Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

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

The GRGEN.NET User Manual

Refers to GRGEN.NET Release 1.4

www.grgen.net

Jakob Blomer   Rubino Geiß

February 13, 2008

ii

ABSTRACT

GRGEN.NET is a graph rewrite tool enabling elegant and convenient development of graph rewriting applications with comparable performance to conventionally developed ones. GR-GEN.NET uses attributed, typed, and 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.

FOREWORD

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 was the only tool that brought together the areas of compiler construction and graph rewriting [Ass00]. 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 an 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, BKG07] 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.

If you find that any statement in this manual needs improvement, we encourage you to contact the maintainer of GRGEN.NET: rubino@ipd.uni-karlsruhe.de. Thank you for using GRGEN.NET.

Karlsruhe in July 2007, the authors of GRGEN.NET

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 What is GRGEN.NET?

GRGEN (Graph Rewrite GENerator) is a generative programming system for graph rewriting. For the potentially expensive matching problem, GRGEN applies several novel heuristic optimizations. According to Varró's benchmark[VSV05], it is at least one order of magnitude faster than any other tool known to us.

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 [BKG07, Bat06]. For the rewrite step, our tool implements the well-founded *single-pushout approach* (SPO, for explanation see [EHK$^+$99]). We also support the *double-pushout approach* (DPO, for explanation see [CMR$^+$99]).

For ease of use, GRGEN features an expressive specification language and generates program code with a convenient interface. In contrast to systems like Fujaba [Fuj07] our pattern matching algorithm is fully automatic and does not need to be tuned or partly be implemented by hand. GRGEN.NET [Gei07] 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].

1.2 Features of GRGEN.NET

The process of graph rewriting can be divided into four steps: Representing a graph according to a model, searching a pattern aka finding a match, performing changes to the matched spot in the host graph, and, finally, selecting the next rule(s) for application. We have organized the features of GRGEN.NET according to this breakdown of graph rewriting.

- The graph model (meta-model) supports:

  - Directed graphs
  - Undirected and arbitrarily directed edges (will be implemented in version 2.0)
  - Typed nodes and edges, with multiple inheritance on types
  - Node and edge types can be equipped with typed attributes (like structs)
  - Multigraphs (including multiple edges of the same type)
  - 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

- Three modes for the semantics of a rule application additional to SPO: matching of exact patterns only, matching of induced subgraphs only, and DPO semantics
- Attribute conditions (including arithmetic operations on the attributes)
- Type conditions (including powerful instanceof-like type expressions)
- Parameter passing to rules
- Dynamic patterns with iterative or recursive paths and graphs (to be implemented in version 2.0)

- The rewrite language supports:

  - Attribute re-calculation (using arithmetic operations on the attributes)
  - Retyping of nodes/edges (a stronger version of casts known from common programming languages)
  - Creation of new nodes/edges of statically as well as dynamically computed types (some kind of generic templates)
  - Two modes of specification: A rule can either express changes to be made to the match or replace the whole match (the semantics is always mapped to SPO)
  - Returning certain edges/nodes for further computations
  - Copying (duplicating) of elements form the match—comparable with sesqui-pushout rewriting [CHHK06] (to be implemented in version 2.0)
  - Composing several rules with logical and iterative sequence control (called graph rewrite sequences, GRS), including support for nested transactions
  - Emitting user-defined text to `stdout` during the rewrite steps

- The rule application language (GrShell) supports:

  - Various methods for creation/deletion/input/output of graphs/nodes/edges
  - Stepwise and graphic debugging of rule application

- Alternatively to GrShell, you can access the match and rewrite facility through libGr. In this way you can build your own algorithmic rule applications in a .NET language of your choice.

- As of version 2.0 GrGen.NET's functionality will be available as an embedded extension to C# (like embedded SQL)

1.3   System Overview

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

A graph rewrite system[1] is defined by a rule set file (*.grg) and zero or more graph model description files (*.gm). Such a graph rewrite system is generated from these specifications by GrGen.exe and can be used by applications such as GrShell. Figure 1.2 shows the generation process.

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).

---

[1]In this context, system is not a CH0-like grammar rewrite system, but rather a set of interacting software components.
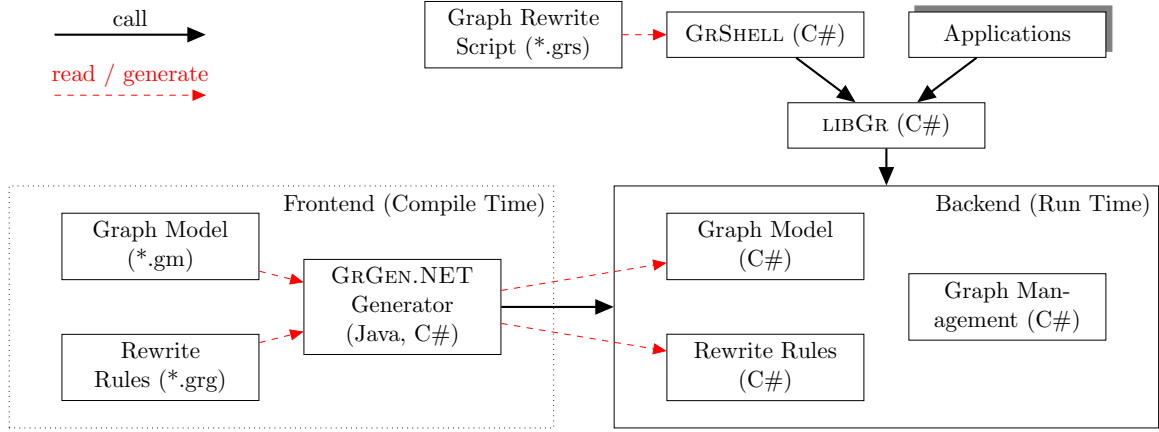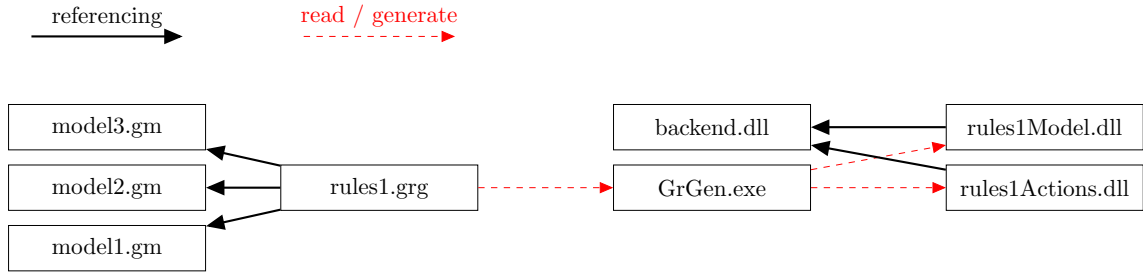
Figure 1.1: GRGEN.NET system components [Kro07]

Figure 1.2: Generating a graph rewrite system

Generating a GRGEN.NET graph rewrite system may be considered as 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—the .NET library LIBGR. For most issues—in particular for experimental purposes—you might rather want to work with the GRSHELL because of its more convenient interface. However, GRSHELL does not provide the full power of the LIBGR; see also note 10 on page 31.

1.4 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. In the simplest case such a graph rewrite rule consists of a tuple $L \to R$, 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*.

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: $H \xRightarrow{m,p} H'$.
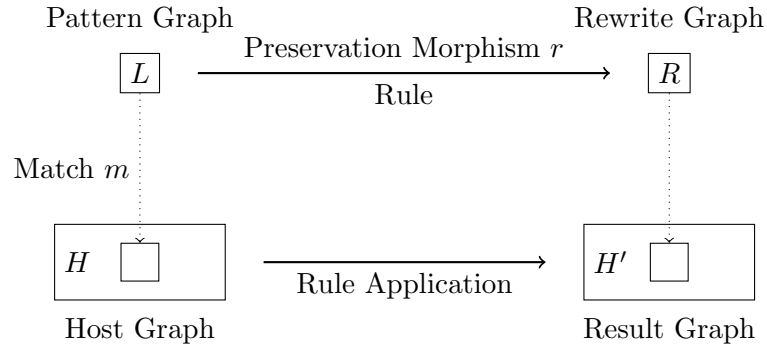
Pattern Graph   Rewrite Graph

L $\xrightarrow{\displaystyle \text{Preservation Morphism } r}$ R
                        Rule

Match m

$H \;\; \square$ $\xrightarrow{\text{Rule Application}}$ $H' \;\; \square$

Host Graph   Result Graph

Figure 1.3: Basic Idea of Graph Rewriting

1.5   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
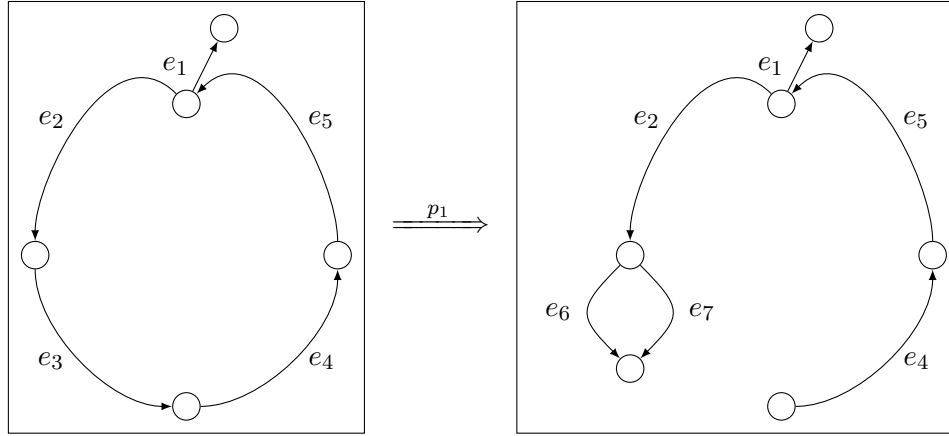
$p_0 : \; \emptyset \; \longrightarrow$
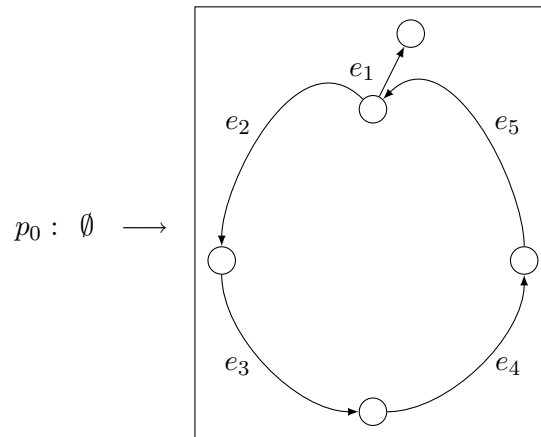
to the empty host graph. As the result we get an apple as new host graph H. Now we want to rewrite our apple with stem to an apple with a leaflet. So we apply

$p_1 :$
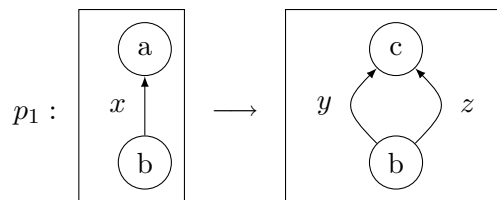
[2]Because of the uniqueness of the total and totally undefined morphism.

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

What happened? GRGEN.NET has arbitrarily chosen one match out of the set of possible matches, because x matches edge e_3 as well as e_1. A correct solution could make use of edge type information. We have to change rule p_0 to generate the edge e_1 with a special type "stem". And this time we will even keep the stem. So let
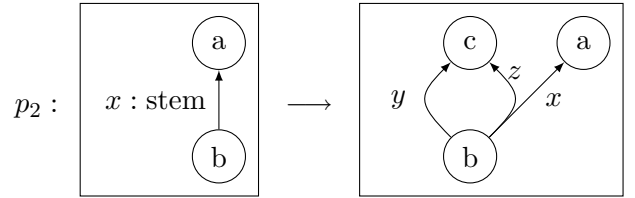
If we apply p_2 to the modified H_1 this leads to
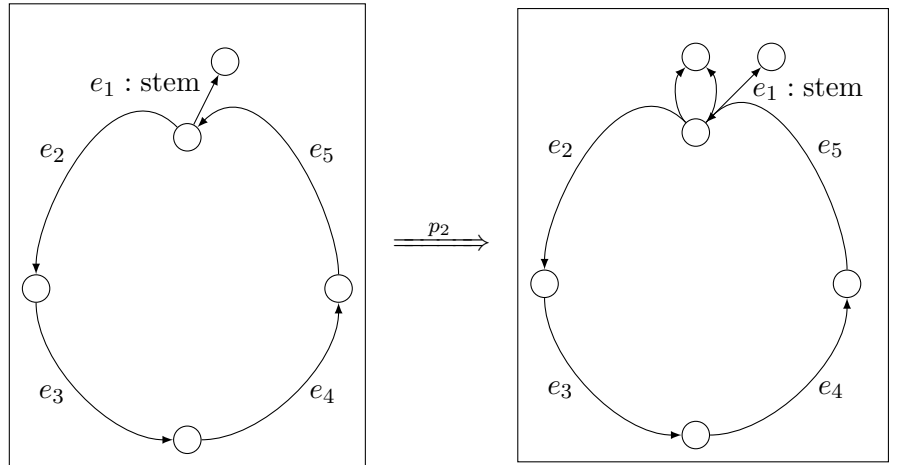
1.6 The Tools

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.

1.6.1 GrGen.exe

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

Usage

```
[mono] GrGen.exe [-keep [<dest-dir>]] [-use <existing-dir>]
                   [-debug] [-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 the C# source file of the *rule-set*'s actions assembly. This file is for internal debug purposes only.
  - *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.
- `-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).

1.6.2 GrShell.exe

The GRSHELL is a shell application 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 6.

Usage

```
[mono] grShell.exe [-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.

Options

- `-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).

1.6.3 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/.

---

[3]n is an increasing number.

1.6.4 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>]
```

The *graph-file* may be any graph file in a supported format. YCOMP will open this file on startup.

Hints

Do not use the compiler graph layout algorithm (YCOMP's default setting). Instead `organic` or `orthogonal` might be good choices. Use the rightmost blue play button to start layout process. This may take a while, depending on the graph size:
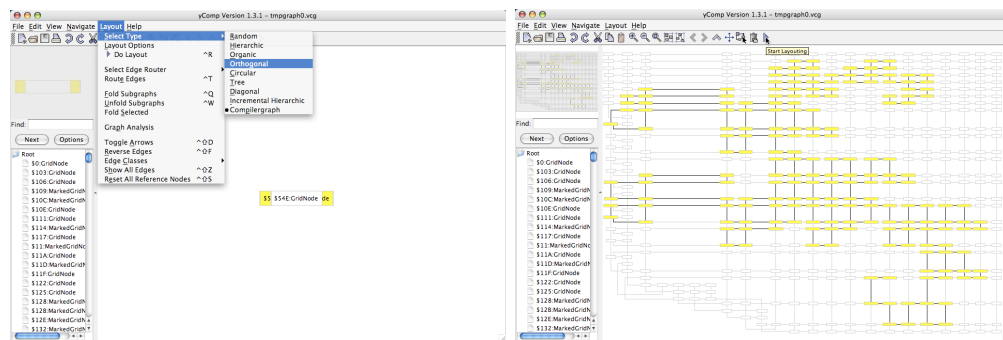
Requires

Java Runtime Environment 1.5 (or above).

In this chapter we'll build a GRGEN.NET system from scratch. You should already have read chapter 1 to have a glimpse of the GRGEN.NET system and its components. We will use GRGEN.NET to construct non-deterministic state machines. We further show some actual graph rewriting by removing ε-transitions from our state machines. This is not too much about details but rather about the GRGEN.NET look and feel.

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

- Java 1.5 or above

- Mono 1.2.3 on Unix-like platforms / .NET 2.0 or above on Microsoft Windows

Unpack the package to a directory of your choice, for example into `/opt/grgen` and `/opt/ycomp`:

```
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`.

2.2 Creating a Graph Model

In the directory `/home/grgen` we create a text file `state_machine.gm` that contains the graph meta model for our state machine. 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 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.

```
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

2.3 Creating Graphs

Let's test our graph meta model by creating a state machine graph. We will use the GRSHELL (see chapter chapgrshell) 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 `remove_epsilons.grg` in the `/home/grgen` directory, containing only the line

```
1  using state_machine;
```

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 `remove_epsilons.grs`, in the `/home/grgen` directory. Figure 2.2 shows this script. Run the script by executing `grshell remove_epsilons.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 you get a window similiar

```
1  new graph remove_epsilons "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

to figure 2.3 after clicking the blue "layout graph" button on the very right side of the button bar (see also section 1.6.4). 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`.

This script starts with creating an empty graph of the meta model `StateMachine` (which is referenced by the rule set `remove_epsilons.grg`) with the name `StateMachineGraph`.
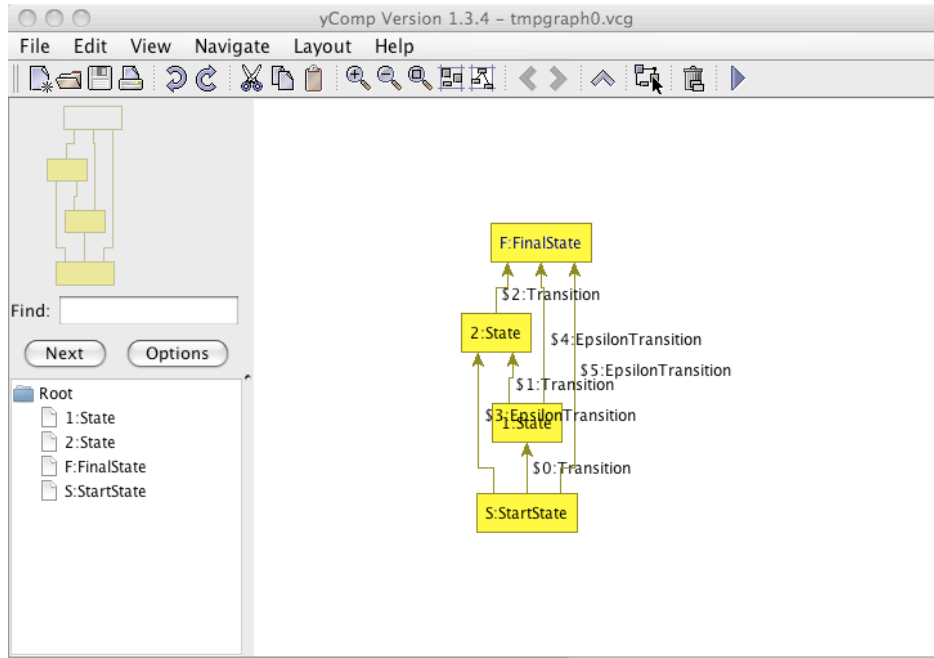
Figure 2.3: A first state machine

Thereafter we create nodes and edges. The double point 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 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 actual rewrite rules to the rule set file `remove_epsilons.grg`. The idea is to "forward" the ε-transitions one after another, i.e. if we have a pattern like `a:State -EpsilonTransition-> b:State -e:Transition-> c:State` we forward to `a -e-> c`. After all such transitions are forwarded we can remove the ε-transitions 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 has 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 operations. 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 couse 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, as we have already seen it 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). Such a 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 general rule: If you want to do something with your found graph element, define a name; otherwise an anonymous graph element will do fine. Also

```
 1 using state_machine;
 2
 3 test CheckStartState {
 4     x: StartState;
 5     negative {
 6         x;
 7         y: StartState;
 8     }
 9 }
10
11 test CheckDoublettes {
12     negative {
13         x:State -e:Transition-> y:State;
14         hom(x,y);
15         x -doublette:Transition-> y;
16         if { typeof(doublette) == typeof(e); }
17         if { ((typeof(e) == EpsilonTransition) || (e.Trigger == doublette.Trigger)); }
18     }
19 }
20
21 rule ForwardTransition {
22     x:State -:EpsilonTransition-> y:State -e:Transition-> z:State;
23     hom(x,y,z);
24     negative {
25         x -exists:Transition-> z;
26         if { typeof(exists) == typeof(e); }
27         if { ((typeof(e) == EpsilonTransition) || (e.Trigger == exists.Trigger)); }
28     }
29     modify {
30         x -forward:typeof(e)-> z;
31         eval { forward.Trigger = e.Trigger; }
32     }
33 }
34
35 rule AddStartFinalState {
36     x:StartState -:EpsilonTransition-> :FinalState;
37     modify {
38         y:StartFinalState<x>;
39         emit ("Start state (" + x.id +  ") mutated into a start-and-final state");
40     }
41 }
42
43 rule AddFinalState {
44     x:State -:EpsilonTransition-> :FinalState;
45     if { typeof(x) < SpecialState; }
46     modify {
47         y:FinalState<x>;
48     }
49 }
50
51 rule RemoveEpsilonTransition {
52     -:EpsilonTransition->;
53     replace {}
54 }
```

Figure 2.4: Rule set for removing ε-transitions

have a look at example 8 on page 27 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 in the replace part. The modify part deletes nothing (by default), but just adds or adjusts graph elements.

What else can we do? We have negative application conditions (NACs), expressed by `negative {...}`; this prevents a rule to be applicated 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 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 kind of a type cast.

Now that we have the rules as rewrite primitives we have to compose them to a sequence of rules. For instance we don't want to forward just one ε-transition as `ForwardTransition` would do; we want to forward them all. Such a rule composing is done by a *graph rewrite sequence* (see section 4.6). We add the following line to our shell script `remove_epsilons.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 successful evaluated if a match could be found. We first check for a valid state machine, i.e. if the host graph has 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 disjunctive 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

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.

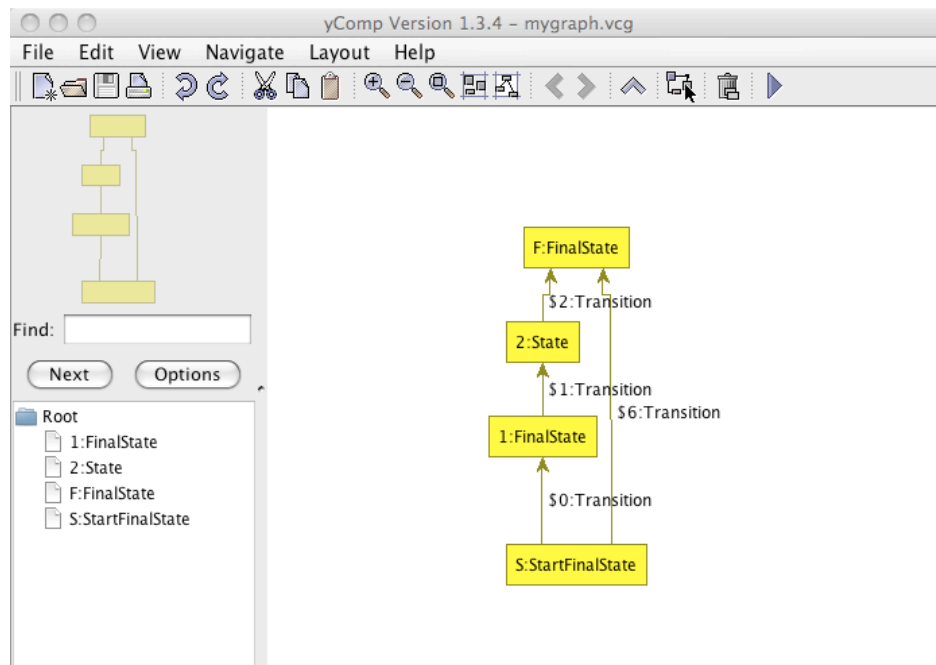Figure 2.5: A state machine without ε-transitions

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.

Attributes
>   Nodes and edges can 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 nodes, we include connection assertions. Furthermore the number of outgoing and incoming edges can be constrained.

In this chapter as well as in chapter 6 (GRSHELL) we use excerpts of Example 1 (the `Map` model) for illustration purposes.

3.1 Building Blocks

NOTE (1)

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 abandoned_city extends city;
14 node class ghost_town extends abandoned_city;
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 5.4 for annotations of declarations.

NOTE (2)

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.

3.2   Type Declarations

GraphModel
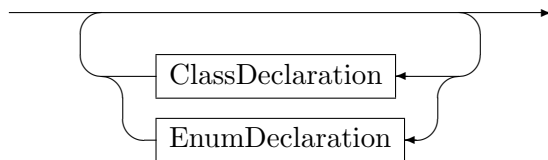
The graph model consists of zero or multiple type declarations. Whereas *ClassDeclaration* defines a node type or an edge type, *EnumDeclaration* defines an enum type for use as attribute of nodes or edges. Like all identifier definitions, types do not need to be declared before they are used.

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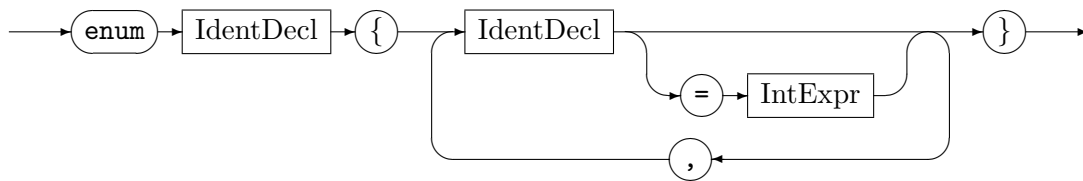
Defines an enum type. An enum type is a collection of so called *enum items* that are associated with integral numbers, each. Accordingly, a GRGEN.NET enum is internally represented as `int` (see Section 5.1).

NOTE (3)

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

NOTE (4)

Normally, assignments of `int` values to something that has an enum type are forbidden (see section 5.1). Only inside a declaration of an enum type an int value may be assigned to the enum item that is currently declared. This also includes the usage of items taken from other enum types (because they are implicitly casted to `int`). However, items from other enum types must be written fully qualified in this case (which, e.g., looks like `my_enum::a`, where `my_enum` 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 (5)

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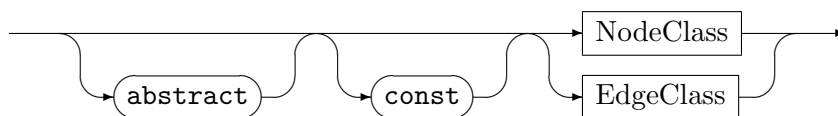
Defines a new node type or edge type. The keyword `abstract` indicates that you cannot instantiate graph elements of this type. Instead you have to derive non-abstract types to create graph elements. The abstract-property will not be inherited by subclasses, of course.

EXAMPLE (3)

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

```
1  abstract node class city {
2      size: int;
3  }
4  abstract node class abandoned_city extends city;
5  node class ghost_town extends abandoned_city;
```

You will be able to create nodes of type **ghost_town**, but not of type **city** or **abandoned_city**. However, nodes of type **ghost_town** are also of type **abandoned_city** 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.

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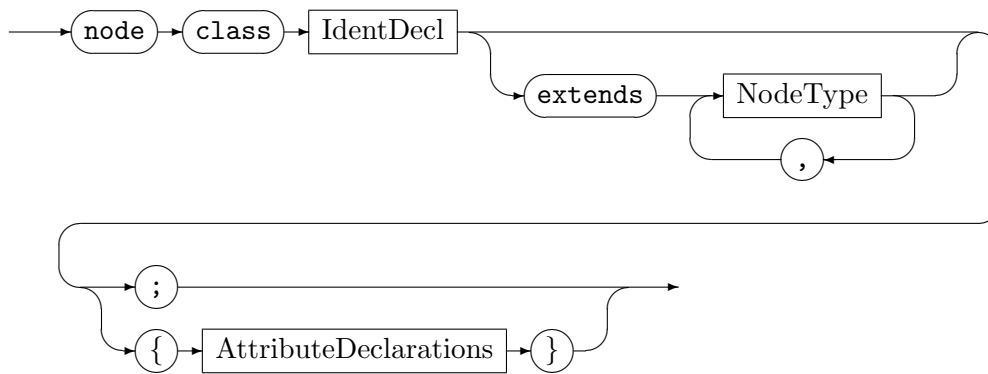
Defines a new node type. Node types can inherit from other node types defined within the same file. If the `extends` clause is omitted, *NodeType* will inherit from the built-in type `Node`. Optionally nodes can possess attributes.

EdgeClass
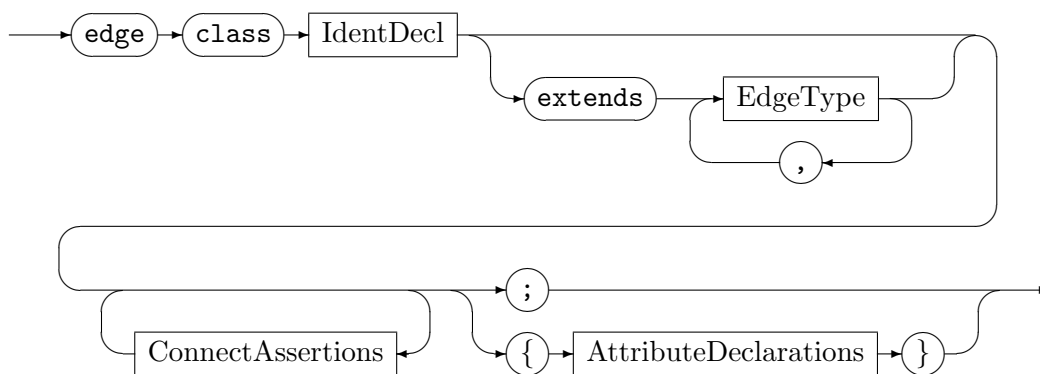
Defines a new edge type. Edge types can inherit from other edge types defined within the same file. If the `extends` clause is omitted, *EdgeType* will inherit from the built-in type `Edge`. Optionally edges can possess attributes. A *connection assertion* specifies that certain edge types should only connect specific nodes. Moreover, the number of outgoing and incoming edges can be constrained.

NOTE (6)

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 6.2.2, `validate`.

ConnectAssertions
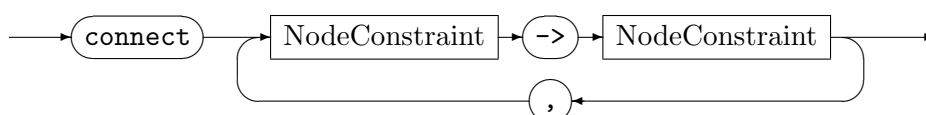
NodeConstraint
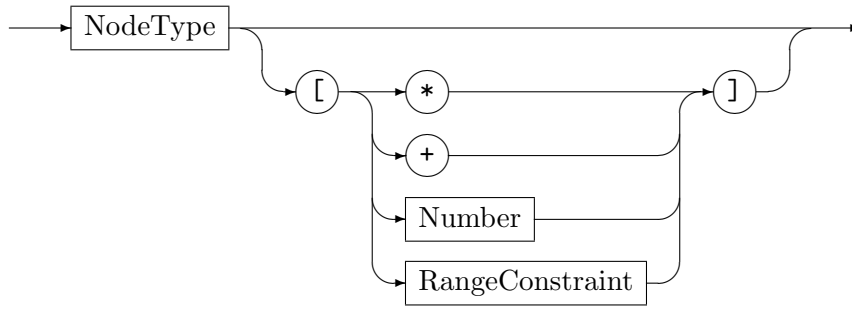
RangeConstraint
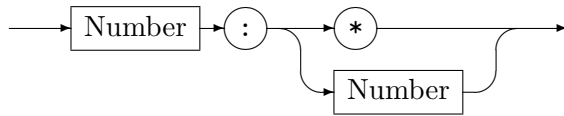
A connection assertion is denoted as a pair of node types in conjunction with their multiplicities. A corresponding edge may connect a node of the first node type or one of its subtypes (source) with a node of the second node type or one of its subtypes (target). The multiplicity is a constraint on the out-degree and in-degree of the source and target node type, respectively. *Number* is an `int` constant as defined in Section 5.2. See Section 6.2.2, `validate`, for an example. Table 3.1 describes the multiplicity definitions. Multiple connection assertions are connected by conjunction, i.e. all of them must be fulfilled.

NOTE (7)

Connection assertions must be consistent according to the type hierarchy. That means—roughly spoken—the assertions for subtypes must get monotonously more restrictive or monotonously less restrictive. For instance, the following connection assertions are illegal for node types `A`, `B`, `C`, `D` with `A` inherits from `B` and `C` inherits from `B`:

```
1  connect A[*] -> D, B[0:5] -> D, C[*] -> D
```

The GRGEN.NET compiler will warn you, if you're using inconsistent connection assertions.

[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

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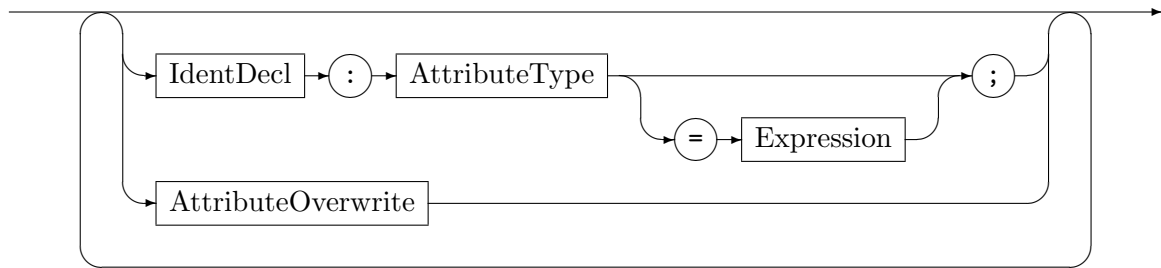
AttributeType
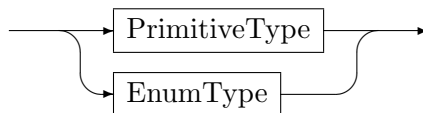
AttributeOverwrite
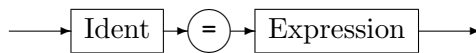
Defines a node or edge attribute. Possible types are enumeration types (`enum`) and primitive types. See Section 5.1 for a list of built-in primitive types. Optionally attributes may be initialized with a constant expression. The expression has to be of a compatible type of the declared attribute, of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

EXAMPLE (4)

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

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

CHAPTER 4

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 (5)

```
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 {
22       n5.a3 = n3.a1*n1.a2;
23     }
24   }
25 }
```

In this chapter we use excerpts of Example 5 (`SomeRule`) for illustration purposes.

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 5.4 for annotations of declarations.

NOTE (8)

As in the GRGEN.NET model language (see note 2) every declaration is also a definition. Using an identifier before defining it is allowed. Every used identifier has to be defined exactly once.

ModelIdent, TypeIdent, NodeType, EdgeType

These are (semantic) specializations of *Ident*. *TypeIdent* matches every type identifier, i.e. a node type, an edge type, an enumeration type or a primitive type. All the type identifiers are actually type *expressions*. See Section 5.3 for the use of type expressions.

Graphlet
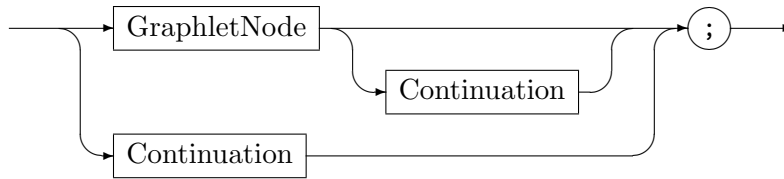
Continuation
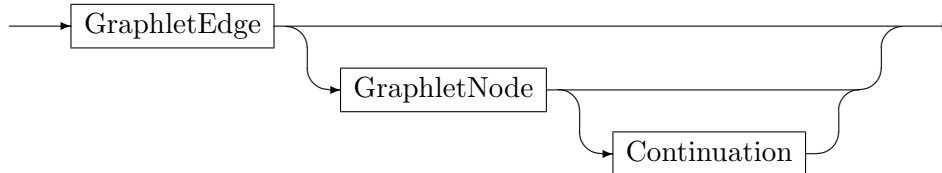
A graphlet specifies a connected subgraph. GRGEN.NET provides graphlets as a descriptive notation to define both, patterns to search for as well as the subgraphs that replace or modify matched spots in a host graph. Any graph can be specified piecewise by a set of graphlets. In Example 5, line 7, the 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 (6)

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 contrast to pure syntactic scoping, the replace/modify part is a direct inner scope of the pattern part. In Example 5, 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
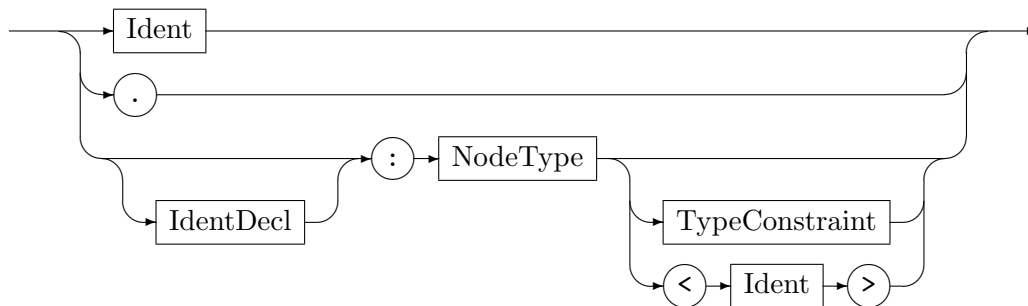
Specifies a node of type *NodeType* with respect to *TypeConstraint* (see Section 5.3, *TypeConstraint*). Type constraints are allowed in the pattern part only. The `.` is an anonymous node of the base type `Node`. Remember that every node type has `Node` as super type. The `<>` operator retypes a node. Retyping is allowed in the replace/modify part only (see Section 4.4, *Retyping*).

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
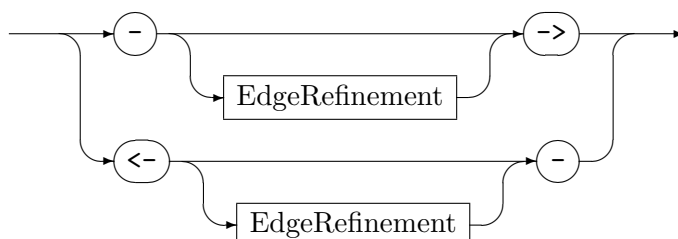
EdgeRefinement
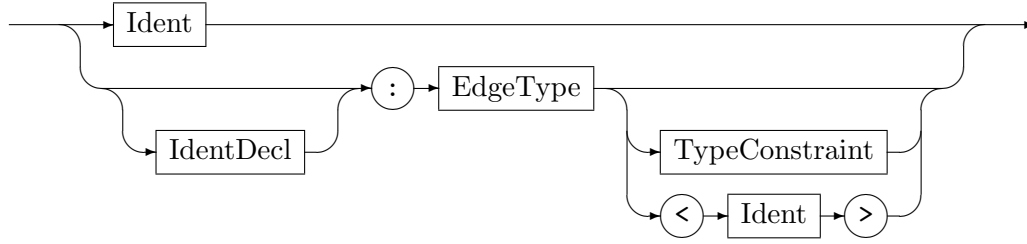
Specifies an edge. Anonymous edges are specified by `-->` or `<--` resp. `-:T->` or `<-:T-` for an edge type `T`. For a more detailed specification you can use the inplace notated *EdgeRefinement* clause. Type constraints are allowed in the pattern part only (see Section 5.3, *TypeConstraint*). The `<>` operator retypes an edge. Retyping is allowed in the replace/modify part only (see Section 4.4, *Retyping*).

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-> ;`
`-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.

EXAMPLE (7)

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.

---

[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 (8)

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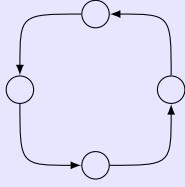
```
x:Node --> . --> . --> . --> x;
```
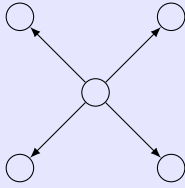
```
. <-- x:Node --> .;
. <-- x --> .;
```
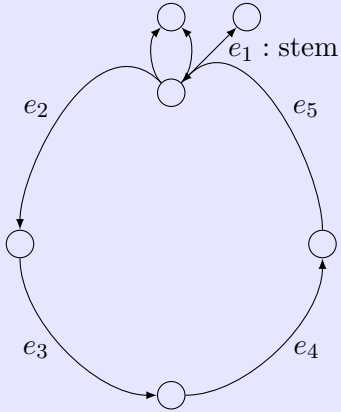
```
. <-e1:stem- n1:Node -e2:Edge-> . -e3:Edge-> .
  -e4:Edge-> . -e5:Edge-> n1;
n1 --> n2:Node;
n1 --> n2;
```

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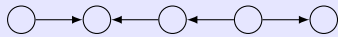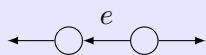
```
-e:Edge->
<-- . <-e- . --> ;
```

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`.
`x:Node; negative {y:Node; hom(x,y)}`	Here `x` must not occur in the `hom` statement. See Section 4.3 for further information.
`<-- --> ;`	There must be at least a node between the edges.

NOTE (9)

Although both, the pattern part and the replace/modify part use graphlets, there are subtle differences between them. These concern the *TypeConstraint* clause, the retype operator `<>`, and the scope of defined graph element names: Names defined within the pattern part are valid in the pattern part as well as in the replace/modify part. Names defined within the replace/modify part are unknown to the pattern part.

4.2 Rules and Tests

The structure of a rule set file is as follows:

RuleSet
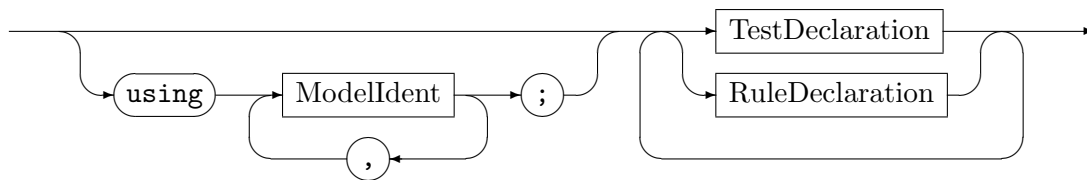
A rule set consists of the underlying graph models and several rewrite rules. In case of multiple graph models, GrGen.NET uses the union of these models. In this case beware of conflicting declarations. There is no built-in conflict resolution mechanism like packages or namespaces for models. If necessary you can use prefixes as you might do in C.

TestDeclaration
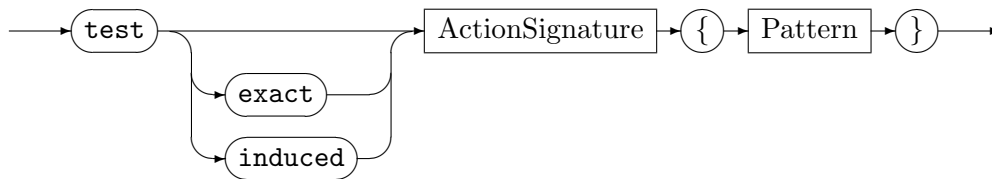
RuleDeclaration
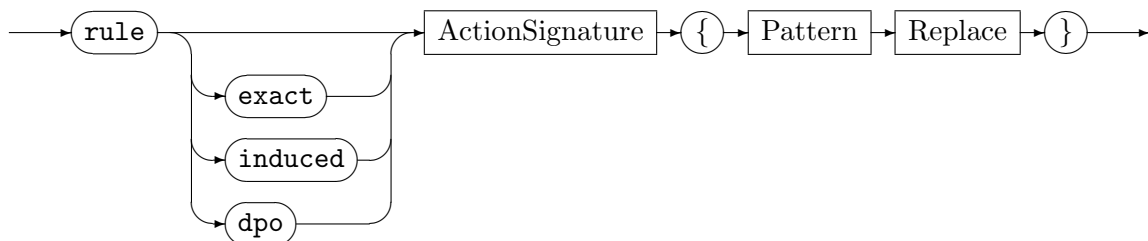
Declares a single rewrite rule such as `SomeRule`. It consists of a pattern part (see Section 4.3) in conjunction with its rewrite/modify part (see Section 4.4). A *test* has no rewrite specification. It's intended to check whether (and maybe how many times) a pattern occurs (see example 9). For an explanation of the `exact`, `induced`, and `dpo` pattern modifiers see section 4.5.

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 6.2.6.

ActionSignature
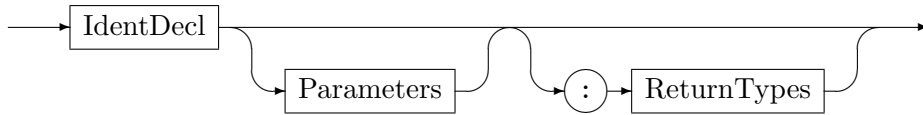
The signature sets the name of a rewrite rule to *IdentDecl* and optionally names and types of formal parameters as well as a list of return types. Parameters and return types provide users with the ability to exchange graph elements with rule. This is similar to parameters of procedural languages.

Parameters
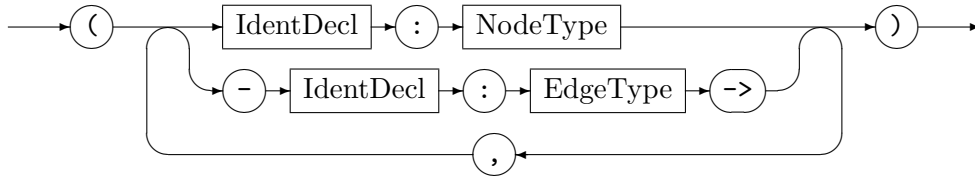
Within a rule, parameters are treated as (predefined) graph elements of the pattern. Even if a supplied parameter value is undefined, it is treated as valid node or edge definition. So in any case a graph element of the specified type has to be mapped. GRGEN.NET assumes the lookup operation for parameters to be in $\mathcal{O}(1)$. In case of an undefined parameter value this might lead to bad search plans, because GRGEN.NET has to actually search for such a graph element.

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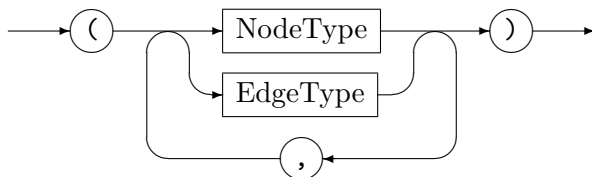
The return types specify edge and node types of graph elements that are returned by the replace/modify part. If return types are specified, the `return` statement is mandatory. Otherwise no `return` statement must occur. See also Section 4.4, `return`.

EXAMPLE (10)

Assume the following test that checks whether the edge e is not incident to x:

```
1  test r(-e:Edge->, x:Node) {
2    negative {
3      -e-> x;
4    }
5    negative {
6      x -e->;
7    }
8  }
```

If x and e are undefined, test r is equivalent to test s:

```
1   test s {
2     x:Node;
3     -e:Edge->;
4     negative {
5       -e-> x;
6     }
7     negative {
8       x -e->;
9     }
10  }
```

In particular, test s is successful if there is *some* edge in the host graph that is *not* incident to x.

EXAMPLE (11)

We extend SomeRule (Example 5) 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.

4.3 Pattern Part

Pattern

A pattern consists of zero or more pattern statements. 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`, `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 (10)

The application of a rule is not deterministic (remember the example of the introduction in Section 1.5); 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. LIBGR allows for splitting a rule application into two steps: Find all the subgraphs of the host graph that match the pattern and 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, IGraphElement[] parameters);
IGraphElement[] 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;
   }
}
```

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 Section 4.6 for further information on graph rewrite sequences.

PatternStatement
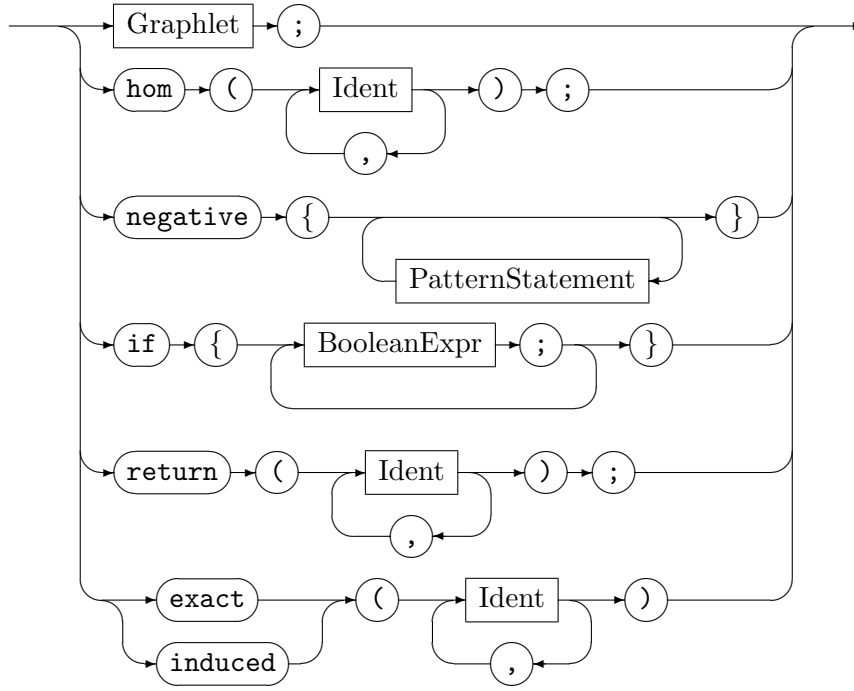
The semantics of the various pattern statements are given below:

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 5 nodes `n1` and `n2` may be the same node. This is possible because they are of the same type (`NodeTypeA`). A name may not occur in multiple `hom` statements. For instance it is illegal to write `hom(a, b); hom(b, c);`. Instead write `hom(a, b, c);`. Inside a NAC the `hom` operator may only operate on graph elements that are either defined or used in the NAC.

Negative Application Conditions (NACs)

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 must not be nested. NACs possess a scope of their own. Names defined within a NAC do not exist outside the NAC. Identifiers from surrounding scopes must not be redefined. In general NACs do not care about bindings within the outer scope. Nevertheless, if you use an identifier that is defined in the outer scope, this specifies exactly the graph element, the identifier is bound to in the outer scope.

> **EXAMPLE (12)**
>
> We specify a node `x` which is not connected to a node of type `BadType`:
>
> ```
> 1  x:Node;
> 2  negative {
> 3    x --> :BadType;
> 4  }
> 5  negative {
> 6    x <-- :BadType;
> 7  }
> ```

Attribute Conditions

    The Java-like attribute conditions (keyword `if`) in the pattern part allow for further restriction of the applicability of a rule.

Return values

    The return statement is only allowed for tests. Otherwise the `return` statement belongs to the replace part. See Section 4.4, *Return Values*.

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.

Keep in mind that using type constraints or the `typeof` operator might be helpful. See Section 5.3 for further information.

4.4 Replace/Modify Part

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

4.4.1 Implicit Definition of the Preservation Morphism r

In theory the transformation specification is done by defining the preservation morphism r. In GRGEN.NET the preservation morphism r is defined implicitly by using names both in pattern graphlets and replace graphlets. Remember that to each of the graph elements a name is bound to, either user defined or internally defined. If such a name is used in a replace graphlet, the denoted graph element will be kept. Otherwise the graph element will be deleted. By defining a name in a replace graphlet a corresponding graph element will be newly created. So in a replace pattern anonymous graph elements will always be created. Using a name multiple times has the same effect as a single using occurrence. In case of a conflict between deletion and preservation, deletion is prioritized. If an incident node of an edge gets deleted, the edge will be deleted as well (in compliance to the SPO semantics).

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*.
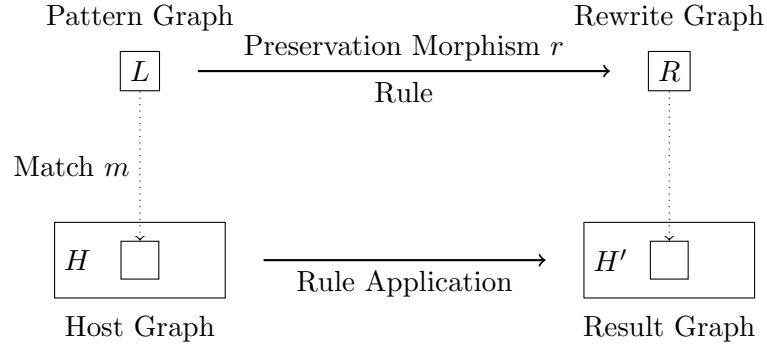
Figure 4.1: Process of Graph Transformation

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

Table 4.1: Definition of the preservation morphism r

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 11 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).

4.4.3 Syntax

Replace
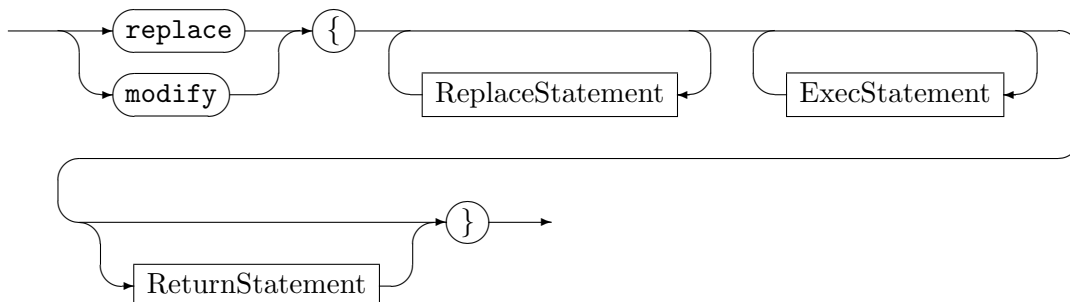
EXAMPLE (13)

How might Example 11 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        ...
```

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
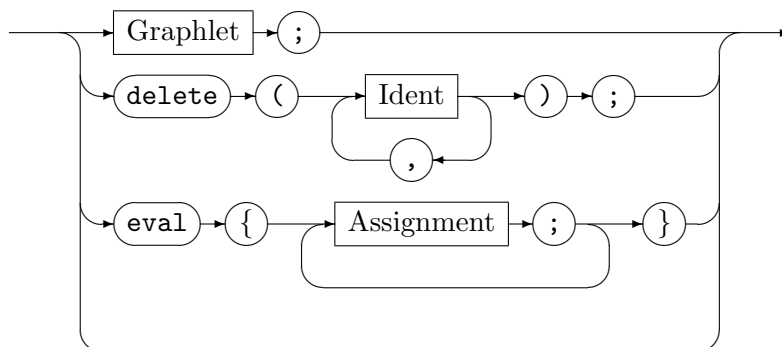
The semantics of the various replace statements are given below:

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.

Retyping

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 4.2 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 5 node n5 is a retyped node stemming from node n1. Note, that—conceptually—the retyping is performed *after* the SPO conform rewrite.

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 4.2: Retyping of preserved nodes and edges

Assignment
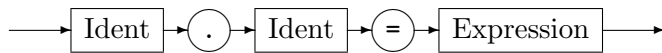
Several evaluation parts are allowed within the replace part. Multiple evaluation statements will be internally concatenated, preserving their order. Evaluation statements have imperative semantics. In particular, GRGEN.NET does not care about data dependencies. Evaluation takes place before any graph 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.

EXAMPLE (14)

```
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    }
```

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

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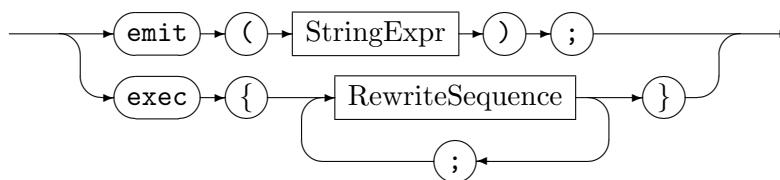
The statements `emit` and `exec` enhance the declarative rewrite part by imperative clauses. That means, these statements are executed in the order as they appear within the rule. The execution statements take place after all the rewrite operations are done, i.e. they operate on the modified host graph.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

Text Output

The `emit` statement prints a string to `stdout`. See section 5.2 for a description of string expressions.

XGRS Execution

The `exec` statement executes a graph rewrite sequence, which is a composition of graph rewrite rules. See section 4.6 for a description of graph rewrite sequences.

ReturnStatement
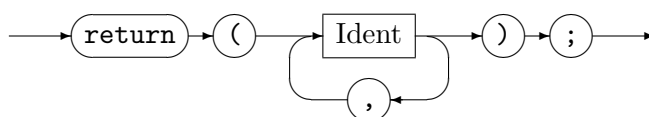
Graph elements of the replace/modify part can be returned according to the return types in the signature (see Section 4.2, `ActionSignature`). The graph element names have to be in the same order as the corresponding return types in the signature. The named elements must be compatible to the declared type.

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 *rule / pattern 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 modifier in front of the `pattern`-keyword is equivalent to one modifier-statement inside the pattern containing all the specified nodes (including anonymous nodes). Table 4.3 lists the modifiers with their semantics. Example 15 explains the modifiers by small toy-graphs.

NOTE (11)

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 $\mathcal{O}(n^2)$, where n is the number of specified nodes.

Modifier	Meaning
`exact`	Switches to the most restricitive mode. An exactly-matched node is matched, iff all its incident edges in the host graph are specified in the pattern. This modifier superseeds all the other modifiers.
`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)`.
`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. Accordingly nodes going to be deleted due to the rewrite part have to be specified exactly (`exact` semantics) in order to be matched. Furthermore if you specify two pattern nodes to be homomorphically matched but only one of them is subject to deletion during rewrite, those pattern nodes will never actually be matched to the same host graph node. In contrast to `exact` and `induced` this modifier applies neither to a pattern as such nor to a single statement but only to an entire rule. See Corradini et al.[CMR+99] for a DPO reference.
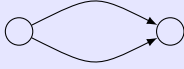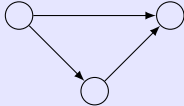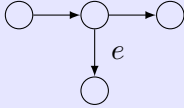
Table 4.3: Semantics of rule and pattern modifiers

4.6   Extended Graph Rewrite Sequences (XGRS)

Graph rewrite sequences are an imperative enhancement to the rule set language. Graph rewrite seuences are syntactically expressed as expressions similar to boolean expressions. Composing single graph rewrite rules to an expression not only yields more complex host graph operations but also determines sort of a control flow by means of evaluation order of the operands. Graph rewrite sequences have a boolean return value; For a single rule, `true` means the rule could successfully applied to the host graph. A `false` return value means that the pattern was not found in the host graph. Graph rewrite sequences can be processed transactionally by using angle brackets (`<>`). If the return value is `false`, all the related operations on the host graph will be rolled back. Nested transactions are supported.

   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 is defined by assigning a graph element, a boolean value, or another variable to it. You may use named

EXAMPLE (15)

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 -->; }` `modify{ delete(x); }`	Produces no match in DPO-mode because of edge `e`

graph elements of the enclosing rule as read-only variables. A parameter is alive from its first assignment until the end of the enclosing `exec` statement.

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. Additionally rules may return graph elements. These return values can be assigned to variables on the fly (see example 16).

EXAMPLE (16)

The graph rewrite sequence

```
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`.

RewriteSequence
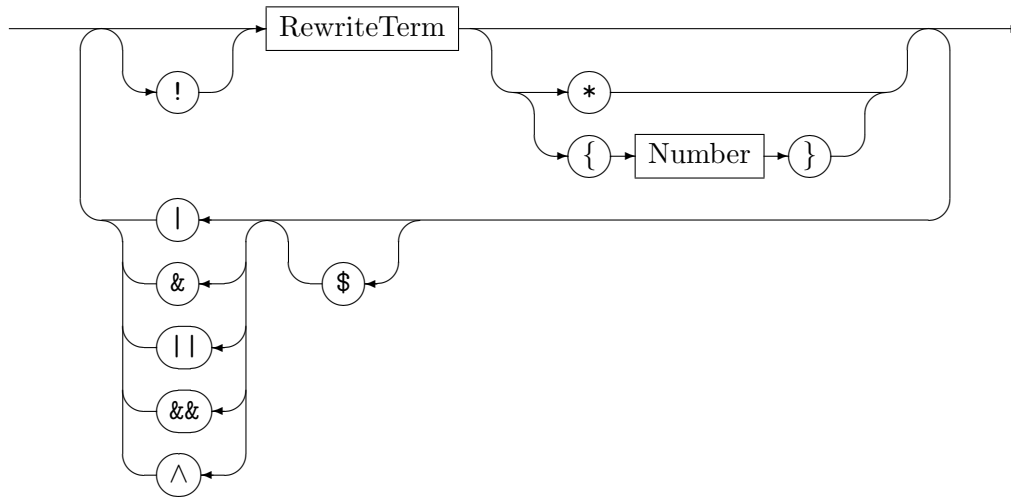
A graph rewrite sequence consists of several, logically linked rewrite terms. The modifier `$`
flags the following operator to act commutative: Usually operands are executed / evaluated
from left to right with respect to bracketing (left-associative). In contrast the sequences `s`,
`t`, `u` in `s $<op> t $<op> u` are executed / evaluated in arbitrary order. A sequence can
further be executed multiple times: The star (`*`) executes a sequence repeatedly as long as
its execution does not fail. Such a sequence always returns `true`. The braces (`{}`) let execute
a sequence repeatedly as long as its execution does not fail but *Number* times at most.

RewriteTerm
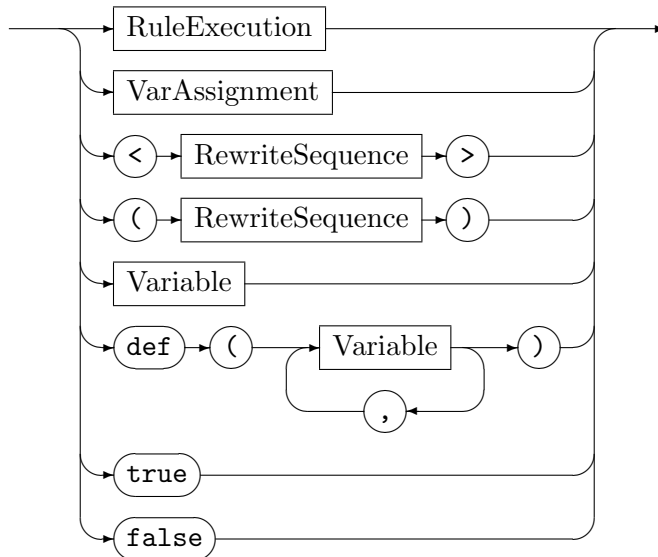
Rewrite terms are the building blocks of graph rewrite sequences. They either act as a rule
application or assign a value to a variable A variable in the *Variable* term must contain a
boolean value, accordingly. A `def` term is successful iff all the the variables are defined.

RuleExecution
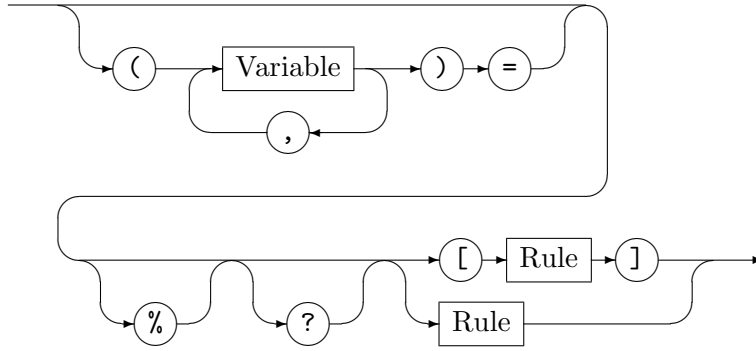
Rule
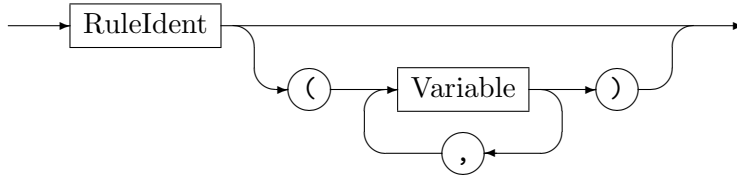
The *RuleExecution* clause applies a single rule or test. In case of a rule, the first found pattern match will be rewritten. Variables and named graph elements from the enclosing rule can be passed. The returned graph elements can be assigned to variables again. The operator ? switches the rule to a 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 6) it dumps the matched graph elements to `stdout`; in `debug`-mode (see section 6.3) 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 1.6.3). The sqare braces (`[]`) introduce a special kind of multiple rule application: Every pattern match produced by the will be rewritten. **Be careful:** Its semantics is not equal to `Rule*`. Instead this operator collects all the matches first before starting rewritings. 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.

VarAssignment
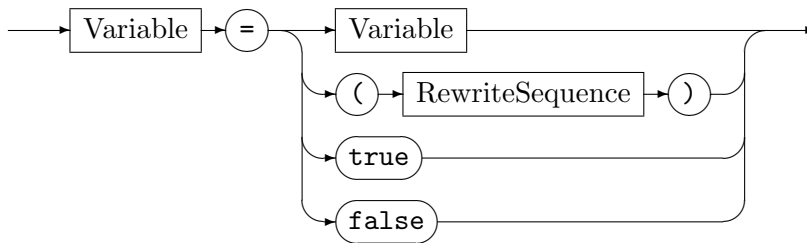
Variables can hold graph elements or boolean values. Graph elments are initially assigned by the *RuleExecution* statement. An *VarAssignment* rewrite term ist always true.

Table 4.4 lists operations of graph rewrite expressions at a glance.

`s	t`	First execute `s` afterwards execute `t`. The sequence `s	t` is successfully executed iff `s` or `t` is successfully executed.
`s || t`	The same as `s	t` but with lazy evaluation, i.e. if `s` is successful, `t` will not be executed,
`s & t`	First execute `s`, afterwards execute `t`. The sequence `s & t` is successfully executed iff `s` and `t` is successful.
`s && t`	The same as `s & t` but with lazy evaluation, i.e. if `s` fails, `t` will not be executed.
`s ^ t`	First execute `s` afterwards execute `t`. The sequence `s ^ t` is successfully executed iff `s` is successful or `t` successfully but not both; i.e. this an XOR sequence.
`<s>`	Execute `s` transactionally.
`!s`	Switch the result of `s` from successful to fail and vice versa.
`$<op>`	Flag the operator `<op>` as commutative.
`s *`	Execute `s` repeatedly as long as its execution does not fail.
`s {n}`	Execute `s` repeatedly as long as its execution does not fail but `n` times at most.
`?Rule`	Switches *Rule* to a test.
`%Rule`	This is the multi-purpose flag when accessed from . 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.

Let `s`, `t`, `u` be graph rewrite sequences, `v`, `w` variable identifiers, `<op>` $\in$ $\{|, ||, \&, \&\&\}$ and `n` $\in \mathbb{N}_0$.

Table 4.4: GRS expressions at a glance

EXAMPLE (17)

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) = DuplicateNode(s) & ConnectNeighbors(s, x)* }
7      exec { [DuplicateCriticalEdge] }
8      exec { MaxCapacityIncidentEdge(t)* }
9      emit ("Redundancy added");
10   }
11 }
```

CHAPTER 5

TYPES AND EXPRESSIONS

In the following sections *Ident* refers to an identifier of the graph model language (see Section 3.1) or the rule set language (see Section 4.1). *TypeIdent* is an identifier of a node type or an edge type, *NodeOrEdge* is an identifier of a node or an edge.

5.1 Built-In Types

Besides user-defined node types, edge types, and enumeration types, GRGEN.NET supports the built-in primitive types in Table 5.1. The exact type format is backend specific. The LGSPBackend maps the GRGEN.NET primitive types to the corresponding C# primitive types. Table 5.2 lists GRGEN.NET's implicit type casts and the allowed explicit type casts.

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

Table 5.1: GRGEN.NET built-in primitive types

Of course you are free to express an implicit type cast by an explicit type cast as well as "cast" a type to itself.

According to table 5.2 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.

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

Table 5.2: GRGEN.NET type casts

43

EXAMPLE (18)

- 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 = (myenum1_type) int; and myenum2 = (myenum2_type) myenum1; where
  ```
  `myenum1` and `myenum2` are different enum types.

NOTE (12)

Unlike an `eval` part (which must not contain assignments to node or edge attributes) the declaration of an enum type can contain assignments of `int` values to enum items (see section 3.2). The reason is, that the range of an enum type is just defined in that context.

5.2 Expressions

Expression
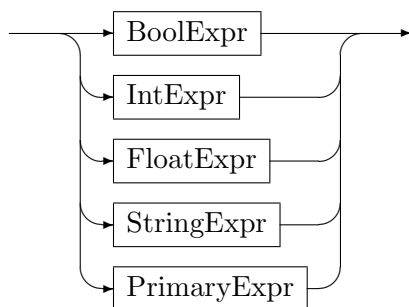
BoolExpr
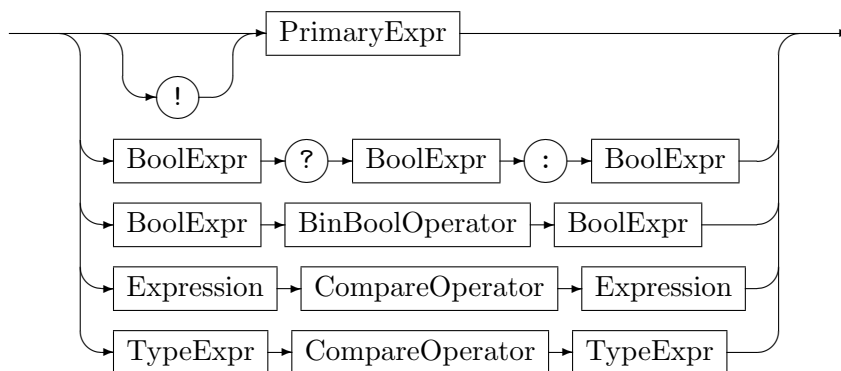
The `!` operator negates a Boolean. Table 5.4 lists the binary operators for Boolean expressions. The `?` operator is a simple if-then-else: If the first *BoolExpr* is evaluated to `true`, the operator returns the second *BoolExpr*, otherwise it returns the third *BoolExpr*. The *BinBoolOperator* is one of the operators in Table 5.4. The *CompareOperator* is one of the following operators:

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

These operators are supported by `int` types and `float`/`double` types (but 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. Table 5.3 describes the semantics of compare operators on type expressions.

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

Table 5.3: Compare operators on type expressions

NOTE (13)

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

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

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

IntExpr
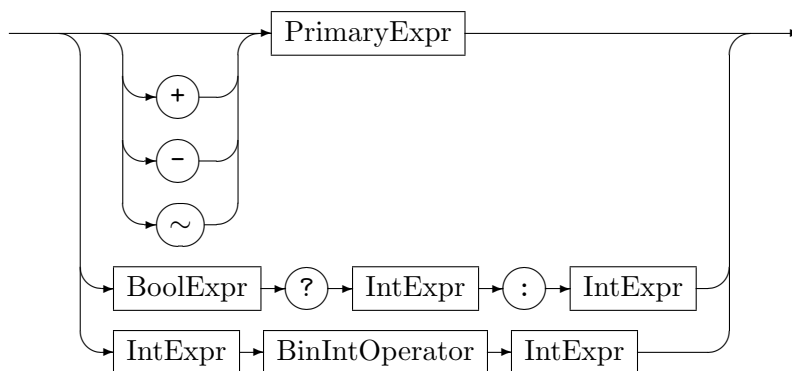
The $\sim$ operator is the bitwise complement. That means every bit of an integer value will be flipped. The `?` operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *IntExpr*, otherwise it returns the second *IntExpr*. The *BinIntOperator* is one of the operators in Table 5.5.

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

Table 5.5: Binary integer operators, in ascending order of precedence

FloatExpr
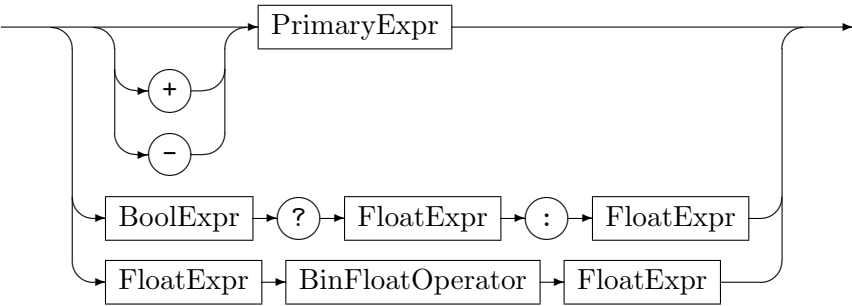
The `?` operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *FloatExpr*, otherwise it returns the second *FloatExpr*. The *BinFloatOperator* is one of the operators in Table 5.6.

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

Table 5.6: Binary float operators, in ascending order of precedence

NOTE (14)

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

StringExpr
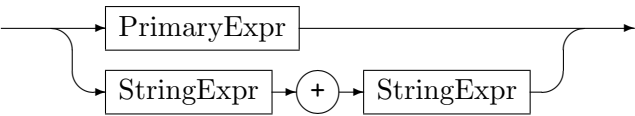
The operator `+` concatenates two strings.

PrimaryExpr
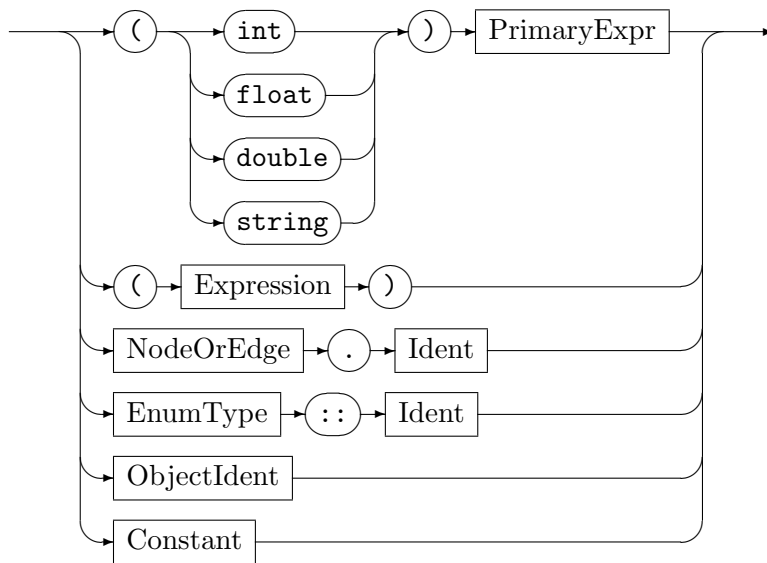
Constant
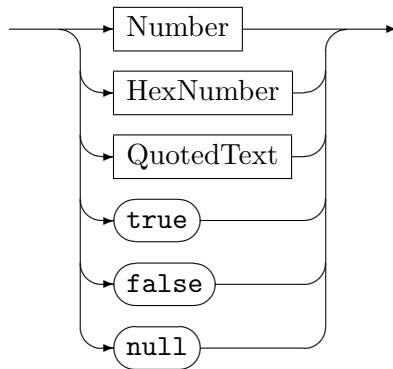
Number

> Is an `int`, `float`, or `double` constant in decimal notation.

HexNumber

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

QuotedText

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

5.3 Type Related Conditions

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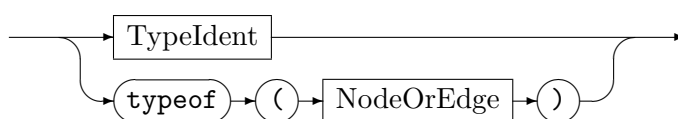
A type expression identifies a type (and—in terms of matching—also its subtypes). A type expression is either a type identifier itself or the type of a graph element. The type expression `typeof(x)` stands for the type of the host graph element `x` is actually bound to.

EXAMPLE (19)

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.

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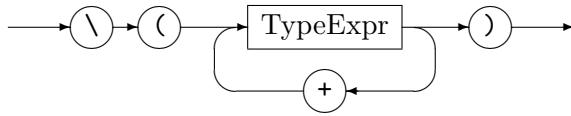
A type constraint is used to exclude parts of the type hierarchy. The operator `+` is used to create a union of its operand types. So the following pattern statements are identical:

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

EXAMPLE (20)
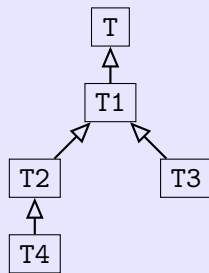
The expression `T\(T2+T3)` applied to the type hierarchy on the left side yields only the types `T` and `T1` as valid.

5.4   Annotations

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

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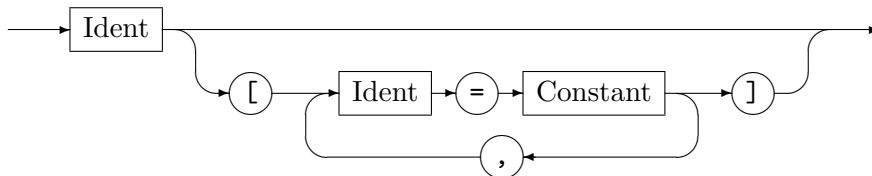
Although you can use any key-value pairs between the brackets, only the identifier has an effect so far.

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 5.7: Annotations

EXAMPLE (21)

We search the pattern `v:NodeTypeA -e:EdgeType-> w:NodeTypeB`. We have a host graph with about 100 nodes of `NodeTypeA`, 1,000 nodes of `NodeTypeB` and 10,000 edges of `EdgeType`. Furthermore we know that between each pair of `NodeTypeA` and `NodeTypeB` there exists at most one edge of `EdgeType`. GrGen.NET can use this information to improve the initial search plan if we adjust the pattern like `v[prio=10000]:NodeTypeA -e[prio=5000]:EdgeType-> w:NodeTypeB`.

CHAPTER 6

GRSHELL LANGUAGE

GRSHELL is a shell application 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.

6.1 Building Blocks

GRSHELL is case sensitive. A comment starts with a `#` and is terminated by end-of-line or end-of-file. Any text left of the `#` will be treated as a statement. 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 5.1).

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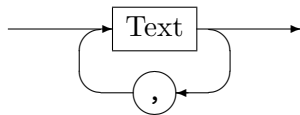
SpacedParameters
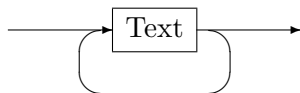
In order to describe the commands more precisely, the following (semantic) specializations of *Text* are defined:

Filename
    A fully qualified file name without spaces (e.g. `/Users/Bob/amazing_file.txt`) or a single quoted or double quoted fully qualified file name that may contain spaces (`"/Users/Bob/amazing file.txt"`).

Variable
>   Identifier of a variable that contains a graph element.

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
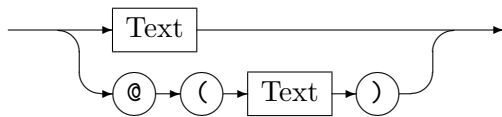
The elements of a graph (nodes and edges) can be accessed both by their variable identifier and by their *persistent name* specified through a constructor (see Section 6.2.3). The specializations *Node* and *Edge* of *GraphElement* require the corresponding graph element to be a node or an edge respectively.

EXAMPLE (22)

We insert a node, anonymously and with a constructor (see also Section 6.2.3):

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

NOTE (15)

Persistent names belong to a specific graph whereas variables belong to the current GRSHELL environment. Persistent names will be saved (`save graph...`, see Section 6.2.5) and, if you visualize a graph (`dump graph...`, see Section 6.2.5), graph elements will be labeled with their persistent names. Persistent names have to be unique for a graph.
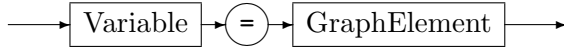
Assigns the variable or persistent name *GraphElement* to *Variable*. If *Variable* has not been defined yet, it will be defined implicitly. As usual for scripting languages, variables have neither static types nor declarations.

6.2 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
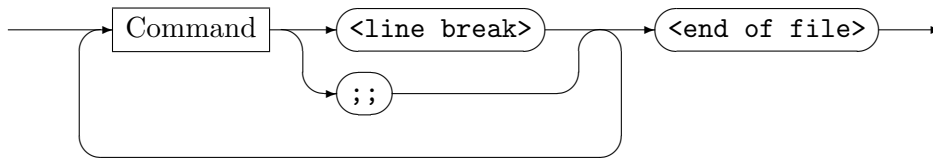
6.2.1 Common Commands
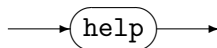
Displays an information message describing supported commands.
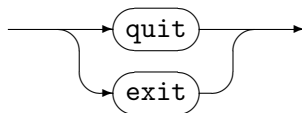
Quits GRSHELL. If GRSHELL is opened in debug mode, a currently active graph viewer (such as YCOMP) will be closed as well.
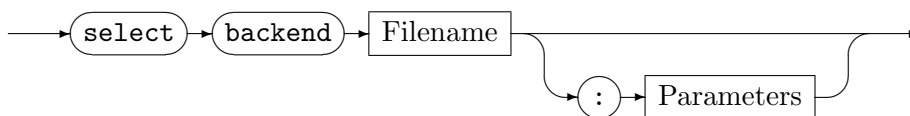
Selects a backend that handles graph and rule representation. *Filename* has to be a .NET assembly (e.g. `lgspBackend.dll`). Comma-separated parameters can be supplied optionally; if so, the backend must support these parameters. By default the LGSPBackend is used.

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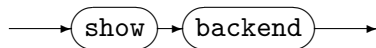
List all the parameters supported by the currently selected backend. The parameters can be provided to the `select backend` command.

```
──→(include)→│Filename│──→
```

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 errors. If a parser error occurs, execution of the script will stop immediately.

```
──→(debug)─┬→(enable)─┬──→
           └→(disable)┘
```
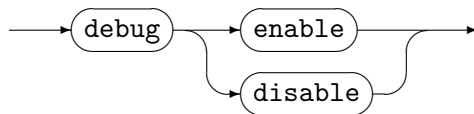
Enables and disables the debug mode. The debug mode shows the current working graph in a YCOMP window. All changes to the working graph are tracked by YCOMP immediately.

```
──→(debug)→(set)→(layout)─┬──────────┬──→
                          └→│ Text │──┘
```
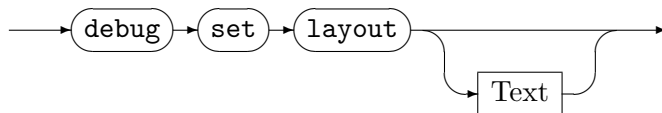
Sets the default graph layout algorithm to *Text*. If *Text* is omitted, a list of available layout algorithms is displayed. See Section 1.6.4 on YCOMP layouters.

```
──→(echo)→│Text│──→
```
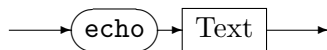
Prints *Text* onto the GRSHELL command prompt.

```
──→(!)→│CommandLine│──→
```
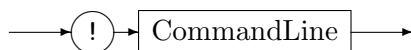
CommandLine is an arbitrary text, the operating system attempts to execute.

EXAMPLE (23)

On a Linux machine you might execute

```
1  !sh -c "ls␣|␣grep␣stuff"
```

6.2.2 Graph Commands

```
──→(new)→(graph)→│Filename│→│Text│──→
```
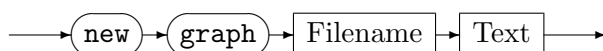
Creates a new graph with the model specified in *Filename*. Its name is set to *Text*. The model file can be either source code (e.g. `turing_machineModel.cs`) or a .NET assembly (e.g. `lgsp-turing_machineModel.dll`). It's also possible to specify a rule set file as *Filename*. In this case the necessary assemblies will be created on the fly.
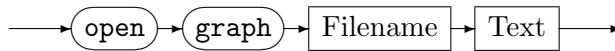
Opens the graph *Text* stored in the backend. However, the *LGSPBackend* doesn't support persistent graphs. The *LGSPBackend* is the only backend so far. Therefore this command is currently useless.

Displays a list of currently available graphs.
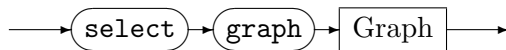
Selects the current working graph. This graph acts as *host graph* for graph rewrite sequences (see also Sections 1.4 and 6.2.6). Though you can define multiple graphs, only one graph can be the active "working graph".
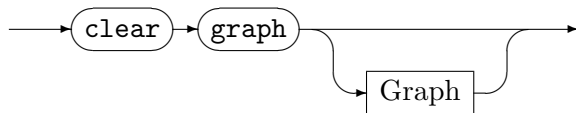
Deletes all graph elements of the current working graph resp. the graph *Graph*.
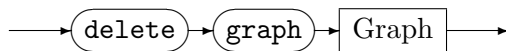
Deletes the graph *Graph* from the backend storage.
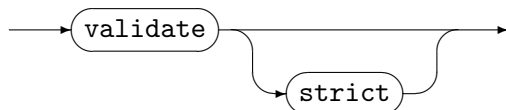
Validates if the current working graph fulfills the connection assertions specified in the corresponding graph model. The *strict* mode additionally requires all the edges of the working graph to be specified in order to get a "valid". Otherwise edges between nodes without specified constraints are ignored.

EXAMPLE (24)

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:
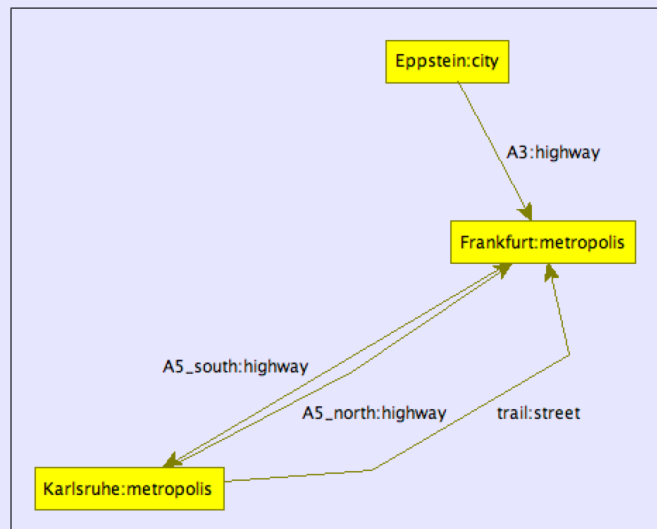
This graph is valid but not strict valid.

```
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  >
```
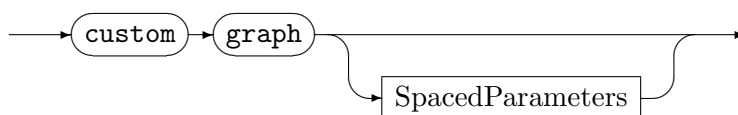

Executes a command specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSP backend see Sections 6.4).

6.2.3   Graph Manipulation Commands

Graph manipulation commands alter existing graphs including creating and deleting graph elements and setting attributes.

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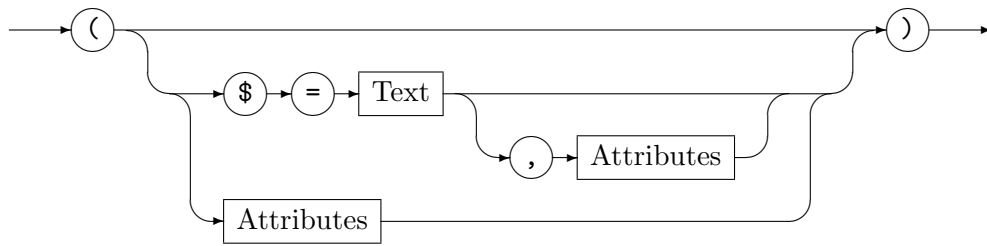
Attributes
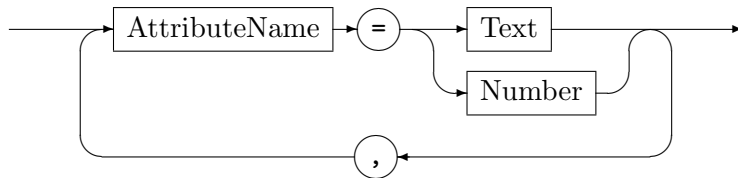
A constructor is used to initialize a new graph element (see `new ...` below). A comma separated list of attribute declarations is supplied to the constructor. Available attribute names are specified by the graph model of the current working graph. All the undeclared attributes will be initialized with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

The `$` is a special attribute name: a unique identifier of the new graph element. This identifier is also called *persistent name* (see Example 22). This name can be specified by a constructor only.
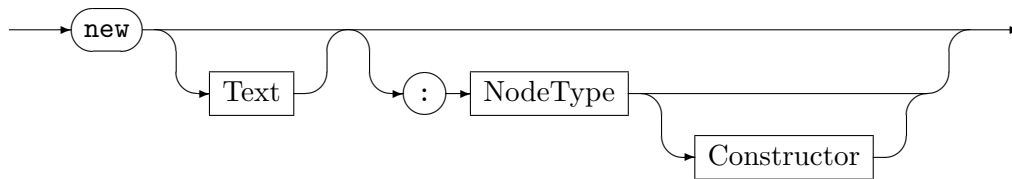
Creates a new node within the current graph. Optionally a variable *Text* is assigned to the new node. If *NodeType* is supplied, the new node will be of type *NodeType* and attributes can be initialized by a constructor. Otherwise the node will be of the base node class type *Node*.

NOTE (16)

The GRSHELL can reassign variables. This is in contrast to the rule language (chapter 4), where we use *names*.
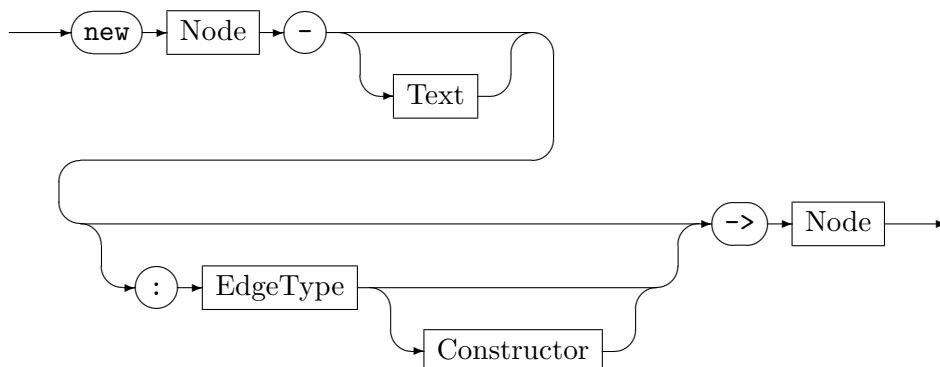
Creates a new edge within the current graph between the specified nodes, directed towards the
second *Node*. Optionally a variable *Text* is assigned to the new edge. If *EdgeType* is supplied,
the new edge will be of type *EdgeType* and attributes can be initialized by a constructor.
Otherwise the edge will be of the base edge class type *Edge*.
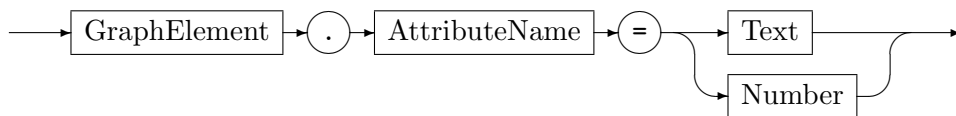
Set the attribute *AttributeName* of the graph element *GraphElement* to the value of *Text* or
Number.
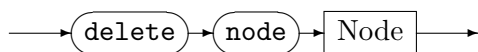
Deletes the node *Node* from the current graph. Incident edges will be deleted as well.
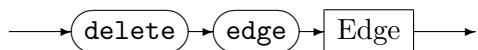
Deletes the edge *Edge* from the current graph.

6.2.4 Graph Query Commands
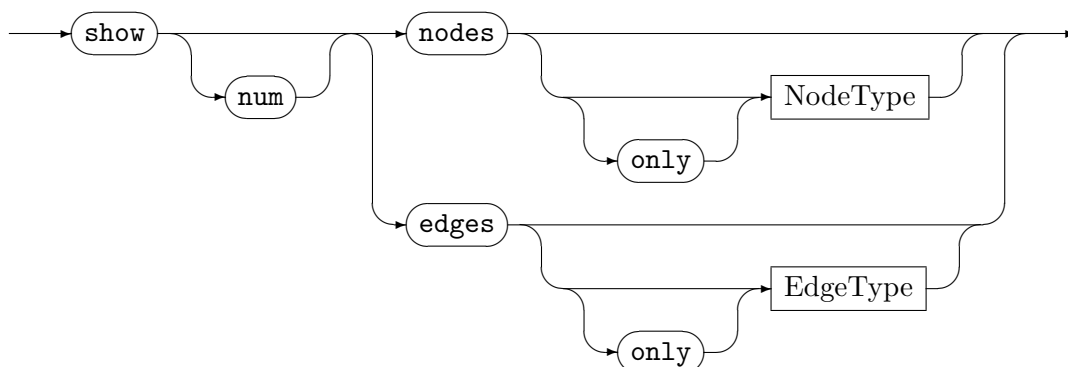
Gets the persistent names and the types of all the nodes/edges of the current graph. If a
node type or edge type is supplied, only elements compatible to this type are considered. The
`only` keyword excludes subtypes. Nodes/edges without persistent names are shown with a
pseudo-name. If the command is specified with `num`, only the number of nodes/edges will be
displayed.
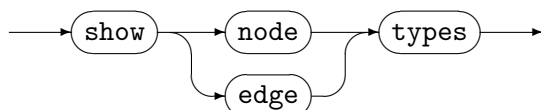
Gets the node/edge types of the current graph model.
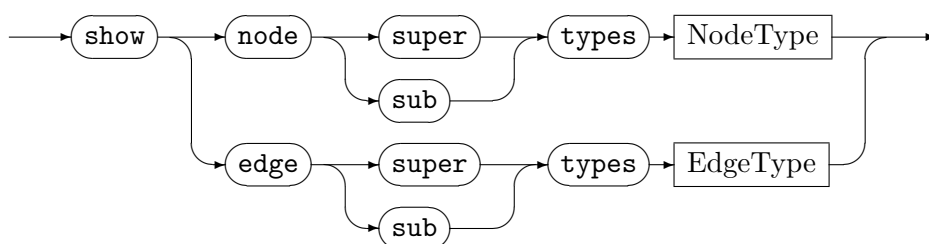
Gets the inherited/descendant types of *NodeType*/*EdgeType*.
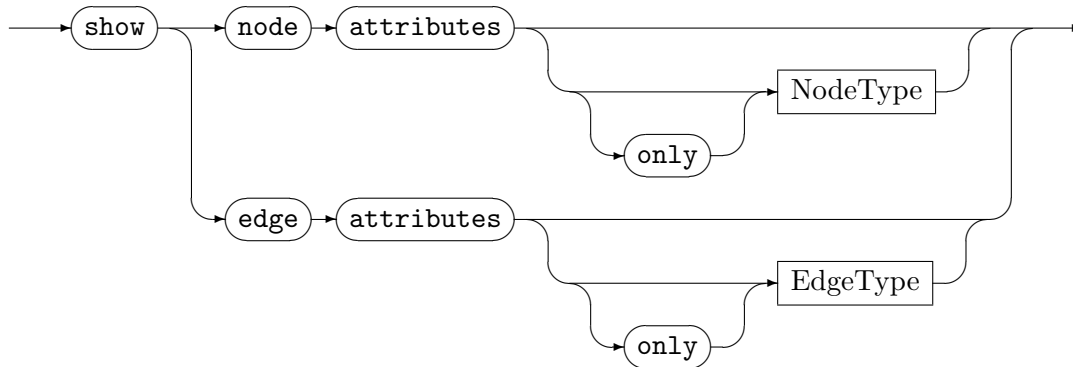
Gets the available node/edge attribute types. If *NodeType*/*EdgeType* is supplied, only attributes defined in *NodeType*/*EdgeType* are diplayed. The `only` keyword excludes inherited attributes.

NOTE (17)

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.
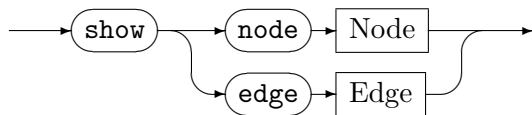
Gets the attribute types and values of a specific graph element.
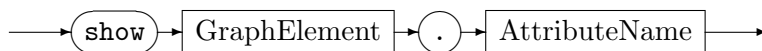
Gets the value of a specific attribute.
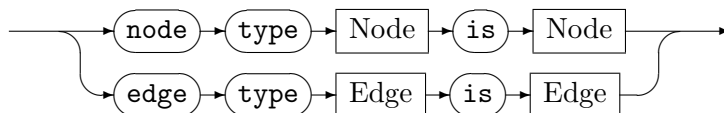
Gets the information whether the first element is type-compatible to the second element.

6.2.5 Graph Output Commands
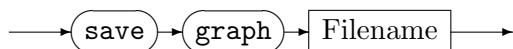
Dumps the current graph as GRSHELL script into *Filename*. The created script includes

- selecting the backend
- creating all nodes and edges

- restoring the persistent names (see Section)

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).
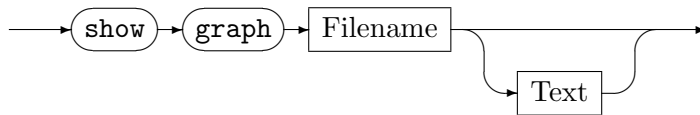
Dumps the current graph as VCG formatted file into a temporary file. *Filename* specifies an executable. The temporary VCG file will be passed to *Filename* as last parameter. Additional parameters, such as program options, can be specified by *Text*. If you use YCOMP[1] as executable, this may look like
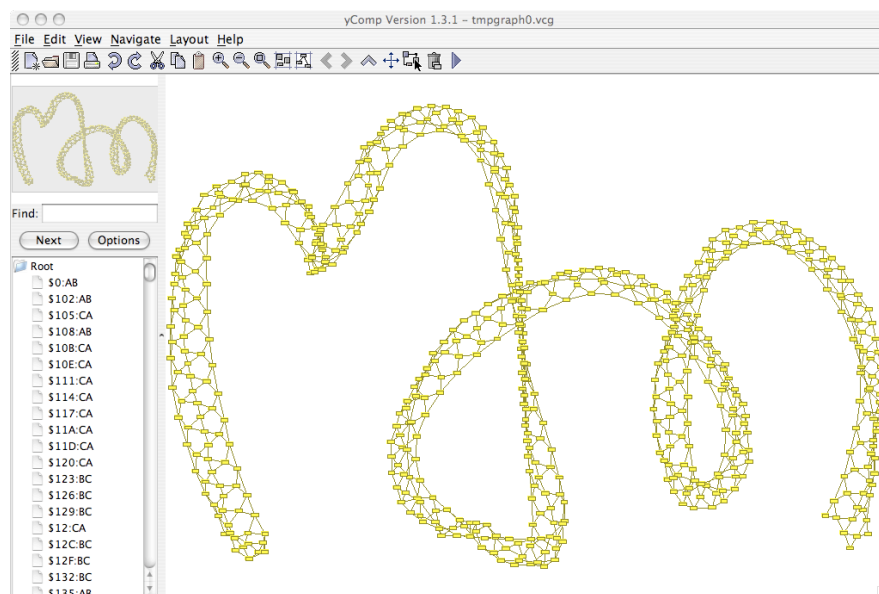
The temporary file will be deleted, when the application *Filename* is terminated if GRSHELL is still running at this time.
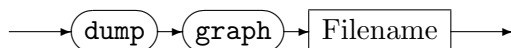
Dumps the current graph in VCG format into the file *Filename*.

Visualization Styles

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

---

[1]See Section .
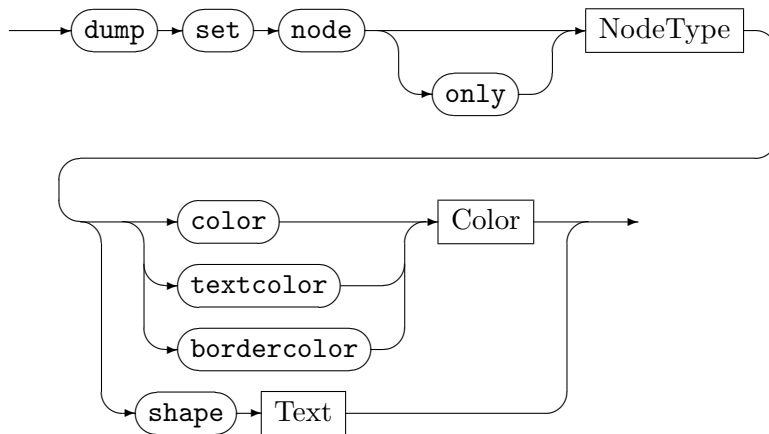
Sets the color, text color, border color, or the shape of the nodes of type *NodeType* and all of its subtypes. The keyword `only` excludes the subtypes. The following shapes are supported: `box`, `triangle`, `circle`, `ellipse`, `rhom`, `hexagon`, `trapeze`, `uptrapeze`, `lparallelogram`, `rparallelogram`. Those are shape names of yCOMP (for a VCG definition see [San95]).
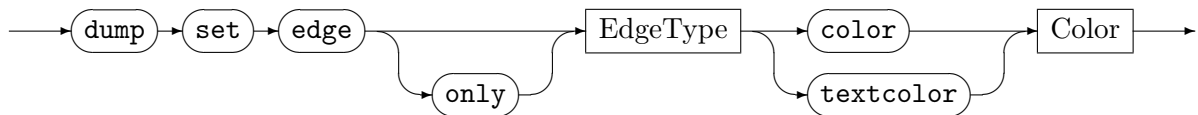
Sets the color or text color of the edges of type *EdgeType* and all of its subtypes. The keyword `only` excludes the subtypes.
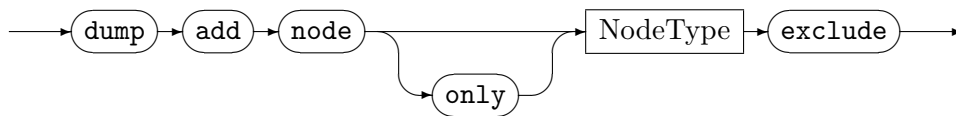
Excludes nodes of type *NodeType* and all of its subtypes as well as their incident edges from output. The keyword `only` excludes the subtypes, i.e. subtypes of *NodeType* are dumped.
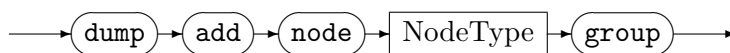
Declares *NodeType* and subtypes of *NodeType* as group node type. All the differently typed nodes that point to a node of type *NodeType* (i.e. there is a directed edge between such nodes) will be grouped and visibly enclosed by the *NodeType*-node. The following example shows *metropolis* ungrouped and grouped:
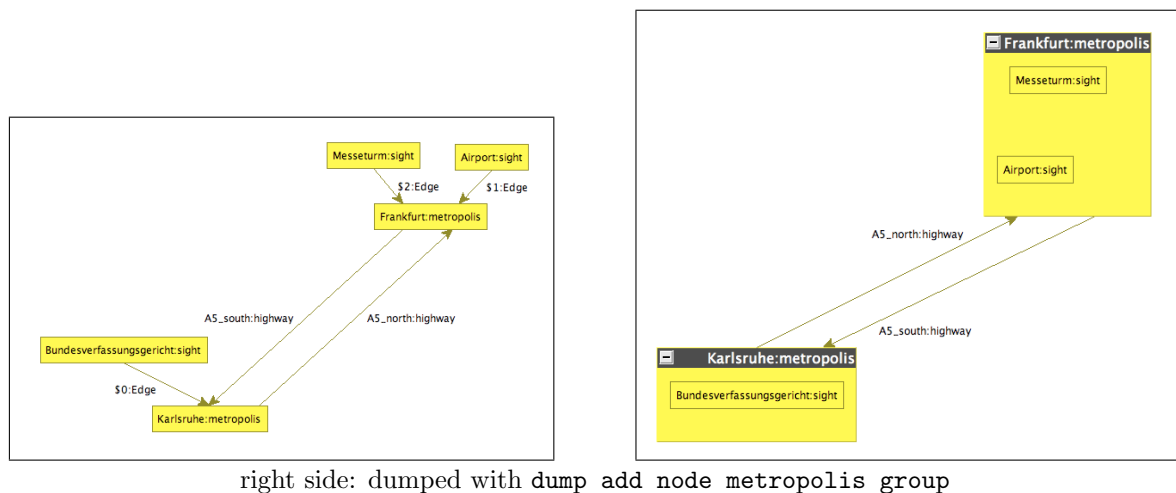
right side: dumped with `dump add node metropolis group`
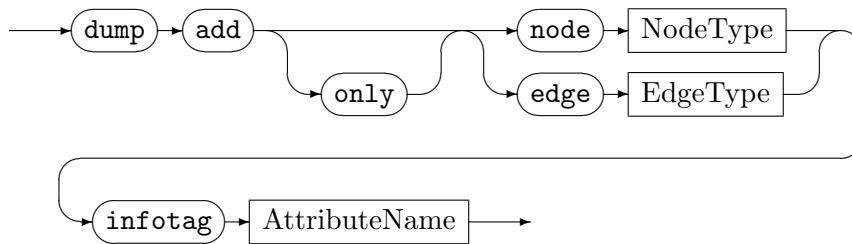
Declares the attribute *AttributeName* to be an "info tag". Info tags are displayed like additional node/edge labels. The keyword `only` excludes the subtypes of *NodeType* resp. *EdgeType*. In the following example *river* and *jam* are info tags:
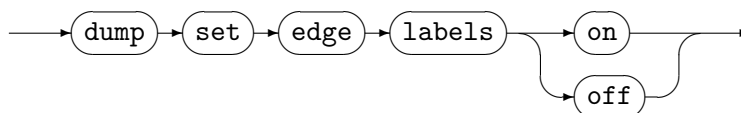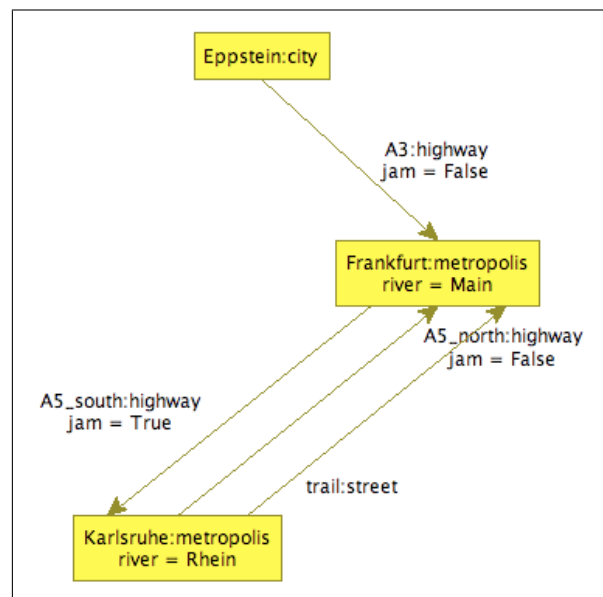
Specifies whether edge labels will be displayed or not (defaults to "on").
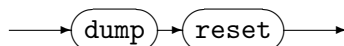
Reset all style options (`dump set...`) to their default values.

6.2.6   Action Commands (XGRS)

An *action* denotes a graph rewrite rule.
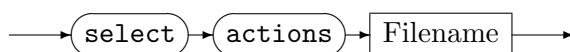
Selects a rule set. *Filename* can either be a .NET assembly (e.g. "rules.dll") or a source file ("rules.cs"). Only one rule set can be loaded simultaneously.
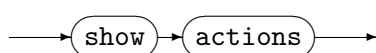
Lists all the rules of the loaded rule set, their parameters, and their return values. Rules can return a set of graph elements.
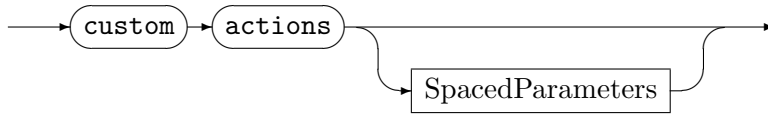
Executes an action specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSPBackend see Section 6.4).

GraphRewriteSequence
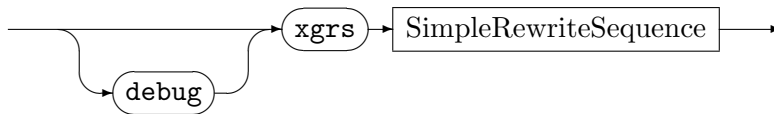
This executes the graph rewrite sequence *SimpleRewriteSequence*. See section 4.6 for graph rewrite sequences. Additionally to the variable assignment in rule-embedded graph rewrite sequences, you are also able to assign *persistant names* to parameters via `Variable = (Text)`.

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*.

6.3 Graphical Debugger

The GRSHELL together with YCOMP build GRGEN.NET's graphical debugger. Use the `debug grs` option to trace the rewriting process step-by-step. 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 6.1.

`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 25)
(step) `o`(ut)	Continue a rewrite sequence until the end of the current loop. If the execution is not in a loop at this moment, the complete sequence will be executed
`r`(un)	Continue execution without further stops
`a`(bort)	Cancel the execution immediately

Table 6.1: GRSHELL debug commands

6.4 LGSPBackend Custom Commands

The LGSPBackend supports the following custom commands:

---

[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.

We demonstrate the debug commands with a slightly adjusted script for the Koch snowflake from GRGEN.NET's examples (see also Section 7.1). 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`):
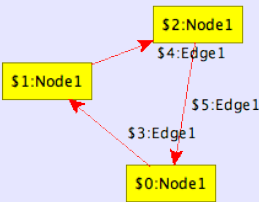
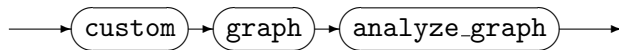We type `d`(etailed step) to apply `makeFlake1` step by step resulting in the following graphs:
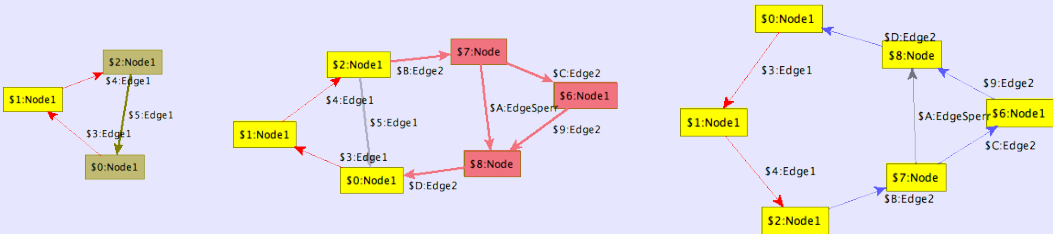
The following table shows the "break points" of further debug commands, entered one after another:

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

6.4.1 Graph Related 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.

6.4.2 Action Related Commands
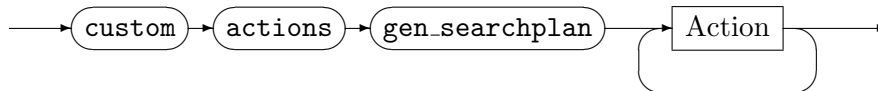
Creates a search plan for each rewrite rule *Action* using a heuristic method and the analyzes data (if the graph has been analyzed by `custom graph analyze_graph`). Otherwise a default search plan is used. For efficiency reasons it is recommended to do analyzing and search plan creation during the rewriting procedure. Therefore the host graph should be in a stage "similar" to the final result. Indeed there might be some trial-and-error steps necessary to get an 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.
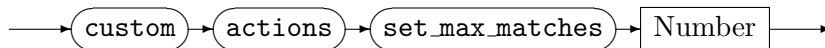
Sets the maximum amount of possible pattern matches to *Number*. This command affects the expression `[Rule]`. If *Number* is less or equal to zero, the constraint is reset.

CHAPTER 7

EXAMPLES

7.1 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 6.2.6 for further information. The resulting graphs should look like Figures 7.1 and 7.2.

---

[1]Be careful: The running time increases exponentially in the number of iterations.
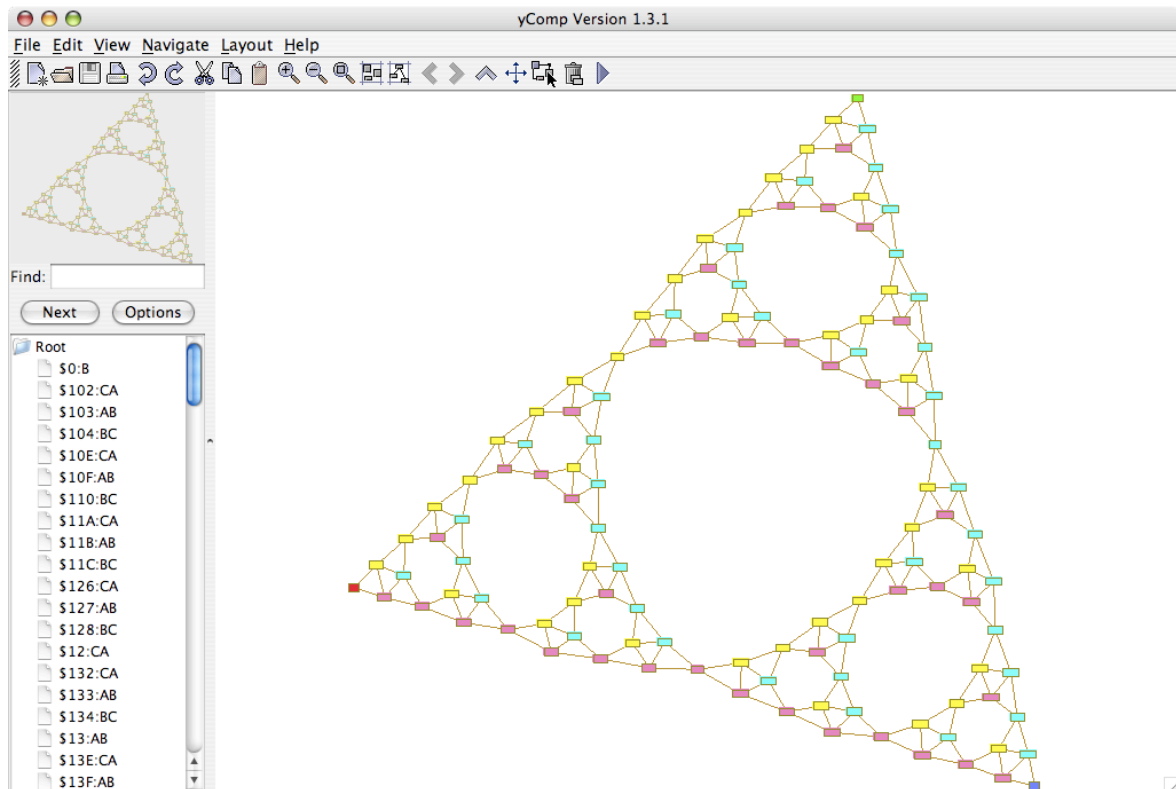
Figure 7.1: Sierpinski triangle
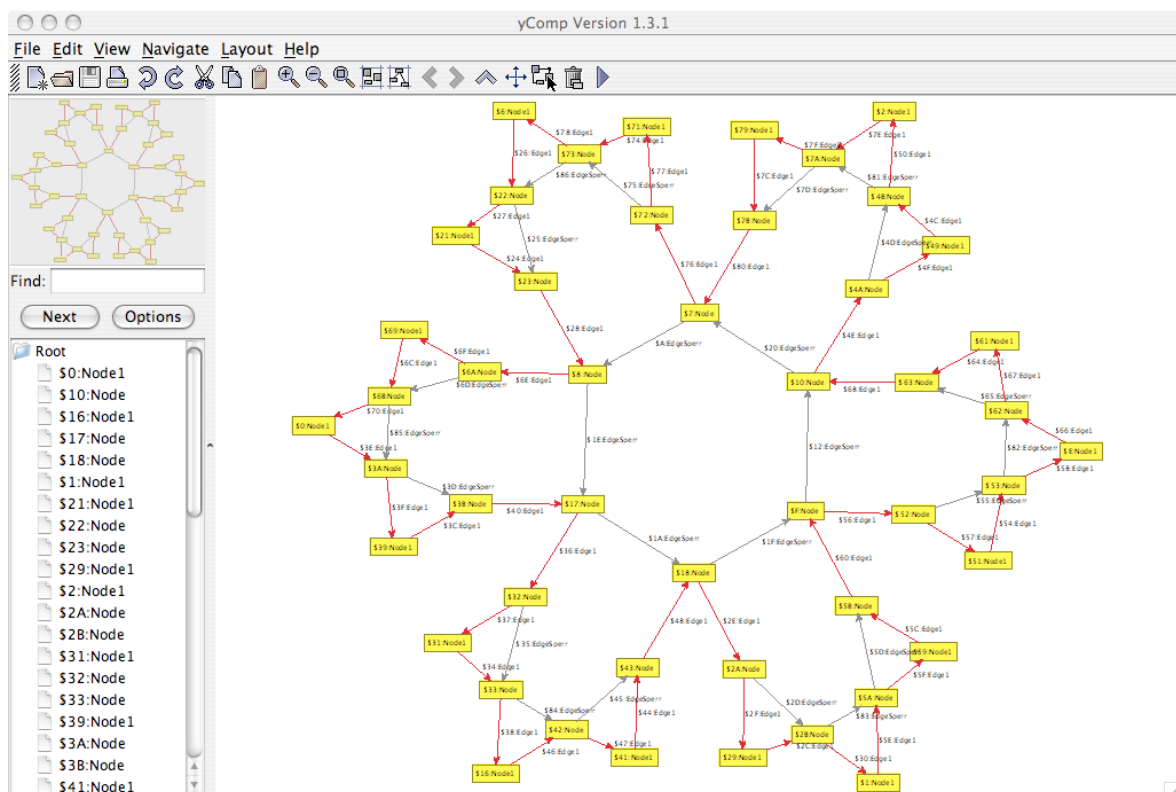
Figure 7.2: Koch snowflake

7.2   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.

7.2.1   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 $[0:1]$.

```
1  node class TapePosition;
2  edge class right
3    connect TapePosition[0:1] -> TapePosition[0:1];
4
5  edge class value
6    connect TapePosition[1] -> TapePosition[1];
7  edge class zero   extends value;
8  edge class one    extends value;
9  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 73).

```
10  node class State;
11
12  edge class RWHead;
13
14  node class WriteValue;
15  node class WriteZero extends WriteValue;
16  node class WriteOne extends WriteValue;
17  node class WriteEmpty extends WriteValue;
18
19  edge class moveLeft;
20  edge class moveRight;
21  edge class dontMove;
```

7.2.2   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.

```
2  rule readZeroRule {
3      s:State -h:RWHead-> tp:TapePosition -:zero-> tp;
4      s -:zero-> wv:WriteValue;
5      modify {
6          delete(h);
7          wv -:RWHead-> tp;
8      }
9  }
```

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:

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

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

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 7.1).

7.2.3 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:
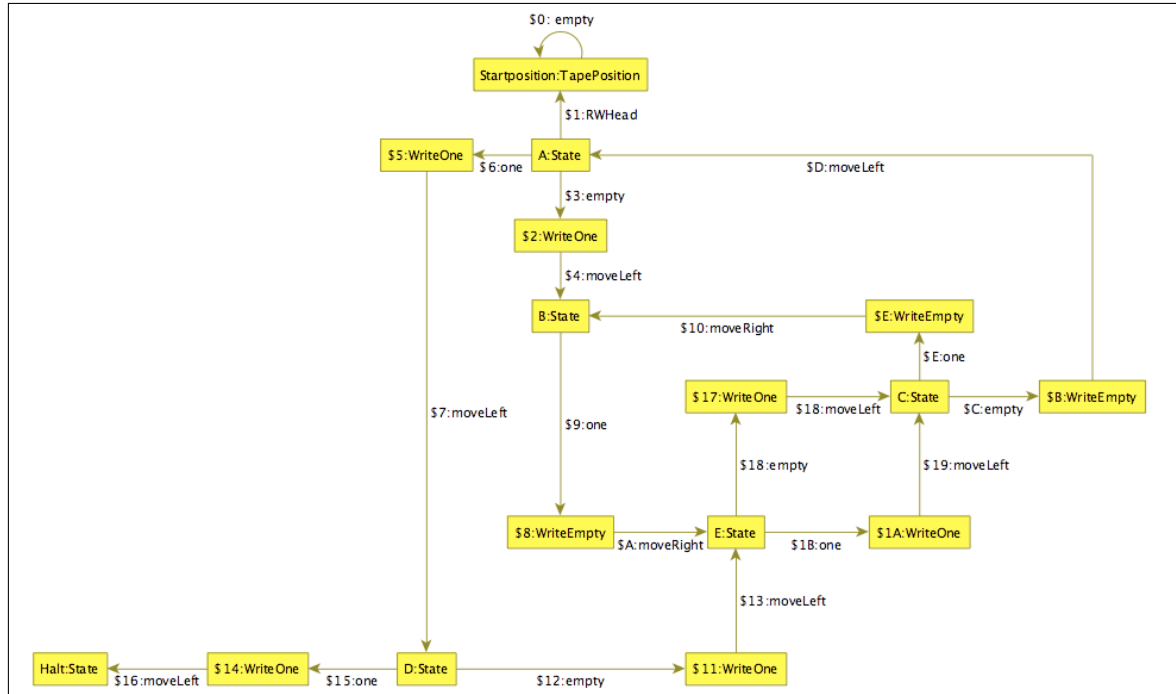
We have an initial host graph now. The graph rewrite sequence is quite straight forward and generic to the Turing graph model. Note that for each state the "`...Empty...` — `...One...`" selection is unambiguous.

```
63   xgrs ((readOneRule | readEmptyRule) & (writeOneRule | writeEmptyRule) &
         (ensureMoveLeftValidRule | ensureMoveRightValidRule) & (moveLeftRule |
         moveRightRule)){32}
```

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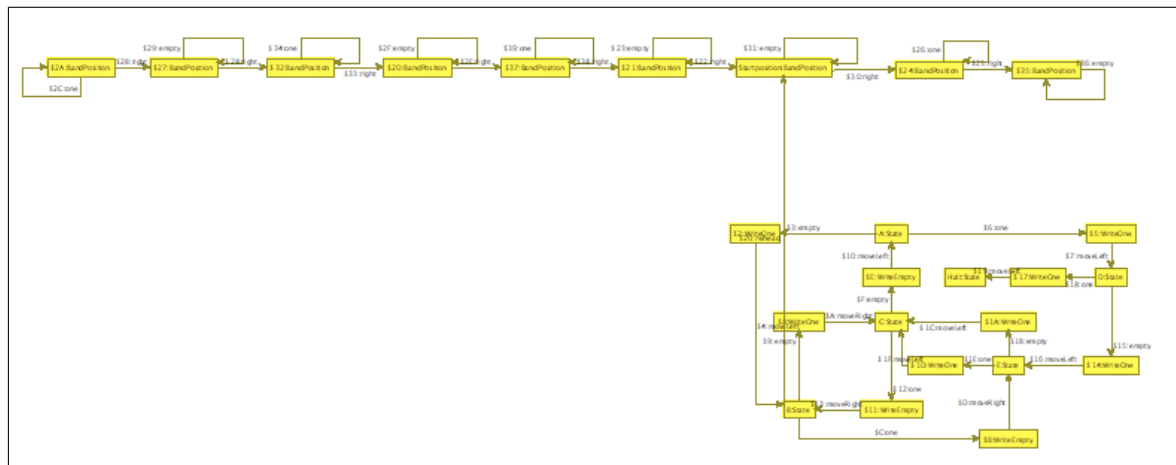
In order to improve the performance we generate better search plans. This is a crucial step for execution time: With the initial search plans the beaver runs for 1 minute and 30 seconds. With improved search plans after the first 32 steps he takes about 8.5 seconds[2].

---

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

```
64  custom graph analyze_graph
65  custom actions gen_searchplan readOneRule readEmptyRule writeOneRule writeEmptyRule
        ensureMoveLeftValidRule ensureMoveRightValidRule moveLeftRule moveRightRule
```

Let the beaver run:

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

DEPRECATED SYNTAX

This appendix describes deprecated GRGEN.NET constructs of versions prior to 1.4. The following constructs may or may not work with the current GRGEN.NET release. Support for such constructs will eventually be terminated.

A.1   Graph Model and Rule Set Language

The graph model and the rule set were previously introduced by keywords:

GraphModel
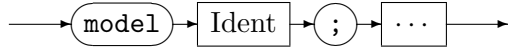
RuleSet
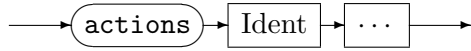
These keywords are not used any more. The name of a graph model resp. a rule set is determinated by its file name.

In previous GRGEN.NET versions the pattern was a syntactical block within a rule:

RuleDeclaration
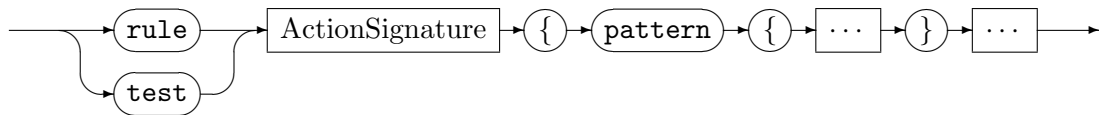
The current version does not use the `pattern` keyword but expects the pattern statements to be placed as direct members of the rule block (see sections 4.2, 4.3). The `pattern` keyword will be used in GRGEN.NET version 2.0 for a different purpose.

A.2   Graph Rewrite Sequences (GRS)

The non-extended graph rewrite sequence was part of the GRSHELL. The extended graph rewrite sequences (XGRS) are available within the GRSHELL as well as within the rule set language (see section 4.6).

GraphRewriteSequence
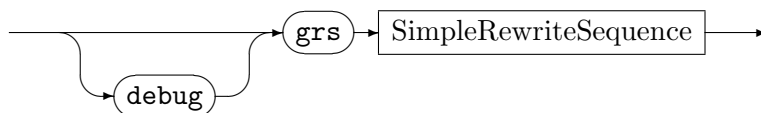
This executes the graph rewrite sequence *SimpleRewriteSequence*.

SimpleRewriteSequence
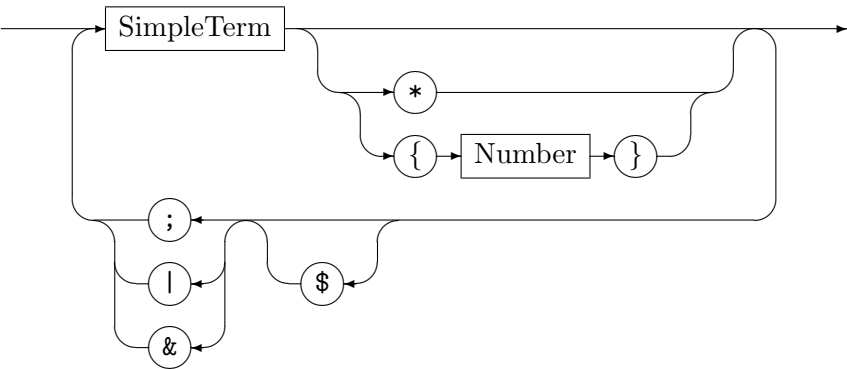
SimpleTerm
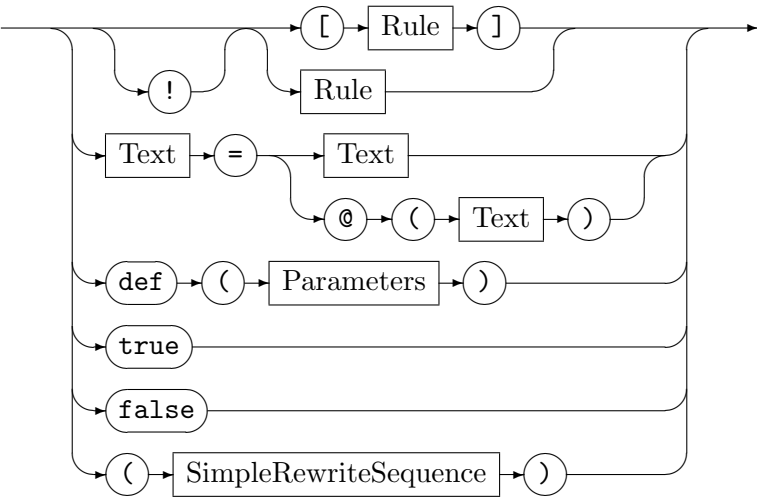
Rule
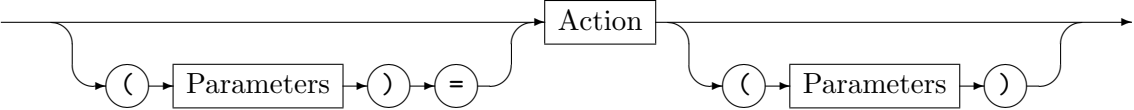
Table A.1 lists graph rewrite expressions at a glance. The operators hold the following order of precedence, starting with the lowest priority:

<div align="center">

; | &

</div>

`s ; t`	**Concatenation.** First execute `s` afterwards execute `t`. The sequence `s ; t` is *successfully* executed iff `s` or `t` is successfully executed.
`s	t`	**XOR.** First execute `s`. Only if `s` fails (i.e. can not be executed successfully) then execute `t`. The sequence `s	t` is successfully executed iff `s` or `t` is successfully executed.
`s & t`	**Transactional AND.** First execute `s`, afterwards execute `t`. If `s` or `t` fails, the action will be terminated and a rollback to the state before `s & t` is performed.
`$<op>`	Flag the operator `<op>` as commutative. Usually operands are executed/evaluated from left to right with respect to bracketing (left-associative). But the sequences `s, t, u` in `s $<op> t $<op> u` are executed/evaluated in arbitrary order.
`s *`	Execute `s` repeatedly as long as its execution does not fail.
`s {n}`	Execute `s` repeatedly as long as its execution does not fail but `n` times at most.
`!`	Dump found matches into VCG formatted files (for a VCG definition see [San95]). Every match produces three files within the current directory: 1. The complete graph that has the matched graph elements marked 2. The complete graph with additional information about matching details 3. A subgraph containing only the matched graph elements
Rule	Rewrite the first found pattern match produced by the action *Rule*.
[*Rule*]	Rewrite every pattern match produced by the action *Rule*. **Note:** This operator is mainly added for benchmark purposes. Its semantics is not equal to `Rule*`. Instead this operator collects all the matches first before starting rewritings. In particular one needs to avoid deleting a graph element that is bound by another match.
`v = w`	Assign the variable `w` to `v`. If `w` is undefined, `v` will be undefined, too.
`v = @(x)`	Assign the graph element identified by `x` to the variable `v`. If `x` is not defined, `v` will be undefined, too.
`def(`*Parameters*`)`	Is *successful* if all the graph elements in *Parameters* exist, i.e. if all the variables are defined.
`true`	A constant acting as a successful match.
`false`	A constant acting as a failed match.

Let `s, t, u` be graph rewrite sequences, `v, w` variable identifiers, `x` an identifier of a graph element, `<op>` $\in \{;, |, \&\}$ and `n` $\in \mathbb{N}_0$.

Table A.1: Graph rewrite expressions

[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.

[BKG07]   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 2007*, 2007. preliminary version, submitted to AGTIVE 2007.

[CEM⁺06]   Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors. *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *Lecture Notes in Computer Science*. Springer, 2006.

[CHHK06]   Andrea Corradini, Tobias Heindel, Frank Hermann, and Barbara König. Sesquipushout rewriting. In Corradini et al. [CEM⁺06], pages 30–45.

[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.

[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.

[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 Corradini et al. [CEM⁺06], pages 383–397.

[Gei07]   R. Geiß. GRGEN.NET. http://www.grgen.net, June 2007.

[Hac03]   Sebastian Hack. Graphersetzung für Optimierungen in der Codeerzeugung. Master's thesis, IPD Goos, 12 2003.

[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.

[Kro07]   Moritz Kroll. GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen, 5 2007. Studienarbeit, Universität Karlsruhe.

[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.

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

[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.

[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.

[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.

[yWo07]   yWorks. yFiles. http://www.yworks.com, 2007.

Keywords

Non-Terminals

General Index