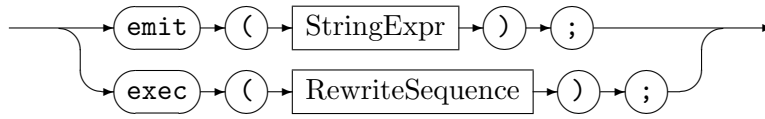


The statements `emit` and `exec` enhance the declarative rewrite part by imperative clauses. That means, these statements are executed in the same order as they appear within the rule. The execution statements take place after all the rewrite operations are done, i.e. they operate on the modified host 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. See Chapter 7 for a description of graph rewrite sequences.

*Text Output*

The `emit` statement prints a string to the currently associated output stream (default is `stdout`). See Chapter 6 for a description of string expressions. With subpatterns used in the pattern, there are `emitpre` and `emitpost` statements available. The `emitpre` is executed before emitting is taking place in the subpatterns. The `emitpost` is executed after emitting in the used subpatterns, is is a synonym for `emit`.

---

**EXAMPLE (18)**

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

# CHAPTER 5

# NESTED AND SUBPATTERNS

The extension of the rule set language described in the previous chapter 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 (19)**

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

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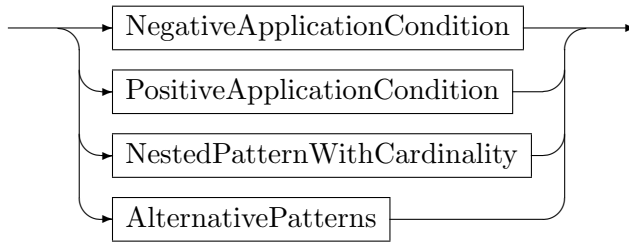


## 5.1   Negative Application Condition (NAC)

*NegativeApplicationCondition*
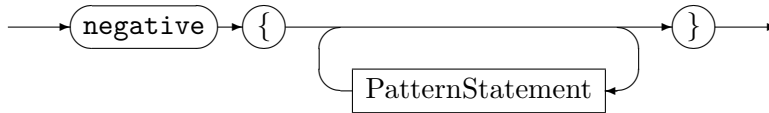


With negative application conditions (keyword `negative`) we can specify graph patterns which forbid the application of a rule if any of them is present in the host graph (cf. [Sza05]). NACs possess a scope of their own, i.e. names defined within a NAC do not exist outside the NAC. Identifiers from surrounding scopes must not be redefined. If they are not explicitely 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 (20)**

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 (21)**

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 (22)**

A fabricated example using parallel as well as nested `negative`s:

```
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 }
```

## 5.2   Positive Application Condition (PAC)

*PositiveApplicationCondition*
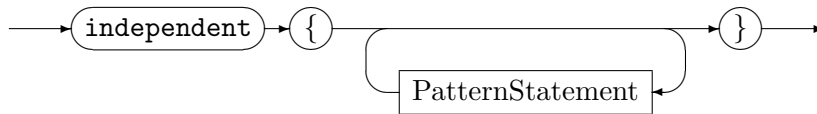


With positive application conditions (keyword `independent`) we can specify graph patterns which, in contrast to negative application conditions, must be present in the host graph to cause the matching of the enclosing pattern to succeed. Together with NACs they share the property of opening a scope, 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 (23)**

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  }
```

## 5.3   Pattern Cardinality

*NestedPatternWithCardinality*
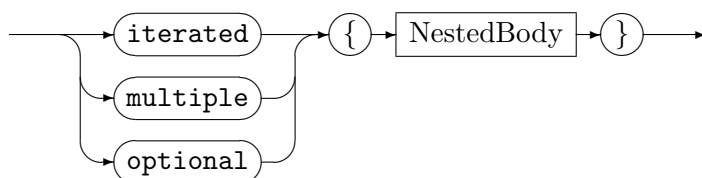
*NestedBody*
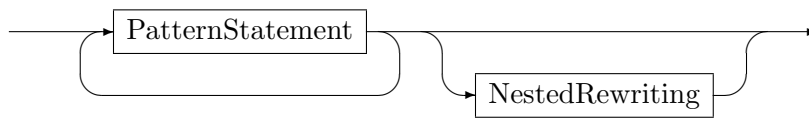


The blocks allow to specify how often the nested pattern – opening a scope – is to be matched. Matching will be carried out eagerly, i.e. if the construct is not limiting the number of matches and a further match is possible it will be done. (The nested body will be explained in Section 5.6.)

### 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 (24)**

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

### 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 (25)**

```
test oneOrMoreMethods
{
  c:Class;
  multiple {
    c --> m:Method;
  }
}
```

### The optional block

is working like the iterated block, but matches the contained subpattern at most once, further occurences 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.

```
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  }
```

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

## 5.4 Alternative Patterns

*AlternativePatterns*
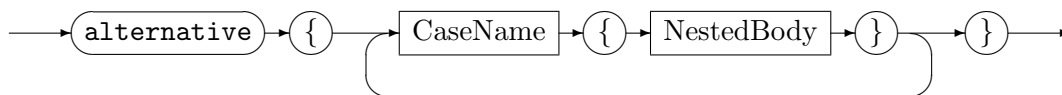


    With the alternative block you can specify several nested alternative patterns. One of them must get matched for the matching of the alternative (and thus its directly nesting pattern) to succeed, and only one of them is matched per match of the alternative / overall pattern. The order of matching the 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 can not 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.

```
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  }
```

```
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 }
```

## 5.5   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*
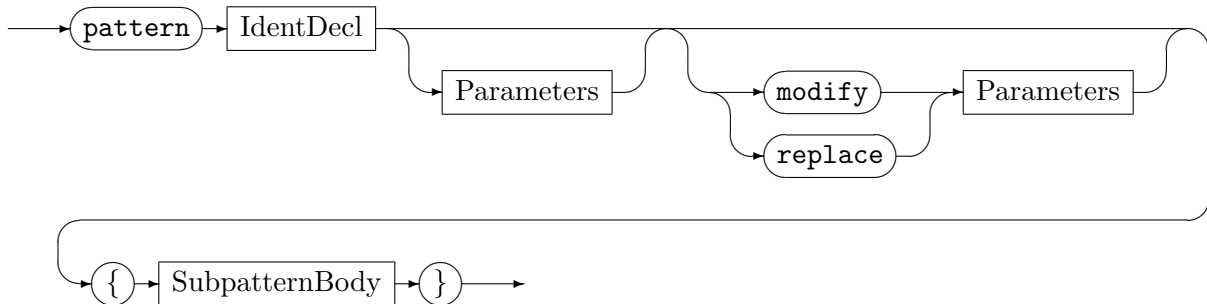


*SubpatternBody*
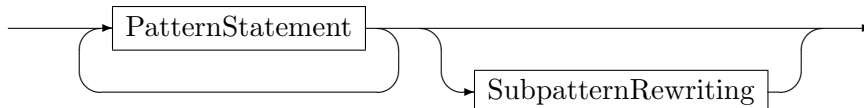


Subpattern declarations define a subpattern type denoting the specified shape in global namespace; the parameters specify some context elements the pattern may refer to, but which are not part of the pattern itself. So they are only syntactically the same as test/rule-parameters (which are members of the pattern part), a further difference is the lack of Re-turnTypes, 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 elements 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 5.7.)

*SubpatternEntityDeclaration*

```
⟶[ Ident ]⟶(:)⟶[ SubpatternType ]⟶(()⟶())⟶(;)⟶
```

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.

---

**EXAMPLE (29)**

```
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  }
```

---

### 5.5.1   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 (30)**

```
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 assinment 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 (31)**

```
 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 chech 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 altready 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(todo:indexed) in the graph) giving you a very powerful and flexible notation to capture large, complex patterns built up in a stuctured way from simple, connected pieces (as e.g. abstract syntax trees of programming languages).

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

## 5.6 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*
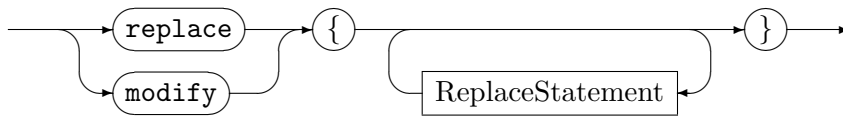


Syntactically the rewrite is specified by a modify or replace clause nested directly within the scope of each nested pattern; in addition to the rewrite clause nested within the top level pattern, which must be present even if the top level pattern is empty. Semantically for every instance of a pattern 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 neither `exec` statements nor `return` statements are available as in the top level rewrite part of a rule.

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 not available for deletion inside the nested rewrite; only the rewrite part of the pattern in which they are declared is allowed to delete them. They are available for retyping though, but only in case the retyping is unambigous – it must be guaranteed that only one retyping can occur. So only the rewrites of alternative cases and optional patterns may contain retypings of elements not declared in the alternative case or optional pattern (in contrast to iterated and multiple pattern rewrites).

**EXAMPLE (33)**

```
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 (34)**

```
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 }
```

## 5.7   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 withing the rewrite part of the pattern containing the subpattern entity declaration requesting the rewrite to be actually applied.

*SubpatternRewriting*
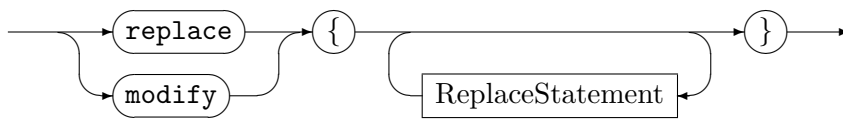


The subpattern rewriting specifications within the subpattern declaration looks like a nested rewriting specification, but additionally there may be rewrite parameters given in the subpattern header (cf. 5.5) which can be referenced in the rewrite body. (Most elements can be handed in with normal parametes, but elements created in the rewrite part of the user of the subpattern can only be handed in at rewrite time.)

*SubpatternRewriteApplication*
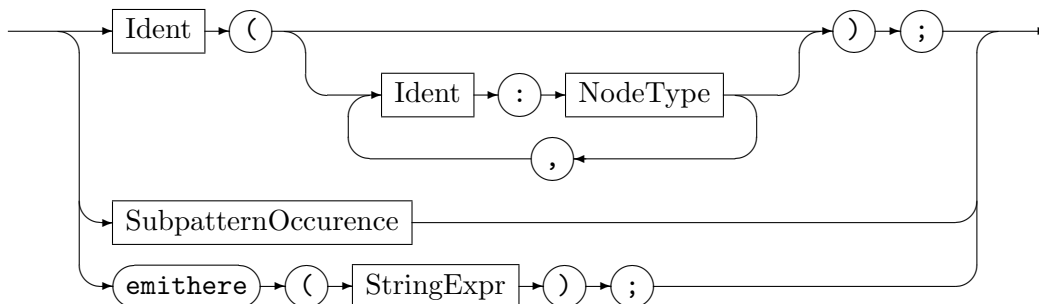


The SubpatternRewriteApplication is part of a ReplaceStatement. The subpattern rewrite application is given within the rewrite part of the pattern containing the subpattern entity declaration, in call notation on the declared subpattern identifier. The `emithere`-statements are executed before the emit statements, and in the order in between the subpattern rewrite applications they are specified syntactically.

**EXAMPLE (35)**

```
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 (36)**

```
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 }
```

Reversing the direction of the edges in an iterated path.

**EXAMPLE (37)**

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

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

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

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

  replace {
    from; to;
  }
}
```

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

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

### 5.7.1 Deletion and Preservation of Subpatterns

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

*SubpatternOccurence*
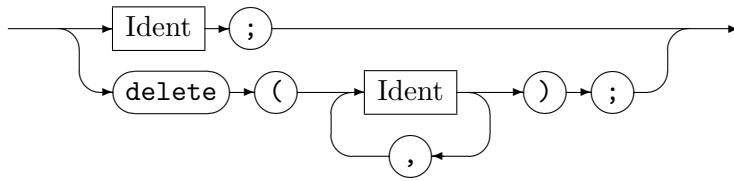


In modify mode, they are kept by default, but deleted if the name of the declared sub-pattern entity is mentioned within a delete statement. In replace mode, they are deleted by default, but kept if the name of the declared subpattern entity is mentioned (using occurence, same as with nodes or edges).

**EXAMPLE (38)**

```
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 }
```

## 5.8  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). Table 5.1 lists the corresponding (/equivalent) language constructs; Example 5.8 is a version of the introductionary example 5 modified to use the new syntax.

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

Table 5.1: Map of nested patterns in keyword syntax to regular expression syntax

**EXAMPLE (39)**

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

# TYPES AND EXPRESSIONS

## 6.1 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 6.1 and the built-in generic types in Table 6.2. 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 6.1: 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 |
| `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 |

Table 6.2: GRGEN.NET built-in generic types

Table 6.3 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.

| 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 6.3: GRGEN.NET type casts

According to table 6.3 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.NETprocessing (and the im/exporters can't handle them), but they can be used on the API level.

**EXAMPLE (40)**

- 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 (17)**

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.

## 6.2  Expressions

GRGEN.NETsupports 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*
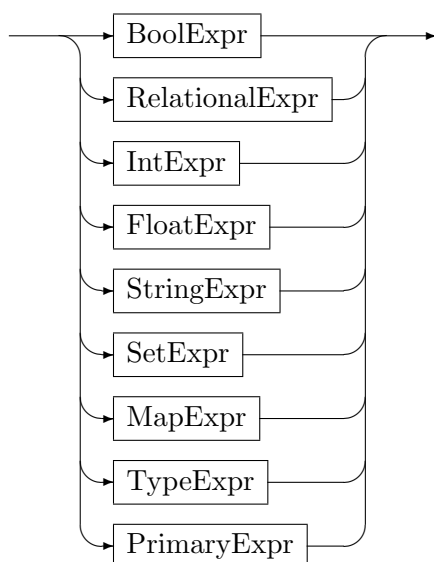
## 6.3  Boolean Expressions

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

*BoolExpr*
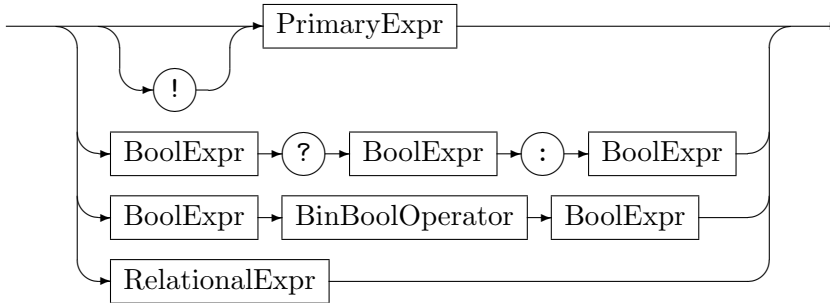


The unary `!` operator negates a Boolean. The binary *BinBoolOperator* is one of the operators in Table 6.4. The ternary `?` operator is a simple if-then-else: If the first *BoolExpr* is evaluated

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

Table 6.4: Binary Boolean operators, in ascending order of precedence

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

## 6.4  Relational Expressions

The relational expressions compare enitites of different kinds, mapping them to the type boolean. The bind stronger than the boolean expressions but weaker than all the other non-boolean expressions.

*RelationalExpr*
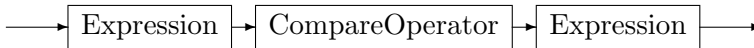


The *CompareOperator* is one of the following operators:

$$< \quad <= \quad == \quad != \quad >= \quad >$$

Their semantics are type dependent.

For arithmetic expressions on `int` and `float` or `double` types the semantics is given by Table 6.5 (by implicit casting they can also by used with all enum types).

`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 the tell whether the references are the same, thus the objects identical.

For set and map expressions, table 6.6 describes the semantics of the compare operators.

For type expressions the semantics of compare operators are given by table 6.7, the rule to remember is: types grow larger with extension/refinement. An example is given in 6.9.

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

Table 6.5: Compare operators on arithmetic expressions

| | |
|---|---|
| `A == B` | True, iff $A$ and $B$ are identical. |
| `A != B` | True, iff $A$ and $B$ are not identical. |
| `A <  B` | True, iff $A$ is a subset/map of $B$, but $A$ and $B$ are not identical. |
| `A >  B` | True, iff $A$ is a superset/map of $B$, but $A$ and $B$ are not identical. |
| `A <= B` | True, iff $A$ is a subset/map of $B$ or $A$ and $B$ are identical. |
| `A >= B` | True, iff $A$ is a superset/map of $B$ or $A$ and $B$ are identical. |

Table 6.6: Compare operators on set/map expressions

**NOTE (18)**

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

**NOTE (19)**

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

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

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

## 6.5   Arithmetic and Bitwise Expressions

The arithmetic and bitwise expressions combinde 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).

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

Table 6.7: Compare operators on type expressions

*IntExpr*
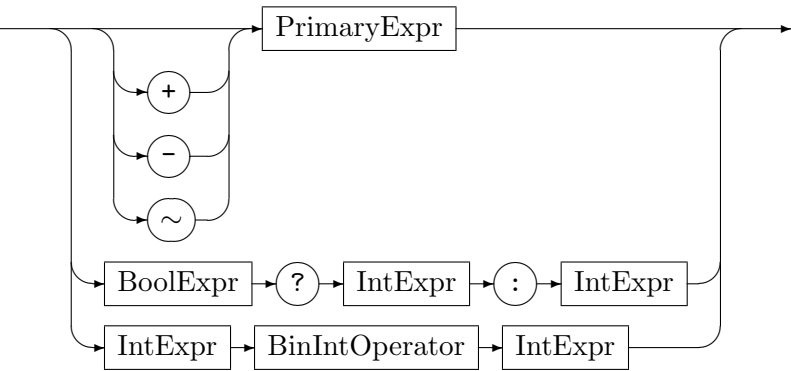


The $\sim$ operator is the bitwise complement. That means every bit of an integer value will be flipped. The `?` operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *IntExpr*, otherwise it returns the second *IntExpr*. The *BinIntOperator* is one of the operators in Table 6.8.

| | |
|---|---|
| `^`<br>`&`<br>`\|` | Bitwise XOR, AND and OR |
| `<<`<br>`>>`<br>`>>>` | Bitwise shift left, bitwise shift right and<br>bitwise shift right preserving the sign |
| `+`<br>`-` | Addition and subtraction |
| `*`<br>`/`<br>`%` | Multiplication, integer division, and modulo |

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

*FloatExpr*
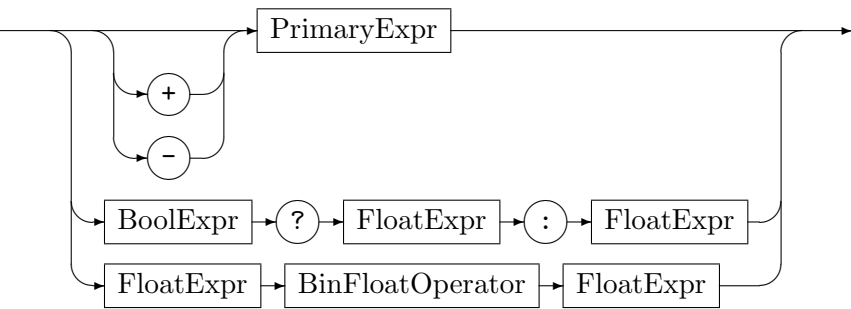


The `?` operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *FloatExpr*, otherwise it returns the second *FloatExpr*. The *BinFloatOperator* is one of the operators in Table 6.9.

**NOTE (20)**

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

| | |
|---|---|
| +<br>− | Addition and subtraction |
| *<br>/<br>% | Multiplication, division and modulo |

Table 6.9: Binary float operators, in ascending order of precedence

## 6.6  String Expressions

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

*StringExpr*
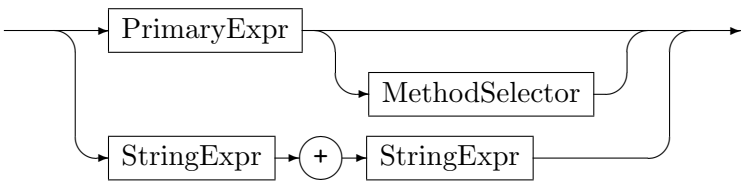


The operator + concatenates two strings. There are several operations on strings available in method call notation (MethodSelector), these are

`.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

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

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

**EXAMPLE (41)**

```
For n.str == "foo bar foo" the operations yield
n.str.length()==11
n.str.indexOf("foo")==0
n.str.lastIndexOf("foo")==8
n.str(4,3)=="bar"
n.str(4,3,"foo")=="foo foo foo"
```

## 6.7  Set Expression

Set expressions consist of the known mathematical set operations and size counting.

*SetExpr*
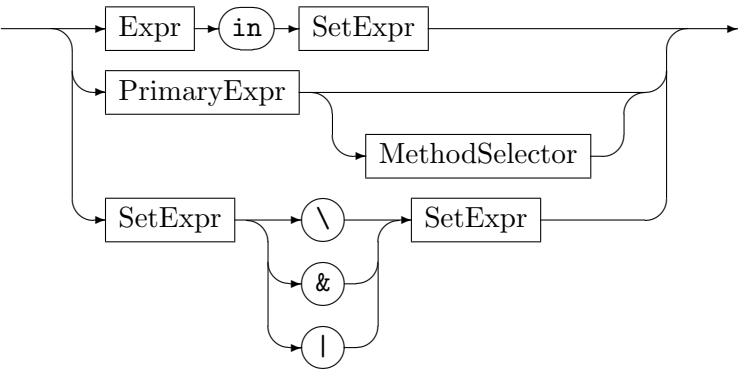


| | |
|---|---|
| \| | Set union (contained in resulting set as soon as contained in one of the sets) |
| & | Set intersection (contained in resulting set only if contained in both of the sets) |
| \\ | Set difference (contained in resulting set iff contained in left but not right set) |

Table 6.10: Binary set operators, in ascending order of precedence

The binary set operators require the left and right operands to be of identical type `set<T>`. The operator `x in s` denotes set membership $x \in s$, returning whether the set contains the given element, as `boolean`. Furthermore there is one operations on sets available in method call notation (MethodSelector):

*.size()*
> returns the number of elements in the set, as `int`

> **NOTE (21)**
>
> 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. 6.10).
>
> ```
> 1 s | { "foo" }
> 2 s \ { n.a }
> ```
>
> Used in this way they get internally optimized to set addition and removal.

## 6.8   Map Expression

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

*MapExpr*
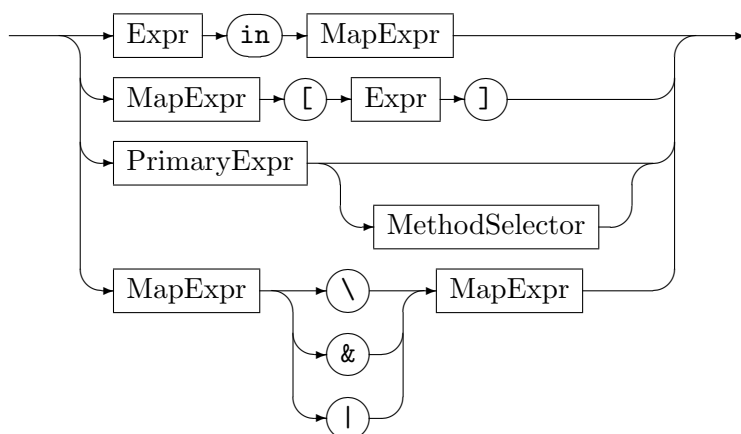


| | Map union: returns new map with elements which are in at least one of the maps, |
|---|---|
| **\|** | with the value of map2 taking precedence |
| **&** | Map intersection: returns new map with elements which are in both maps, |
| | with the value of map2 taking precedence |
| **\\** | Map difference: returns new map with elements from map1 |
| | without the elements with a key contained in map2 |

Table 6.11: 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<T>`, too. The operator `x in s` denotes map domain membership $x \in dom(s)$, 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>`

**NOTE (22)**

The declarative rule language comes without the imperative map `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. 6.10).

```
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.

## 6.9   Type Expressions

*TypeExpr*
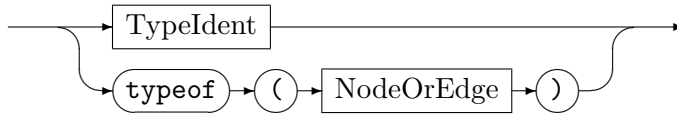


A type expression identifies a type (and—in terms of matching—also its subtypes). A type expression is either a type identifier itself or the type of a graph element. The type expression `typeof(x)` stands for the type of the host graph element `x` is actually bound to.

---

**EXAMPLE (42)**
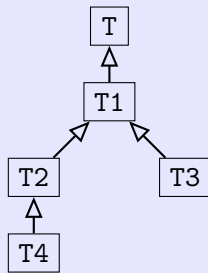
The expression `typeof(x)<=T2` applied to the type hierarchy on the left side yields `true` if `x` is a graph element of type `T` or `T1` or `T2`. The expression `typeof(x)>T2` only yields `true` for `x` being a graph element of type `T4`. The expression `T1<T3` always yields `true`.



---

## 6.10   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*
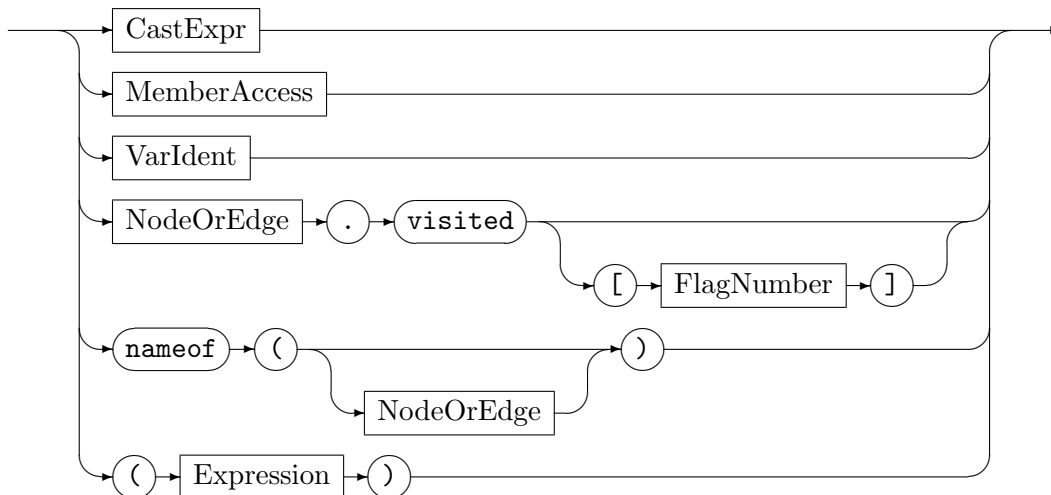


The `visited` query returns the status of the visited flag of the given number for the given graph element as `boolean`. If no flag number is given, the default number for the first visited flag of 0 is used. The visited flags are written in the assignments of the eval statements (see section 4.4). Make sure to allocate7.4/10.3 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 of the given graph element as `string`; graphs elements at the API level bear no name, the operator can only be used with Shell-graphs. If no graph element is given, the name of the graph is returned.

*CastExpr*

$$\longrightarrow \bigcirc\!\!( \rightarrow \boxed{\text{PrimitiveType}} \rightarrow \bigcirc\!\!) \rightarrow \boxed{\text{PrimaryExpr}} \longrightarrow$$

*MemberAccess*

$$\longrightarrow \boxed{\text{NodeOrEdge}} \rightarrow \bigcirc\!\!. \rightarrow \boxed{\text{Ident}} \longrightarrow$$
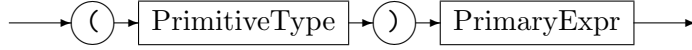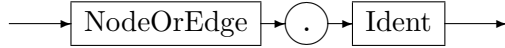
The cast expression returns the original value casted to the new prefixed type. The member access `n.a` returns the value of the attribute `a` of the graph element `n`.

*Literal*

$$\longrightarrow \boxed{\text{Constant}} \longrightarrow$$
$$\boxed{\text{SetConstructor}}$$
$$\boxed{\text{MapConstructor}}$$

*Constant*
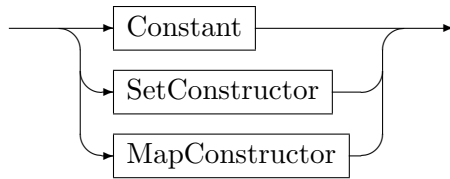
$$\longrightarrow \boxed{\text{EnumLit}} \longrightarrow$$
$$\boxed{\text{Number}}$$
$$\boxed{\text{HexNum}}$$
$$\boxed{\text{DoubleNum}}$$
$$\boxed{\text{FloatNum}}$$
$$\boxed{\text{QuotedText}}$$
$$\boxed{\text{BoolLit}}$$
$$\boxed{\text{NullLit}}$$
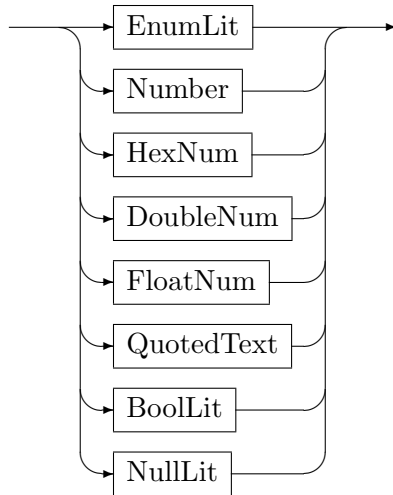
The Constants are:

*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.

*BoolLit*

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

*NullLit*

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

*SetConstructor*



*MapConstructor*



The set/map constructors are constant if only primitive type literals, enum literals, or constant expressions are used; this is required for set/map initializations in the model. They are non-constant if they contain member accesses, which is the common case if used in rules. If the type of the set/map is given before the constructor, the elements given in the type constructor are casted to the specified member types if needed and possible. Without the type prefix all elements given in the constructor must be of the same type.

**EXAMPLE (43)**

Some examples of literals:

```
1  Apple::ToffeeApple // an enum literal
2  42 // an integer number in decimal notation
3  0xdeadbeef // an integer number in hexadecimal notation
4  3.14159 // a double number
5  3.14159f // a float number
6  "ve␣rule␣and␣Own␣ze␣vorld" // a text literal
7  true // a bool literal
8  null // the object literal
9  { "foo", "bar" } // a constant set<string> constructor
10 map<string,int>{ (n.strVal+m.strVal)->(m.intVal+n.intVal), intVal->strVal, "fool"->42 } // a
       non-constant map constructor with type prefix
```

## 6.11   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 6.12: All operators, in ascending order of precedence

CHAPTER 7

# RULE APPLICATION 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 programms.

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

Graph rewrite sequences are 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 return values of rewrite terms and to pass 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 implicitely declared upon first assignment to their name, sequence local variables are explicitely 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. 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).

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 term 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 44).

**EXAMPLE (44)**

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` mached or not to variable `a`.

## 7.1 Rewrite sequence (logical connectives)

*RewriteSequence*
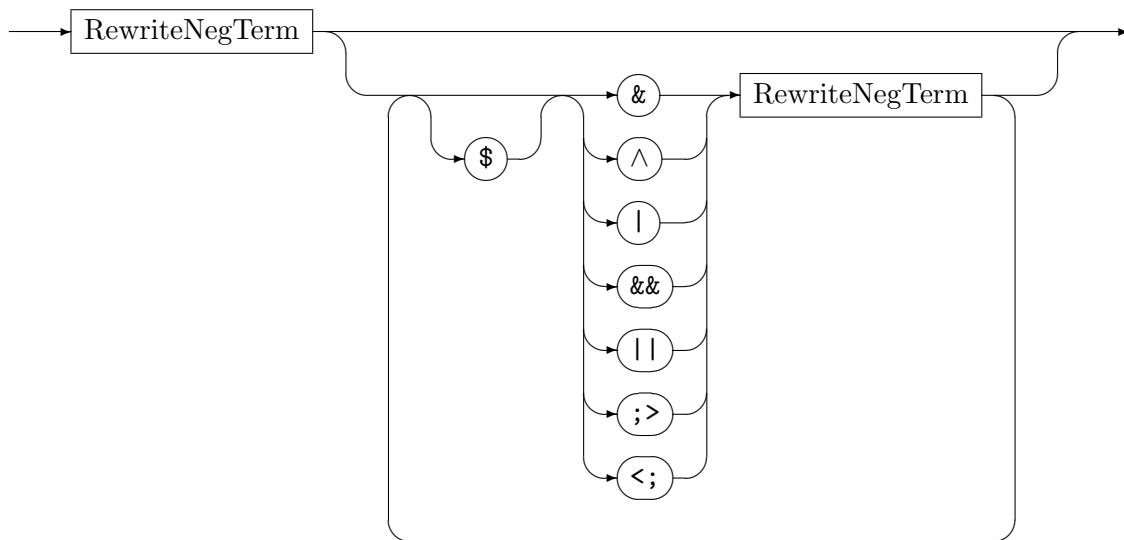


*RewriteNegTerm*
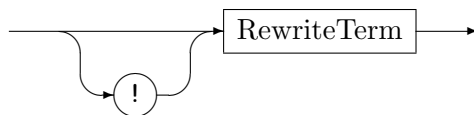


A graph rewrite sequence consists of several rewrite terms linked by operators. Table 7.1 gives the priorities and semantics of the operators, priorities in ascending order. The modifier `$` flags the following operator to randomly execute the left or the right operand first; usually operands are executed / evaluated from left to right if not altered by bracketing (left-associative). In contrast the sequences `s`, `t`, `u` in `s $<op> t $<op> u` are executed / evaluated in arbitrary order. Then-Left and Then-Right are of the same priority and their execution order can't be altered by the `$` operator prefix.

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

Table 7.1: Semantics and priorities of rewrite sequence operators

## 7.2 Rewrite term (loops)

*RewriteTerm*
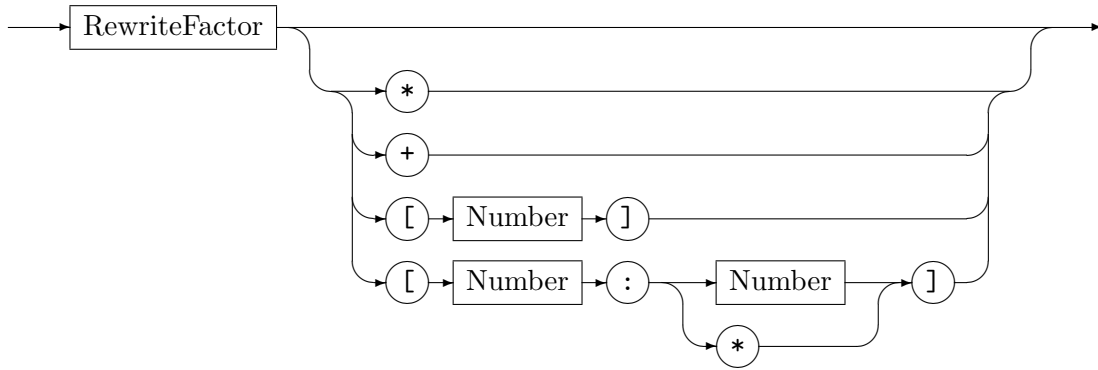


A rewrite term consists of a rewrite factor which can be executed multiple times. The star (*) executes a sequence repeatedly as long as its execution does not fail. Such a sequence always returns `true`. A sequence s+ is equivalent to s && s*. The brackets ([m]) execute a sequence repeatedly as long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

## 7.3   Rewrite factor (rule application and variable handling)

*RewriteFactor*
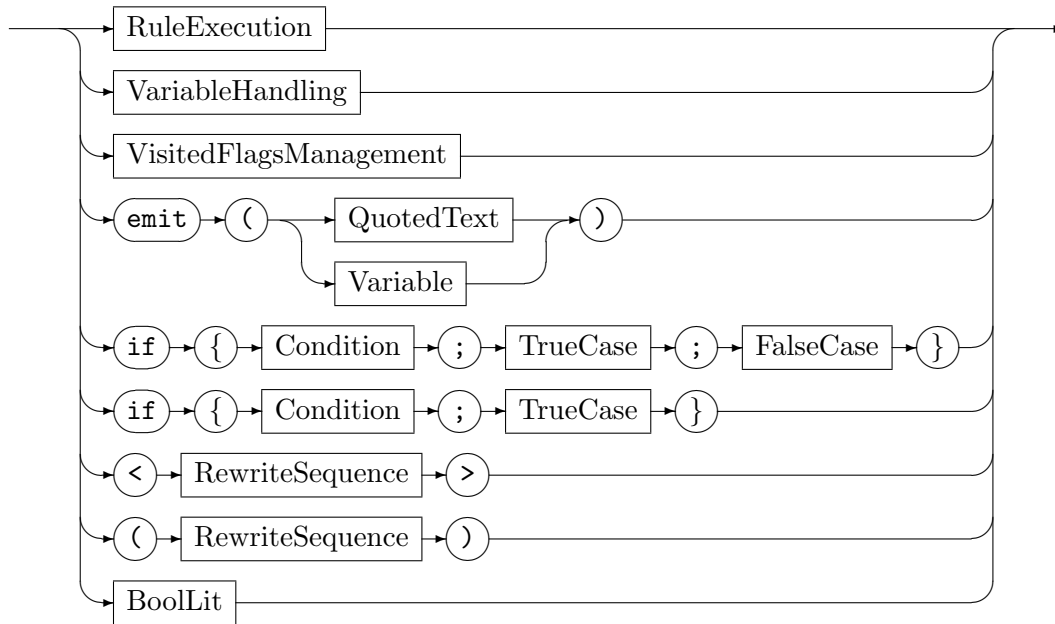


Rewrite factors are the building blocks of graph rewrite sequences. They mainly act as a rule application or variable assignment. Further on, there are commands for visited flags management. The `emit` sequence writes a double quoted string or the value of a variable to the emit target (stdout as default, or a file specified with the shell command `redirect emit`). In addition, they comprise the condition execution statement `if`, which executes the condition xgrs, 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. 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. Forcing execution orders can be achived by parentheses, an boolean literal `true`/`false` comes in handy if a sequence is to be evaluated but its result must be a predefined value.

*RuleExecution*

*Rule*
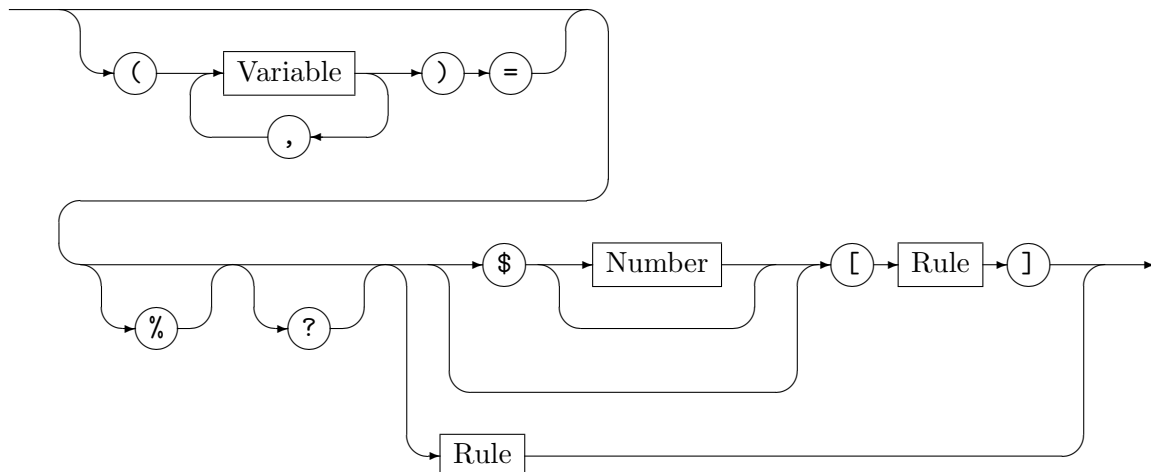
RuleIdent ( Variable , )
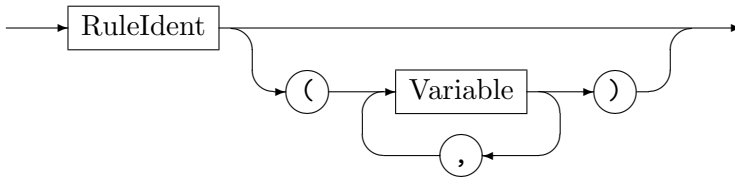
The *RuleExecution* clause applies a single rule or test. In case of a rule, the first found pattern match will be rewritten. Variables and named graph elements from the enclosing rule can be passed. The returned graph elements can be assigned to variables again. The operator `?` switches the rule to 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 8) it dumps the matched graph elements to `stdout`; in debug-mode (see Section 8.3) it acts as a break point; 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 sqare braces (`[]`) introduce a special kind of multiple rule application: Every pattern match produced by the will be rewritten. Attention: Its semantics is **not** equal to `Rule*`. Instead this operator collects all the matches first before starting to rewrite. In particular the semantics is unsafe, i.e. one needs to avoid deleting a graph element that is bound by another match. If *Rule* returns values, the values of *one* of the executed rules will be returned. The random-match-selector `$n` searches for all matches and then randomly selects `n` of them to be rewritten, with `$[r1]` being equivalent to `$1[r1]`.

*VariableHandling*

Variable : Type = Variable
( RewriteSequence )
Literal
Variable . Attribute
@ ( NameString )
Variable . Attribute = Variable
Variable
def ( Variable , )
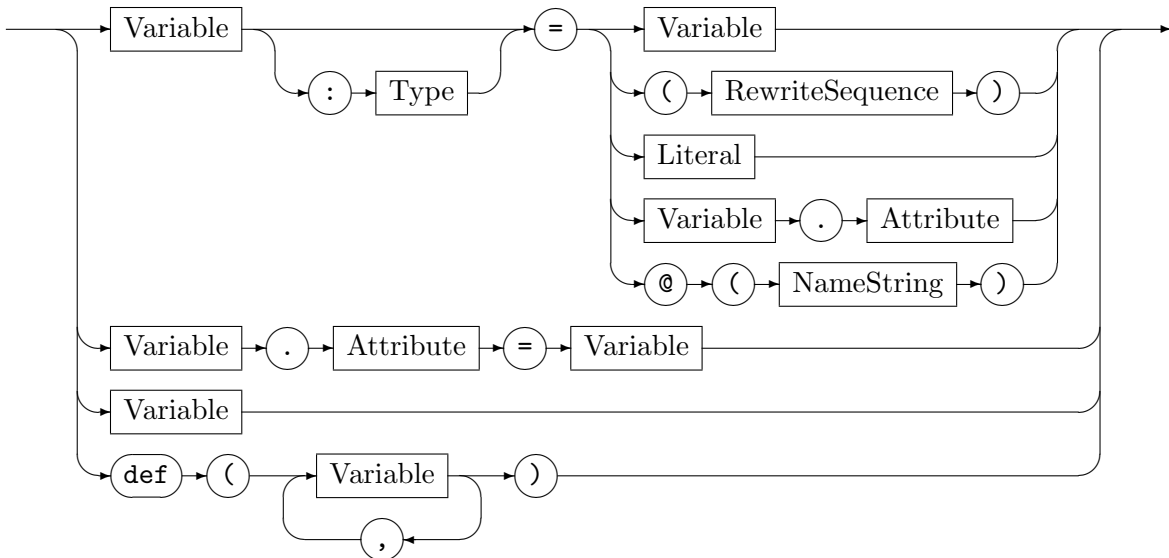
Variables can hold graph elements, or values of value/attribute types. The typed explicit declaration introduces a sequence local variable, if the double colon and the type are missing, a graph global variable gets implicitely declared if not existing yet. Graph elements are initially assigned by the element returns of the *RuleExecution* statement. A *VarAssignment* rewrite factor ist always true. A variable in the *Variable* factor must contain a boolean value. A `def` term is successful iff all the the variables are defined. Assigning graph elements by name to variables is not supported in compiled sequences (names are not available on this level).

## 7.4 Visited Flags

*VisitedFlagsManagement*
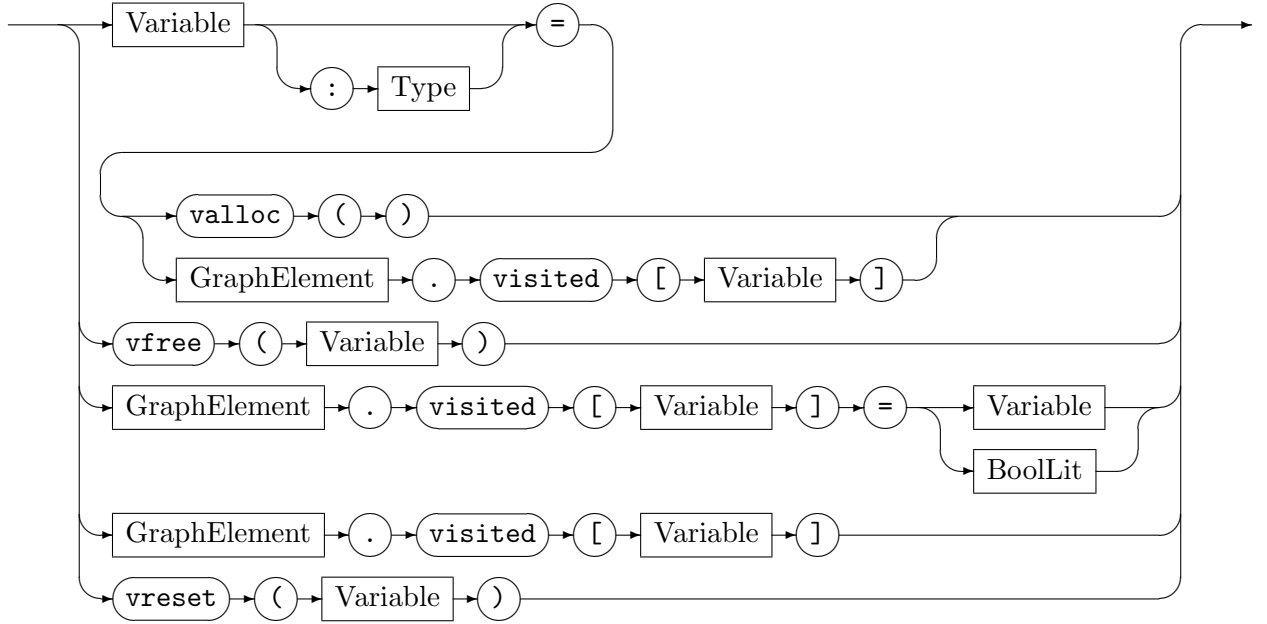


The visited flags are stored in some excess bits of the graph elements, but if this pool is exceeded they are stored in additional dictionaries, one per visited flag requested. Due to this flags must get allocated/deallocated, and all flag related operations require an integer number – the flag id – 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.

## 7.5 Storages

Storages are variables of set or map type used in the sequences; in contrast to the sets in the graph model and rewrite rules, their elements can be of node or edge type. Storing nodes and edges is in fact their primary usage. 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 further 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.

*VariableHandling*
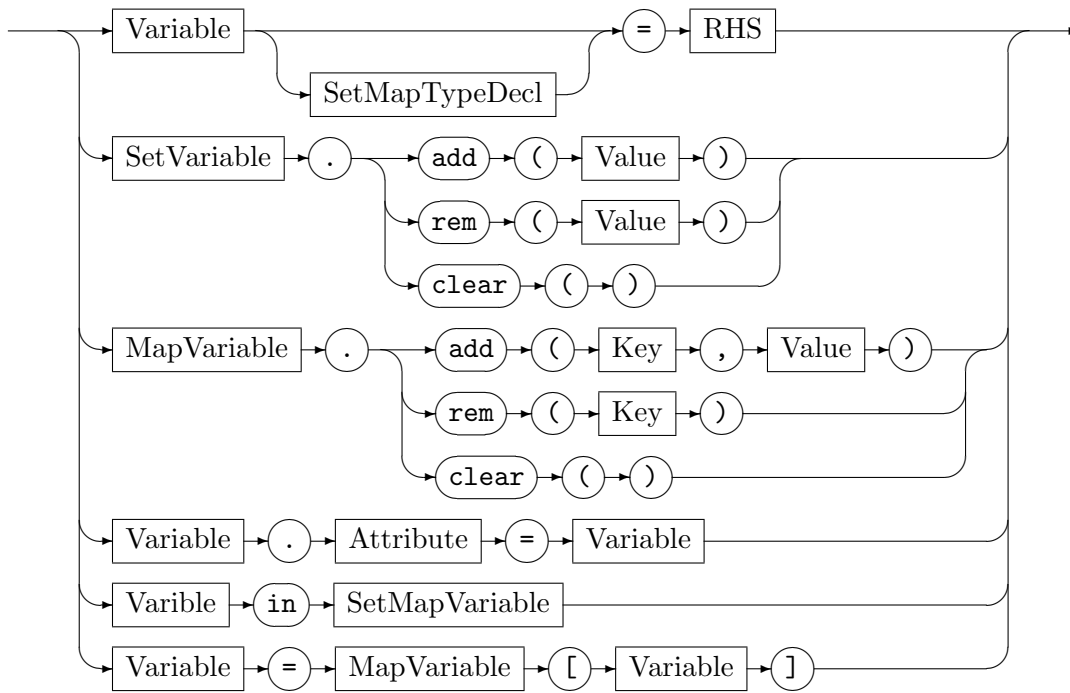


*SetMapTypeDecl*
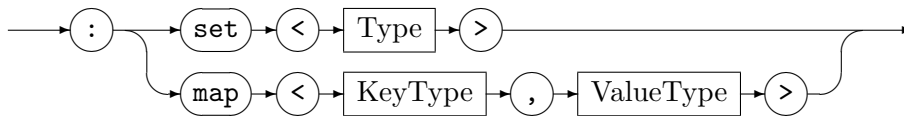


*RHS*
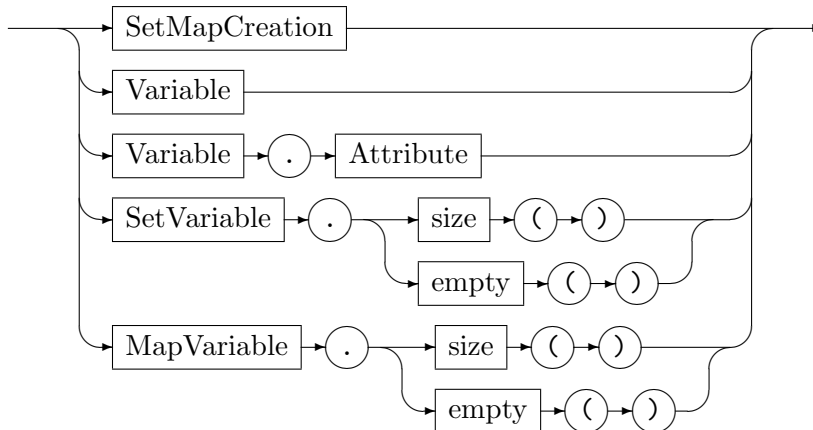
*SetMapCreation*
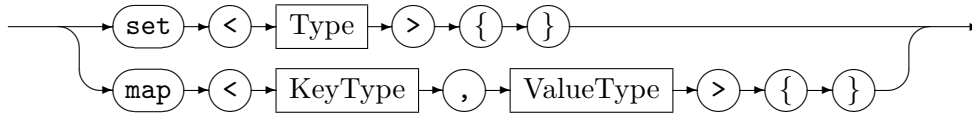


In the interpreted sequences, variables can be declared to be of set/map type; in the compiled sequences, they must be declared to be of set/map type (all variables must be declared in the compiled sequences). Both types of sequences require a set/map to be created and assigned to a variable before it can be used.

**EXAMPLE (45)**

```
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.

*Emptyness assignment:*
   v=sm.empty() writes to the variable v whether the set or map sm is empty, succeeds always.

*Map lookup assignemt:*
   v=m[el] assigns the result of the map lookup to the variable v,
   succeeds iff el is contained in m,
   fails otherwise, not touching the variable v.

*RewriteFactor*
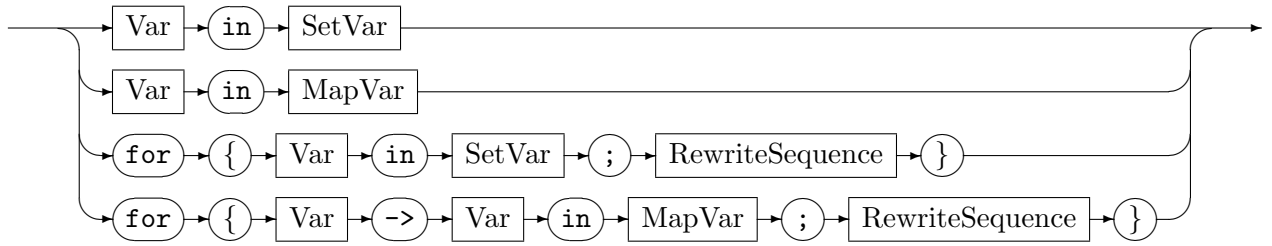


Handling of the storages is completed by the rewrite factors for membership query and storage iteration. The binary operator `el in sm` checks for set/map membership; it returns true if `el` is contained in the set or the domain of the map, otherwise false. The `for` command iterates over all elements 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 (46)**

The following XGRS is a typical storage usage. First an empty set `x` is created, which gets populated by an 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()`

**NOTE (23)**

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

## 7.6 Quick reference table

Table 7.2 lists most of the operations of the graph rewrite expressions at a glance.

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

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

Table 7.2: GRS expressions at a glance

# 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.

## 8.1 Building Blocks

GRSHELL is case sensitive. A line may be emty, 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 (`''`).

*Number*
Is an `int` or `float` constant in decimal notation (see also Section 6.1).

*Parameters*

Text , (railroad diagram)

*SpacedParameters*

Text (railroad diagram)
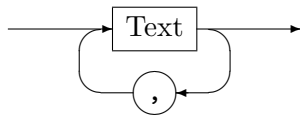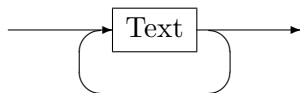
In order to describe the commands more precisely, the following (semantic) specializations of *Text* are defined:

*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*
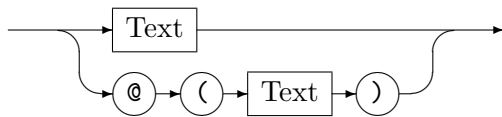


The elements of a graph (nodes and edges) can be accessed both by their (graph global) variable identifier and by their *persistent name* specified through a constructor (see Section 8.2.5). The specializations *Node* and *Edge* of *GraphElement* require the corresponding graph element to be a node or an edge respectively.

---

**EXAMPLE (47)**

We insert a node, anonymously and with a constructor (see also Section 8.2.5):

```
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 (24)**

Persistent names will be saved (`save graph...`, see Section 8.2.4) and exported, and, if you visualize a graph (`dump graph...`, see Section 8.2.4), graph elements will be labeled with their persistent names. Persistent names have to be unique for a graph (the graph they belong to).
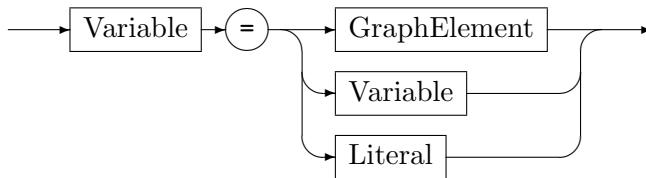


Assigns the variable or persistent name *GraphElement* or literal to *Variable*. If *Variable* has not been defined yet, it will be defined implicitly. As usual for scripting languages, variables have neither static types nor declarations. The variables known to GRSHELL are the graph global variables (see 7 for the distinction between graph global and sequence local variables).
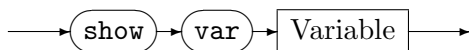


Prints the content of the specified variable.

## 8.2 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*
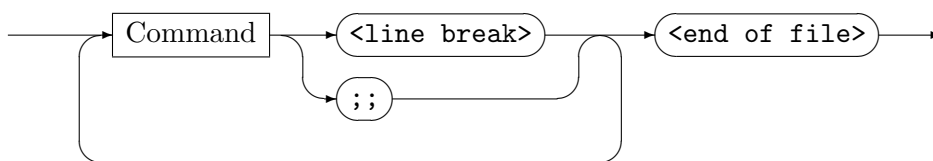


### 8.2.1 Common Commands
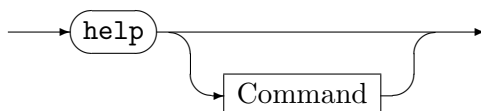


Displays an information message describing all the supported commands. A command `Command` displayed with `...` has further help available, which can be displayed with `help Command`.
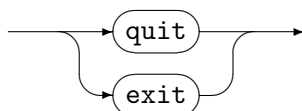
Quits GRSHELL. If GRSHELL is opened in debug mode, a currently active graph viewer (such as YCOMP) will be closed as well.
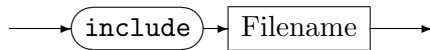


Executes the GRSHELL script *Filename*. A GRSHELL script is just a plain text file containing GRSHELL commands. They are treated as they would be entered interactively, except for parser error If a parser error occurs, execution of the script will stop immediately.
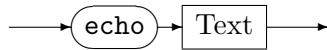


Prints *Text* onto the GRSHELL command prompt.
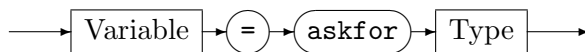


The `askfor` command interactively asks the user for a value of the specified type. The entered value is type checked against the expected type, and assigned to the given variable in case it matches. If the type is a value type, the user is prompted to enter a value literal with the keyboard. If the 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.

---

**EXAMPLE (48)**

`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 (49)**

On a Linux machine you might execute

```
!sh -c "ls␣|␣grep␣stuff"
```

---
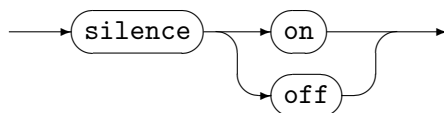


Switches the new node / edge created / deleted messages on(default) or off. Switching them off allows for much faster execution of scripts containing a lot of creation commands.
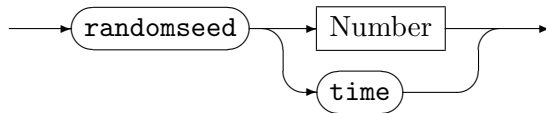
Sets the random seed to the given number for reproducible results when using the $-operator-prefix or the random-match-selector, whereas time sets the random seed to the current time in ms.
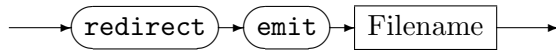


Redirects the output of the emit-statements in the rules from stdout to the given file.
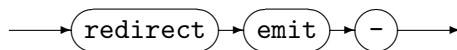


Redirects the output of the emit-statements in the rules to stdout (again).

## 8.2.2 Graph Commands
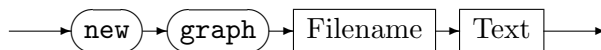


Creates a new graph with the model specified in *Filename*. Its name is set to *Text*. The model file can be either source code (e.g. turing_machineModel.cs) or a .NET assembly (e.g. lgsp-turing_machineModel.dll). It's also possible to specify a rule set file as *Filename*. In this case the necessary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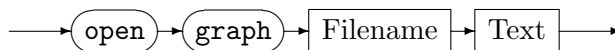


Opens the graph *Text* stored in the backend. However, the *LGSPBackend* doesn't support persistent graphs, and as the *LGSPBackend* is the only backend available at the moment, this command is currently useless. You may achieve persistence by using import/export or save/include instead.
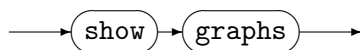


Displays a list of currently available graphs.
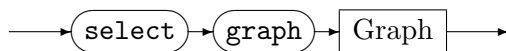


Selects the current working graph. This graph acts as *host graph* for graph rewrite sequences (see also Sections 1.5 and 8.2.8). Though you can define multiple graphs, only one graph can be the active "working graph".
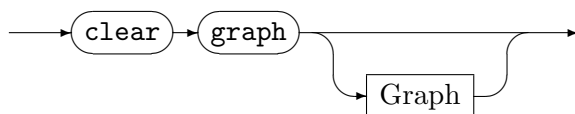


Deletes all graph elements of the current working graph resp. the graph *Graph*.
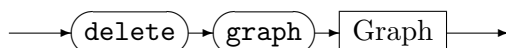


Deletes the graph *Graph* from the backend storage.

### 8.2.3 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.
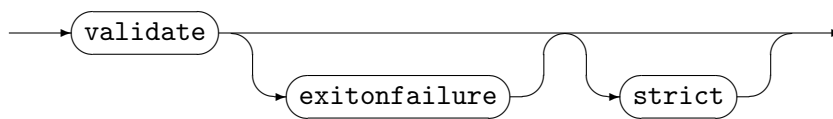


Validates if the current working graph fulfills the connection assertions specified in the corresponding graph model. The *strict* mode additionally requires all the edges available in the instance graph to be specified in the model in order to be "valid". Otherwise edges between nodes without specified constraints are ignored.
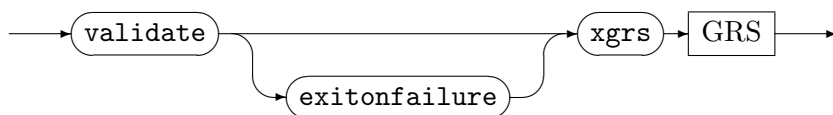


Validates if the current working graph satisfies the graph rewrite sequence given. Before the graph rewrite sequence is executed, the instance graph gets cloned; the sequence operates on the clone, allowing you to change the graph as you want to, without influence on the host graph. Validation fails 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 (50)**

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:
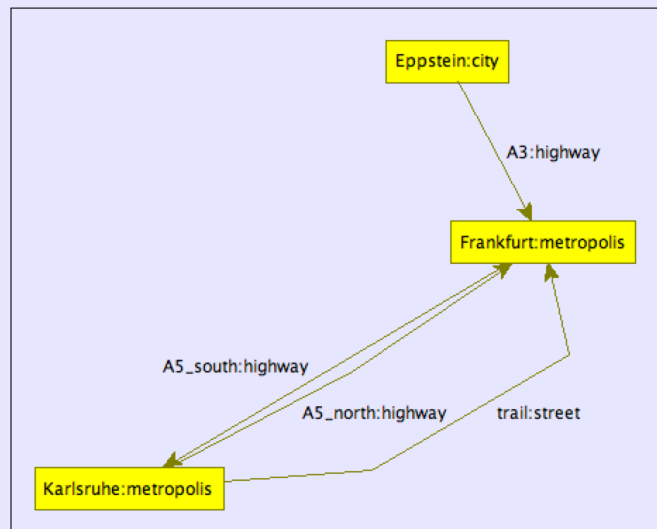


This graph is valid but not strict valid.

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

### 8.2.4   Graph Input and Output Commands

```
⟶( save )⟶( graph )⟶[ Filename ]⟶
```
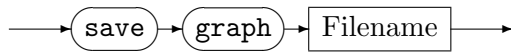
Dumps the current graph as GRSHELL script into *Filename*. The created script includes

- 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 8.2.1).

```
⟶( export )⟶[ Filename.grs ]⟶
              [ Filename.grsi ]⟶[ withvariables ]⟶
```
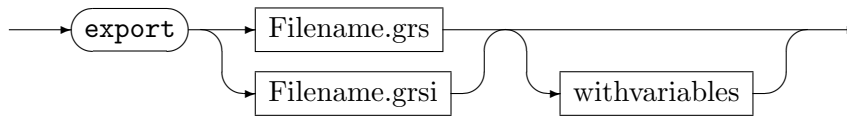
Exports an instance graph in GRS (.grs/.grsi) format, which is a reduced GRSHELL script (it can get imported and exported on API level 10.3 without using the GRSHELL). It contains the `new graph` command, followed by `new node` commands, followed by `new edge` commands. If `withvariables` is specified, 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 a GRSHELL session including visualization styles, the goal of the `export` command is graph rewrite system interoperability.

```
⟶( export )⟶[ Filename.gxl ]⟶
```
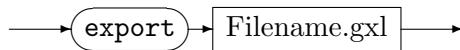
Exports an instance graph and a graph model in GXL format [WKR02, HSESW05], which is somewhat of a standard format for graphs of graph rewrite systems, but suffers from the well-known XML problems – it is barely human-readable and bloated. Exporting fails if the graph model contains attributes of `set<S>-`,`map<S,T>-`, or `object`-type.

```
⟶( import )⟶[ Filename.grs ]⟶
              [ Filename.grsi ]⟶
```
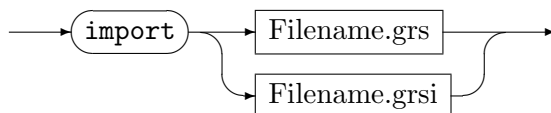
Imports the specified graph instance in GRS (.grs/.grsi) format (the *reduced* GRSHELL script, a saved graph can only be imported by `include` (but an exported graph can be imported by `include`, too)). The referenced graph model must be available as `.gm`-file.

```
⟶( import )⟶[ Filename.gxl ]⟶
                           [ ModelOverride ]⟶
```
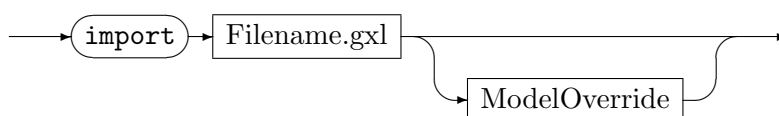
Imports the specified graph instance and model in GXL format. If a model override of the form `Filename.gm` is specified, the given model will be used instead of the model in the GXL file. The `.gxl`-graph must be compatible to the `.gm`-model.

**NOTE (25)**

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 Your-Name.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. 8.2.8).

4. Now you are ready to use the actions.

### 8.2.5  Graph Manipulation Commands

Graph manipulation commands alter existing graphs; they allow to create and delete graph elements and change attributes. These are tasks which should be carried by the rules of the rule language – the commands are mainly used as elementary instructions in graph input and output.
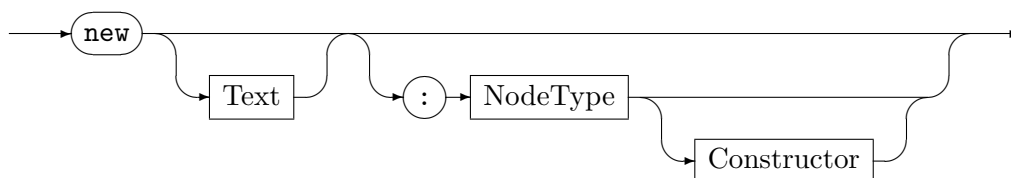


Creates a new node within the current graph. Optionally a variable *Text* is assigned to the new node. If *NodeType* is supplied, the new node will be of type *NodeType* and attributes can be initialized by a constructor. Otherwise the node will be of the base node class type *Node*.

**NOTE (26)**

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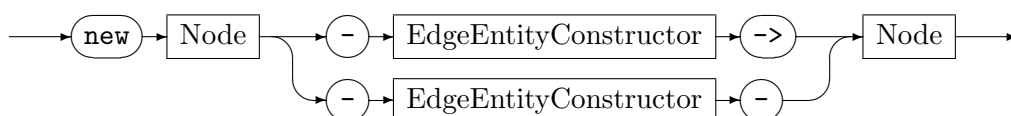
*EdgeEntityConstructor*
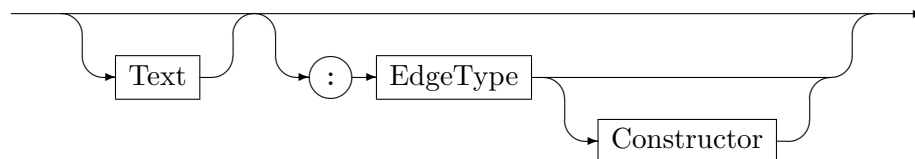


Creates a new edge within the current graph between the specified nodes, directed from the first to the second *Node* in the case of `-->`, or undirected in the case of `--`. Optionally a variable *Text* is assigned to the new edge. If *EdgeType* is supplied, the new edge will be of type *EdgeType* and attributes can be initialized by a constructor. Otherwise the edge will be of the base edge class type `Edge` for `-->` or `UEdge` for `--`.

*Constructor*
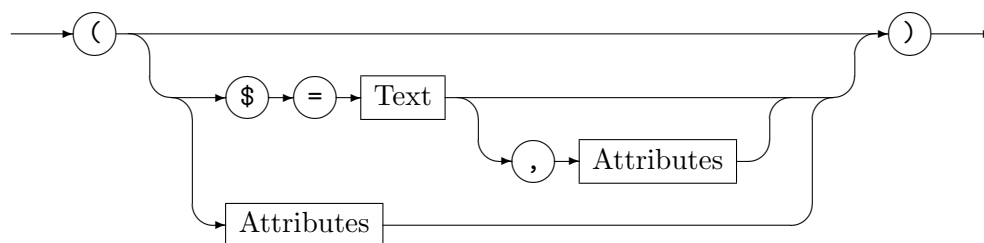


*Attributes*
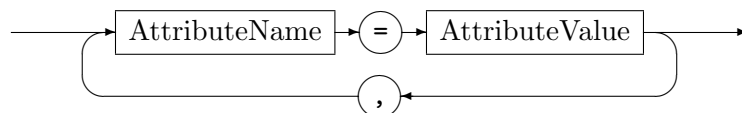


*AttributeValue*
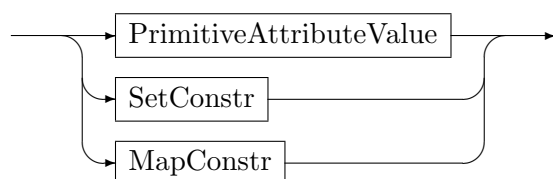


*PrimitiveAttributeValue*
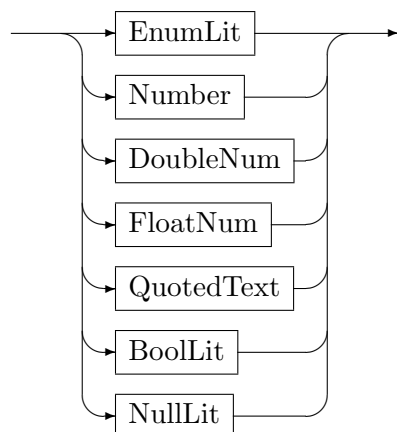
*SetConstr*
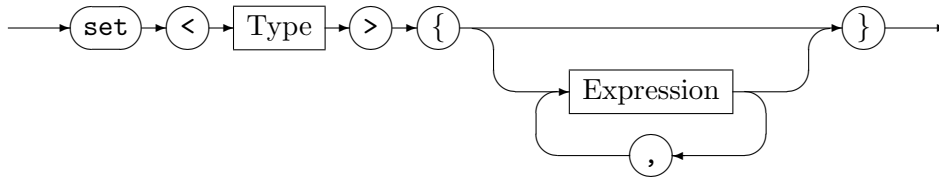


*MapConstr*
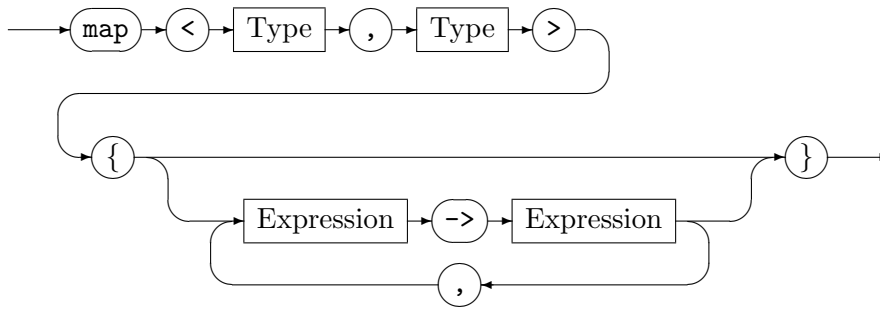


A constructor is used to initialize a new graph element (see `new ...` below). A comma separated list of attribute declarations is supplied to the constructor. Available attribute names are specified by the graph model of the current working graph. All the undeclared attributes will be initialized with default values, depending on their type (`int` ← 0; `boolean` ← `false`; `float` ← 0.0f; `double` ← 0.0; `string` ← `""`; `set<T>` ← `set<T>{}`; `map<S,T>` ← `map<S,T>{}`; `enum` ← unspecified;).

The `$` is a special attribute name: a unique identifier of the new graph element. This identifier is also called *persistent name* (see Example 47). This name can be specified by a constructor only.
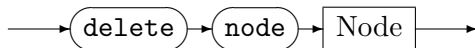


Deletes the node *Node* from the current graph. Incident edges will be deleted as well.
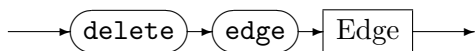


Deletes the edge *Edge* from the current graph.
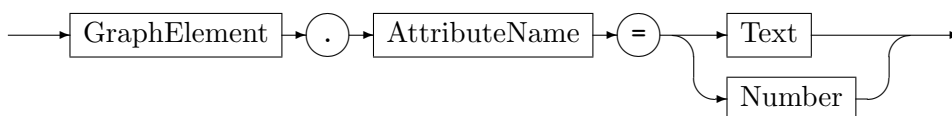


Set the attribute *AttributeName* of the graph element *GraphElement* to the value of *Text* or *Number*.

### 8.2.6 Graph and Model Query Commands
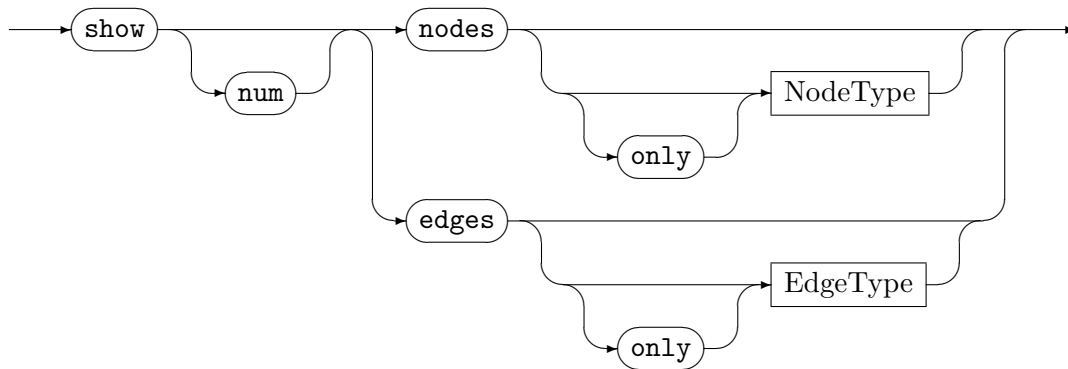


Gets the persistent names and the types of all the nodes/edges of the current graph. If a node type or edge type is supplied, only elements compatible to this type are considered. The `only` keyword excludes subtypes. Nodes/edges without persistent names are shown with a pseudo-name. If the command is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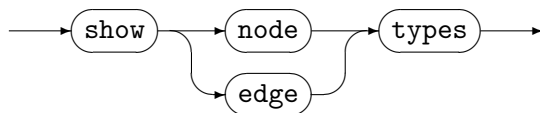


Gets the node/edge types of the current graph model.
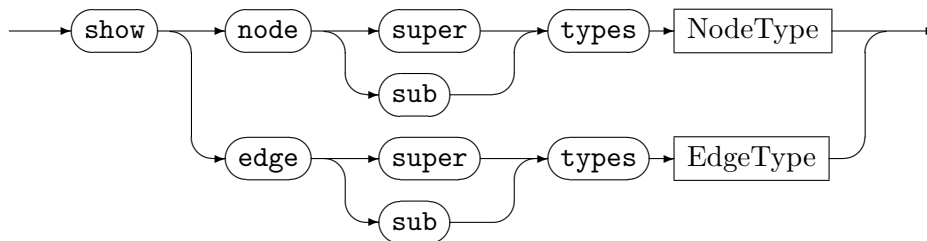


Gets the inherited/descendant types of *NodeType*/*EdgeType*.
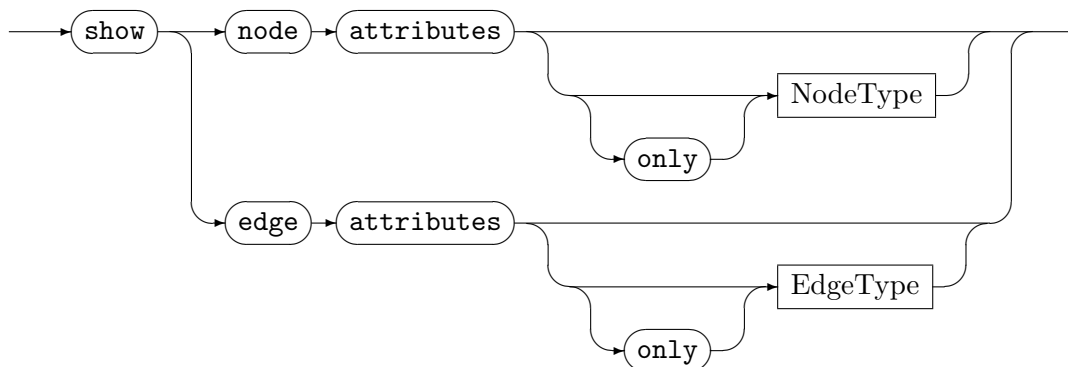


Gets the available node/edge attribute types. If *NodeType*/*EdgeType* is supplied, only attributes defined in *NodeType*/*EdgeType* are diplayed. The `only` keyword excludes inherited attributes.

**NOTE (27)**

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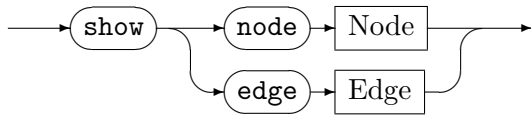


Gets the attribute types and values of a specific graph element.
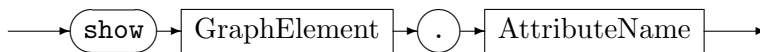


Displays the value of the specified attribute.
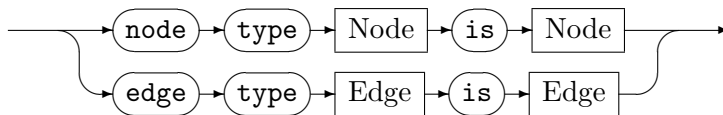


Gets the information whether the first element is type-compatible to the second element.

### 8.2.7 Graph Visualization Commands
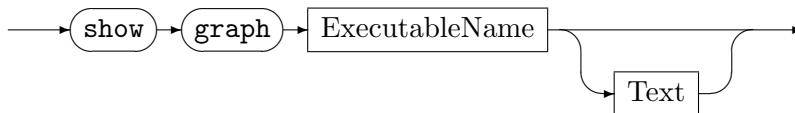


Dumps the current graph in VCG format into a temporary file. The temporary VCG file will be passed to the program *ExecutableName* as first parameter; further parameters, such as program options, can be specified by *Text*. If you use YCOMP[1] as executable (`show graph ycomp`), this may look like



---
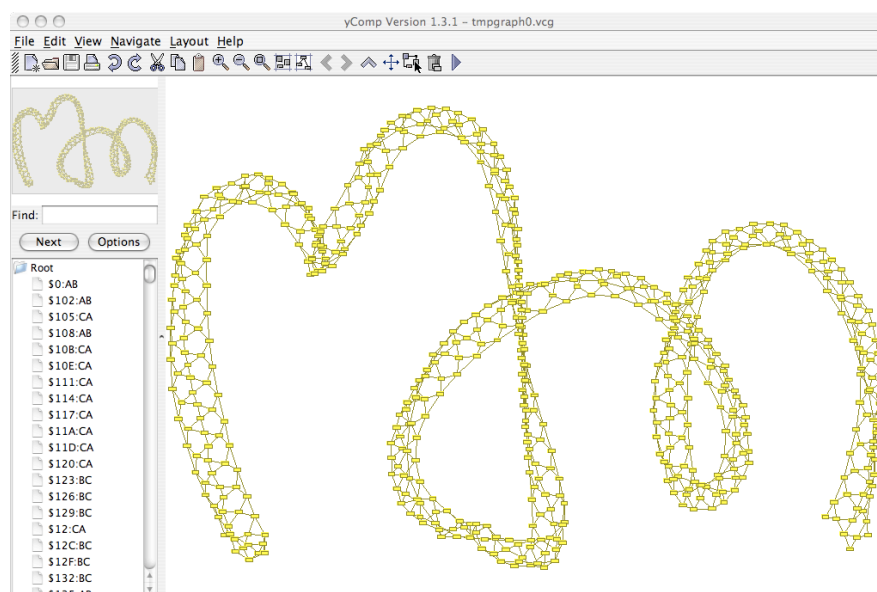
[1]See Section 1.7.4.

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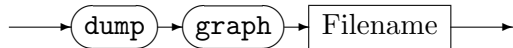


Dumps the current graph in VCG format into the file *Filename.*

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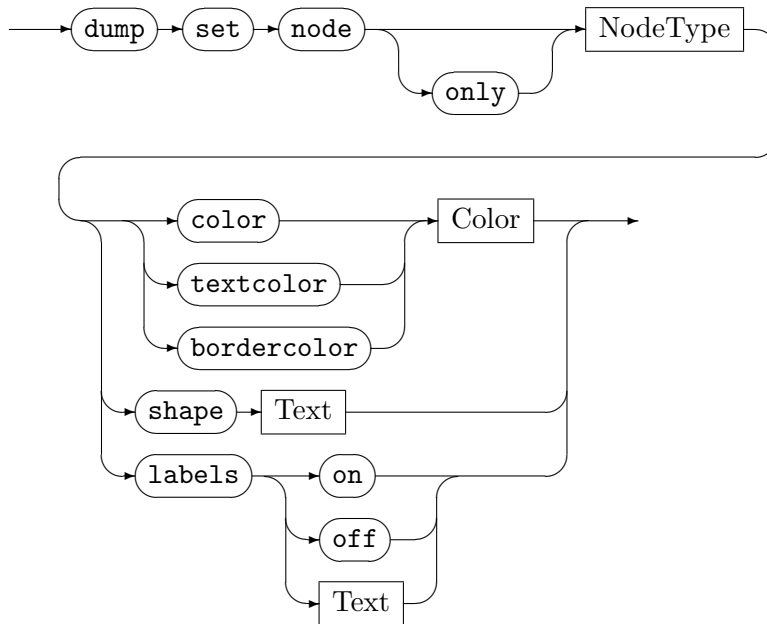


Sets the color, text color, border color, the shape or the label of the nodes of type *NodeType* and all of its subtypes. The keyword `only` excludes the subtypes. The available colors are specified at the begin of this chapter. The following shapes are supported: `box`, `triangle`, `circle`, `ellipse`, `rhomb`, `hexagon`, `trapeze`, `uptrapeze`, `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.
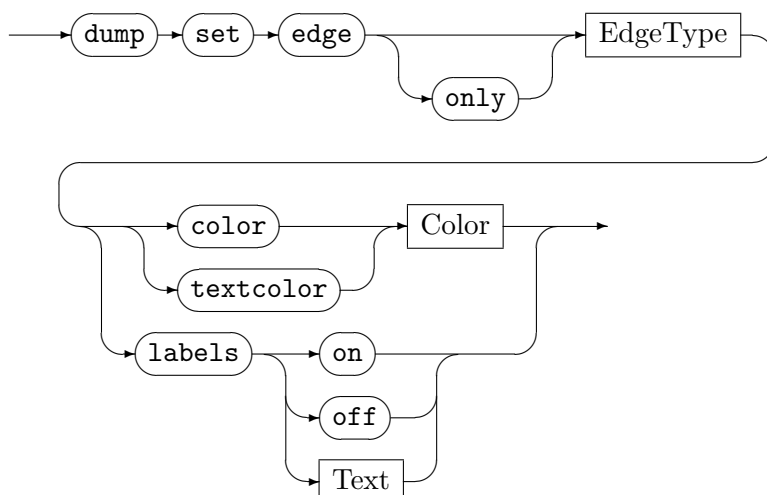
Sets the color, text color or label of the edges of type *EdgeType* and all of its subtypes. The keyword `only` excludes the subtypes. The available colors are specified at the begin of this chapter. The default labeling is set to `on` with `Name:Type`, it can be overwritten by an specified label string or switched off.
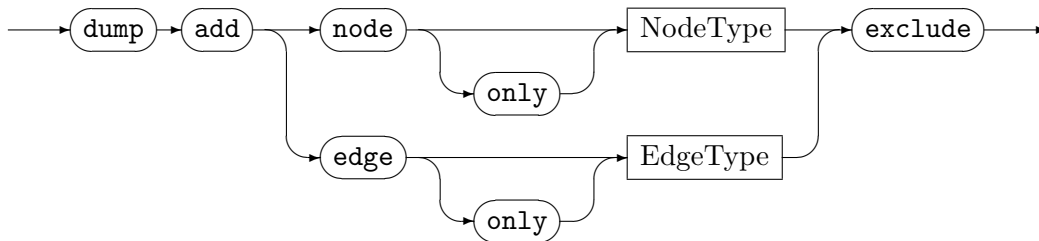


Excludes nodes/edges of type *NodeType*/*EdgeType* and all of its subtypes from output, for a node it also excludes its incident edges. The keyword `only` excludes the subtypes from exlusion, i.e. subtypes of *NodeType*/*EdgeType* are dumped.
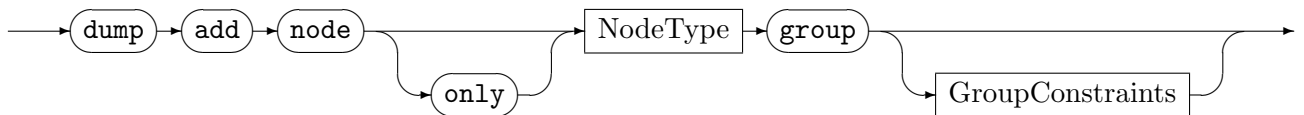


*GroupConstraints*
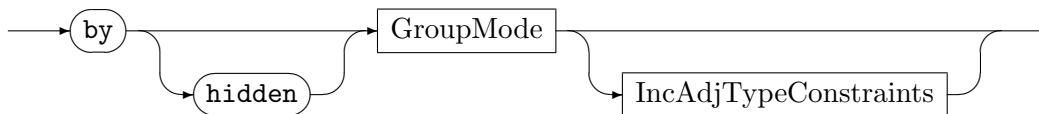


*IncAdjTypeConstraints*
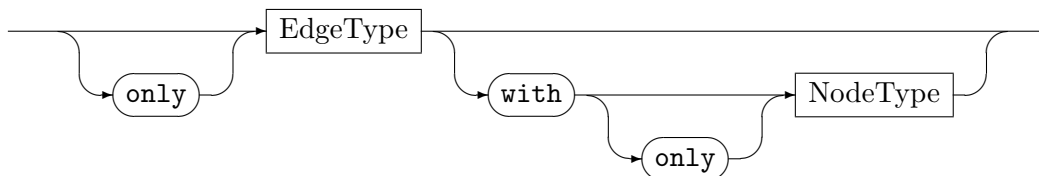


Declares *NodeType* and subtypes of *NodeType* as group node type. All the differently typed nodes that point to a node of type *NodeType* (i.e. there is a directed edge between such nodes) will be grouped and visibly enclosed by the *NodeType*-node. `GroupMode` is one of `no`,`incoming`,`outgoing`,`any`; `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 (28)**

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 resultion 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:
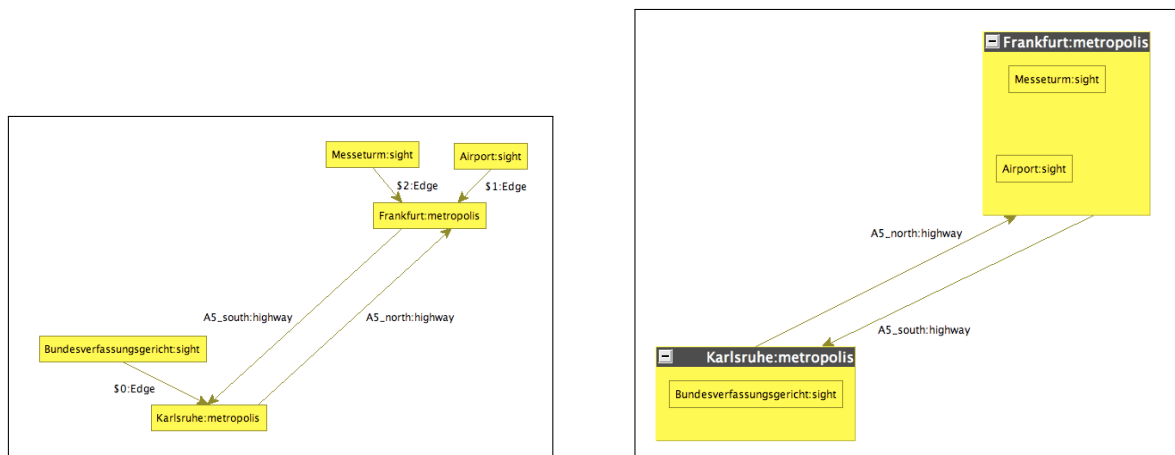


right side: dumped with `dump add node metropolis group`
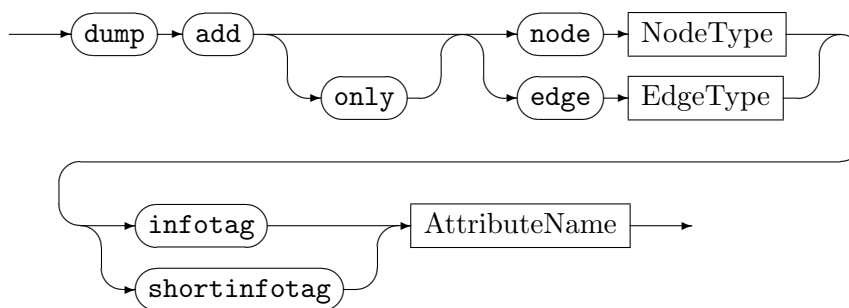


Declares the attribute *AttributeName* to be an "info tag" or "short info tag". Info tags are displayed like additional node/edge labels, in format `Name=Value`, or `Value` only for short info tags. The keyword `only` excludes the subtypes of *NodeType* resp. *EdgeType*. In the following example *river* and *jam* are info tags:
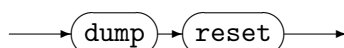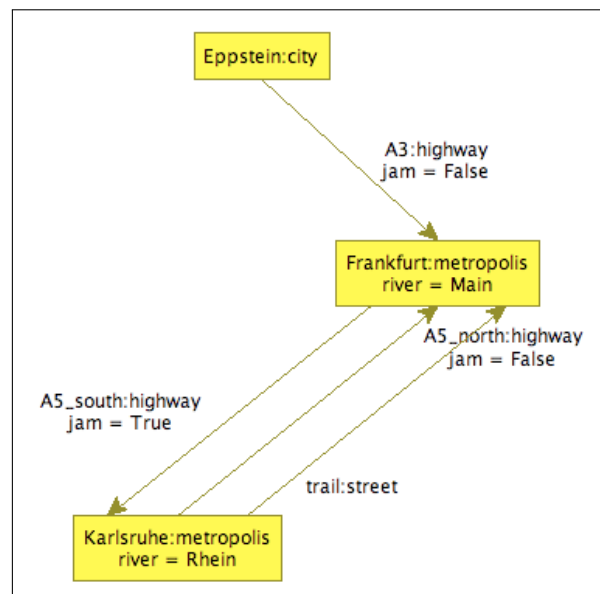




Resets all style options (`dump set...`) and (`dump add...`) to their default values.

**NOTE (29)**

Small graphs allow for fast visual understanding, but with an increasing number of nodes and edges they quickly loose 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/infotags/shortinfotags to display the most important information directly. Choose the layout algorithm and the options delivering the best results for your needs, organic and hierarchic or compiler graph (an extension of hierarchic with automatic edge cutting – marking cut edges by fat dots, showing the edge only on mouse over and allowing to jump to the other end on a mouse click) should be tried first.

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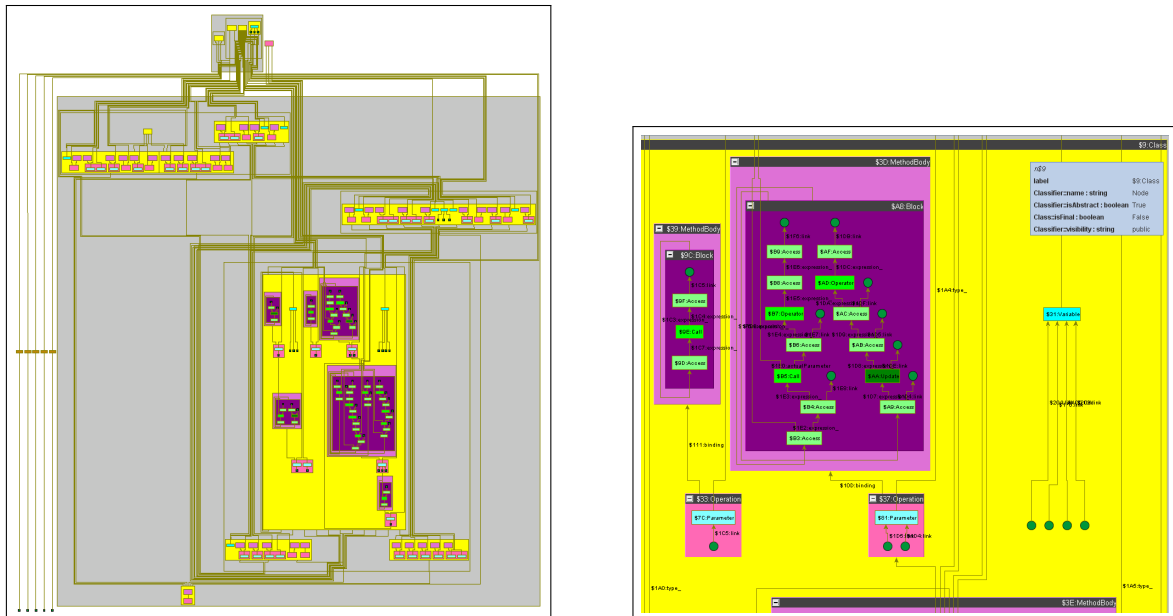


Overview of the initial program graph and some details of the "Node" class

### 8.2.8   Action Commands (XGRS)

An *action* denotes a graph rewrite rule.
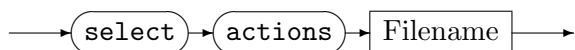


Selects a rule set. *Filename* can either be a .NET assembly (e.g. "rules.dll") or a source file ("rules.cs"). Only one rule set can be loaded simultaneously.
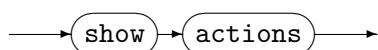


Lists all the rules of the loaded rule set, their parameters, and their return values. Rules can return a set of graph elements.
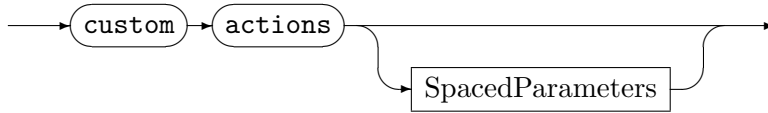
Executes an action specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSPBackend see Section 8.4.2).

*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*.

## 8.3   Graphical Debugger

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

### 8.3.1   Debugging Related Commands
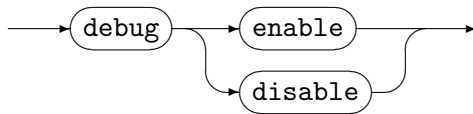


Enables and disables the debug mode. The debug mode shows the current working graph in a YCOMP window. All changes to the working graph are tracked by YCOMP immediately.
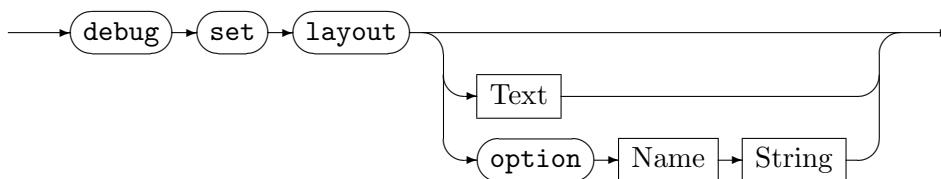


Sets the default graph layout algorithm to *Text*. If *Text* is omitted, a list of available layout algorithms is displayed. See Section 1.7.4 on YCOMP layouters. The `option` version allows to specify layout options by name value pairs. The available layout options can be listed by the following command.
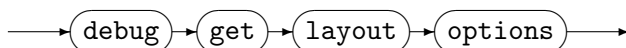


Prints a list of the available layout options of the layout algorithm.
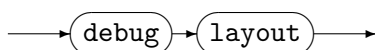


Forces re-layout of the graph shown in yComp (same as pressing the play button within yComp).

*GraphRewriteSequence*

```
——→(debug)→(xgrs)→| SimpleRewriteSequence |——→
```
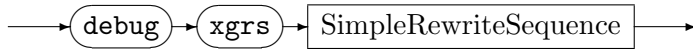
This executes the graph rewrite sequence *SimpleRewriteSequence* in the debugger. Same as `xgrs SimpleRewriteSequence` in the previous section, but allows tracing the rewriting process step-by-step.

### 8.3.2 Using the Debugger

During execution yComp[2] will display every single step. The debugger can be controlled by GRSHELL. Remember that the `%` modifier before a rule works as break point in a graph rewrite sequence. The debug commands are shown in Table 8.1. A run is shown in the following example 51.

| | |
|---|---|
| s(tep) | Execute the next rewrite rule (match and rewrite) |
| d(etailed step) | Execute a rewrite rule in a three-step procedure: matching, highlighting elements to be changed, doing rewriting |
| n(ext) | Ascend one level up within the Kantorowitsch tree of the current rewrite sequence (see Example 51) |
| (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 |
| (toggle) b(reakpoint) | Toggle a breakpoint at a rewrite rule, a true, or a false |
| r(un) | Continue execution |
| a(bort) | Cancel the execution immediately |

Table 8.1: GRSHELL debug commands

## 8.4 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.

### 8.4.1 Backend selection and custom commands

```
——→(select)→(backend)→| Filename |——————————→
                              └→(:)→| Parameters |┘
```
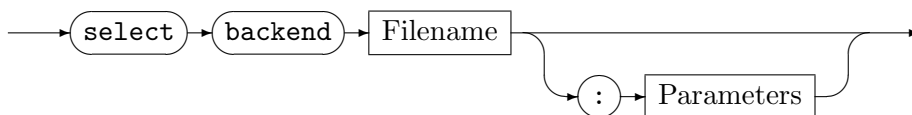
Selects a backend that handles graph and rule representation. *Filename* has to be a .NET assembly (e.g. `lgspBackend.dll`). Comma-separated parameters can be supplied optionally; if so, the backend must support these parameters. By default the LGSPBackend is used.

```
——→(show)→(backend)——→
```
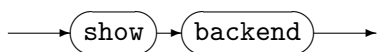
List all the parameters supported by the currently selected backend. The parameters can be provided to the `select backend` command.

---

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

**EXAMPLE (51)**

We demonstrate the debug commands with a slightly adjusted script for the Koch snowflake from GRGEN.NET's examples (see also Section 9.1). 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`):
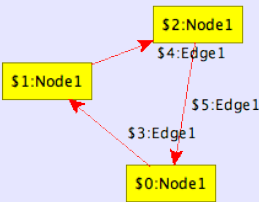


We type **d**(etailed step) to apply `makeFlake1` step by step resulting in the following graphs:
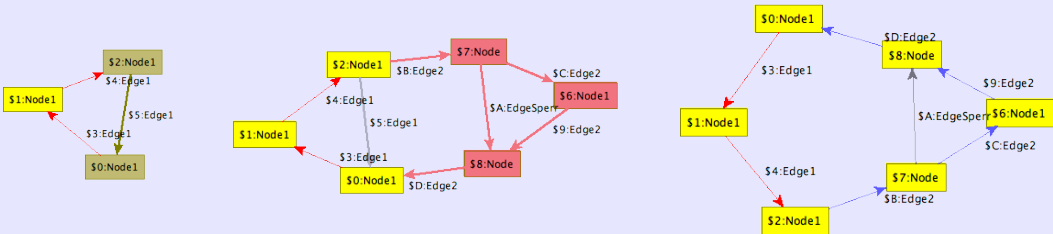


The following table shows the "break points" of further debug commands, entered one after another:

| Command | Active rule |
|---------|-------------|
| s | makeFlake1 |
| o | beautify |
| s | doNothing |
| s | beautify |
| n | beautify |
| o | makeFlake2 |
| r | — |
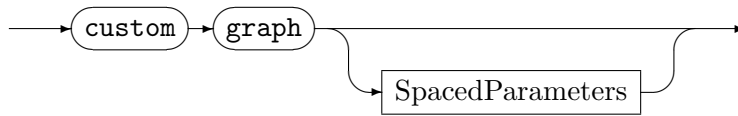
Executes a command specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSP backend see Sections 8.4.2).

### 8.4.2  LGSPBackend Custom Commands

The LGSPBackend supports the following custom commands:
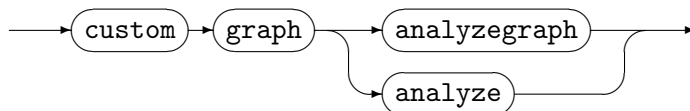


Analyzes the current working graph. The analysis data provides vital information for efficient search plans. Analysis data is available as long as GRSHELL is running, i.e. when the working graph changes, the analysis data is still available but maybe obsolete.
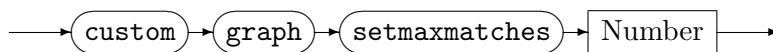


Sets the maximum amount of possible pattern matches to *Number*. This command affects the expression `[Rule]`. If *Number* is less or equal to zero, the constraint is reset.
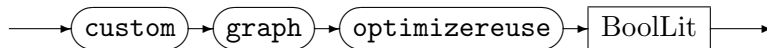


If set to false it prevents deleted elements from getting reused in a rewrite (i.e. it disables a performance optimization). If set to true, new elements may not be discriminable anymore from already deleted elements using object equality, hash maps, etc.
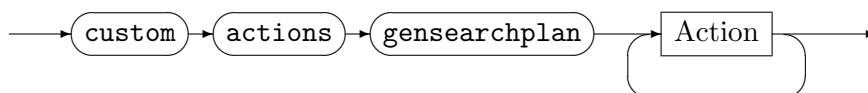


Creates a search plan (and executable code from it) for each rewrite rule *Action* using the data from analyzing the graph (`custom graph analyze`). Otherwise a default search plan is used. Analyzing and search plan/code generation themselves take some time, but they can lead to massively faster pattern 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.
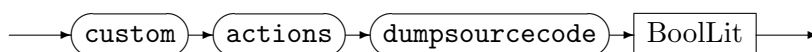


If set to true, C# files will be dumped for the newly generated searchplans (similar to the `-keep` option of the generator).

# CHAPTER 9

# EXAMPLES

## 9.1 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
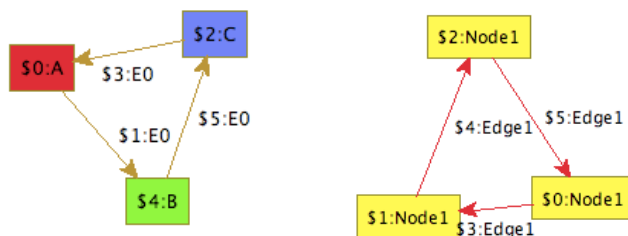


for the Sierpinski triangle resp. the Koch snowflake. First of all we have to compile the model and rule set files. So execute in GRGEN.NET's `bin` directory

```
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 graphs[1]. 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 8.2.8 for further information. The resulting graphs should look like Figures 9.1 and 9.2.

---

[1]Be careful: The running time increases exponentially in the number of iterations.
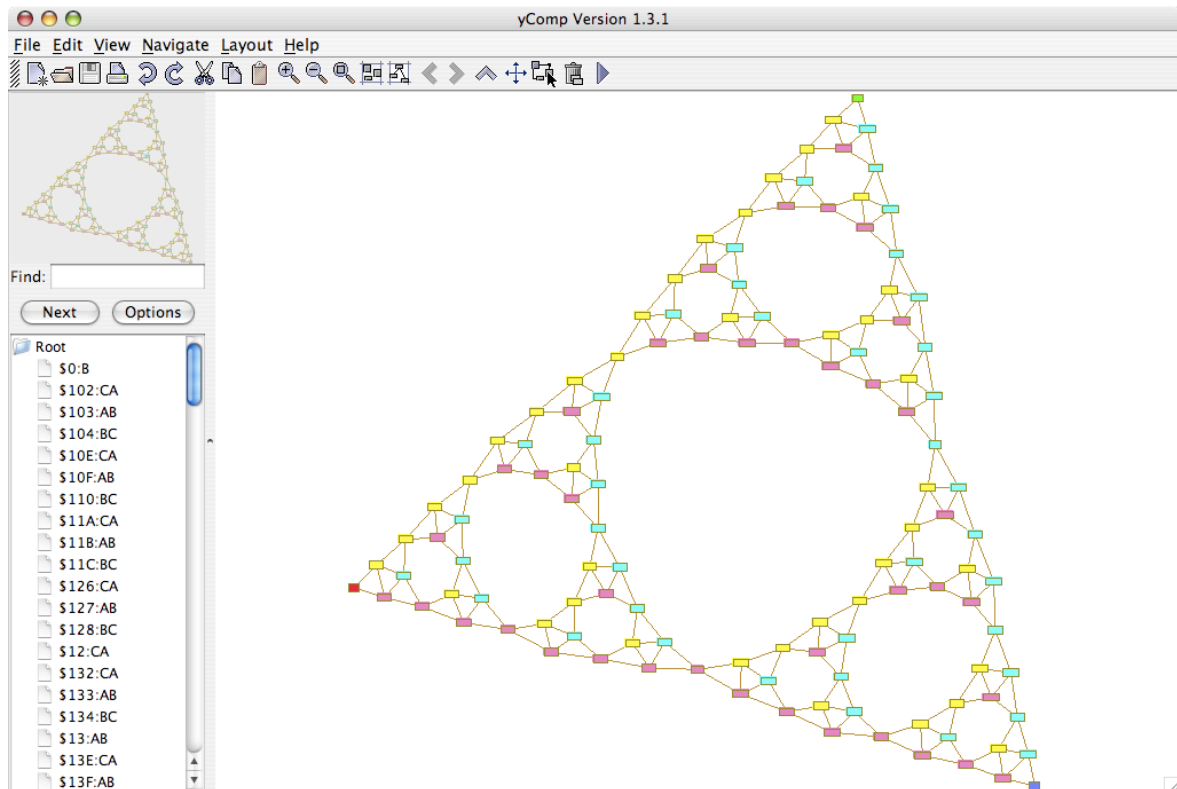
Figure 9.1: Sierpinski triangle
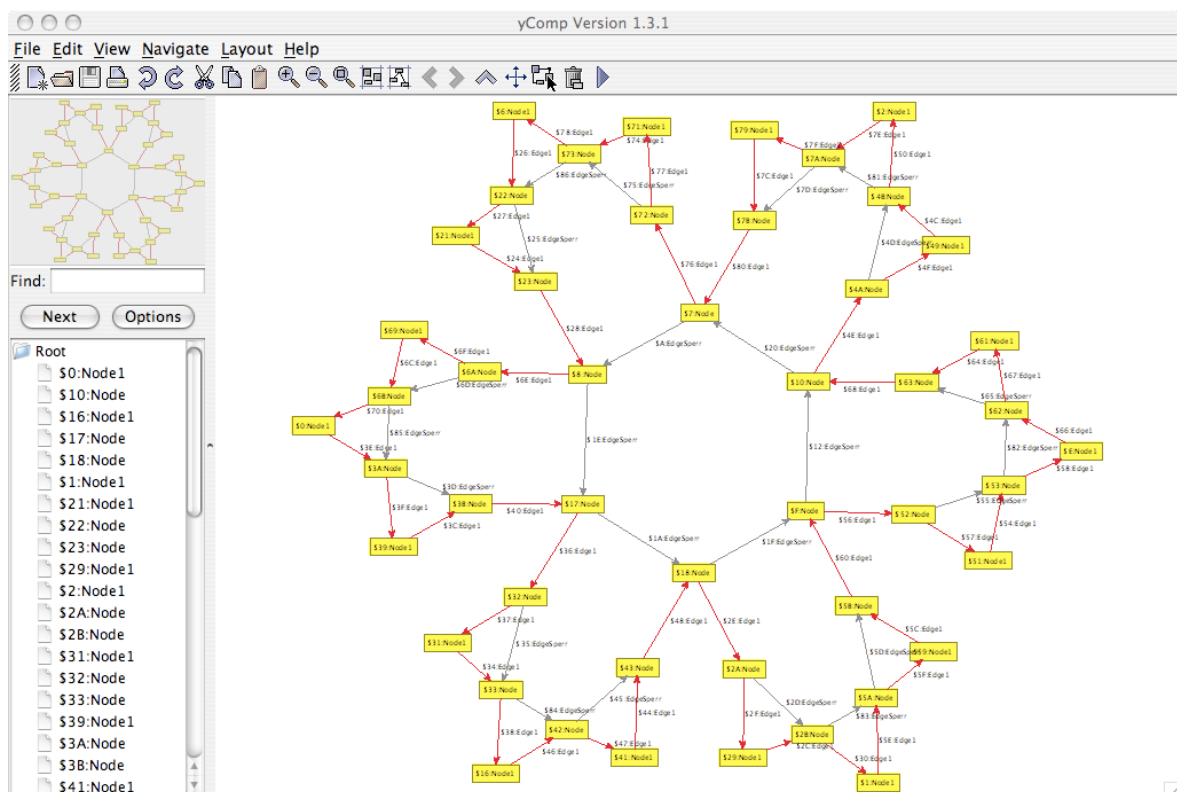


Figure 9.2: Koch snowflake

## 9.2 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.

### 9.2.1 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 113).

```
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;
```

### 9.2.2 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      wv -h:RWHead-> tp:TapePosition <-r:right- ltp:TapePosition;
54      modify {
55          delete(h);
56          s -:RWHead-> ltp;
57      }
58  }
59
60  rule moveRightRule {
61      wv:WriteValue -:moveRight-> s:State;
62      wv -h:RWHead-> tp:TapePosition -r:right-> rtp:TapePosition;
63      modify {
64          delete(h);
65          s -:RWHead-> rtp;
66      }
67  }
68
69  rule dontMoveRule {
70      wv:WriteValue -:dontMove-> s:State;
71      wv -h:RWHead-> tp:TapePosition;
72      modify {
73          delete(h);
74          s -:RWHead-> tp;
75      }
76  }
77
78  rule ensureMoveLeftValidRule {
79      wv:WriteValue -:moveLeft-> :State;
80      wv -: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      wv:WriteValue -:moveRight-> :State;
91      wv -: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 9.1).

### 9.2.3  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:
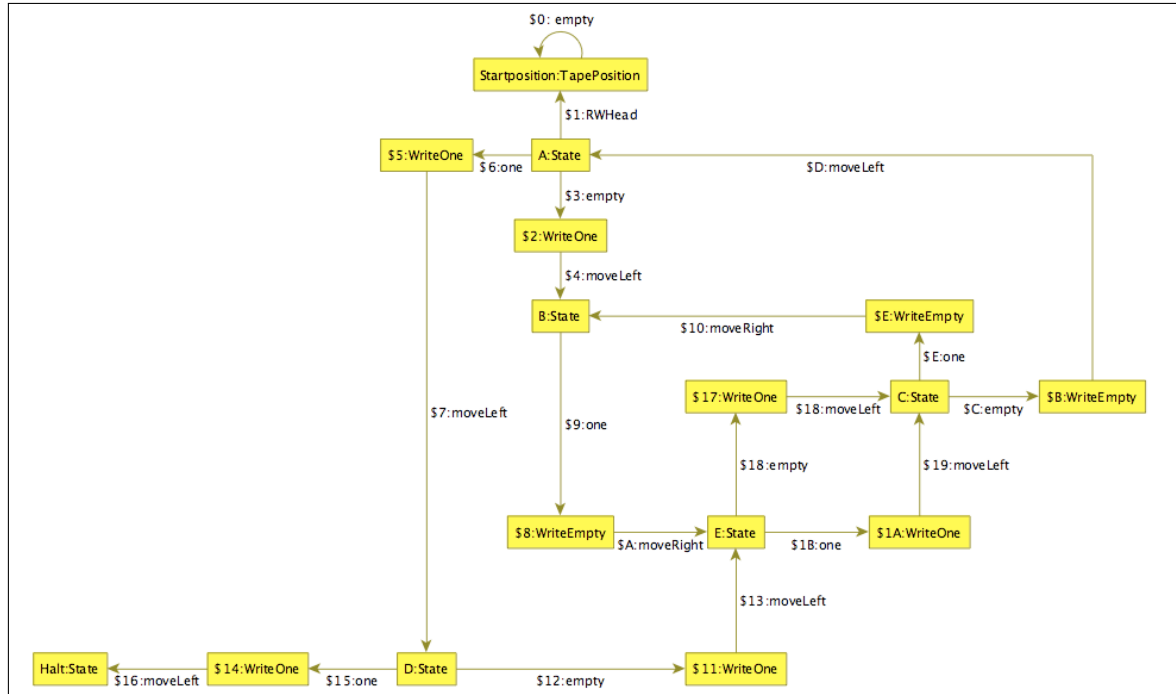


We have an initial host graph now. The graph rewrite sequence is quite straight forward and generic to the Turing graph model. Note that for each state the "`...Empty...` — `...One...`" selection is unambiguous.

```
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:
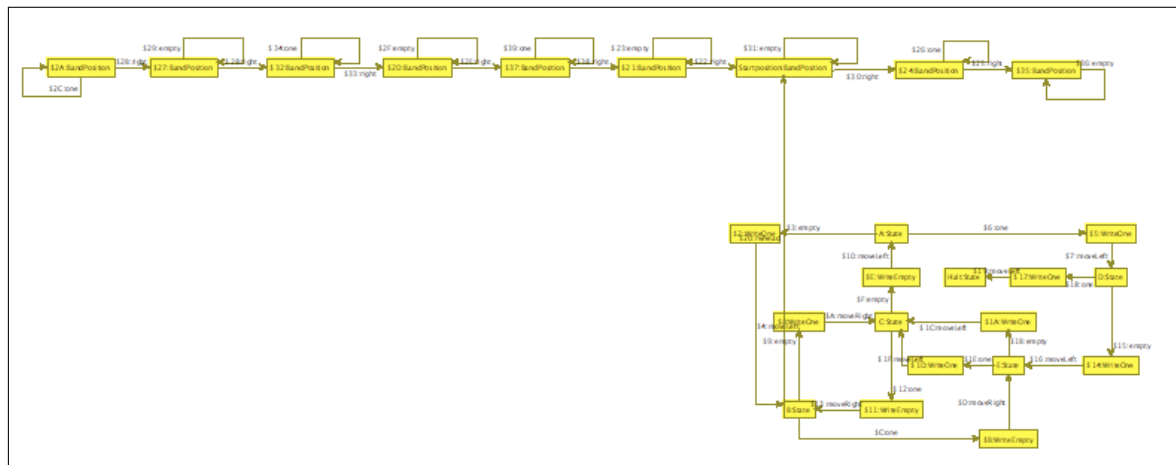


In order to improve the performance we generate better search plans. This is a crucial step for execution time: With the initial search plans the beaver runs for 1 minute and 30 seconds. With improved search plans after the first 32 steps he takes about 8.5 seconds[2].

_____

[2]On 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 10

# 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 model

- the interface to the rules

- the porter module for importing and exporting of graphs and miscellaneous stuff

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` with the entities from both points above.

**NOTE (30)**

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.

## 10.1 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 (31)**

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`.

## 10.2   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 (32)**

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 (33)**

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 (52)**

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 (53)**

This is an example doing mostly the same as the previous example 52, in a slightly more complicated way. 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);
```

## 10.3 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 25), the EMF/ECORE importer expects first one or more `.ecore` files and following optionally a `.xmi` files and/or a `.grg` file. 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`

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 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 communicate with yComp on the API level (from your own code) is shown in
  `examples-api/YCompExample/YCompExample.cs`.

---

**NOTE (34)**

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 (35)**

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;
```

## 10.4   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`.

## 10.5 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 mimimum 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`).

[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. *Scientic American*, 251(2):10–12, 16, 17, 8 1984.

[Dia]        DiaGen Developer Team. The Diagram Editor Generator. `http://www.unibw.de/inf2/DiaGen/`.

[Dör95]      Heiko Dörr. *Efficient Graph Rewriting and its Implementation*, volume 922 of *LNCS*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[EHK⁺99]    H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout A. and Comparison with Double Pushout A. In *[Roz99]*, volume 1, pages 247–312. 1999.

[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.

[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.

[Roz99]     G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation.* World Scientific, 1999.

[RVG08]     Arend Rensink and Pieter Van Gorp. Graph-based tools: The contest. Proceedings 4th Int. Conf. on Graph Transformation (ICGT '08), 2008.

[SAI⁺90]    Herbert Schildt, American National Standards Institute, International Organization for Standardization, International Electrotechnical Commission, and ISO/IEC JTC 1. *The annotated ANSI C standard: American National Standard for Programming Languages C: ANSI/ISO 9899-1990.* 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 suchprogrammbasierte 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

127

General Index