

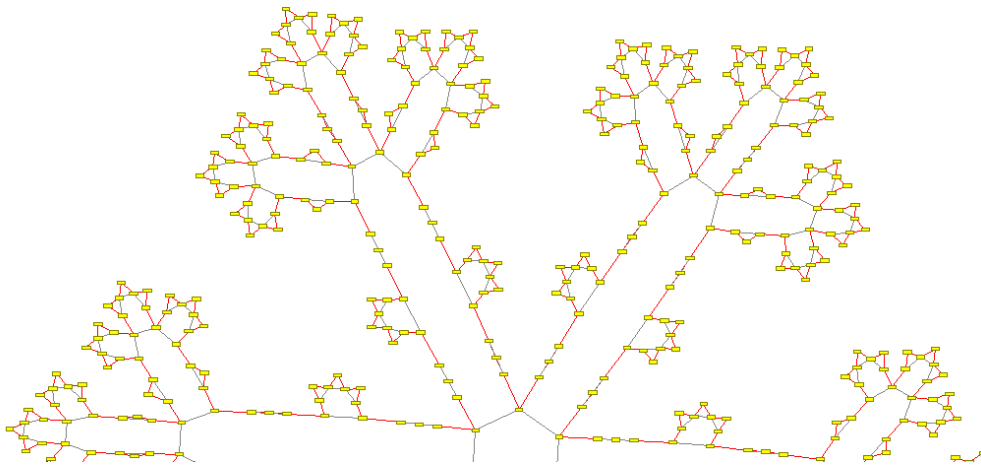


Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

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

DRAFT

The GRGEN User Manual



Jakob Blomer

Rubino Geiß

May 24, 2007

Technical Report 2007-5
ISSN 1432-7864

ABSTRACT

This is the abstract. TODO!

CONTENTS

1	Introduction	1
1.1	What is GRGEN?	1
1.2	Features of GRGEN	1
1.3	System Overview	2
1.4	What is Graph Rewriting?	3
1.5	An Example	4
1.6	The Tools	5
1.6.1	GrGen.exe	5
1.6.2	GrShell.exe	6
1.6.3	LibGr.dll	6
1.6.4	yComp.jar	6
2	Graph Model Language	9
2.1	Building Blocks	9
2.2	Type Declarations	11
3	Rule Set Language	15
3.1	Building Blocks	16
3.2	Rules and Tests	20
3.3	Pattern Part	23
3.4	Replace/Modify Part	25
4	Types and Expressions	29
4.1	Built-In Types	29
4.2	Expressions	30
4.3	Type Related Conditions	33
4.4	Annotations	34
5	GrShell Language	35
5.1	Building Blocks	35
5.2	GRSHELL Commands	37
5.2.1	Common Commands	37
5.2.2	Graph Commands	38
5.2.3	Graph Manipulation Commands	40
5.2.4	Graph Query Commands	42
5.2.5	Graph Output Commands	44
5.2.6	Action Commands (GRS)	46
5.3	LGSPBackend Custom Commands	48
5.3.1	Graph Related Commands	48
5.3.2	Action Related Commands	48

6	Examples	53
6.1	Fractals	53
6.2	Busy Beaver	55
6.2.1	Graph Model	55
6.2.2	Rule Set	55
6.2.3	Rule Execution with GRSHELL	58

CHAPTER 1

INTRODUCTION

1.1 What is GRGEN?

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 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. For the rewrite step, our tool implements the well-founded single-pushout approach (SPO, for explanation see [?]).

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

1.2 Features of GRGEN

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 found match, and, finally, selecting the next rule(s) for application. We have organized the features of GRGEN according to this breakdown of graph rewriting.

- The graph model (meta-model) supports:
 - Directed graphs
 - Typed nodes and edges, with multiple inheritance on types
 - Typed attributes can be assigned to node and edge types
 - Multigraphs (including multiple edges of the same type)
 - Connection assertions to restrict the "shape" of graphs
- The pattern language supports:
 - Plain isomorphic subgraph matching (injective mapping)
 - Homomorphic matching for a selectable set of nodes/edges, so that the matching is not injective
 - Attribute conditions (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 (yet to be implemented)
- The rewrite language supports:
 - Attribute re-calculation (using arithmetic operations on the attributes)
 - Retyping of nodes/edges (a stronger version of casts of 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 [?] (yet to be implemented)
- The rule application language (GRSHELL) supports:
 - Composing several rules with logical and iterative sequence control (called regular graph rewrite sequences, RGS)
 - Various methods for creation/deletion/input/output of graphs/nodes/edges
 - Stepwise and graphic debugging of rule application
 - Regular graph rewrite sequences that can contain optionally nested transactions (yet to be implemented)
- 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.

1.3 System Overview

Figure 1.1 gives an overview of the GRGEN system components, whereas table 1.1 shows the GRGEN directory structure.

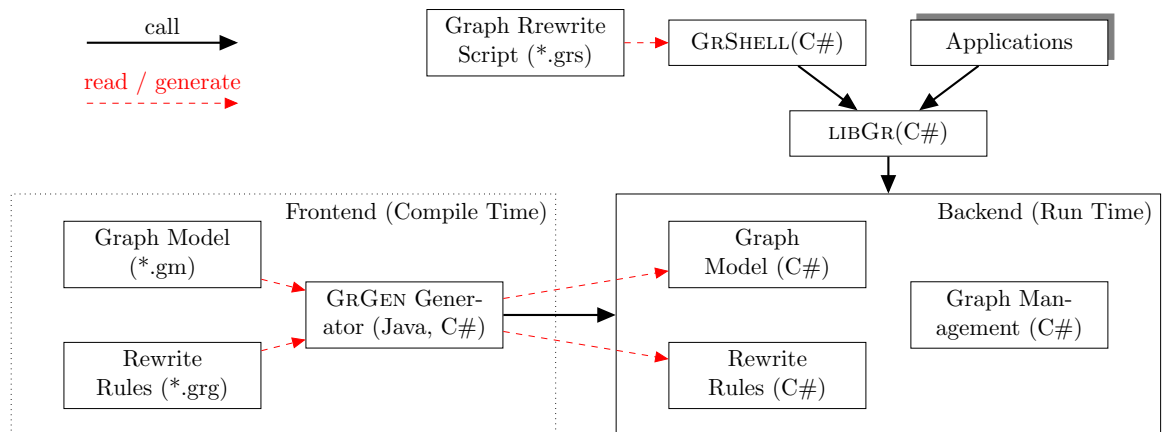


Figure 1.1: GRGEN system components [8]

A graph rewrite system¹ 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

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

bin	Contains the .NET assemblies, in particular GrGen.exe (the graph rewrite system generator), LGSPBackend.dll (a GRGEN backend) and the shell GrShell.exe.
lib	Contains the GRGEN generated assemblies (*.dll).
specs	Contains the graph rewrite system source documents (*.gm and *.grg).

Table 1.1: GRGEN directory structure

by GrGen.exe and can be used by applications such as GRShell. Figure 1.2 shows the generation process.

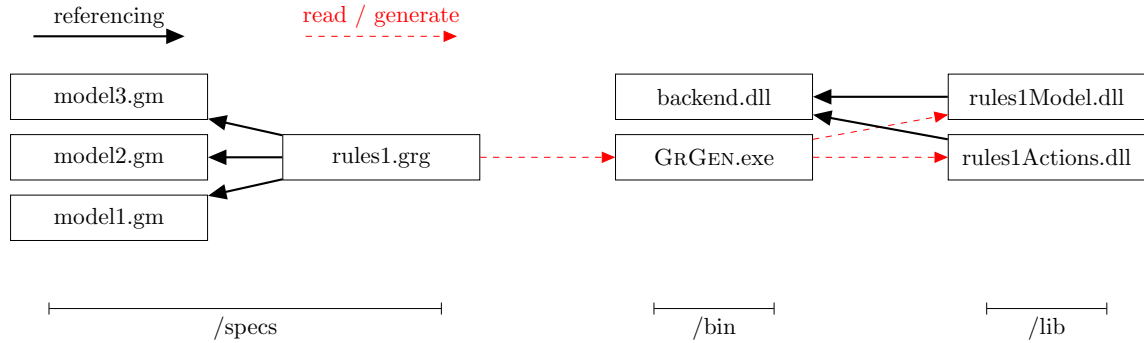


Figure 1.2: Generating a graph rewrite system

In general you have to distinguish carefully between a graph model (meta level), a host graph, a pattern graph and a rewrite rule. In GRGEN 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).

1.4 What is Graph Rewriting?

The notion of graph rewriting as understood by GRGEN 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 implements an SPO-based approach. In the simplest case such a graph rewrite rule consists of a tuple $L \rightarrow R$, whereas L – the left hand side of the rule – is called *pattern graph* and R – the right hand side of the rule – is the *replacement graph*.

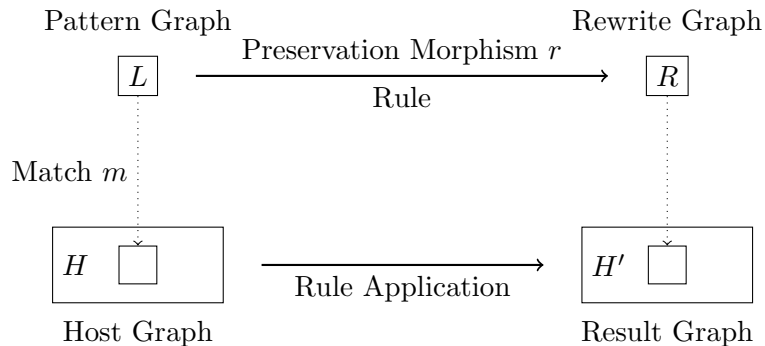


Figure 1.3: Basic Idea of Graph Rewriting

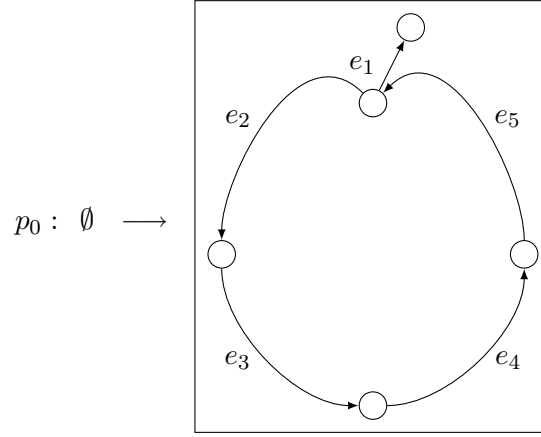
Moreover we need to identify graph elements (nodes or edges) of L and R for preserving them during rewrite. This is done by a *preservation morphism* r mapping elements from L

to R ; the morphism r is injective, but needs to be neither surjective nor total. Together with a rule name p we have $p : L \xrightarrow{r} R$.

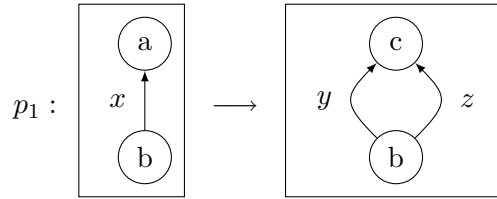
The transformation is done by *application* of a rule to a *host graph* H . To do so, we have to find an occurrence of the pattern graph in the host graph. Mathematically speaking, such a *match* m is an isomorphism from L to a subgraph of H . This morphism may not be unique, i.e. there may be several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

1.5 An Example

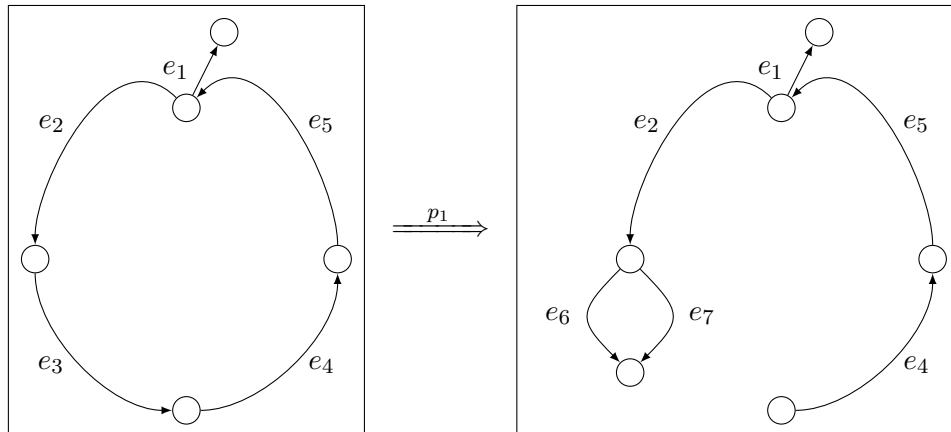
We'll have a look at a small example. We start using a special case to construct our host graph: an empty pattern always produces exactly one² match (independent of the host graph). So starting with an empty host graph we construct an apple by applying



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

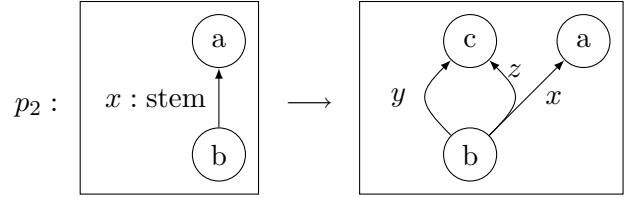


to H and get the new host graph H_1 , something like this:

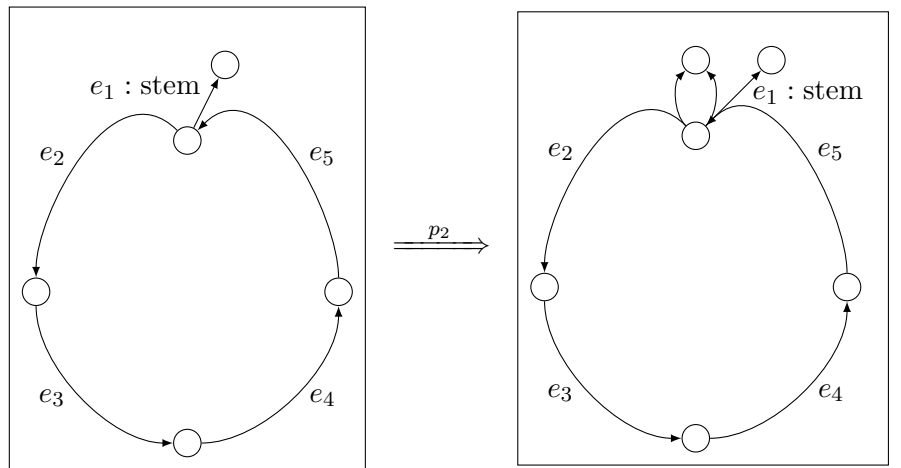


²Because of the uniqueness of the total and totally undefined morphism.

What happened? GRGEN 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’ll even keep the stem. So let



If we apply p_2 to the modified H_1 this leads to



Note: If we had applied p_1 to H_1 and repetitive to the resulting graphs, this would not have led to a fixed point because each rewrite produces one new candidate (one deleted, two added) for matching.

1.6 The Tools

1.6.1 GrGen.exe

The **GrGen.exe** assembly implements the GRGEN generator. The GRGEN generator parses a rule set and its model files and compiles them into .NET assemblies. The compiled assemblies interact with the GRGEN backend.

Usage

```
[mono] GrGen.exe [-use <existing-dir>] [-d] <rule-set> [<output-dir>]
```

rule-set is a file containing a rule set specification according to chapter 3. Usually such a file has the suffix **.grg**. The suffix **.grg** may be omitted.

By default GRGEN tries to write the compiled assemblies to the directory **../lib** relative to the path of **GrGen.exe**. This can be changed by the optional parameter *output-dir*.

Options

- d Enables debug output. A subdirectory `tmpgrgenn3` within the current directory will be created. This directory contains:
 - `printOutput.txt` – a snapshot of `stdout` during program execution.
 - `NameActions.cs` – the C# source file of the `rule-setActions.dll` assembly.
 - `NameModel.cs` – the C# source file(s) of the `rule-setModel1.dll` assembly.
- use Don't re-generate C# source files. Instead use the files in *existing-dir* to build the assemblies.

Requires

.NET 2.0 (or above) or Mono 1.2.2 (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 and debugging graph rewriting. For further information about the GRShell language see chapter 5.

Usage

```
[mono] grShell.exe [-c "<commands>" | <grshell-script>]
```

Opens the interactive shell. The GRShell will execute the commands in *grshell-script* (usually a **.grs* file) immediately.

Options

- c Execute the quoted GRShell commands immediately. Instead of a line break use a double semicolon `;;` as command separation terminal.

Requires

.NET 2.0 (or above) or Mono 1.2.2 (or above).

1.6.3 LibGr.dll

The LIBGR is a .NET assembly implementing GRGEN's API. See the extracted HTML documentation for interface descriptions.

1.6.4 yComp.jar

YCOMP is a graph visualization tool based on YFILES. It is well integrated in GRGEN, but it's not a part of GRGEN. YCOMP implements several graph layout algorithms and has file format support for VCG, GML and YGF among others. Notice that YCOMP is not free software. However, it's free for use in academic and non-commercial areas.

Usage

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

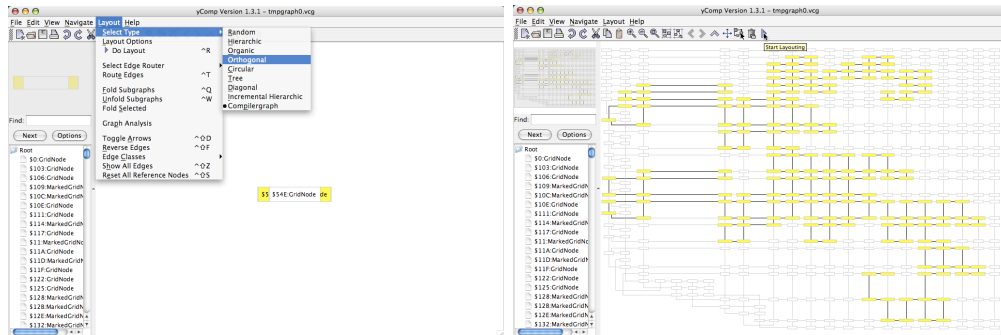
```
java -jar yComp.jar [<graph-file>]
```

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

³*n* is an increasing number.

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 layouting. This may take a while, depending on the graph size:

*Requires*

Java Runtime Environment 1.5 (or above).

CHAPTER 2

GRAPH MODEL LANGUAGE

The key features of GRGEN graph models as described by Geiß et al. [4]:

Types.

Nodes and edges can have types (classes). This is similar to common programming languages, except GRGEN types have no concept of methods.

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 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 5 (GRSHELL) we use excerpts of example 1 (the Map model) for illustration purposes.

2.1 Building Blocks

NOTE (1)

The following syntax specifications make heavy use of syntax diagrams (also known as rail diagrams). Syntax diagrams provide a visualization of EBNF^a grammars. Follow a path along the arrows from left to right through a diagram to get a valid sentence (or sub sentence) of the language. Ellipses are terminals whereas rectangles are non-terminals. For further information on syntax diagrams see [11].

^aExtended Backus–Naur Form.

Basic elements of the GRGEN graph model language are identifiers to denominate types, attributes, and the model itself. The GRGEN graph model language is case sensitive.

EXAMPLE (1)

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

```

1 model Map;
2
3 enum resident {village = 500, town = 5000, city = 50000}
4
5 node class sight;
6
7 node class city {
8     size: resident;
9 }
10
11 const node class metropolis extends city {
12     river: string;
13 }
14
15 abstract node class abandoned_city extends city;
16 node class ghost_town extends abandoned_city;
17
18 edge class street;
19 edge class trail extends street;
20 edge class highway extends street
21     connect metropolis [+] -> metropolis [+]
22 {
23     jam: boolean;
24 }
```

Ident, IdentDecl

A non-empty character sequence of arbitrary length consisting of letters, digits or under-scores. The first character must not be a digit. *Ident* and *IdentDecl* differ in their role: While *IdentDecl* is a *defining* occurrence of an identifier, *Ident* is a *using* occurrence. An *IdentDecl* non-terminal can be annotated. See 4.4 for annotations of declarations.

NOTE (2)

The GRGEN model language does not distinguish between declarations and definitions. More precisely, every declaration is also a definition. For instance, there are no C-like constructs as the following:

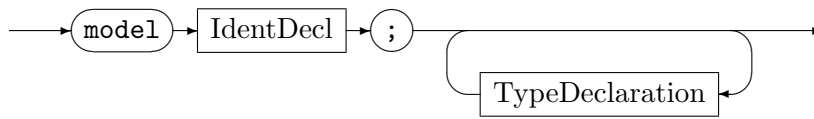
```

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

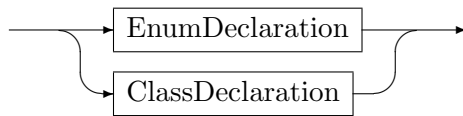
NodeType, EdgeType, EnumType

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

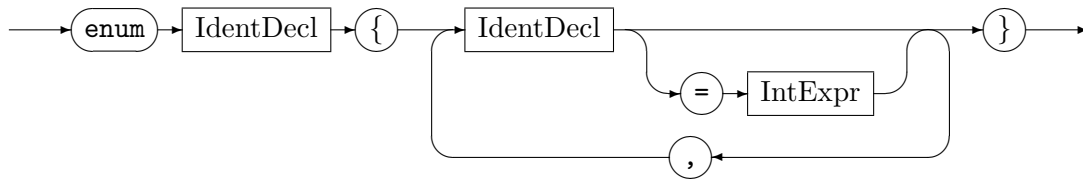
2.2 Type Declarations

GraphModel

The graph model consists of its name *IdentDecl* and type declarations defining specific node and edge types as well as enums.

TypeDeclaration

ClassDeclaration defines a node type or an edge type. *EnumDeclaration* defines an enum type for use as attribute of nodes or edges. Types do not need to be declared before they are used.

EnumDeclaration

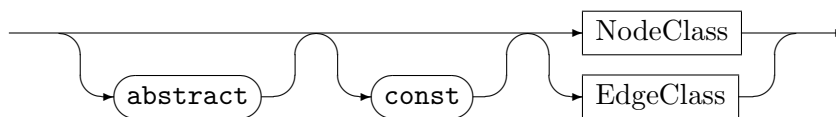
Defines an enum type.

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 = Resident::village + c}
  
```

The semantics is as in C [6]. So, the following holds: **red** = 0, **green** = 1, **blue** = 2, **a** = 2, **b** = 3, **c** = 1, **d** = 2, and **e** = 501.

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:

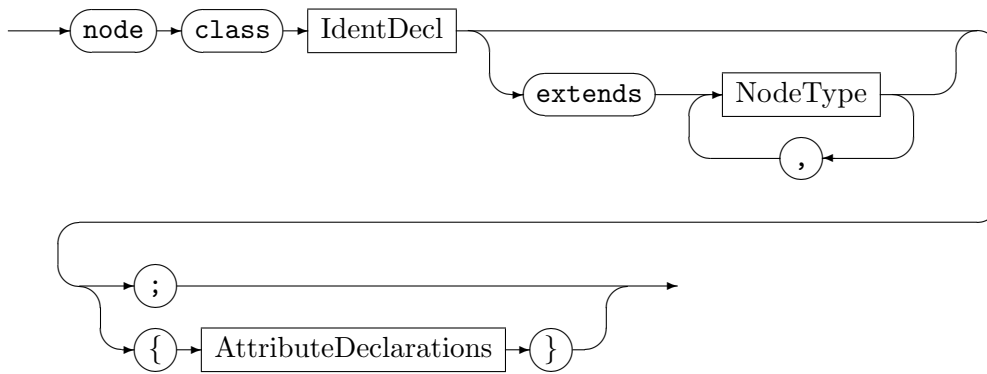
```

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;

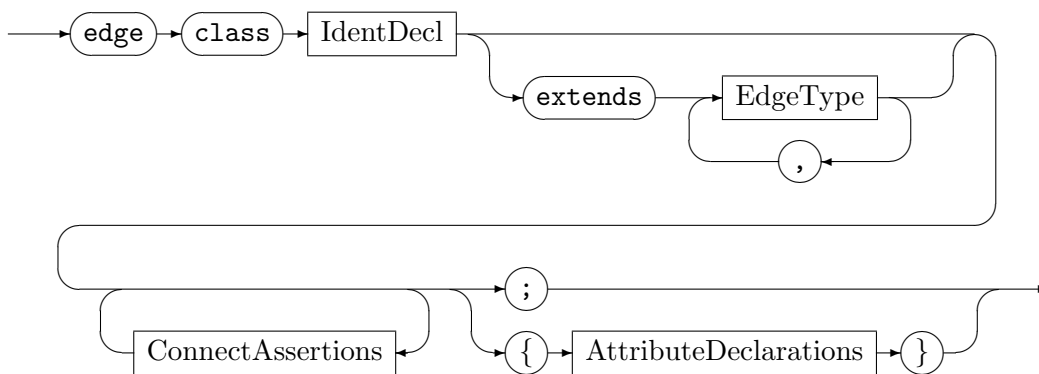
```

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 3.4, `eval`). However, such attributes are still writable by `LIBGR` and `GRSHELL` directly. This property will not be inherited by subclasses. If you want a subclass to have the `const` property, you have to set the `const` modifier explicitly.

NodeClass

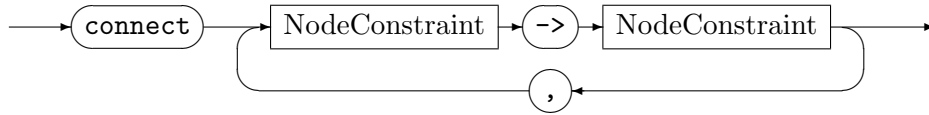
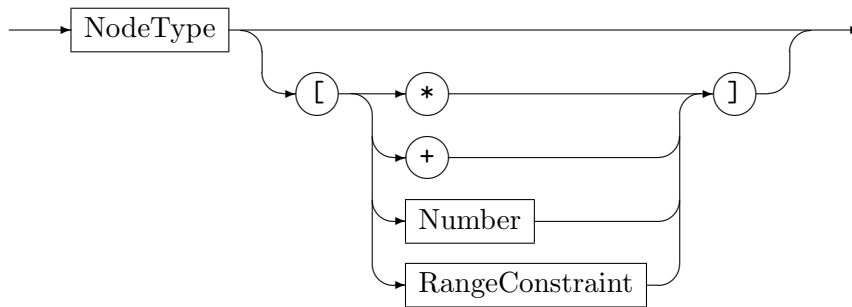
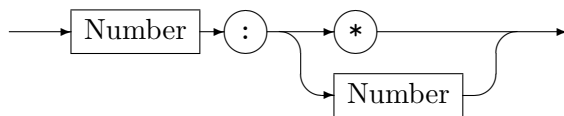
Defines a new node type. Node types can inherit from other node types defined within the same file. If the `extends` clause is omitted, *NodeType* will inherit from the built-in type `Node`. Optionally nodes can possess attributes.

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 and, moreover, the number of outgoing and incoming edges can be constrained.

NOTE (3)

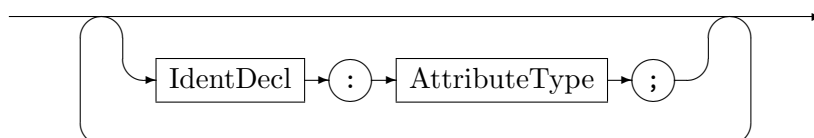
It is not forbidden to create graphs that are invalid according to connection assertions. GR-GEN just enables you to check, whether a graph is valid or not. See also section 5.2.2, `validate`.

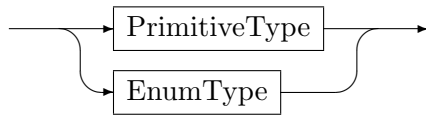
ConnectAssertions*NodeConstraint**RangeConstraint*

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 destination node type, respectively. *Number* is an `int` constant as defined in section 4.2. See 5.2.2, `validate`, for an example. Table 2.1 describes the multiplicity definitions.

$[n:*)$	The number of edges nodes of that type are incident to 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 ($m \geq n$ must hold).
$[*]$	Abbreviation for $[0:*)$.
$[+]$	Abbreviation for $[1:*)$
$[n]$	Abbreviation for $[n:n]$.

Table 2.1: GRGEN node constraint multiplicities

AttributeDeclarations

AttributeType

Defines a node or edge attribute. Possible types are enumerationn types (**enum**) and primitive types. See section [4.1](#) for a list of built-in primitive types.

The rule set language forms the core of GRGEN. Rule files refer to zero¹ or more graph models and specify a set of rewrite rules. The rule language covers the pattern specification and the replace/modify specification. Attributes of graph elements can be re-evaluated during a rule application. The following rewrite rule by Geiß et al. [4] gives a rough picture of the language:

EXAMPLE (4)

```

1  actions SomeActions using SomeModel;
2
3  rule SomeRule {
4    pattern {
5      n1 : NodeTypeA;
6      n2 : NodeTypeA;
7      hom(n1, n2);
8      n1 --> n2;
9      n3: NodeTypeB;
10     negative {
11       n3 -e1:EdgeTypeA-> n1;
12       if {n3.a1 == 42*n2.a1;}
13     }
14     negative {
15       n4: Node \ (NodeTypeB);
16       n3 -e1:EdgeTypeB-> n4;
17       if {typeof(e1) >= EdgeTypeA;}
18     }
19   }
20   replace {
21     n5: NodeTypeC<n1>;
22     n3 -e1:EdgeTypeB-> n5;
23     eval {
24       n5.a3 = n3.a1*n1.a2;
25     }
26   }
27 }
```

In this chapter we use excerpts of example 4 (*SomeRule*) for illustration purposes.

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

3.1 Building Blocks

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

Ident, IdentDecl

A non-empty character sequence of arbitrary length consisting of letters, digits or under-scores. The first character must not be a digit. *Ident* may be an identifier defined in a graph model (see 2.1). *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 4.4 for annotations of declarations.

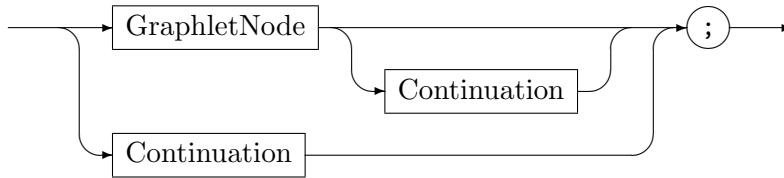
NOTE (4)

The GRGEN rule set language does not distinguish between declarations and definitions. More precisely, every declaration is also a definition. That means, that every element of a LHS (left hand side, see section 3.3) is actually mapped by a match.

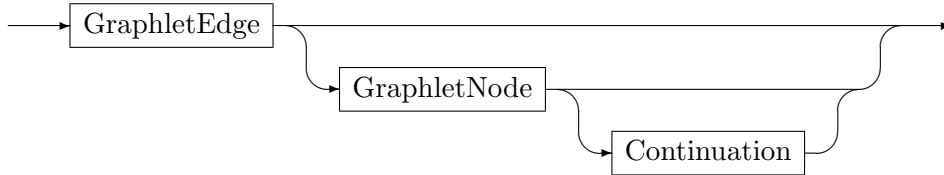
ModelIdent, TypeIdent, NodeType, EdgeType

These are (semantic) specializations of *Ident*. *TypeIdent* matches every type identifier, i.e. a node type, an edge type, an enum type or a primitive type. All the type identifiers are actually type *expressions*. See 4.3 for the use of type expressions.

Graphlet



Continuation



A graphlet specifies a connected subgraph. By graphlets GRGEN provides 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. A graph is specified piecewise by graphlets. In example 4, line 8, the statement `n1 --> n2` is the node identifier `n1` followed by the continuation graphlet `--> n2`.

All the graph elements of a graphlet have got a *name*. The name is either user assigned or an 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. Names may 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 denote a graph element of a graphlet, but also denotes the mapping of the abstract graph element of a graphlet to a concrete graph element of a host graph. So graph elements of different names are pairwise distinct except for homomorphically matched graph elements

(see section 3.3).

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, they are distinct.

Binding of names allows for splitting a single graphlet into multiple graphlets as well as defining cyclic structures.

EXAMPLE (5)

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

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

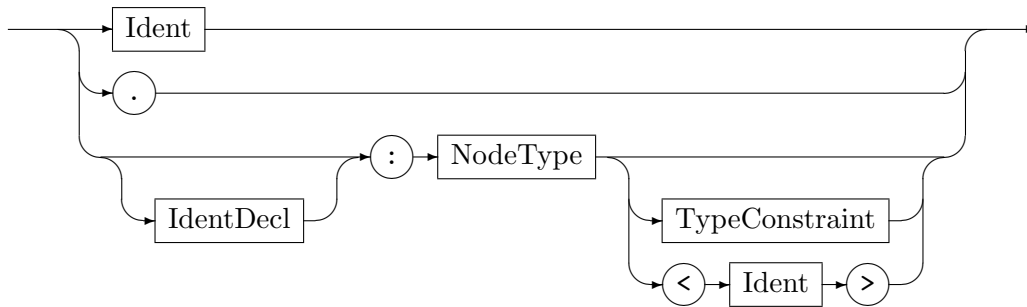
is equivalent to

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

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

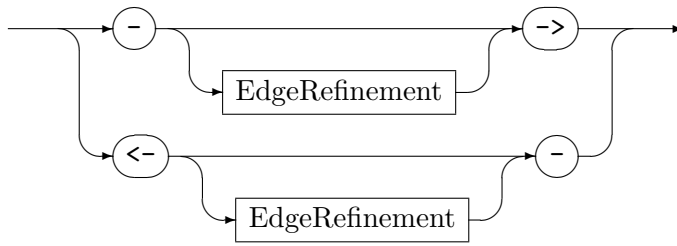
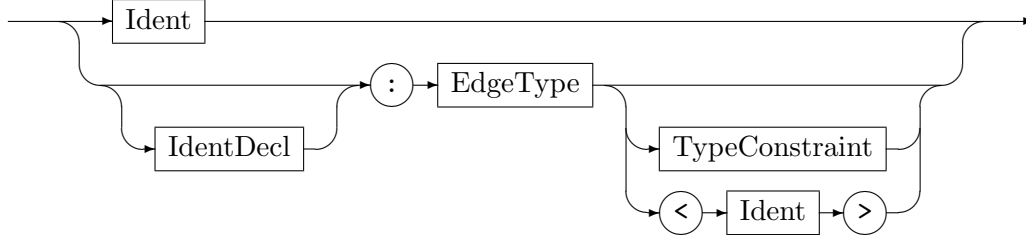
The visibility of names is determined by scopes. Scopes can be nested. Names of surrounding scopes are visible in inner scopes. Usually a scope is defined by `{` and `}`. In terms of scopes the replace part is a direct inner scope of the pattern part. In example 4 the negative condition from lines 14 to 18 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 4.3, *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 3.4, *Retyping*).

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

GraphletEdge*EdgeRefinement*

Specifies an edge. Anonymous edges are specified by `-->` or `<--`. For a more detailed specification you can use an inplace notated edge refinement clause. Type constraints are allowed in the pattern part only (see section 4.3, *TypeConstraint*). The `<>` operator retypes an edge. Retyping is allowed in the replace/modify part only (see section 3.4, *Retyping*).

Graphlet	Meaning
<code>-e:EdgeType-></code> ;	The name <code>e</code> is bound to an edge of type <code>EdgeType</code> or one of its subtypes
<code>-:EdgeType-></code> ;	<code>-\$1:EdgeType-></code> ;
<code>--></code> ;	<code>-\$1:Edge-></code> ;
<code>-e-></code> ;	The edge <code>e</code> 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 an edge: The graphlets

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

are illegal, because they would effect redirecting of edge `e`. However, the graphlets

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

are allowed, but the second graphlet is pointless. In particular the second graphlet does not identify or create any “copies”, neither occurring in the pattern part nor occurring in the replace part.

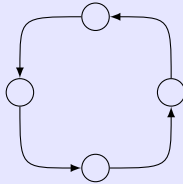
EXAMPLE (6)

Some attempts to specify a loop edge:

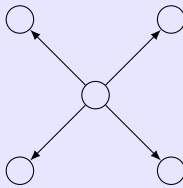
Graphlet	Meaning
<code>x:Node -e:Edge-> x;</code>	The edge <code>e</code> is a loop.
<code>x:Node -e:Edge-> ; -e-> x;</code>	The edge <code>e</code> is a loop.
<code>-e:Edge-> x:Node;</code>	The edge <code>e</code> may or may not be a loop.
<code>. -e:Edge-> .;</code>	The edge <code>e</code> is certainly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

EXAMPLE (7)

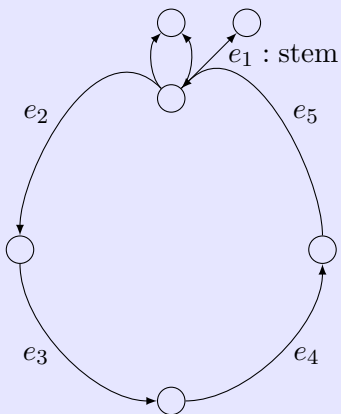
Some graphlets:



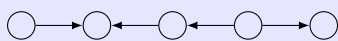
```
x:Node --> . --> . --> . --> x;
```



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



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



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

And some illegal graphlets:

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

Would effect redirecting of edge e.

```
x -e:T-> y; x -e-> x;
```

Would effect redirecting of edge e.

```
x:Node; negative {y:Node; hom(x,y)}
```

Here x must not occur in the hom statement.
See section 3.3 for further information.

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

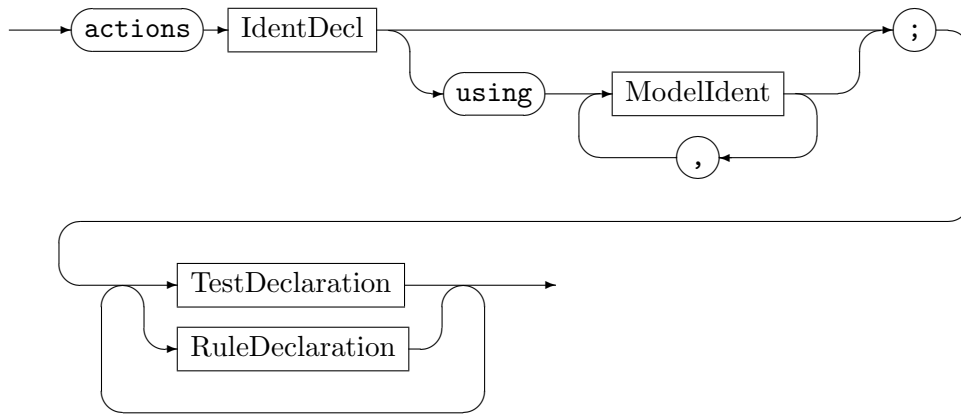
There must be at least a node between the edges.

NOTE (5)

Although both, the pattern part and the replace/modify part, use graphlets, there are subtle differences between them. It concerns the *TypeConstraint* clause, the retype operator $\langle \rangle$, 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.

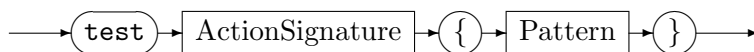
3.2 Rules and Tests

The structure of a rule set file is as follows:

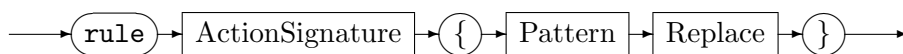


A rule set consists of the underlying graph models and several rewrite rules. In case of multiple graph models GRGEN uses the union of these models. In this case beware of conflicting declarations. There is no built-in conflict resolution mechanism like packages or namespaces for models. If necessary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

TestDeclaration



RuleDeclaration



Declares a single rewrite rule such as `SomeRule`. It consists of a pattern part (see 3.3) in conjunction with its rewrite/modify part (see 3.4). A *test* has no rewrite specification. It's intended to check whether (and maybe how many times) a pattern occurs.

EXAMPLE (8)

We define a test `SomeCond`

```

1 test SomeCond {
2   pattern {
3     n: SeldomNodeType;
4   }
5 }

```

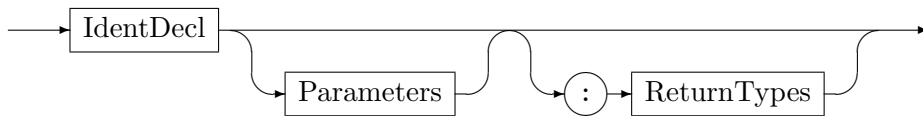
and execute in `GRSHELL`:

```

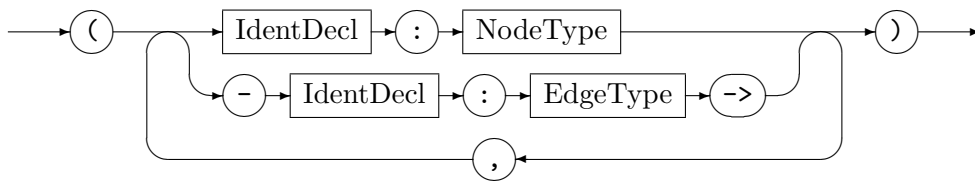
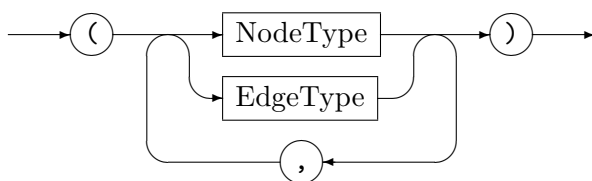
1 grs SomeCond & SomeRule

```

`SomeRule` will only be executed, if a node of type `SeldomNodeType` exists. For regular graph rewrite sequences in `GRSHELL` see 5.2.6.

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 provide users with the ability to pass graph elements to and from a rule. This is similar to parameters of procedural languages.

Parameters*ReturnTypes*

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` 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` has to actually search for such a graph element.

EXAMPLE (9)

Assume the following rule:

```

1 rule r(-e:Edge->; x:Node) {
2   pattern {
3     x --> ;
4     negative {
5       <-- x -->;
6     }
7   }
8   modify {}
9 }
```

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

```

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

In particular x will not be incident to e .

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 3.4, **return**.

EXAMPLE (10)

We extend `SomeRule` with a variable node to find and we want it to return the rewritten graph elements `n5` and `e1`.

```

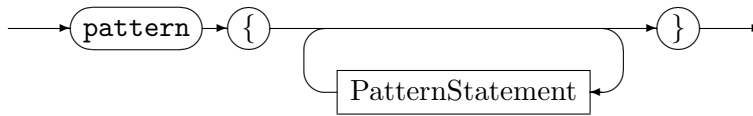
1 rule SomeRuleExt(varnode: Node): (Node, EdgeTypeB) {
2   pattern {
3     n1: NodeTypeA;
4     ...
5   }
6   replace {
7     varnode;
8     ...
9     return(n5, e1);
10   eval {
11     ...

```

We don't define `varnode` within the pattern part because this is already covered by the parameter specification itself.

3.3 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. Even stronger – a graph element of the host graph, that is matched by a statement, is *bound*, i.e. it cannot be part of another pattern statement, unless you use the `hom` operator. An empty pattern always produces exactly one (empty) match. This is caused by the uniqueness of the totally undefined function.

Pattern statements may define names for graph elements for use by other pattern statements or replace statements. Such names may be used before their declaration. See section 3.1 for a detailed explanation of names and graphlets.

NOTE (6)

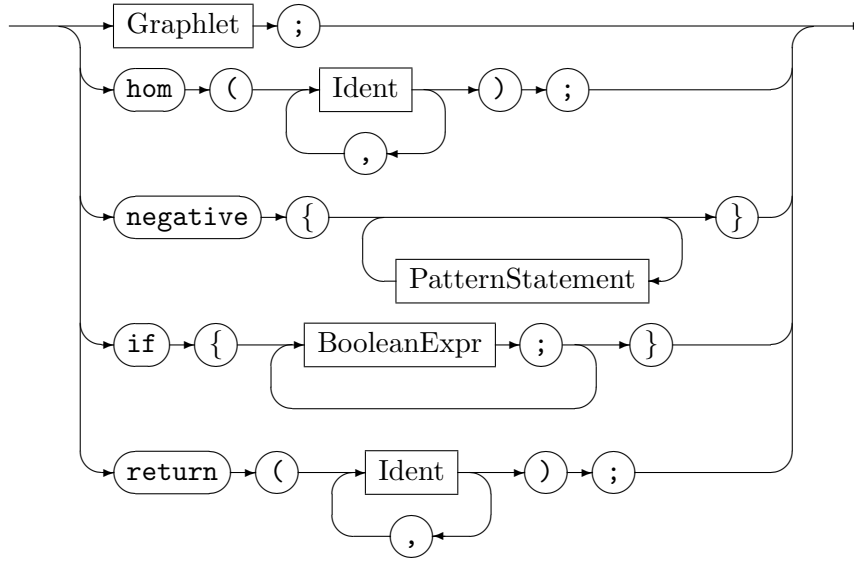
The application of a rule is not deterministic (remember the introducing example in 1.5), in particular there may be more than one subgraph that matches the pattern. Whereas the GRShell selects one of them arbitrarily (without further abilities to control the selection), the underlying LIBGR provides 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 matches the pattern and rewrite one of this matches. By returning a collection of all matches, the LIBGR retains the complete graph rewrite process under control. As a LIBGR user use the following methods of the `IAction` interface:

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

This might look like this in C#:

```
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 the regular graph rewrite sequences introduces a further variant of indeterminism on rule application level: The `$<op>` flag marks the operator `<op>` as commutative, i.e. the execution order of its operands (rules) is indeterministic. See section 5.2.6 for further information on regular graph rewrite sequences.

PatternStatement

The semantics of the various pattern statements:

Graphlet.

Graphlets specify connected subgraphs. See section 3.1 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 shall have a common supertype. Several homomorphically matched graph elements will be mapped to a graph element of a common supertype.

In example 4 node **n1** and node **n2** may be the same node. This is possible because they are of the same type (**NodeTypeA**).

The **hom** operator is transitive, i.e. **hom(a, b); hom(b, c);** implies **hom(a, 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. [10]). NACs must not be nested. NACs possess an own scope. Names defined within a NAC are not alive outside the NAC. Identifiers from surrounding scopes may be overwritten. In general NACs does 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 (11)

We specify a node `x` with out-degree 2:

```

1 pattern {
2   <-- x:Node -->;
3   negative {
4     <-- x -->;
5     x -->;
6   }
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 3.4, *Return Values*.

Keep in mind that using type constraints or the `typeof` operator might be helpful. See section 4.3 for further information.

3.4 Replace/Modify Part

For the task of rewriting GRGEN provides two different modes: A replace mode and a modify mode.

Replace mode.

The semantics of this mode is to delete every graph element of the pattern that is not used (denoted) in the replace part, keep every graph element that is used, and create every additionally defined graph elements. “Using” means using a graph element in a graphlet. Attribute calculations are no using occurrences.

In example 4 `SomeRuleExt` 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.

Modify mode.

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

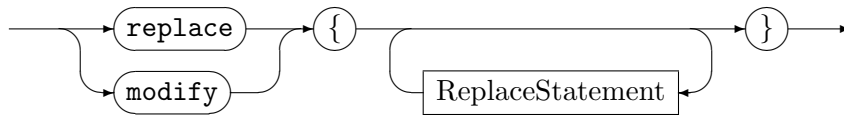
EXAMPLE (12)

How might example 4 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 omitted. So we have

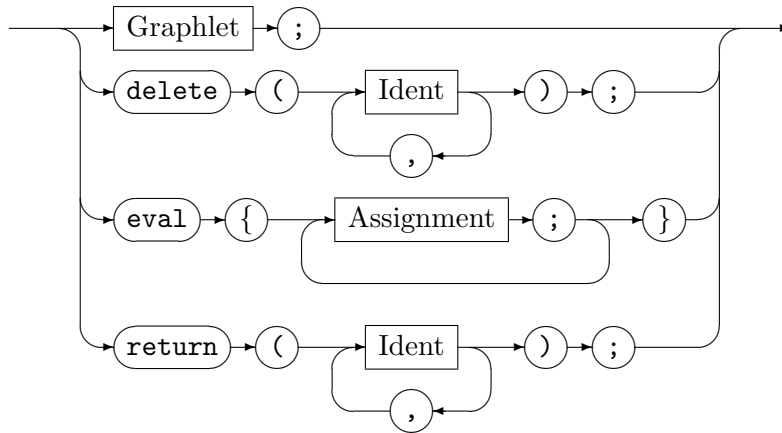
```

1 rule SomeRuleModify {
2   pattern {
3     ...
4     n1 -e0:Edge-> n2;
5     ...
6   }
7   modify {
8     n5 : NodeTypeC<n1>;
9     n3 -e1:EdgeTypeB-> n5;
10    delete(e0);
11    eval {
12      ...

```

Replace

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

ReplaceStatement

The semantics of the various pattern statements:

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 3.1 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.

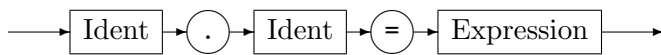
Return Values.

Graph elements of the replace/modify part can be returned according to the return types in the signature (see 3.2, *ActionSignature*). The `return` statement must not occur multiple times. 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.

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. Retyping differs from a type cast: During replacement both of the graph elements are alive. Specifically both of them are available for evaluation. Furthermore the source and destination types need not to be on a path in the directed type hierarchy tree, rather their relation can be arbitrary.

The edge specification as well as *ReplaceNode* supports retyping. In example 4 node `n5` is a retyped node stemming from node `n1`.

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 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 doesn't make sense) attributes of graph elements to be deleted.

EXAMPLE (13)

```

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 nonsense example yields `y.i = 42`, `y.j = 1`.

CHAPTER 4

TYPES AND EXPRESSIONS

In the following sections *Ident* refers to an identifier of the graph model language (see section 2.1) or the rule set language (see section 3.1). *TypeIdent* is an identifier of a type.

4.1 Built-In Types

Besides user-defined node types, edge types and enum types, GrGen supports the built-in primitive types in table 4.1. Table 4.2 lists GRGEN’s implicit type casts and the allowed

boolean	Covers the values true and false .
int	A 32-bit signed integer, in two’s complement representation.
float, double	A floating-point number in IEEE 754 format with single precision or double precision respectively.
string	A sequence of digits, letters, underscores and white spaces. Strings are of arbitrary length and enclosed by double quotes.

Table 4.1: GRGEN built-in types

explicit type casts. Of course you are free to express an implicit type cast by an explicit type cast as well as “cast” a type to itself, except for enum types. The cast operator does never accept an enum type.

EXAMPLE (14)

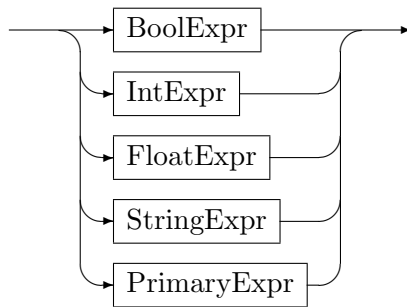
`myfloat = myint; mydouble = (float)myint; mystring = (string)mybool` is allowed, `myenum = (myenum)int; myfloat = mydouble; myint = (int)mybool` is forbidden.

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

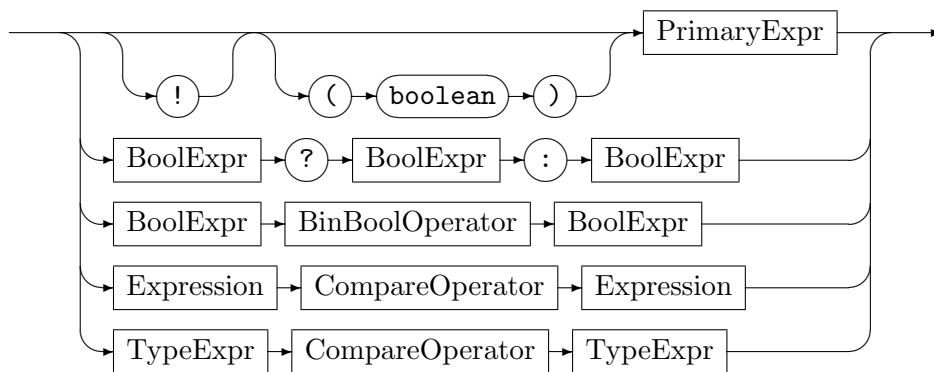
Table 4.2: GRGEN type casts.

4.2 Expressions

Expression



BoolExpr



As in C [6], `!` negates a boolean. Table 4.4 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 *CompareOperator* is one of the following operators:

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

Table 4.2 describes the semantics of compare operators on type expressions.

The *BinBoolOperator* is one of the operators in table 4.4.

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

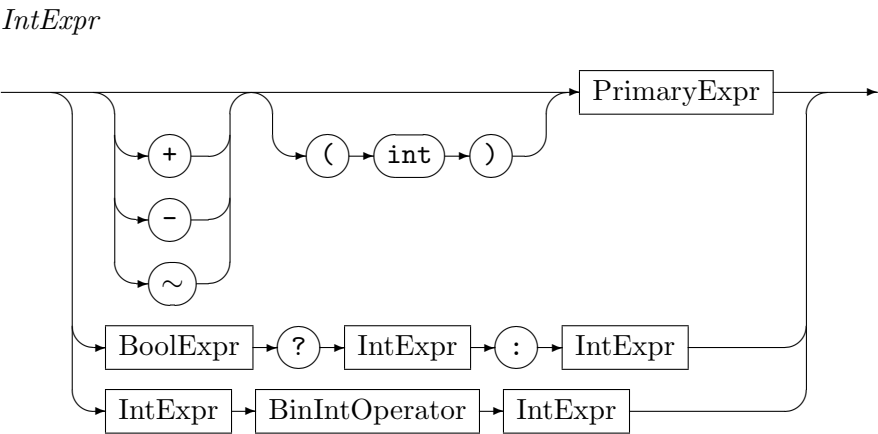
Table 4.3: Compare operators on type expressions

NOTE (7)

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

\sim	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.

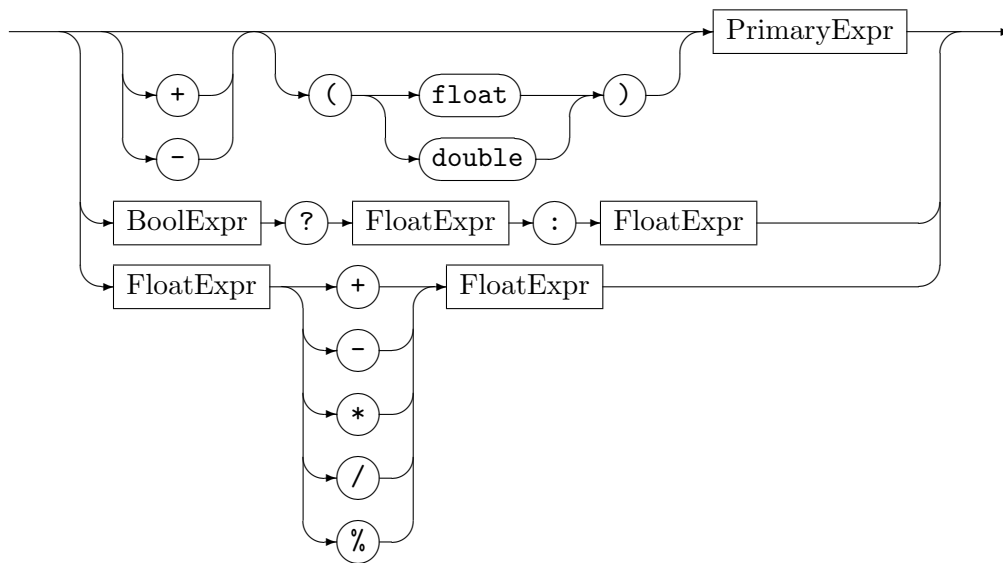
Table 4.4: Binary boolean operators, in ascending order of precedence



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 4.5.

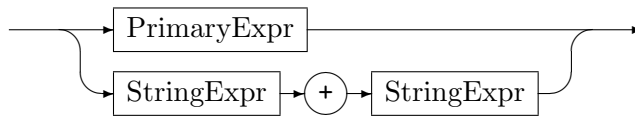
\sim $\&$ $ $	Bitwise XOR, AND and OR
\ll \gg \ggg	Bitwise shift left, bitwise shift right and bitwise shift right preserving the sign
$+$ $-$	Addition and subtraction
$*$ $/$ $\%$	Multiplication, integer division and modulo

Table 4.5: Binary integer operators, in ascending order of precedence

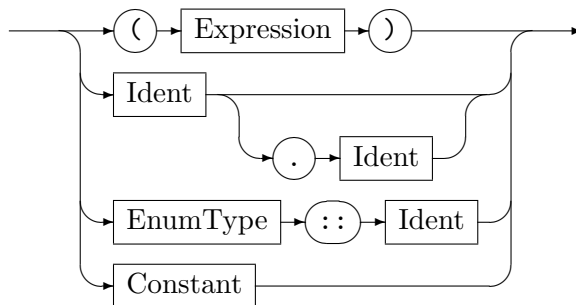
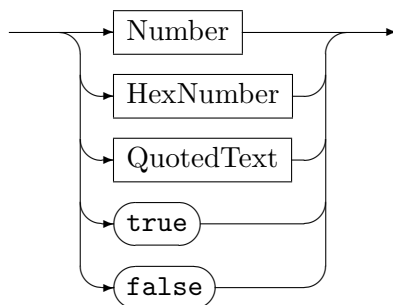
FloatExpr

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 $%$ operator works analogous to the integer modulo operator. For instance $4.5 \% 2.3 == 2.2$. GRGEN implements the Java semantics.

StringExpr

The operator $+$ concatenates two strings.

PrimaryExpr*Constant**Number.*

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

HexNumber.

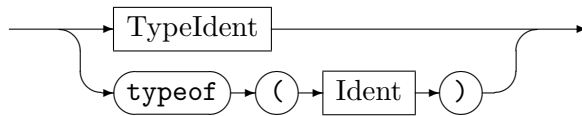
Is an `int` constant in hexadecimal notation starting with `0x`.

QuotedText.

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

4.3 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. So *Ident* has to be a node or edge identifier.

EXAMPLE (15)

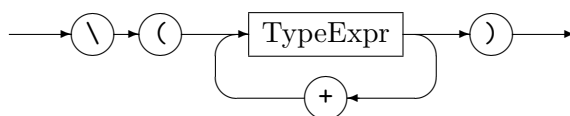
The following rule will add a reverse edge to a one-way street.

```

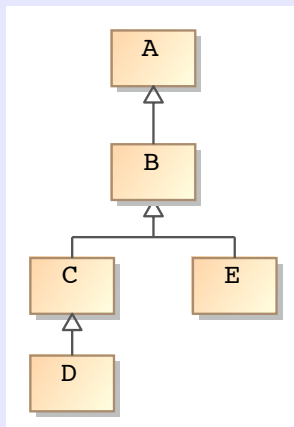
1 rule oneway {
2   pattern {
3     a:Node -x:street-> y:Node;
4     negative{
5       y -:typeof(x)-> a;
6     }
7   }
8   replace {
9     a -x-> y;
10    y -:typeof(x)-> a;
11  }
12 }
```

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.

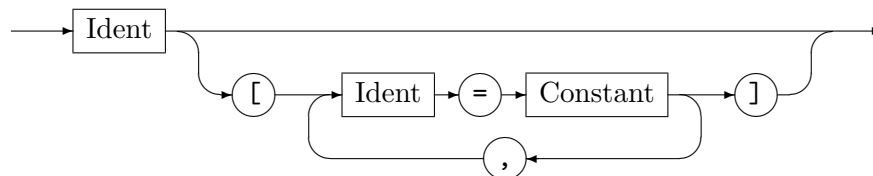
EXAMPLE (16)

The expression $A \setminus (C+E)$ applied to the type hierarchy on the left side covers A and B .

4.4 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 `prio` has an effect so far.

Key	Value Type	Applies to	Meaning
<code>prio</code>	<code>int</code>	node, edge	Changes the ranking of a graph element for search plans. The default is <code>prio=1000</code> . Graph elements with high values are likely to appear prior to graph elements with low values in search plans.

Table 4.6: Annotations

EXAMPLE (17)

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

CHAPTER 5

GRSHELL LANGUAGE

GRSHELL is a shell application of LIBGR. It belongs to GRGEN's standard equipment. GRSHELL is capable of creating, manipulating, and dumping graphs as well as performing graph rewriting and debugging graph rewriting.

The GRSHELL provides a line oriented scripting language. GRSHELL scripts are structured by simple statements separated by line breaks.

5.1 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

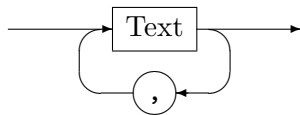
May be one of the following:

- 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 (`' '`).

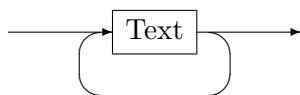
Number

Is an `int` or `float` constant in decimal notation (see also section 4.1).

Parameters



SpacedParameters



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

Filename

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

Variable

Identifier of a variable that contains a graph element.

Node, Edge

Variable identifier or persistent name of a node resp. an edge (see also example 18).

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.

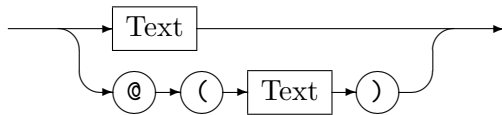
Action

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.

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 5.2.3):

GraphElement

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

EXAMPLE (18)

We insert a node, anonymously and with a constructor:

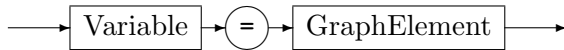
```

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

```

NOTE (8)

Persistent names belong to a specific graph, whereas variables belong to the current GRShell environment. Persistent names will be saved (`save graph...`, see 5.2.5) and, if you visualize a graph (`dump graph...`, see 5.2.5), graph elements will be labeled with their persistent names. Persistent names have to be unique for a graph.

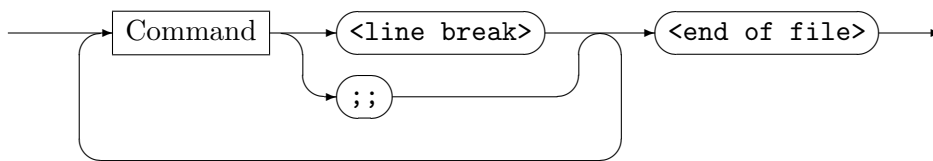


Assigns the variable or persistent name *GraphElement* to *Variable*. If *Variable* has not been defined yet, it will be defined implicitly. As usual for scripting languages, variables have neither types nor declarations.

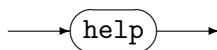
5.2 GRShell Commands

This section describes the GRShell commands. Commands are assembled from basic elements. As stated before commands are terminated by a line breaks. Alternatively commands can be terminated by the `;;` symbol.

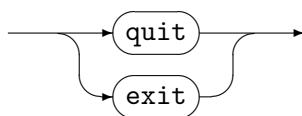
Script



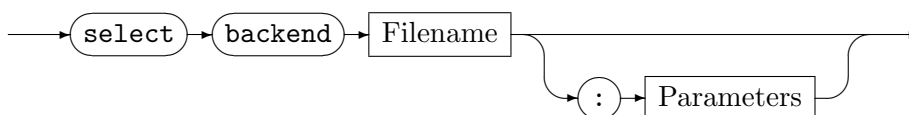
5.2.1 Common Commands



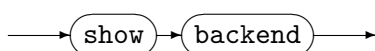
Displays an information message describing supported commands.



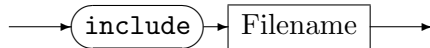
Quits GRShell. If GRShell is opened in debug mode, currently active graph displays (such as YCOMP) will be closed as well.



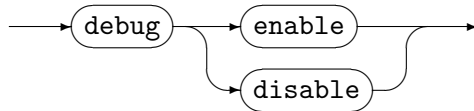
Selects a backend that handles graph and rules representation. *Filename* has to be a .NET assembly (e.g. `lgspBackend.dll`). Comma-separated parameters can be supplied optionally. If so, the backend must support these parameters.



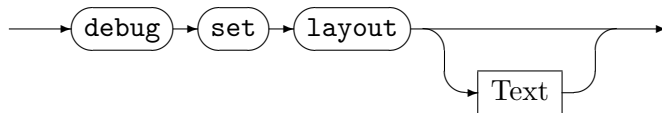
List all the parameters supported by the currently selected. The parameters can be provided to the `select backend` command.



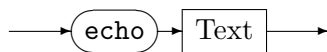
Executes the GRShell script *Filename*. A GRShell script is just a plain text file containing GRShell commands. They are treated as they would be entered interactively, except for parser errors. If a parser error occurs, execution of the script will stop immediately.



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



Sets the default graph layout algorithm to *Text*. If *Text* is omitted, a list of available layout algorithms is displayed. See section 1.6.4 on YCOMP layouters.



Prints *Text* onto the GRShell command prompt.



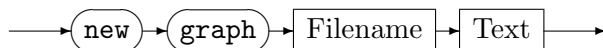
Passes *CommandLine* to the operating system. *CommandLine* is an arbitrary text the operating system attempts to execute.

EXAMPLE (19)

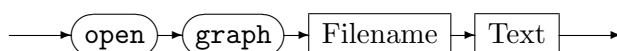
On a Linux machine you might execute

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

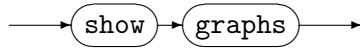
5.2.2 Graph Commands



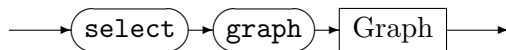
Creates a new graph with the model specified in *Filename*. Its name is set to *Text*. The model file can be either source code (e.g. `turing_machineModel.cs`) or a .NET assembly (e.g. `lgsp-turing_machineModel.dll`).



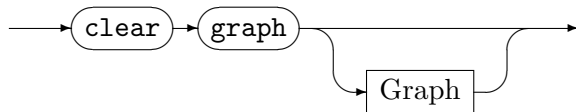
Opens the graph *Text* stored in the backend. However, the *LGSPBackend* doesn't support persistent graphs. *LGSPBackend* is the only backend so far. Therefore this command is currently useless.



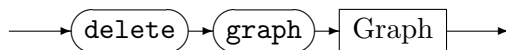
Displays a list of currently available graphs.



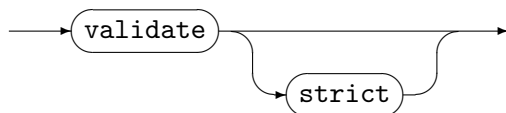
Selects the current working graph. This graph act as *host graph* for regular graph rewrite sequences (see also sections ?? and 5.2.6). Though you can define multiple graphs, only one graph can be the active “working graph”.



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 edge constraints 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 (20)

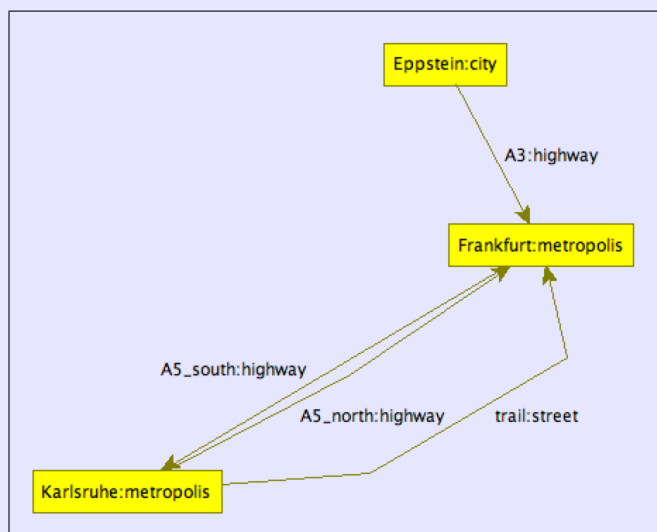
We reuse a simplified version of the street map model from chapter 2:

```

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:

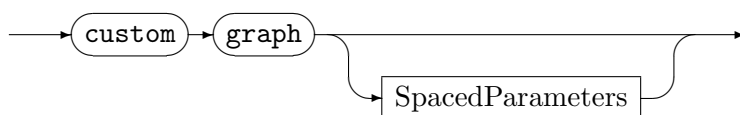


This graph is valid, but not strict valid.

```

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 >

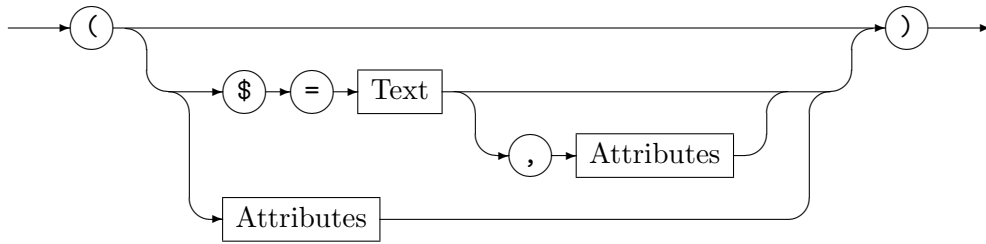
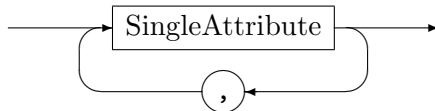
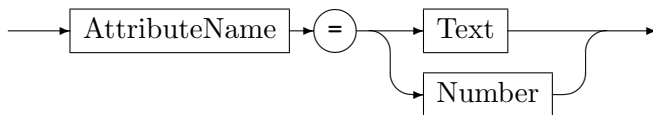
```



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 5.3).

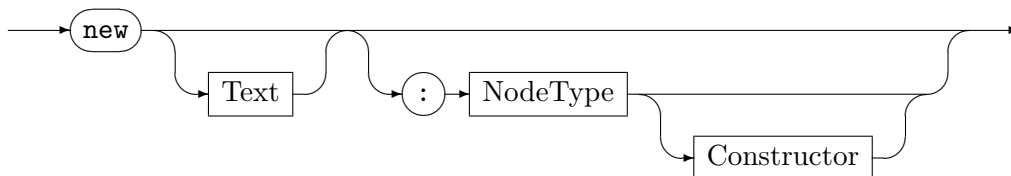
5.2.3 Graph Manipulation Commands

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

Constructor*Attributes**SingleAttribute*

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, enum \leftarrow 0; boolean \leftarrow false; float, double \leftarrow 0.0; string \leftarrow "").

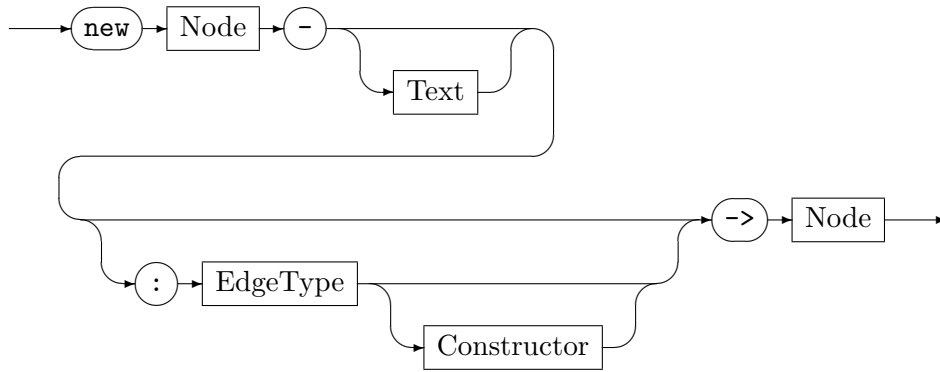
The `$` marks a special attribute: an unique identifier of the new graph element. This identifier is also called *persistent name* (see example 18). This name can be specified by a constructor only.



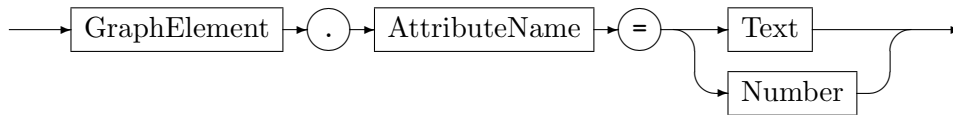
Creates a new node within the current graph. Optionally a variable *Text* is assigned to the new node. If *NodeType* is supplied, the new node will be of type *NodeType* and attributes can be initialized by a constructor. Otherwise the node will be of the base node class type *Node*.

NOTE (9)

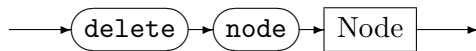
The GRShell can reassign variables. This is in contrast to the rule language (chapter 3), where we use *names*.



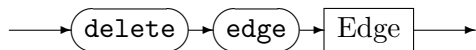
Creates a new edge within the current graph between the specified nodes, directed towards the second *Node*. Optionally a variable *Text* is assigned to the new edge. If *EdgeType* is supplied, the new edge will be of type *EdgeType* and attributes can be initialized by a constructor. Otherwise the edge 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*.

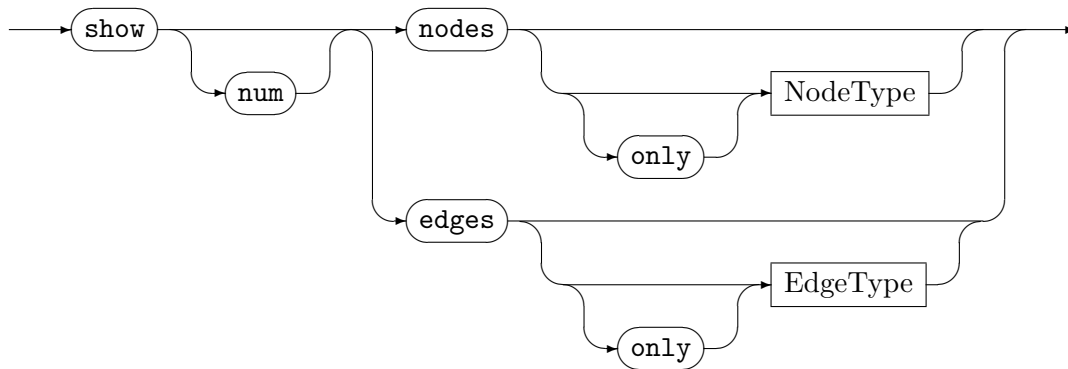


Deletes the node *Node* from the current graph. Incident edges will be deleted as well.



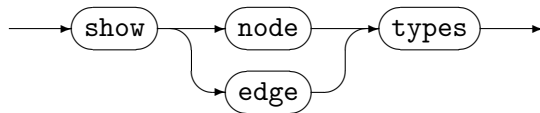
Deletes the edge *Edge* from the current graph.

5.2.4 Graph Query Commands

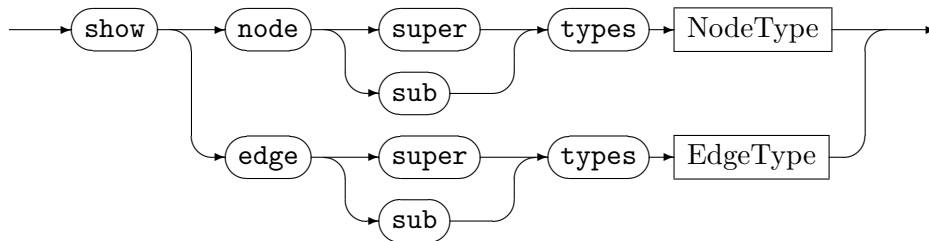


Gets the persistent names and the types of all the nodes/edges of the current graph. If a node type or edge type is supplied, only elements compatible to this type are considered. The **only** keyword excludes subtypes. Nodes/edges without persistent names are shown with a pseudo-name.

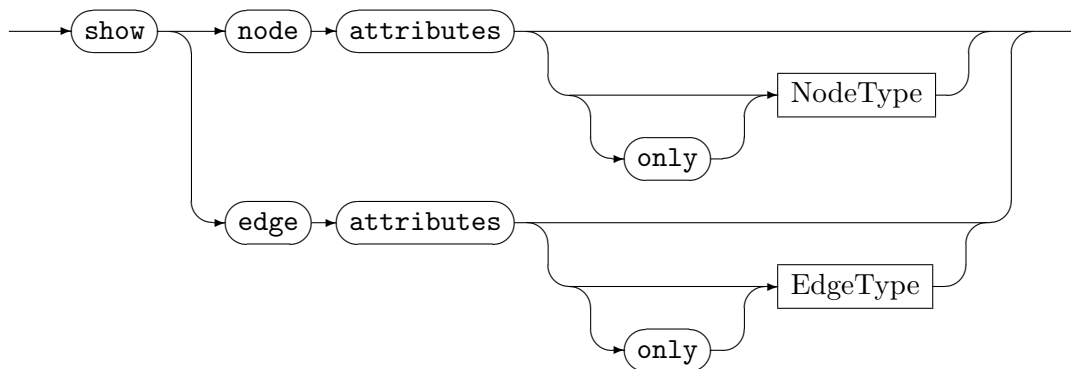
If the command is specified with **num**, only the number of nodes/edges will be displayed.



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



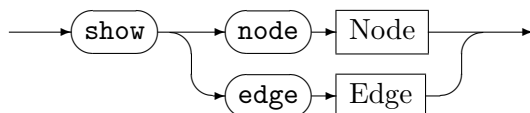
Gets the inherited/descended types of *NodeType*/*EdgeType*.



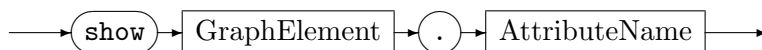
Gets the available node/edge attribute types. If *NodeType*/*EdgeType* is supplied, only attributes defined in *NodeType*/*EdgeType*. The **only** keyword excludes inherited attributes.

NOTE (10)

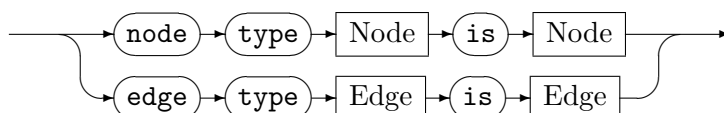
This is in contrast to the `show num...`, `show nodes...` and `show edges...` commands where types and subtypes are specified.



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

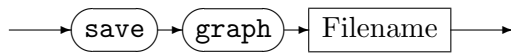


Gets the value of a specific attribute.



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

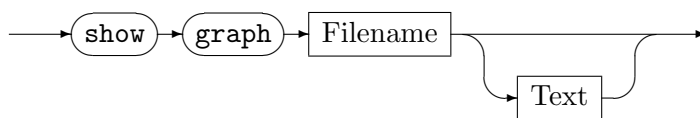
5.2.5 Graph Output Commands



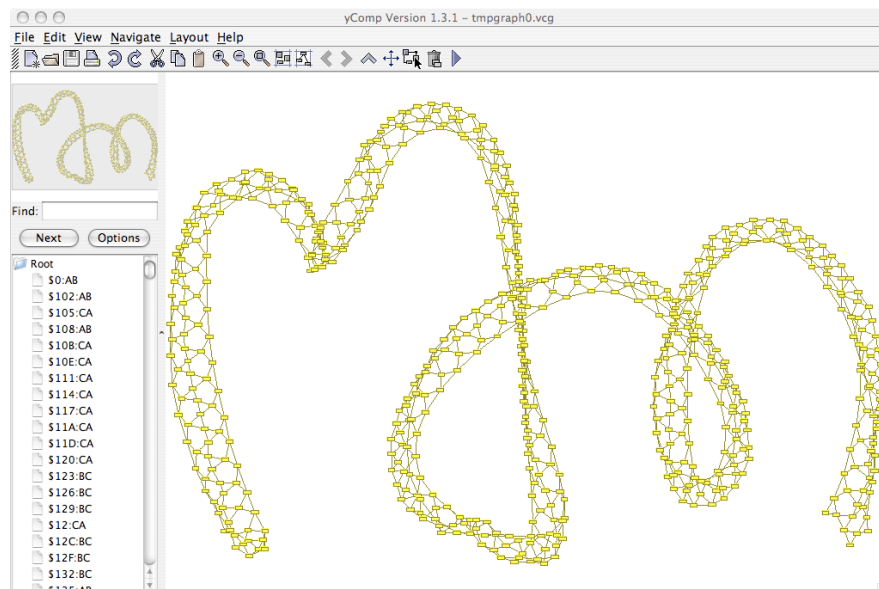
Dumps the current graph as GRShell script into *Filename*. The created script includes

- selecting the backend
- creating all nodes and edges
- restoring the persistent names (see 5.2.3),

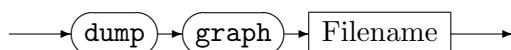
but not necessarily using the same commands you typed in during construction.



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 YCOMP¹ as executable, this may look like



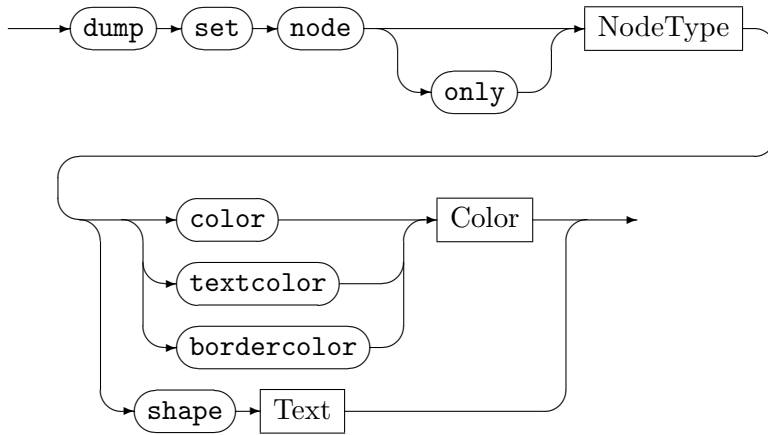
The temporary file will be deleted, when *Filename* is terminated, if GRShell is still running at this time.



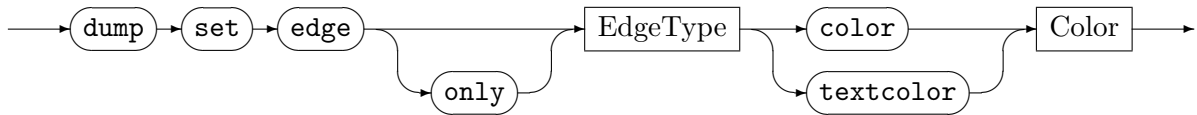
Dumps the current graph as VCG formatted file into *Filename*.

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

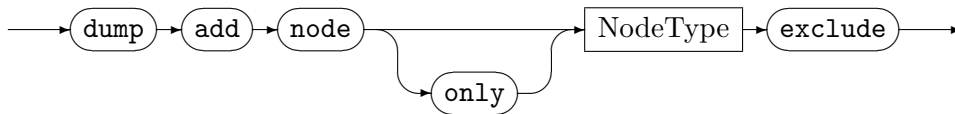
¹See section 1.6.4.



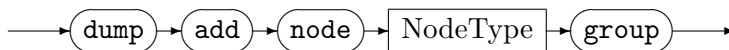
Sets the color or text color or border color resp. the shape of the nodes of type *NodeType* and all of its subtypes. The keyword **only** excludes the subtypes. The following shapes are supported: **box**, **triangle**, **circle**, **ellipse**, **rhom**, **hexagon**, **trapeze**, **uptrapeze**, **lparallelogram**, **rparallelogram**. Those are shape names of YCOMP (see []).



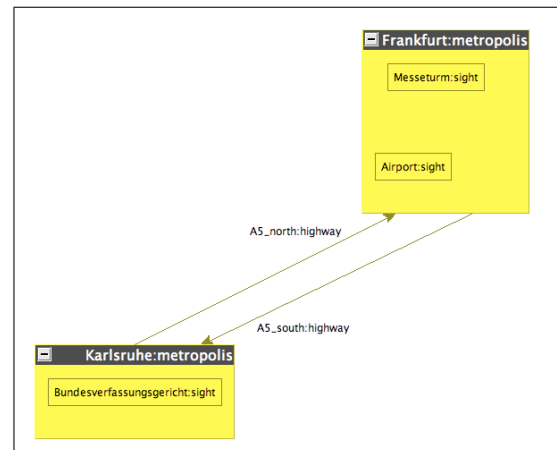
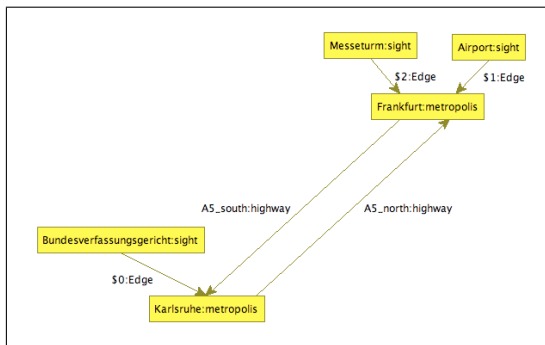
Sets the color or text color of the edges of type *EdgeType* and all of its subtypes. The keyword **only** excludes the subtypes.



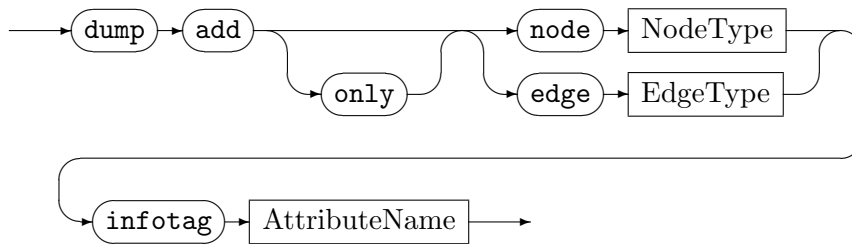
Excludes nodes of type *NodeType* and all of its subtypes as well as their incident edges from output. The keyword **only** excludes the subtypes.



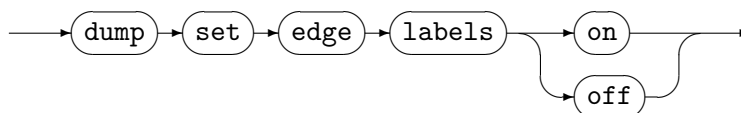
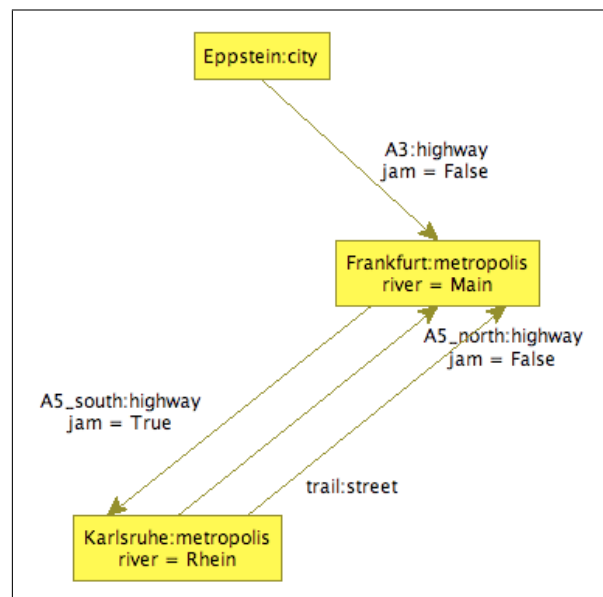
Declares *NodeType* and subtypes of *NodeType* as group node type. All the different typed nodes that points to a node of type *NodeType* (i.e. there is a directed edge between such nodes) will be grouped and visibly enclosed by the *NodeType*-node. The following example shows *metropolis* ungrouped and grouped:



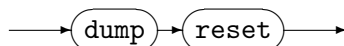
right side: dumped with `dump add node metropolis group`



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:



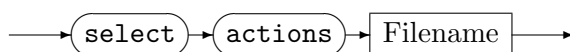
Specifies, whether edge labels will be displayed or not (default to “on”).



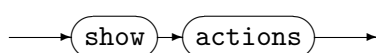
Reset all style options (`dump set...`) to their default values.

5.2.6 Action Commands (GRS)

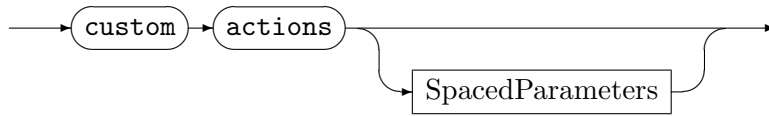
An *action* denotes a graph rewrite rule.



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.



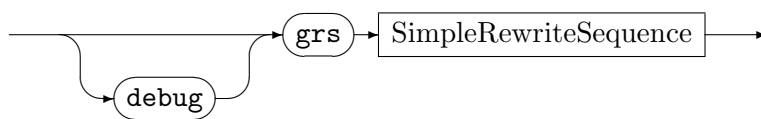
Lists all the rules of the loaded rule set, their parameters, and their return values. Rules can return a set of graph elements.



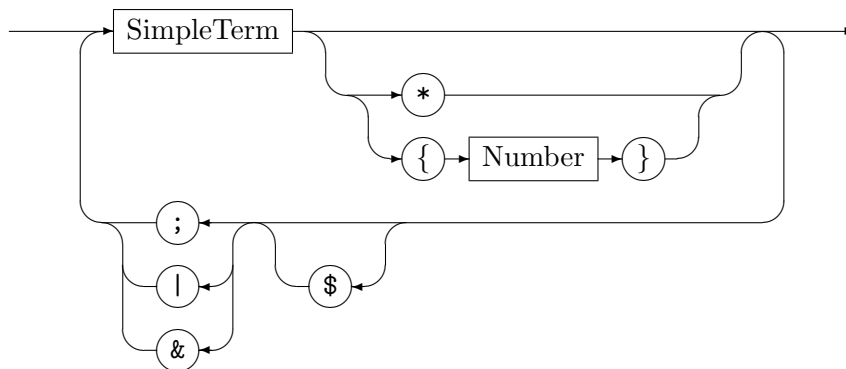
Executes an action specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSPBackend see section 5.3).

Regular Graph Rewrite Sequences

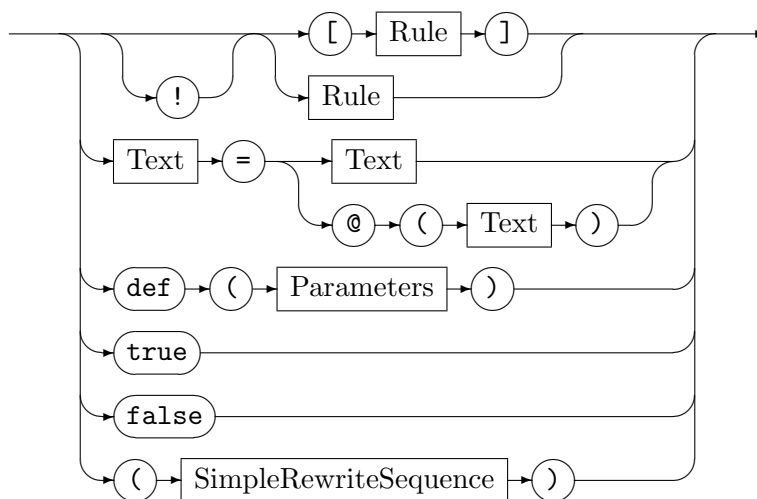
Basically a graph rewrite command looks like this:



SimpleRewriteSequence



SimpleTerm



Rule

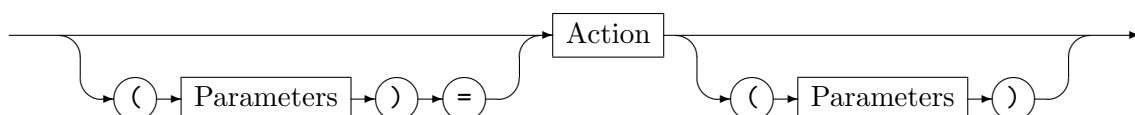


Table 5.1 lists graph rewrite expressions at a glance. The operators hold the following order of precedence, starting with the lowest precedence:

; | &

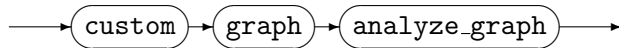
Variables can be assigned to graph elements returned by rules using `(Parameters) = Action`. The desired variable identifiers have to be listed in *Parameters*. Graph elements required by rules must be provided using `Action (Parameters)`, where *Parameters* is a list of variable identifiers. For undefined variables the specific element constraint of *Action* will be ignored (every element matches).

Use the `debug` option to trace the rewriting process step-by-step. During execution YCOMP will display every single step². The debugger can be controlled by GRShell. The debug commands are shown in table 5.2.

5.3 LGSPBackend Custom Commands

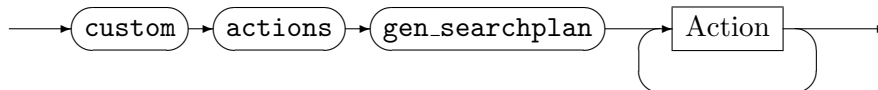
The LGSPBackend supports the following custom commands:

5.3.1 Graph Related Commands



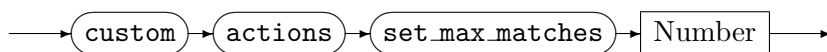
Analyzes the current working graph. The analysis data provides vital information for efficient search plans. Analysis data are available as long as GRShell is running, i.e. when the working graph changes the analysis data is still available but maybe obsolete.

5.3.2 Action Related Commands



Creates a search plan for each rewrite rule *Action* using a heuristic method and the analyze 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. In deed there might be some trial-and-error steps necessary to get a 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]`. For *Number* less or equal to zero, the constraint is reset.

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

<code>s ; t</code>	Concatenation. First, <code>s</code> is executed, afterwards <code>t</code> is executed. The sequence <code>s ; t</code> is <i>successfully</i> executed iff <code>s</code> or <code>t</code> is successfully executed.
<code>s t</code>	XOR. First, <code>s</code> is executed. Only if <code>s</code> fails, <code>t</code> is executed. The sequence <code>s t</code> is successfully executed iff <code>s</code> or <code>t</code> is successfully executed.
<code>s & t</code>	Transactional AND. First, <code>s</code> is executed, afterwards <code>t</code> is executed. If <code>s</code> or <code>t</code> fails, the action will be terminated and a rollback to the state before <code>s & t</code> is performed.
<code>\$<op></code>	Flags the operator <code><op></code> as commutative. Usually operands are executed / evaluated from left to right with respect to bracketing (left-associative). But the sequences <code>s, t, u</code> in <code>s \$<op> t \$<op> u</code> are executed / evaluated in arbitrary order.
<code>s *</code>	Executes <code>s</code> repeatedly as long as its execution does not fail.
<code>s {n}</code>	Executes <code>s</code> repeatedly as long as its execution does not fail, but anyway <code>n</code> times at most.
<code>!</code>	Found matches are dumped into VCG formatted files. Every match produces three files within the current directoy: <ol style="list-style-type: none"> 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
<code>Rule</code>	Only the first pattern match produced by the action <code>Rule</code> will be rewritten.
<code>[Rule]</code>	Every pattern match produced by the action <code>Rule</code> will be rewritten. Note: This operator is mainly added for benchmark purposes. Its semantic is not equal to <code>Rule*</code> . 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.
<code>v = w</code>	The variable <code>v</code> is assigned to <code>w</code> . If <code>w</code> is undefined, <code>v</code> will be undefined, too.
<code>v = @(<i>x</i>)</code>	The variable <code>v</code> is assigned to the graph element identified by <code>x</code> . If <code>x</code> is not defined any more, <code>v</code> will be undefined, too.
<code>def(<i>Parameters</i>)</code>	Is <i>successful</i> if all the graph elements in <i>Parameters</i> exist, i.e. if all the variables are defined.
<code>true</code>	A constant acting as a successful match.
<code>false</code>	A constant acting as a failed match.

Let `s, t, u` be graph rewriting sequences, `v, w` variable identifiers, `x` an identifier of a graph element, `<op> ∈ {;, |, &}` and `n ∈ ℕ0`.

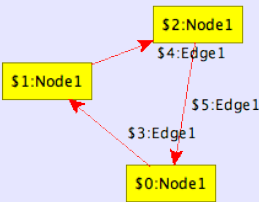
Table 5.1: Graph rewriting expressions

EXAMPLE (21)

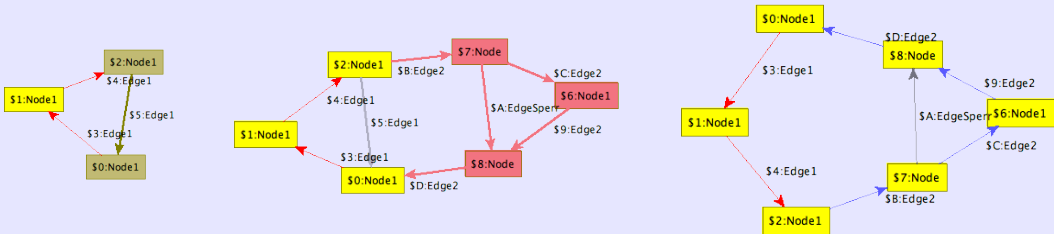
We demonstrate the debug commands with a slightly adjusted script for the Koch snowflake from GRGEN’s examples (see also 6.1). The graph rewriting sequence is

```
1 debug grs (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	—

s (tep)	Executes the next rewriting rule (match and rewrite)
d (etailed step)	Executes a rewriting rule in a three-step procedure: matching, highlighting elements to be changed, doing rewriting
n (ext)	Ascends one level up within the Kantorowitsch tree of the current rewrite sequence
(step) o (ut)	Continues a rewriting 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)	Continues execution without further stops
a (bort)	Cancels the execution immediately

Table 5.2: GRShell debug commands

CHAPTER 6

EXAMPLES

6.1 Fractals

The GRGEN package ships with samples for fractal generation. We will construct the Sierpinski triangle and the Koch snowflake. First of all we have to compile the model and rule set files. So execute in GRGEN'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. 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 [5.2.6](#) for further information. The resulting graphs should look like figures [6.1](#) and [6.2](#).



Figure 6.1: Sierpinski triangle



Figure 6.2: Koch snowflake

6.2 Busy Beaver

We want GRGEN to work as hard as a busy beaver [8, 3]. Our busy beaver is a Turing machine, that has got five states, writes 1,471 bars onto the tape and terminates [9]. So first of all we design a Turing machine as graph model. Besides this example shows that GRGEN 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 GRSELL.

6.2.1 Graph Model

Let's start the model file *TuringModel.gm* with

```
1 model TuringModel;
```

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 cell (*TapePosition*) does not have an incoming and outgoing edge respectively. Therefore we have the node constraint $[0 : 1]$.

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

Finally we need states and transitions. The current configuration is modeled with a *RWHead* edge pointing to a *TapePosition* node. *State* nodes are connected with *WriteValue* nodes via *value* edges and from a *WriteValue* node a *moveLeft*/*moveRight* edge leads to the next state.

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

6.2.2 Rule Set

Now the rule set: we begin the rule set file *Turing.grg* with

```
1 actions Turing 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 propagate a new state as current state.

As you can see a transition of the Turing machine is split into two graph rewriting 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   pattern {
4     s:State ->:RWHead-> tp:TapePosition -zv:zero-> tp;
5     s -zr:zero-> wv:WriteValue;
6   }
7   replace {
8     s -zr-> wv;
9     tp -zv-> tp;
10    wv ->:RWHead->tp;
11  }
12 }

```

We take the current state s and the current cell tp 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:

```

13 rule readOneRule {
14   pattern {
15     s:State ->:RWHead-> tp:TapePosition -ov:one-> tp;
16     s -or:one-> wv:WriteValue;
17   }
18   replace {
19     s -or-> wv;
20     tp -ov-> tp;
21     wv ->:RWHead-> tp;
22   }
23 }
24
25 rule readEmptyRule {
26   pattern {
27     s:State ->:RWHead-> tp:TapePosition -ev:empty-> tp;
28     s -er:empty-> wv:WriteValue;
29   }
30   replace {
31     s -er-> wv;
32     tp -ev-> tp;
33     wv ->:RWHead-> tp;
34   }
35 }
36
37 rule writeZeroRule {
38   pattern {
39     wv:WriteZero -rw:RWHead-> tp:TapePosition ->:value-> tp;
40   }
41   replace {
42     wv -rw-> tp ->:zero-> tp;
43   }
44 }
45
46 rule writeOneRule {
47   pattern {
48     wv:WriteOne -rw:RWHead-> tp:TapePosition ->:value-> tp;
49   }
50   replace {

```

```

51     wv -rw-> tp -:one-> tp;
52 }
53 }
54
55 rule writeEmptyRule {
56   pattern {
57     wv:WriteEmpty -rw:RWHead-> tp:TapePosition -:value-> tp;
58   }
59   replace {
60     wv -rw-> tp -:empty-> tp;
61   }
62 }
63
64 rule moveLeftRule {
65   pattern {
66     wv:WriteValue -m:moveLeft-> s:State;
67     wv -:RWHead-> tp:TapePosition <-r:right- ltp:TapePosition;
68   }
69   replace {
70     wv -m-> s;
71     s -:RWHead-> ltp -r-> tp;
72   }
73 }
74
75 rule moveRightRule {
76   pattern {
77     wv:WriteValue -m:moveRight-> s:State;
78     wv -:RWHead-> tp:TapePosition -r:right-> rtp:TapePosition;
79   }
80   replace {
81     wv -m-> s;
82     s -:RWHead-> rtp <-r- tp;
83   }
84 }
85
86 rule dontMoveRule {
87   pattern {
88     wv:WriteValue -m:dontMove-> s:State;
89     wv -:RWHead-> tp:TapePosition;
90   }
91   replace {
92     tp;
93     wv -m-> s;
94     s -:RWHead-> tp;
95   }
96 }
97
98 rule ensureMoveLeftValidRule {
99   pattern {
100     wv:WriteValue -m:moveLeft-> s:State;
101     wv -rw:RWHead-> tp:TapePosition;
102     negative {
103       tp <-:right- ltp:TapePosition;
104     }
105   }
106   replace {
107     wv -m-> s;
108     wv -rw-> tp <-:right- ltp:TapePosition -:empty-> ltp;
109   }

```

```

110 }
111
112 rule ensureMoveRightValidRule {
113   pattern {
114     wv:WriteValue -m:moveRight-> s:State;
115     wv -rw:RWHead-> tp:TapePosition;
116     negative {
117       tp -:right-> rtp:TapePosition;
118     }
119   }
120   replace {
121     wv -m-> s;
122     wv -rw-> tp -:right-> rtp:TapePosition -:empty-> rtp;
123   }
124 }

```

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 6.1).

6.2.3 Rule Execution with GRSHELL

Finally we construct the busy beaver and let it work with GRSHELL:

```

1 select backend "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($="Startposition")
7
8 # States
9 new sA:State($="A")
10 new sB:State($="B")
11 new sC:State($="C")
12 new sD:State($="D")
13 new sE:State($="E")
14 new sH:State($ = "Halt")
15
16 new sA -:RWHead-> tp
17
18 # Transitions: three lines per state for
19 #   - updating cell value
20 #   - moving read-write-head
21 # respectively
22
23 new sA_0: WriteOne
24 new sA -:empty-> sA_0
25 new sA_0 -:moveLeft-> sB
26
27 new sA_1: WriteOne
28 new sA -:one-> sA_1
29 new sA_1 -:moveLeft-> sD
30
31 new sB_0: WriteOne
32 new sB -:empty-> sB_0
33 new sB_0 -:moveRight-> sC
34
35 new sB_1: WriteEmpty
36 new sB -:one-> sB_1

```

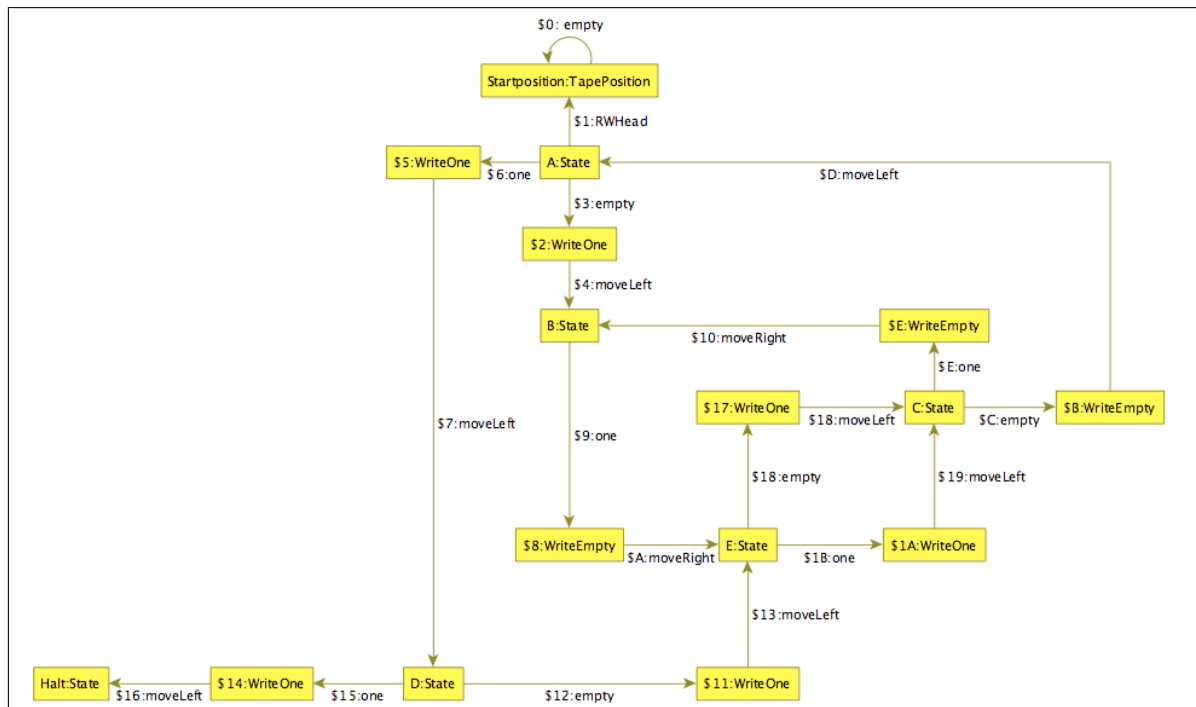


```

37 new sB_1 -:moveRight-> sE
38
39 new sC_0: WriteEmpty
40 new sC -:empty-> sC_0
41 new sC_0 -:moveLeft-> sA
42
43 new sC_1: WriteEmpty
44 new sC -:one-> sC_1
45 new sC_1 -:moveRight-> sB
46
47 new sD_0: WriteOne
48 new sD -:empty-> sD_0
49 new sD_0 -:moveLeft->sE
50
51 new sD_1: WriteOne
52 new sD -:one-> sD_1
53 new sD_1 -:moveLeft-> sH
54
55 new sE_0: WriteOne
56 new sE -:empty-> sE_0
57 new sE_0 -:moveRight-> sC
58
59 new sE_1: WriteOne
60 new sE -:one-> sE_1
61 new sE_1 -:moveLeft-> sC

```

Our busy beaver looks like this:



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.

```

62 grs ((readOneRule | readEmptyRule) ; (writeOneRule | writeEmptyRule) ;
      (ensureMoveLeftValidRule | ensureMoveRightValidRule) ; (moveLeftRule |
      moveRightRule)){32}

```

We interrupt the machine after 32 iterations and look at the result so far:

BIBLIOGRAPHY

- [1] G. Batz: *Graphersetzung für eine Zwischendarstellung im Übersetzerbau* Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2005
- [2] G. Batz: *An Optimization Technique for Subgraph Matching Strategies* Technischer Bericht, IPD Goos, Nr. 2006-7, Fakultät für Informatik, Universität Karlsruhe, 2006
- [3] A. Dewdney: *A computer trap for the Busy Beaver, the hardest-working machine* Scientific American, 251 (2), pages 10-12, 16, 17, August 1984
- [4] R. Geiß et al.: *GRGEN: A Fast SPO-Based Graph Rewriting Tool* in Graph Transformations, number 4178 in LNCS, pages 383-397, Springer, 2006
- [5] D. Grund: *Negative Anwendungsbedingungen für den Graphersetzer GRGEN* Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2004
- [6] *Programming languages – C*, ISO/IEC 9899:1999, 2005
- [7] S. Hack: *Graphersetzung für Optimierung in der Codeerzeugung* Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2003
- [8] M. Kroll: *Portierung des C-Anteils des Graphersetzungssystems GRGEN nach C# mit Erweiterungen*, Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2007
- [9] H. Marxen, J. Buntrock: *Old list of record TMs*.
<http://www.drb.insel.de/~heiner/BB/index.html>. Version: August 2000
- [10] A. Szalkowski: *Negative Anwendungsbedingungen für das suchprogrammbasierte Backend von GrGen* Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2005
- [11] N. Wirth, K. Jensen: *Pascal User Manual and Report* Springer, ⁴1991