Universität Karlsruhe (TH)

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

The GrGen User Manual

Jakob Blomer Rubino Geiß

April 11, 2007

Technical Report 2007-5
ISSN 1432-7864

ABSTRACT

This is the abstract. TODO!

CONTENTS

1 System Overview 1

1.1 Components . . . 2

2 Graph Model Language 4

2.1 Building Blocks . . . 4
2.2 Type Declarations . . . 5

3 Rule Set Language 8

3.1 Building Blocks . . . 8
3.2 Declarations . . . 9
3.3 Pattern Part . . . 10
3.4 Replace Part . . . 11
3.5 Evaluation Part . . . 12

4 Types and Expressions 13

4.1 Built-In Types . . . 13
4.2 Expressions . . . 13
4.3 Type Expressions . . . 16

5 GrShell Language 18

5.1 Building Blocks . . . 18
5.2 GrShell Commands . . . 20
5.2.1 Common Commands . . . 20
5.2.2 Graph Commands . . . 21
5.2.3 Graph Manipulation Commands . . . 22
5.2.4 Graph Query Commands . . . 24
5.2.5 Graph Output Commands . . . 25
5.2.6 Action Commands . . . 28
5.2.6.1 Regular Graph Rewrite Sequences (GRS) . . . 28
5.3 LGSPBackend Custom Commands . . . 29
5.3.1 Graph Related Commands . . . 29
5.3.2 Action Related Commands . . . 29

6 Examples 32

6.1 Busy Beaver . . . 32
6.1.1 Graph Model . . . 32
6.1.2 Rule Set . . . 32
6.1.3 Rule Execution with GrShell . . . 35
6.2 Fractals . . . 38

CHAPTER 1

SYSTEM OVERVIEW

GrGen is a generative programming system for graph rewriting. GrGen is mainly designed for *typed graphs*. That means, graphs are not only nodes and edges, but labeled (*attributed typed*) directed multigraphs. The type system is specified as class hierarchy, like a classes in object oriented languages. Type systems for specific sets of graphs can be specified by user supplied graph meta models. Such a graph model describes a set of well-formed graphs, i.e. allowed node and edge types, their attributes and specific connection assertions. We'll build a graph model for a turing machine in section 6.

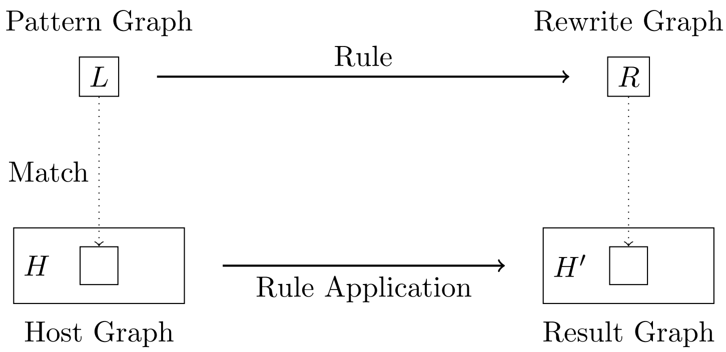
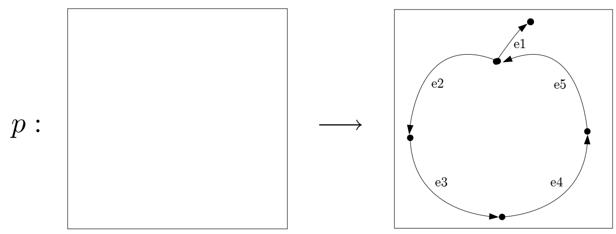


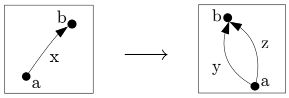
Figure 1.1: Basic Idea of Graph Rewriting

How does graph rewriting work? GrGen implements an SPO-based approach. Given a host graph H, each *rewriting rule* $p : L \longrightarrow R$ consists of a pattern L and a transformation specification R in form of a adopted pattern graph. The process of rewriting searches a match $H_L \trianglelefteq H$ (i.e. a graph homomorphism from L to a subgraph of H) and rewrites H_L to R. Nodes or edges added to R (in compare to L) will be added H_L and nodes or edges deleted in R will be deleted in H_L. The homomorphism may not be unique.

We'll have a look at a small example. First we use a special case to construct our host graph: an empty pattern does always produce exactly one match (independently of the host graph). So starting with an empty host graph H we construct an apple using

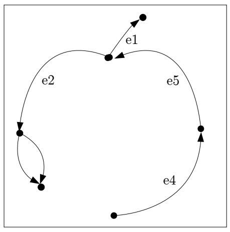


applied to H. We'll get the apple as new host graph H'. Now we want to rewrite our apple with stem to an apple with a leaflet. We use

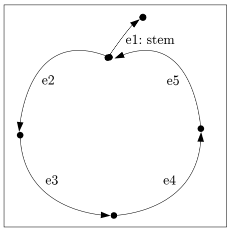
p' : 

,

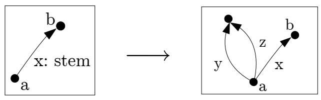
apply p' to H' and get the new host graph H'', something like this:



What happened? GrGen has randomly choosen a match, and $e3$ matches as well as $e1$. A correct solution could make use of edge type information. And this time we'll keep even the stem. So let H'' now be

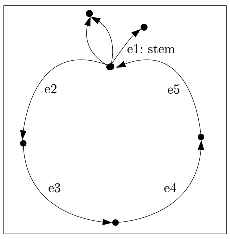


and

p'' : 

.

If we apply p'' to H'' this leads to



Note: If we had applied $(p')*$ to H' (execute p' consecutively until no match is found) this would not have terminated, because each rewrite had produced one new canditate (one deleted, two added) for matching.

1.1 Components

Figure 1.2 gives an overview of the GrGen system components, whereas table 1.1 shows the GrGen directory structure.

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

Table 1.1: GrGen directory structure

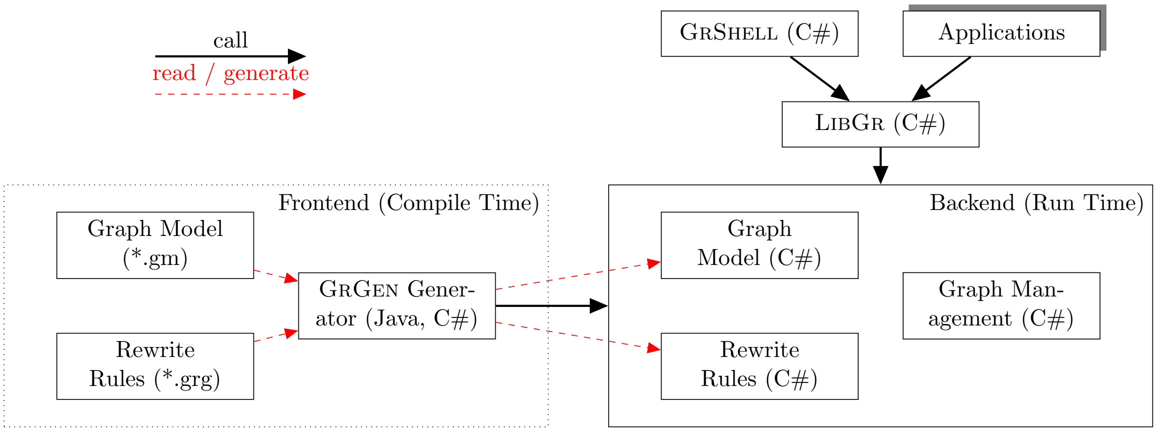


Figure 1.2: GrGen system components [2]

A graph rewriting system is defined by a rule set description file (*.grg) and one or more graph model description files (*.gm).[1] It is generated by GrGen.exe and can be used by GrGen applications such as GrShell. Figure 1.3 shows the generation process.

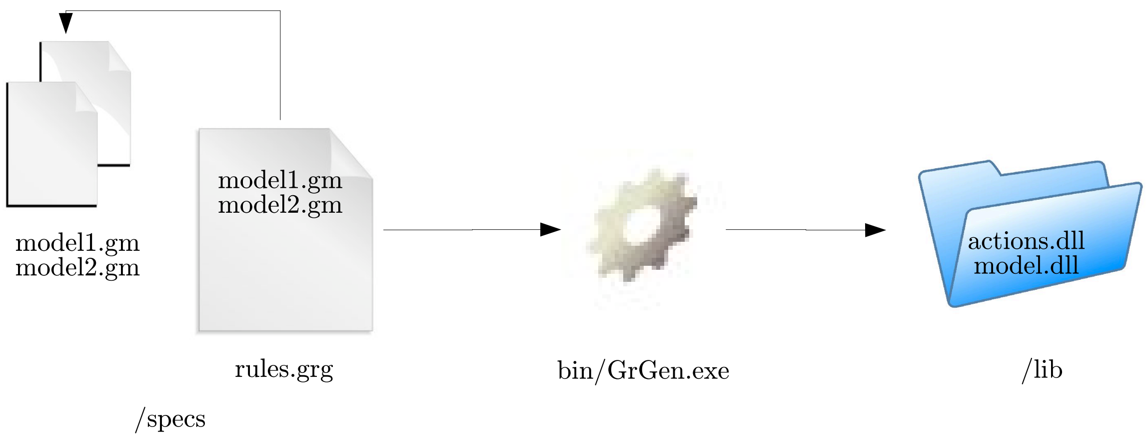


Figure 1.3: Generating a graph rewriting system

In general you have to distinguish carefully between a graph model (meta level), a host graph, a pattern graph and a rewriting 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 rewriting 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]System, in this context, is not a CHO-like grammar rewriting system, but rather a set of interacting software components.

CHAPTER 2

GRAPH MODEL LANGUAGE

The key features of GrGen graph models from [1]:

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 itself are typed, too.

Inheritance.
: 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.

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.

2.1 Building Blocks

Note: The following syntax specifications make heavy use of syntax diagrams (also known as rail diagrams). Syntax diagrams provide an visualization of EBNF 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 [**?**]

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

Ident
A character sequence of arbitrary length consisting of letters, digits or underscores. The first character must not be a digit.

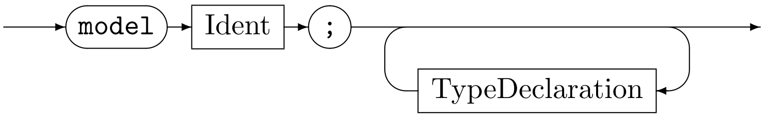
NodeType, *EdgeType*, *EnumType*
These are (semantic) specializations of Ident to restrict an identifier to a specific type.

Number

A sequence of digits. The sequence has to form a non-negative integer in decade system and will be internally stored in 32 bit tow's complement representation.

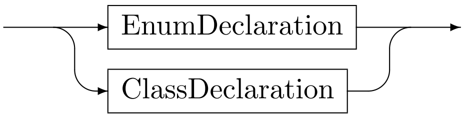
2.2 Type Declarations

GraphModel



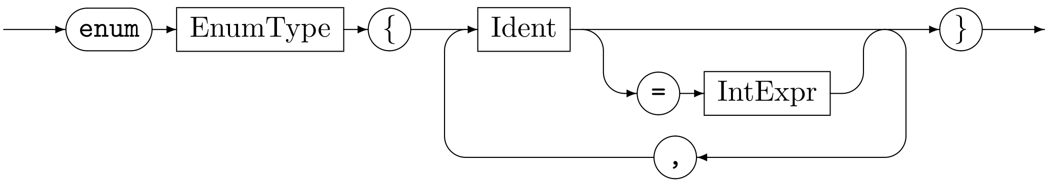
The graph model consists of its name *Ident* and type declarations defining specific nodes and edges.

TypeDeclaration



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

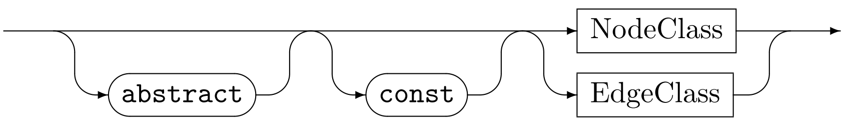
EnumDeclaration



Defines an enum type.
Example:

```
enum Color {red, green, blue}
enum Resident {village = 500, town = 5000, city = 50000}
```

ClassDeclaration

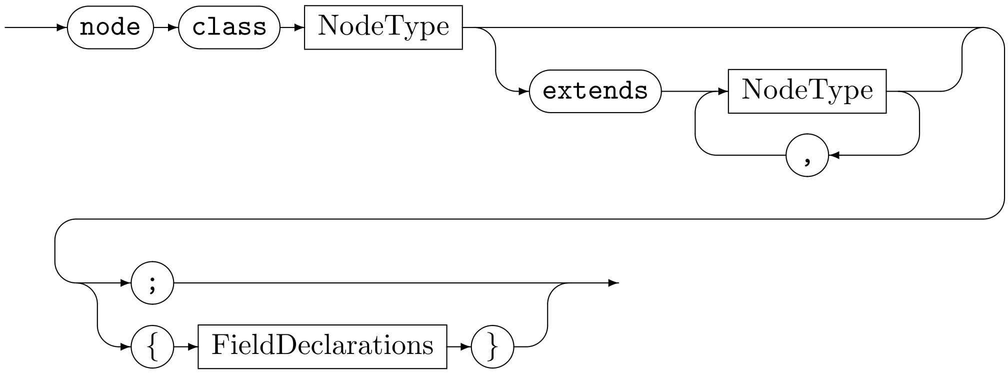


Defines a new node type or edge type. The keyword `abstract` indicates, that you can't create graph elements of this type but rather you have to derive non-abstract types for graph elements.
Example:

```
abstract node class city {
   size: int;
}
abstract node class abandoned_city extends city;
node class ghost_town extends abandoned_city;
```

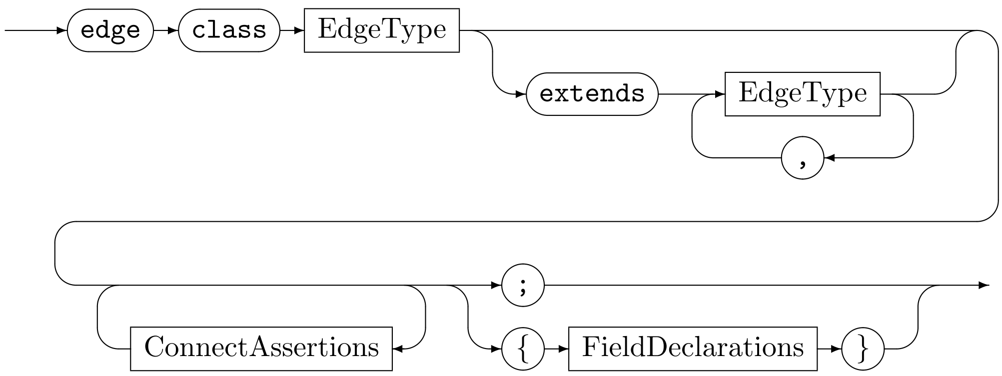
You will be able to create nodes of type `ghost_town`, but not of type `city` or `abandoned_city`. However, nodes of type `ghost_town` are also nodes of type `abandoned_city` and of type `city` and they have got the attribute `size`.

NodeClass



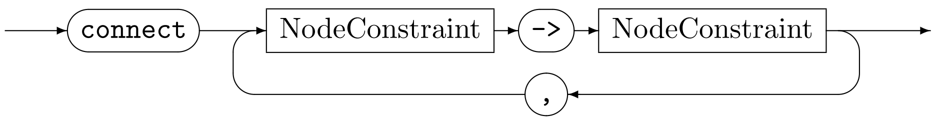
Defines a new node type. Node types can inherit from other node types defined within the same file. If the `extends` clause is omitted, *NodeType* will inherit from the built-in type `Node`. Optionally nodes can possess attributes (fields).

EdgeClass

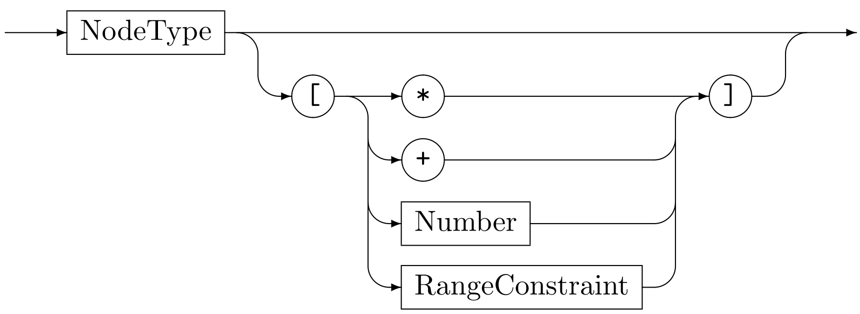


Defines a new edge type. Edge types can inherit from other edge types defined within the same file. If the `extends` clause is omitted, *EdgeType* will inherit from the built-in type `Edge`. Optionally edges can possess attributes (fields). A *connection assertion* specifies that certain edge types should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

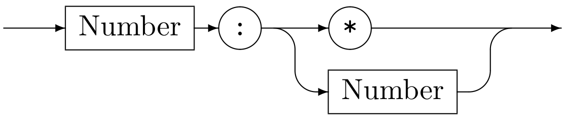
ConnectAssertions



NodeConstraint



RangeConstraint

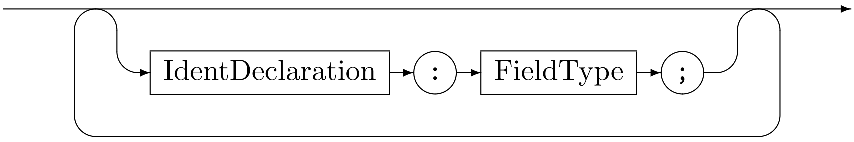


A connection assertion is described as a pair of node types, optionally together with a multiplicity. A corresponding edge may connect a node of the first node type or one of its subtypes (source) with a node of the second node type or one of its subtypes (destination). The multiplicity is a constraint on the out-degree and in-degree of the source and destination node type respectively. See 5.2.2, `validate`, for an example. Table 2.1 describes the multiplicity definitions:

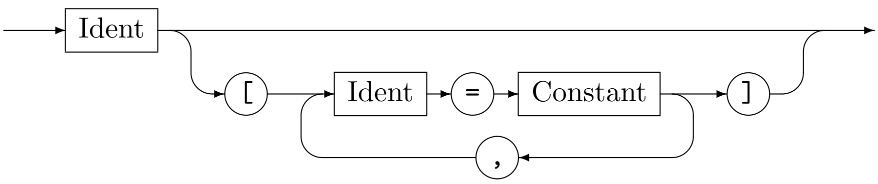
`[*]`	Number of edges the node is adjacent to is unbounded. The node may not be connected to any edge. This is the **default**.
`[+]`	Number of edges the node is adjacent to is unbounded. At least one edge must be adjacent to the node.
`[n:*]`	Number of edges the node is adjacent to is unbounded. At least n edges must be adjacent to the node.
`[n:m]`	At least n edges must be adjacent to the node, but at most m edges may be adjacent to the node ($m \geq n$ holds).
`[n]`	Abbreviation for `[n:n]`.

Table 2.1: GrGen Node constraint multiplicities concerning a specific pair of an edge type and a node type.

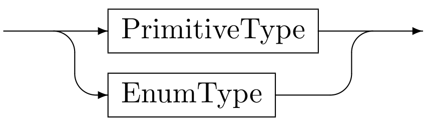
FieldDeclarations



IdentDeclaration



FieldType



Defines a node or edge attribute. Attributes can be initialized with a value. See 4.2 for a definition of *Constant*. Possible types are `enum` and primitive types. See 4.1 for a list of built-in primitive types.

CHAPTER 3

RULE SET LANGUAGE

3.1 Building Blocks

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

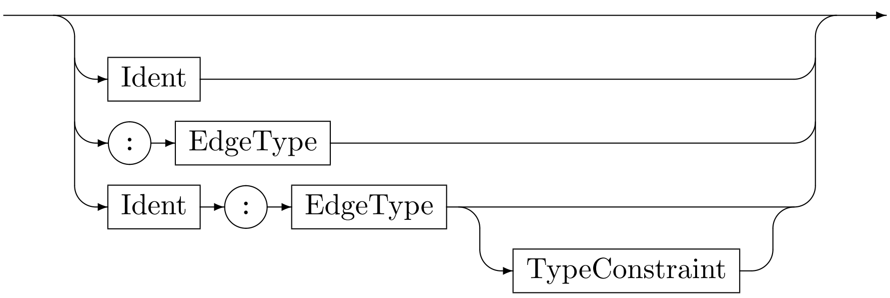
Ident
A character sequence of arbitrary length consisting of letters, digits or underscores. The first character must not be a digit. *Ident* may be an identifier defines in a graph model (see 2.1).

RuleSetIdent, *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.

ForwardEdge

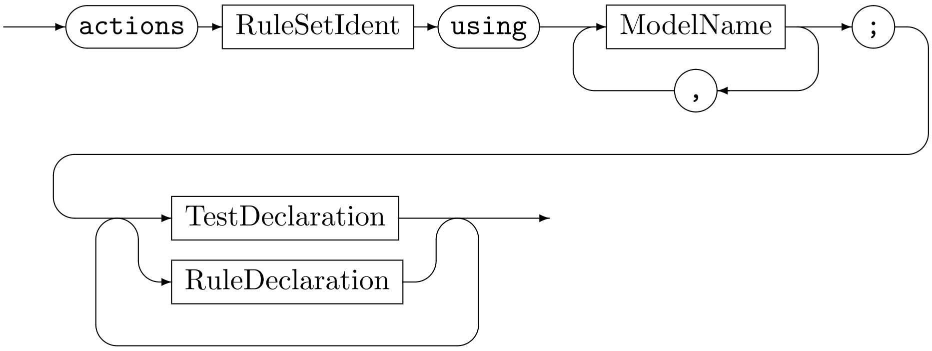
ReverseEdge

EdgeRefinement



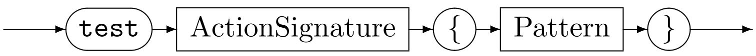
In general edges are specified by `-->` or `<--`. Those edges are called *anonymous*. For a more detailed specification use an edge refinement clause between the arrow dashes. See 4.3 for *TypeConstraint*.

3.2 Declarations

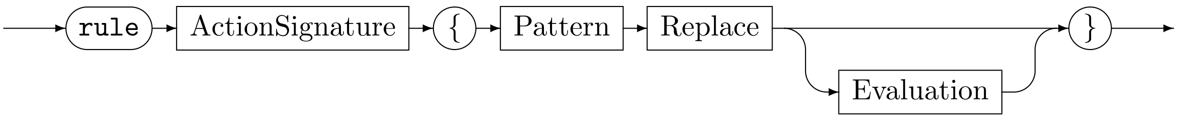


A rule set consists of the underlying graph models and several rewriting rules. In case of multiple graph models GrGen use the unification of the models. In this case beware of conflicting declarations.

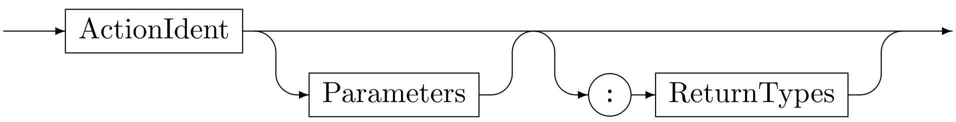
TestDeclaration



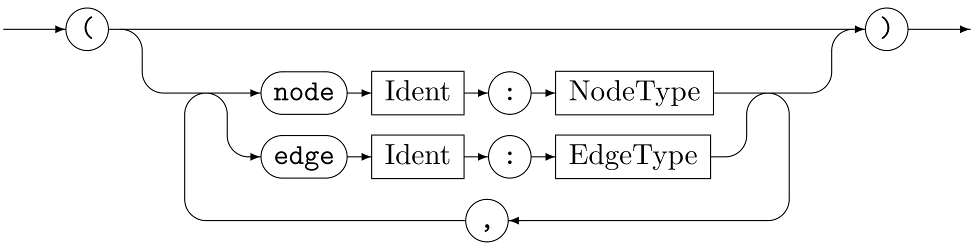
RuleDeclaration



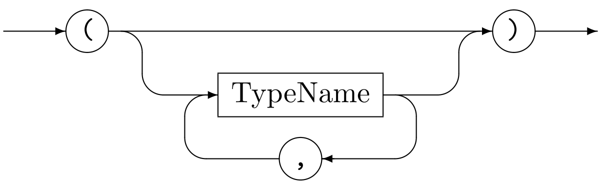
ActionSignature



Parameters

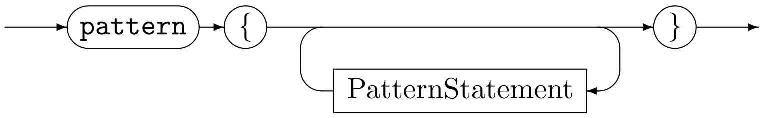


ReturnTypes

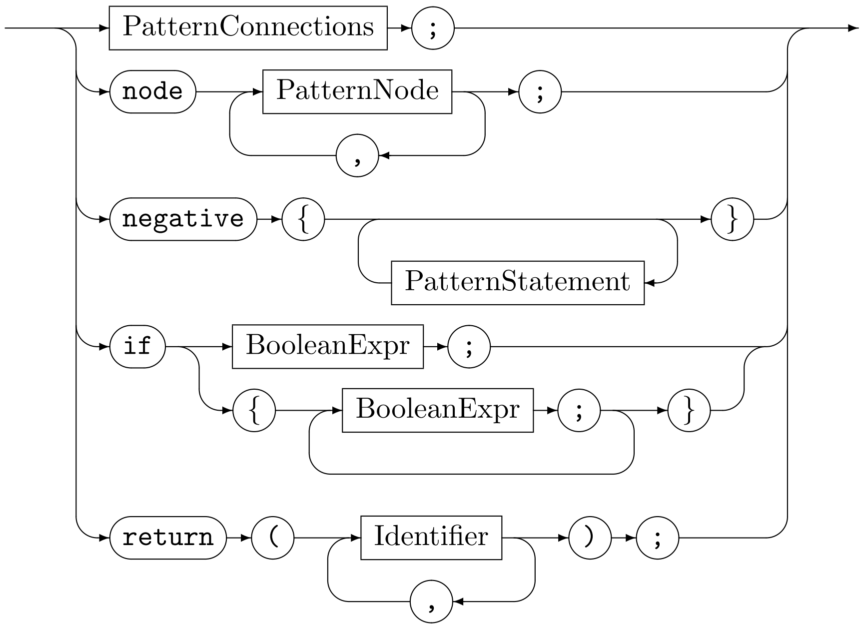


3.3 Pattern Part

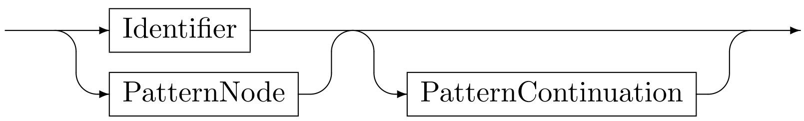
Pattern



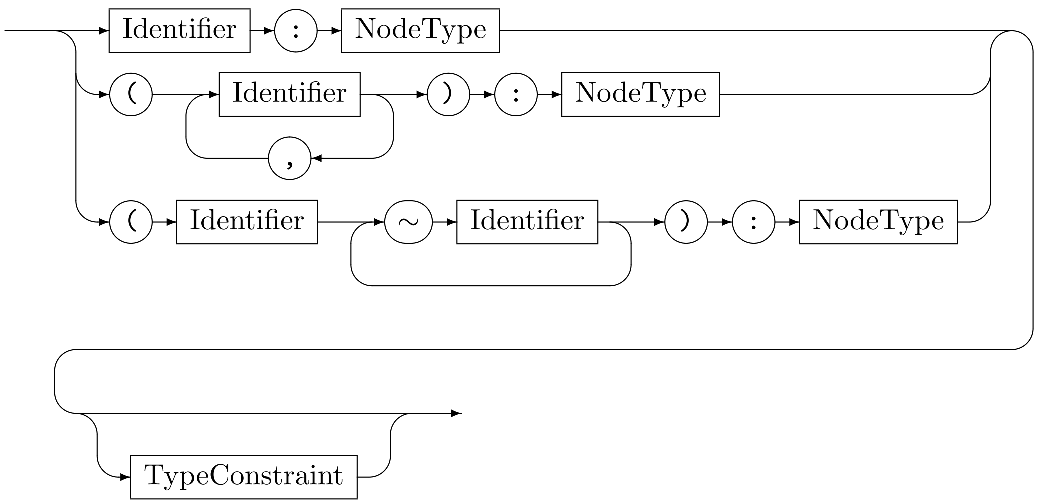
PatternStatement



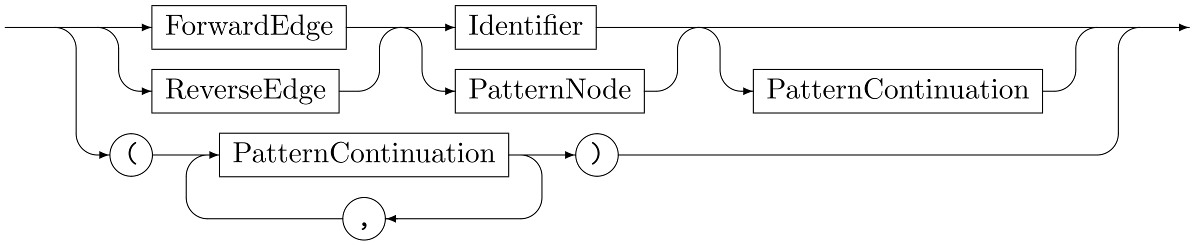
PatternConnections



PatternNode



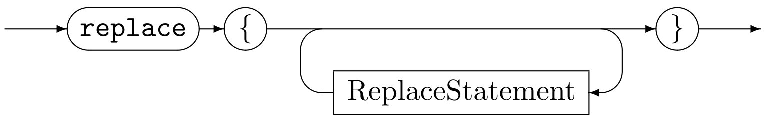
PatternContinuation



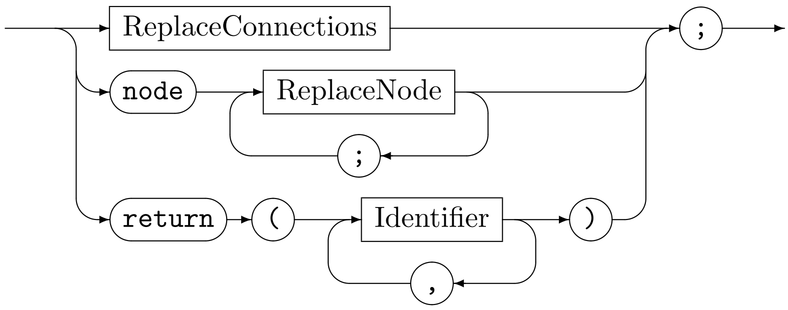
3.4 Replace Part

Note: Although GrGen does – in general – graph re*writing* (also called graph transformation) this part has a re*place* semantics.

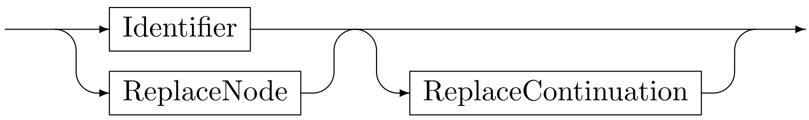
Replace



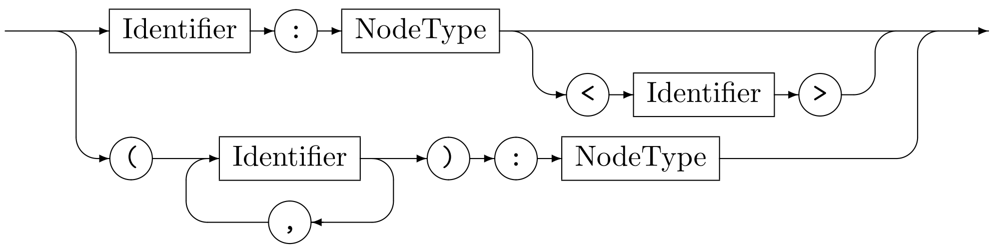
ReplaceStatement



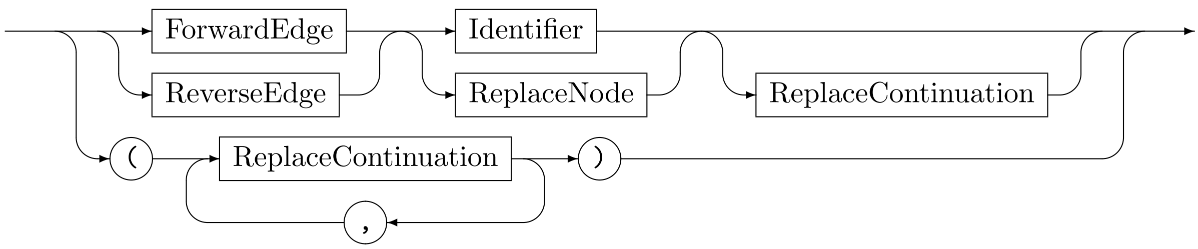
ReplaceConnections



ReplaceNode

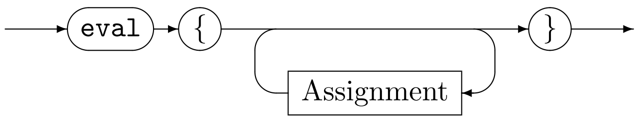


ReplaceContinuation

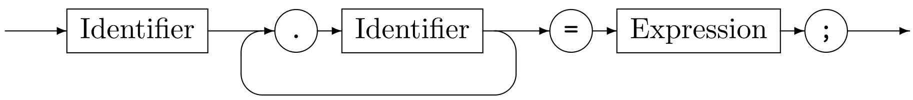


3.5 Evaluation Part

Evaluation



Assignment



CHAPTER 4

TYPES AND EXPRESSIONS

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

4.1 Built-In Types

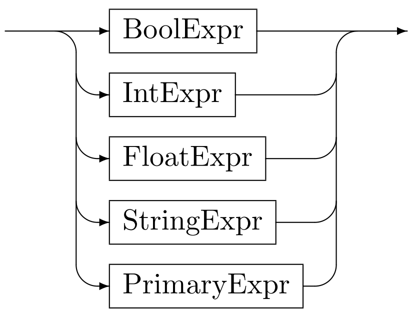
Besides user-defined node types, edge types and enum types, GrGen supports the built-in primitive types in table 4.1:

`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 may be enclosed by double quotes. If the string contains white spaces, double quotes are mandatory.

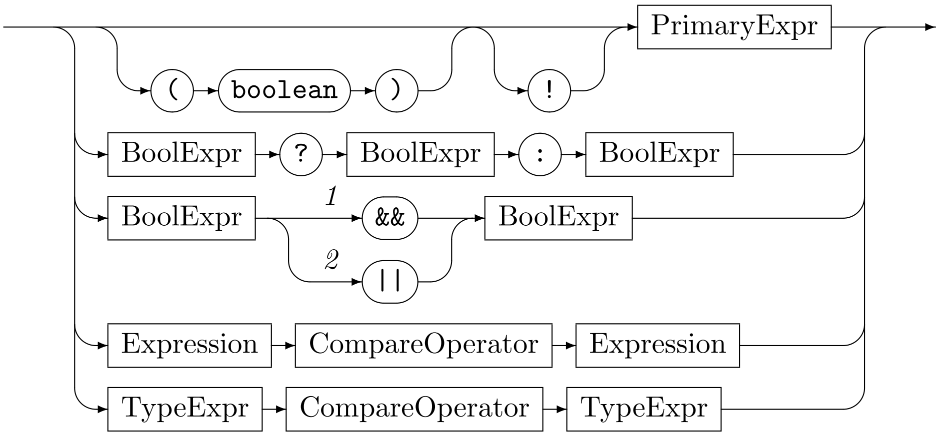
Table 4.1: GrGen built-in types

4.2 Expressions

Expression



BoolExpr



As in C, `!` negates a boolean, `&&` is a logical AND and `||` is a logical OR. The order of precedence is `!`, `&&`, `||`. 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*.
CompareOperator is one of the following operators:

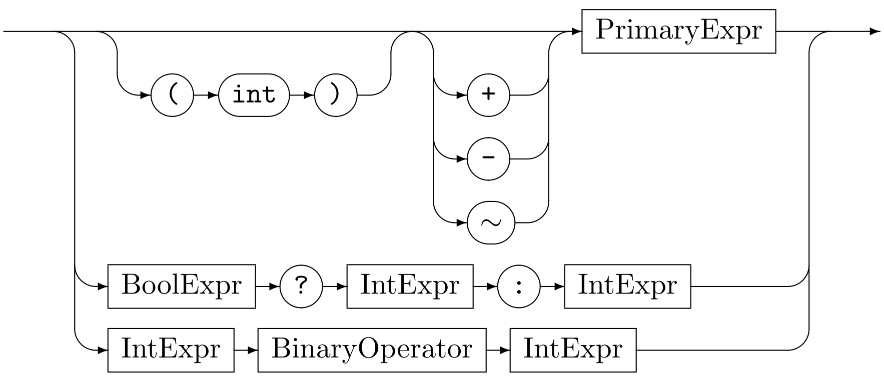
`<`   `<=`   `==`   `!=`   `>=`   `>`

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

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

Table 4.2: Compare operators on type expressions

IntExpr

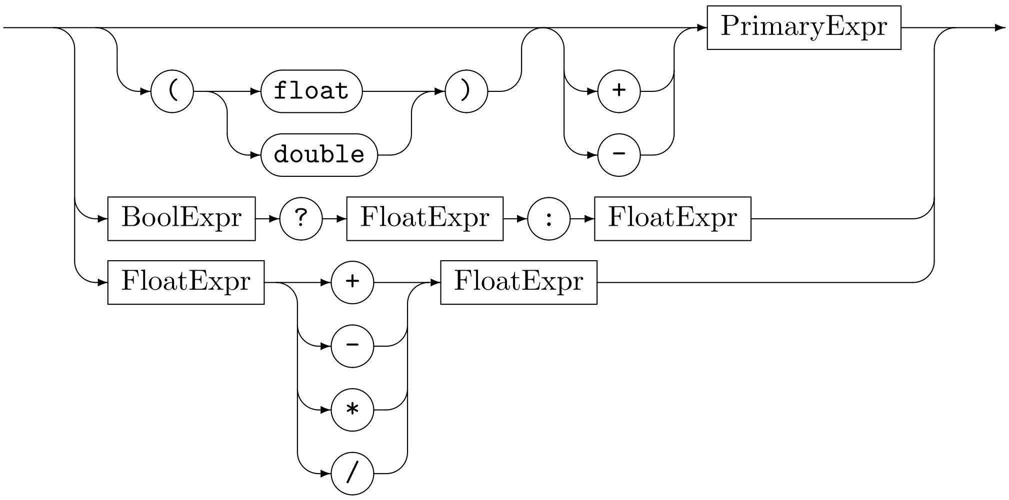


Boolean and enum types (and technically integer types) can be converted to `int` by using the `(int)` construct. The ~ operator is a binary NOT. That means every bit of an integer value will be flipped. The `?` operator is a simple if-then-else: if the *BoolExpr* is evaluated to `true`, the operator returns the first *IntExpr*, otherwise it returns the second *IntExpr*.
BinaryOperator is one of the operators in table 4.3:

<table>
<tr><td><code>|</code>
<code>^</code>
<code>&</code></td><td>Binary OR, XOR and binary AND</td></tr>
<tr><td><code><<</code>
<code>>></code>
<code>>>></code></td><td>Bitwise shift left, bitwise shift right and
bitwise shift right with respect to sign</td></tr>
<tr><td><code>+</code>
<code>-</code></td><td>Addition and subtraction</td></tr>
<tr><td><code>*</code>
<code>/</code>
<code>%</code></td><td>Multiplication, integer division
and modulo</td></tr>
</table>

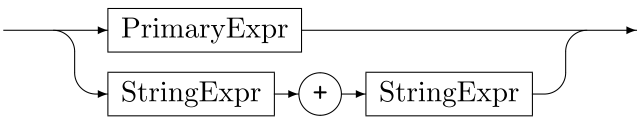
Table 4.3: Binary integer operators, in ascending order of precedence

FloatExpr



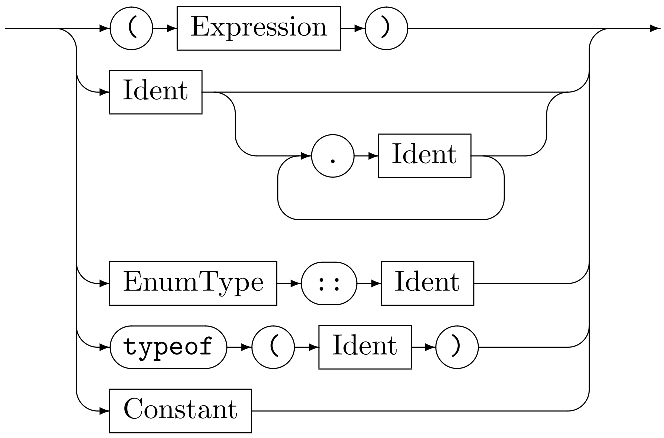
Type conversion by `(float)` and `(double)` is possible between `float` and `double` only. The `?` operator is a simple if-then-else: if the *BoolExpr* is evaluated to `true`, the operator returns the first *FloatExpr*, otherwise it returns the second *FloatExpr*.

StringExpr

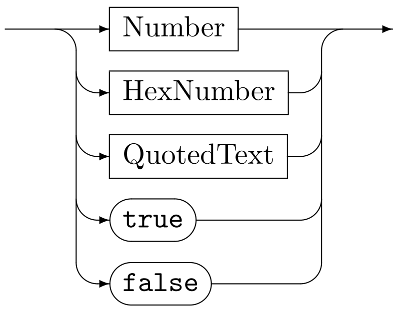


The operator `+` concatenates two strings.

PrimaryExpr

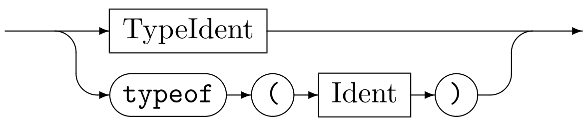


Constant



4.3 Type Expressions

TypeExpr



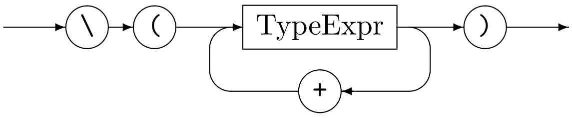
A type expression identifies a type (and – in terms of matching – also its subtypes). A type expression is either a type identifier itself or the type of a variable.

Example:

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

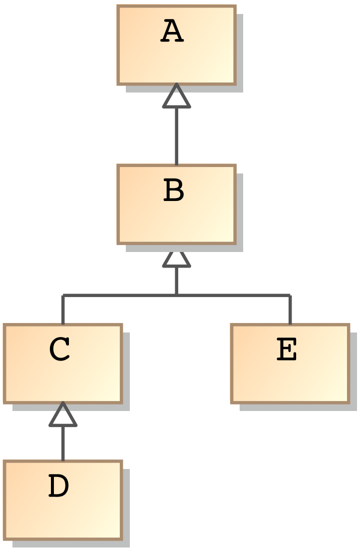
```
rule oneway {
    pattern{
        a:Node -x:street-> y:Node;
        negative{
            y -:typeof(x)-> a;
        }
    }
    replace{
      a -x-> y;
      y -:typeof(x)-> a;
    }
}
```

TypeConstraint



A type constraint is used to exclude parts of the type hierarchy. The operator `+` is used to identify a unification of types.

Example:



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

CHAPTER 5

GRSHELL LANGUAGE

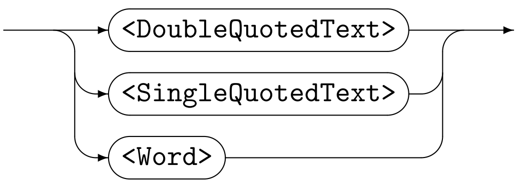
The GrShell is a shell application of the LibGr. It belongs to GrGen's standard equipment. GrShell is capable of creating, manipulating and dumping graphs as well as performing graph rewriting and debugging graph rewriting.

The GrShell language is a line oriented scripting language. It is structured by simple statements separated by line breaks.

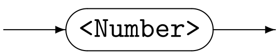
5.1 Building Blocks

The GrShell is case sensitive. A comment starts with a # and is terminated by end-of-line or end-of-file. Anything left from the # will be treated as a statement.

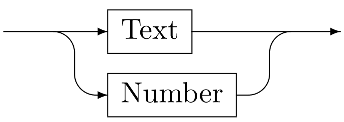
Text



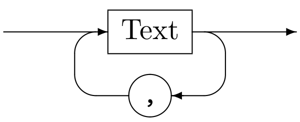
Number



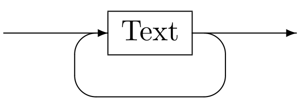
TextOrNumber



Parameters



SpacedParameters



Those items are required for representing text, numbers and parameters within rules. The tokens `<...>` are defined as follows:

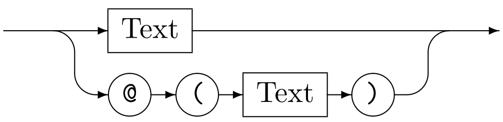
`<$Word$>`:	A character sequence consisting of letters, digits and underscores. The first character must not be a digit.
`<DoubleQuotedText>`:	Arbitrary text enclosed by double quotes (“ ”).
`<SingleQuotedText>`:	Arbitrary text enclosed by single quotes (‘ ’).
`<Number>`:	A sequence of digits.

In order to describe the possible input to some of 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.
NodeType:	Identifier of a node type within 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, Yelloq, White, DarkBlue, DarkRed, DarkGreen, DarkYellow, DarkMagenta, DarkCyan, Gold, Lilac, Turquoise, Aquamarine, Khaki, Pink, Orange, Orchid.

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 5.2.3):

GraphElement

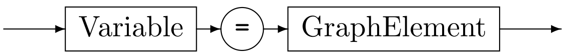


The specializations *Node* and *Edge* of *GraphElement* requires the corresponding graph element to be a node or an edge respectively.

Example: We insert a node, anonymously and with a constructor:

```
> select backend lgspBackend.dll
Backend selected successfully.
> new graph "../lib/lgsp-TuringModel.dll" G
New graph "G" of model "Turing" created.

# insert an anonymous node...
# it will get a persistent pseudo name
> new :State
New node "$0" of type "State" has been created.
> delete node @("$0")

# and now with constructor
> new v:State($=start)
new node "start" of type "State" has been created.
# Variable v is now assigned to start
> new :State($=end)
new node "end" of type "State" has been created.

# actually we want v to be "end":
>  v = @(end)
>
```

Note: 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.

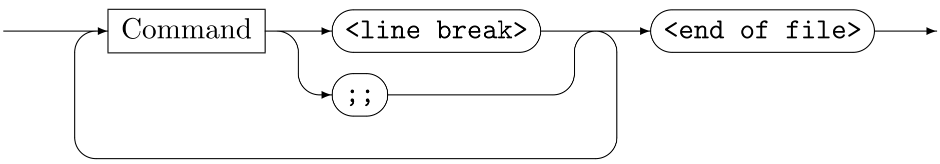


Assigns the variable or persistent name *GraphElement* to *Variable*. If *Variable* is not defined, it will be defined implicitly.

5.2 GrShell Commands

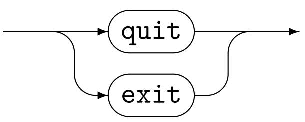
This section describes the GrShell commands. Commands are assembled from basic elements. As stated before commands are terminated by a line breaks. Alternatively commands can be terminated by the `;;` symbol.

Script

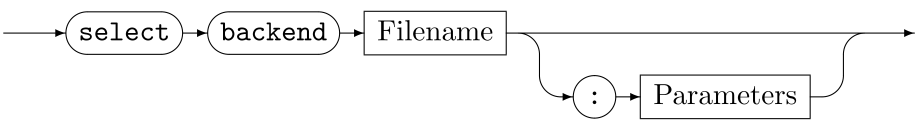


5.2.1 Common Commands

Displays an information message describing supported commands.

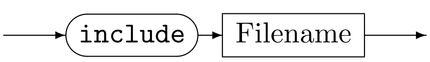


Quits GrShell. If GrShell is opened in debug mode, currently active graph displayers (such as YComp) will be closed as well.

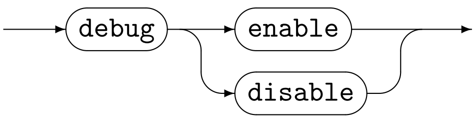


Selects a backend that handles graph and rules representation. *Filename* has to be a .NET assembly (e.g. "lgspBackend.dll"). Comma-separeted parameters can be supplied optionally. If so, the backend must support these parameters.

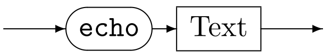
List all the parameters supported by the backend *Filename*, that can be provided to the `select backend` command.



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 an parser error occurs, execution of the script will cancel immediately.

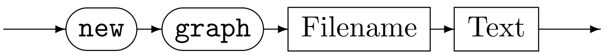


Enables and disables the debug mode. The debug mode shows the current working graph in a YComp window. All changes to the working graph are tracked by YComp immediately.

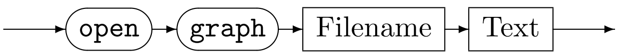


Prints *Text* onto the GrShell command prompt.

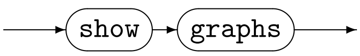
5.2.2 Graph Commands



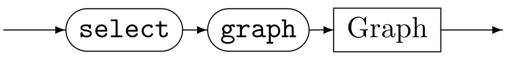
Creates a new graph with the model specified in *Filename*. Its name is set to *Text*. The model file can be either source code (e.g. “turing_machine.cs”) or a .NET assembly (e.g. “turing_machine.dll”).



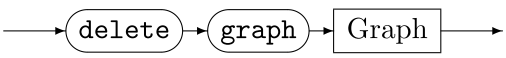
Opens the graph *Text* stored in the backend. Its model is specified in *Filename*. However, the *LGSPBackend* doesn’t support persistent graphs. *LGSPBackend* is the only backend so far. Therefore this command is useless at the moment.



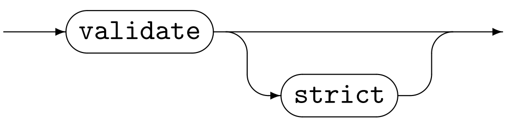
Displays a list of currently available graphs.



Selects the current working graph.



Deletes the graph *Graph* from the backend storage.

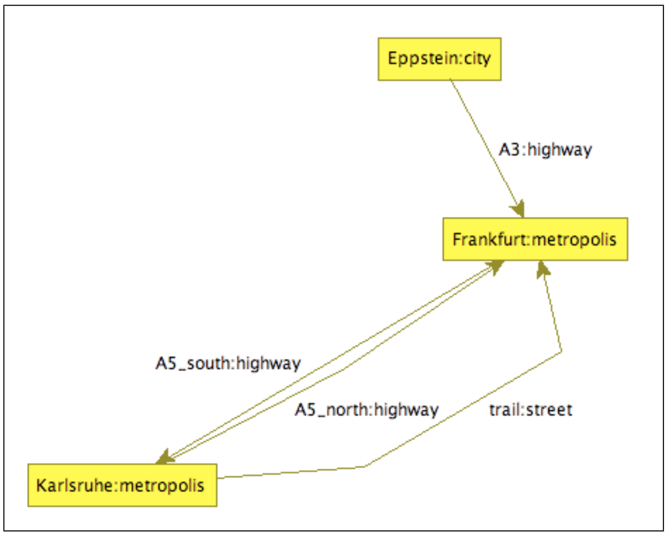


Validates if the current working graph fulfills the edge constraints specified in the corresponding graph model. The *strict* mode additionally requires all the edges of the working graph to be specified in order to get a “valid”. Otherwise edges between nodes without specified constraints are ignored.

Example: We define a micro model of street map graphs:

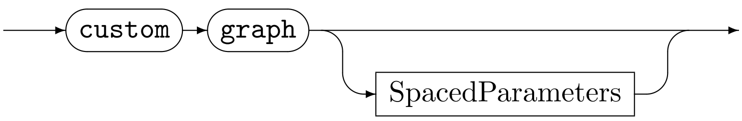
```
model Map;

node class city;
node class metropolis;

edge class street;
edge class highway
      connect metropolis [+] -> metropolis [+];
```

The node constraint on *highway* requires all the metropolises to be connected by highways. Now have a look at the following graph:



This graph is valid, but not strict valid.

```
> validate
The graph is valid.
> validate strict
The graph is NOT valid:
  CAE: city "Eppstein" -- highway "A3" --> metropolis
            "Frankfurt" not specified
  CAE: metropolis "Karlsruhe" -- street "trail" -->
            metropolis "Frankfurt" not specified
>
```

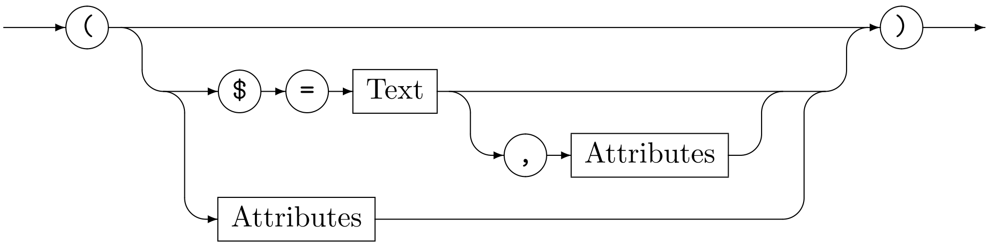


Executes a command specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSP backend see 5.3).

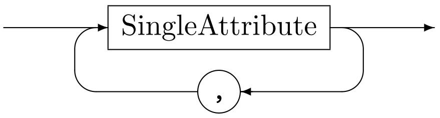
5.2.3 Graph Manipulation Commands

In order to create graph elements an optional constructor can be used.

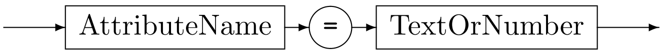
Constructor



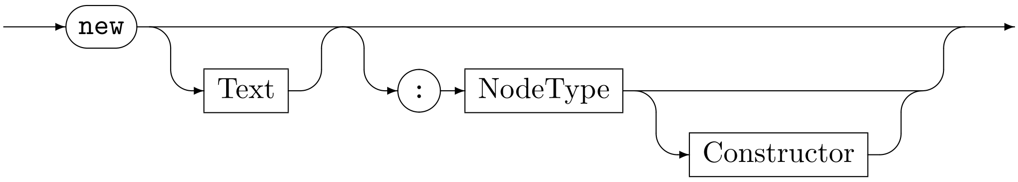
Attributes



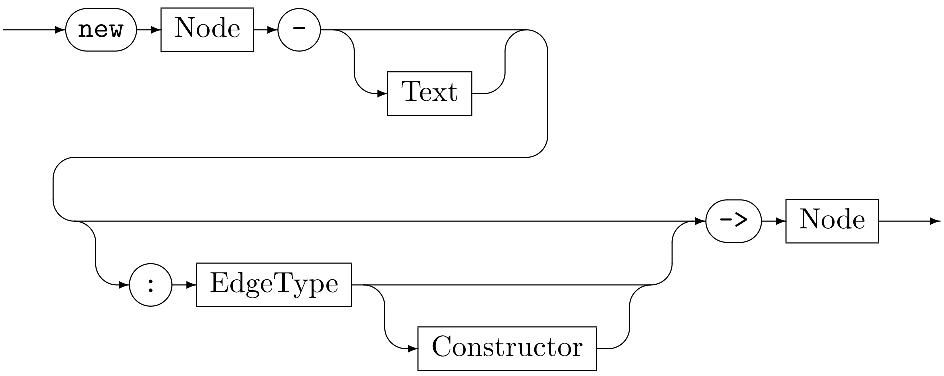
SingleAttribute



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 ← 0; boolean ← false; String ← “”).
The `$` marks a special attribute: an unique identifier of the new graph element. This identifier also is denoted as *persistent name* (see 5.1, *GraphElement*). This name can be specified by a constructor only.

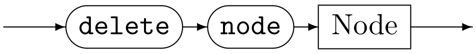


Creates a new node within the current graph. Optionally a variable *Text* is assigned to the new node. If *NodeType* is supplied, the new node will be of type *NodeType* and attributes can be initialized by a constructor. Otherwise the node will be of the base node class type *Node*.

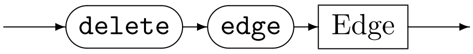


Creates a new edge within the current graph between the specified nodes, directed towards the second *Node*. Optionally a variable *Text* is assigned to the new edge. If *EdgeType* is supplied, the new edge will be of type *EdgeType* and attributes can be initialized by a constructor. Otherwise the edge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

Set the attribute *AttributeName* of the graph element *GraphElement* to the value of *TextOrNumber*.

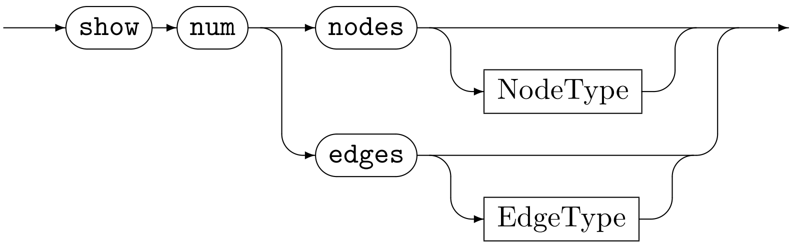


Deletes the node *Node* from the current graph. Incident edges will be deletes as well.

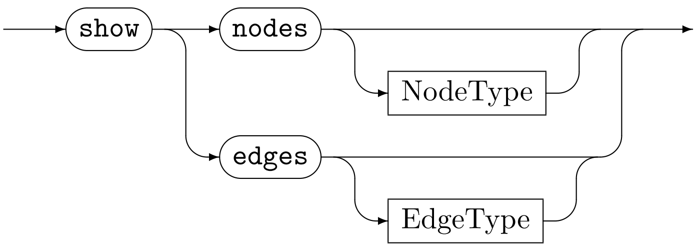


Deletes the edge *Edge* from the current graph.

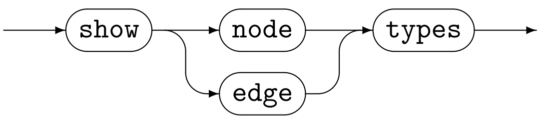
5.2.4 Graph Query Commands



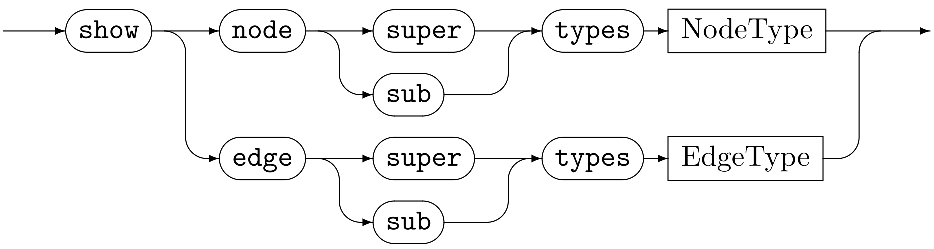
Gets the number of nodes/edges of the current graph. If a node type resp. edge type is supplied, only elements compatible to this type are considered.



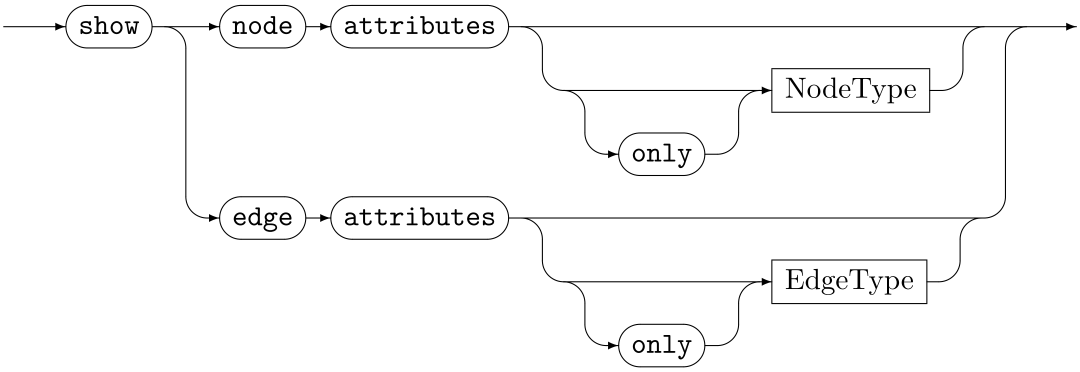
Gets the persistent names and the types of all the nodes / edges of the current graph. If a node type or edge type is supplied, only elements compatible to this type are considered. Nodes / edges without persistent names are shown with a pseudo-name.



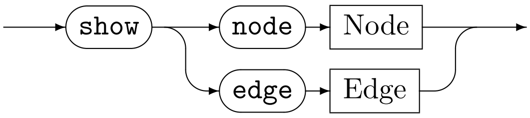
Gets the node / edge types of the current graph model.



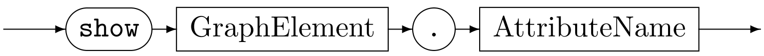
Gets the inherited / descended types of *NodeType* / *EdgeType*.



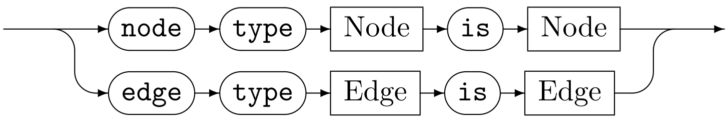
Gets the available node / edge attribute types. If *NodeType* / *EdgeType* is supplied, only attributes defined in *NodeType* / *EdgeType*. The `only` keyword excludes inherited attributes. **Note:** This is in contrast to the `show num...`, `show nodes...` and `show edges...` commands where types and *sub*types are specified.



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

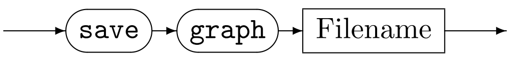


Gets the value of a specific attribute.



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

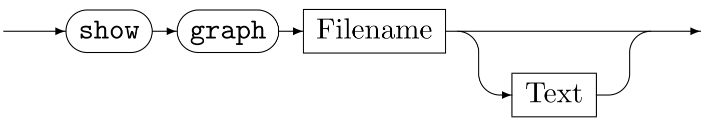
5.2.5 Graph Output Commands



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),

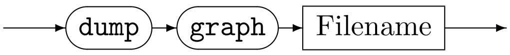
but not necessarily using the same commands you typed in during construction.



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 YComp as executable, this may look like

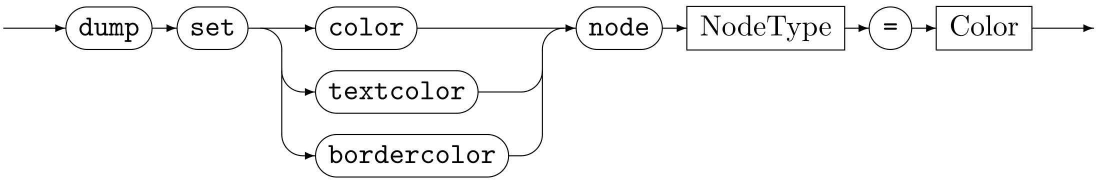
BILD (Dragon?)

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

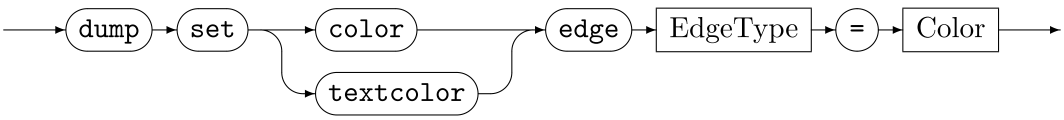


Dumps the current graph as VCG formatted file into *Filename*.

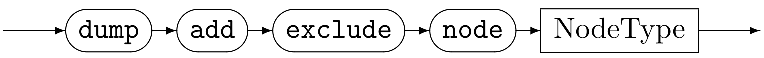
The following commands control the style of the VCG output. This affects `dump graph`, `show graph` and `enable debug`.



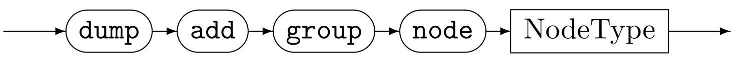
Sets the color / text color / border color of the nodes of type *NodeType*. This doesn't include sub types of *NodeType*.



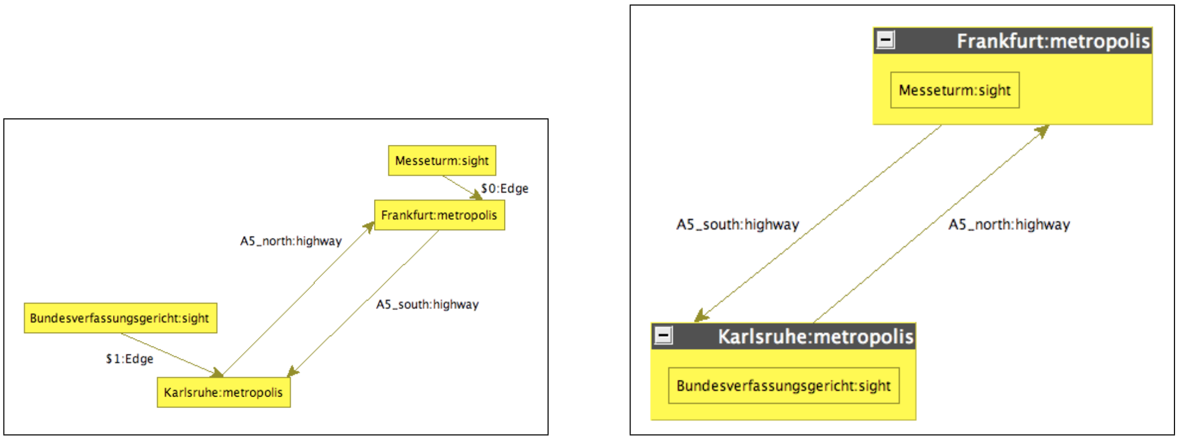
Sets the color / text color of the edges of type *EdgeType*. This doesn't include sub types of *NodeType*.



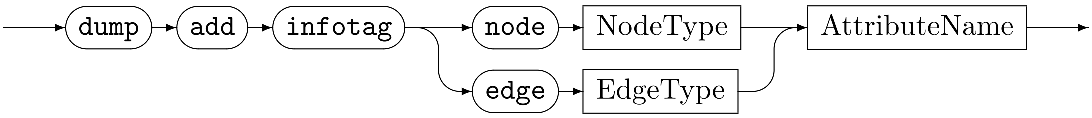
Excludes nodes of type *NodeType* (or sub type of *NodeType*) as well as their incident edges from output.



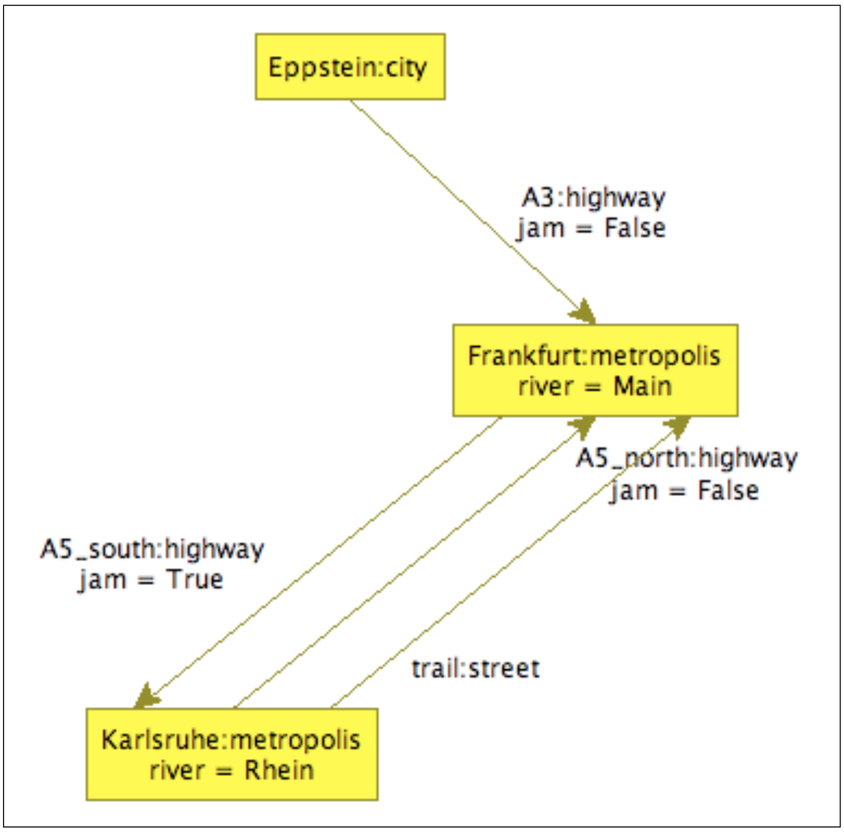
Declares *NodeType* (or sub type of *NodeType*) as group node type. All the different typed nodes that points to a node of type *NodeType* (i.e. there is a directed edge between such nodes) will be grouped and visibly enclosed by the *NodeType*-node. The following example shows *metropolis* ungrouped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

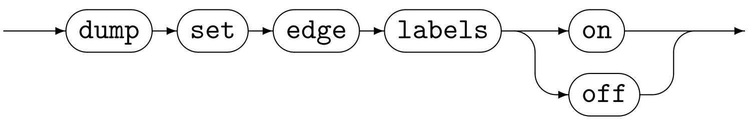


right side: dumped with `dump add group node metropolis`

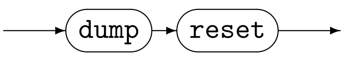


Declares the attribute *AttributeName* to be an “info tag”. Info tags are displayed like additional node / edge labels. In the following example *river* and *jam* are info tags:





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

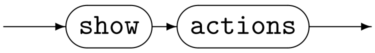


Reset all style options (`dump set`...) to their default values.

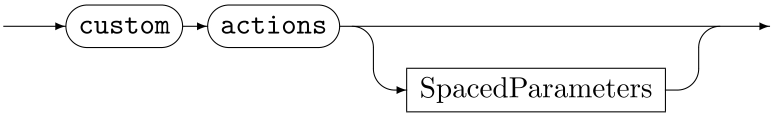
5.2.6 Action Commands

An *action* denotes a graph rewriting rule.

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



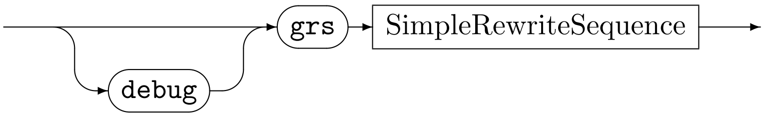
Lists all the rules of the loaded rule set, their parameters and their return values. Rules can return a set of graph elements.



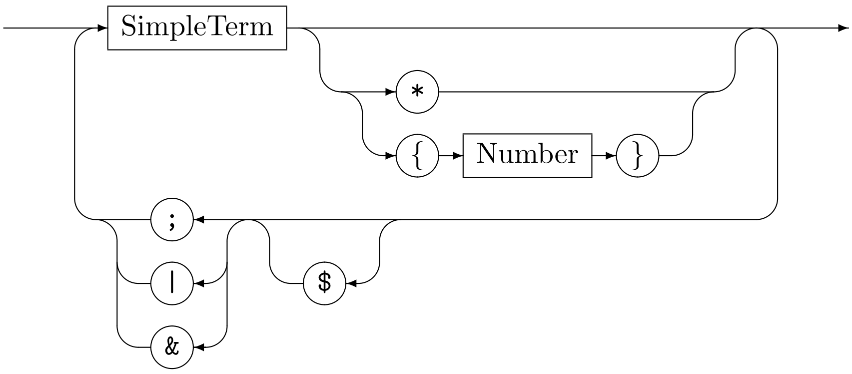
Executes an action specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSPBackend see section 5.3).

5.2.6.1 Regular Graph Rewrite Sequences (GRS)

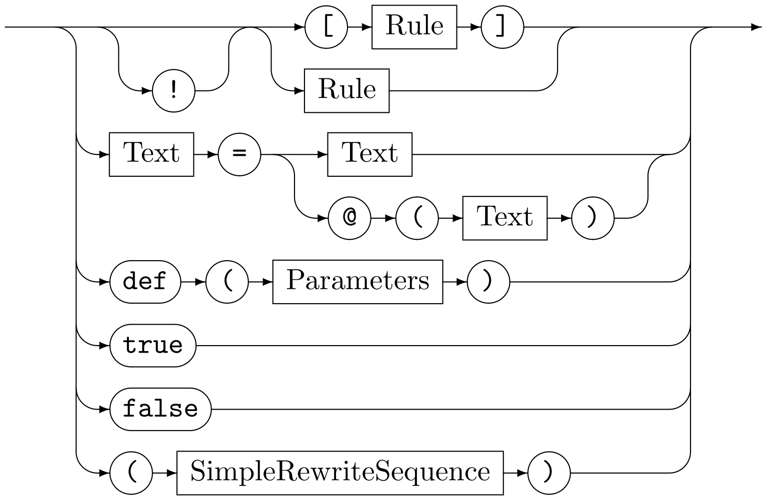
Basically a graph rewriting command looks like this:



SimpleRewriteSequence



SimpleTerm



Rule

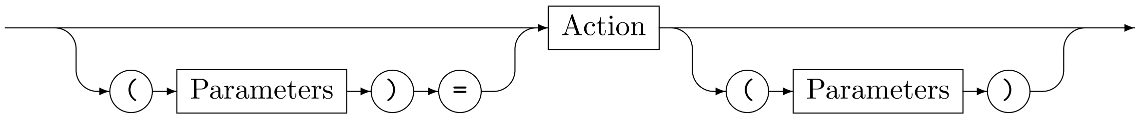


Table 5.1 lists graph rewriting expressions at a glance. The operators hold the following order of precedence, starting with the lowest precedence:

`;` `|` `&`

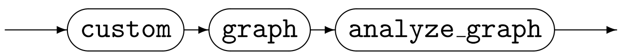
Variables can be assigned to graph elements returned by rules using `(`*`Parameters`*`) =` *`Action`*. The desired variable identifiers have to be listed in *Parameters*. Graph elements required by rules must be provided using *`Action`* `(`*`Parameters`*`)`, where *Parameters* is a list of variable 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 step. The debugger can be controlled by GrShell. The debug commands are shown in table 5.2.

5.3 LGSPBackend Custom Commands

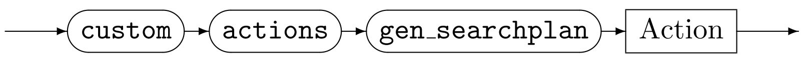
The LGSPBackend supports the following custom commands:

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



<table>
<tr><td>s ; t</td><td>Concatenation. First, s is executed, afterwards t is executed. The sequence s ; t is successfully executed iff s or t is successfully executed.</td></tr>
<tr><td>s | t</td><td>XOR. First, s is executed. Only if s fails, t is executed. The sequence s | t is successfully executed iff s or t is successfully executed.</td></tr>
<tr><td>s & t</td><td>Transactional AND. First, s is executed, afterwards t is executed. If s or t fails, the action will be terminated and a rollback to the state before s & t is performed.</td></tr>
<tr><td>$<op></td><td>Flags the operator <op> as commutative. Usually operands are executed / evaluated from left to right with respect to bracketing (left-associative). But the sequences s, t, u in s $<op> t $<op> u are executed / evaluated in arbitrary order.</td></tr>
<tr><td>s *</td><td>Executes s repeatedly as long as its execution does not fail.</td></tr>
<tr><td>s {n}</td><td>Executes s repeatedly as long as its execution does not fail, but anyway n times at most.</td></tr>
<tr><td>!</td><td>Found matches are dumped into VCG formatted files. Every match produces three files within the current directoy[1]:
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</td></tr>
<tr><td>Rule</td><td>Only the first pattern match produced by the action Rule will be rewritten.</td></tr>
<tr><td>[Rule]</td><td>Every pattern match produced by the action Rule will be rewritten.
Note: This operator is mainly added for benchmark purposes. Its semantic is not equal to Rule*. Instead this operator collects all the matches first before starting rewritings. In particular one needs to avoid deleting a graph element that is bound by another match.</td></tr>
<tr><td>v = w</td><td>The variable v is assigned to w. If w is undefined, v will be undefined, too.</td></tr>
<tr><td>v = @(x)</td><td>The variable v is assigned to the graph element identified by x. If x is not defined any more, v will be undefined, too.</td></tr>
<tr><td>def(Parameters)</td><td>Gets successful if all the graph elements in Parameters exist, i.e. if all the variables are defined.</td></tr>
<tr><td>true</td><td>A constant acting as a successful match.</td></tr>
<tr><td>false</td><td>A constant acting as a failed match.</td></tr>
</table>

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

Table 5.1: Graph rewriting expressions

`s`(tep)	Executes the next rewriting rule (match and rewrite)
`d`(elaied 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

Creates a search plan for the rewriting 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. This is kind of a rough specification. 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.

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.

CHAPTER 6

EXAMPLES

6.1 Busy Beaver

We want GrGen to work as hard as a busy beaver [2, 8]. 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.

6.1.1 Graph Model

Let's start with

```
model TuringMashine;
```

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 $[0:1]$.

```
node class TapePosition;
edge class right
  connect TapePosition[0:1] -> TapePosition[0:1];

edge class value
  connect TapePosition[1] -> TapePosition[1];
edge class zero extends value;
edge class one extends value;
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 *move...* edge leads to the next state.

```
node class RWHead;

node class WriteValue;
node class WriteZero extends WriteValue;
node class WriteOne extends WriteValue;
node class WriteEmpty extends WriteValue;

edge class moveLeft;
edge class moveRight;
edge class dontMove;
```

6.1.2 Rule Set

Now the rule set: we start with

```
actions Turing using TuringModel;
```

We need rewrite rules for the following steps of the turing machine:

1. Reading the value of the current tape cell and select a outgoing edge of the current state.
2. Writing a new value in 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 the right, respectively.

```
rule readZeroRule {
   pattern {
      s:State -:RWHead->tp:TapePosition -zv:zero->tp;
      s -zr:zero-> wv:WriteValue;
   }
   replace {
      s -zr-> wv;
      tp -zv-> tp;
      wv -:RWHead->tp;
   }
}
```

We the state and the current cell (*RWHead* edge) and check, if 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*. Analogous the remaining rules:

```
rule readOneRule {
   pattern {
      s:State -:RWHead-> tp:TapePosition -ov:one-> tp;
      s -or:one-> wv:WriteValue;
   }
   replace {
      s -or-> wv;
      tp -ov-> tp;
      wv -:RWHead-> tp;
   }
}

rule readEmptyRule {
   pattern {
      s:State -:RWHead-> tp:TapePosition -ev:empty-> tp;
      s -er:empty-> wv:WriteValue;
   }
   replace {
      s -er-> wv;
      tp -ev-> tp;
      wv -:RWHead-> tp;
   }
}

rule writeZeroRule {
```

```
    pattern {
        wv:WriteZero -rw:RWHead-> tp:TapePosition -:value-> tp;
    }
    replace {
        wv -rw-> tp -:zero-> tp;
    }
}

rule writeOneRule {
    pattern {
        wv:WriteOne -rw:RWHead-> tp:TapePosition -:value-> tp;
    }
    replace {
        wv -rw-> tp -:one-> tp;
    }
}

rule writeEmptyRule {
    pattern {
        wv:WriteEmpty -rw:RWHead-> tp:TapePosition -:value-> tp;
    }
    replace {
        wv -rw-> tp -:empty-> tp;
    }
}

rule moveLeftRule {
    pattern {
        wv:WriteValue -m:moveLeft-> s:State;
        wv -:RWHead-> tp:TapePosition <-r:right- ltp:TapePosition;
    }
    replace {
        wv -m-> s;
        s -:RWHead-> ltp -r-> tp;
    }
}

rule moveRightRule {
    pattern {
        wv:WriteValue -m:moveRight-> s:State;
        wv -:RWHead-> tp:TapePosition -r:right-> rtp:TapePosition;
    }
    replace {
        wv -m-> s;
        s -:RWHead-> rtp <-r- tp;
    }
}

rule dontMoveRule {
    pattern {
        wv:WriteValue -m:dontMove-> s:State;
        wv -:RWHead-> tp:TapePosition;
    }
    replace {
        tp;
        wv -m-> s;
        s -:RWHead-> tp;
    }
}
```

```

rule ensureMoveLeftValidRule {
   pattern {
      wv:WriteValue -m:moveLeft-> s:State;
      wv -rw:RWHead-> tp:TapePosition;
      negative {
         tp <-:right- ltp:TapePosition;
      }
   }
   replace {
      wv -m-> s;
      wv -rw-> tp <-:right- ltp:TapePosition -:empty-> ltp;
   }
}

rule ensureMoveRightValidRule {
   pattern {
      wv:WriteValue -m:moveRight-> s:State;
      wv -rw:RWHead-> tp:TapePosition;
      negative {
         tp -:right-> rtp:TapePosition;
      }
   }
   replace {
      wv -m-> s;
      wv -rw-> tp -:right-> rtp:TapePosition -:empty-> rtp;
   }
}
```

Have a look at the negative condition within the *ensureMove...* rules. They ensure, that the current cell is in deed at the end of the tape: an edge to a right / left neighbor cell may not exist.

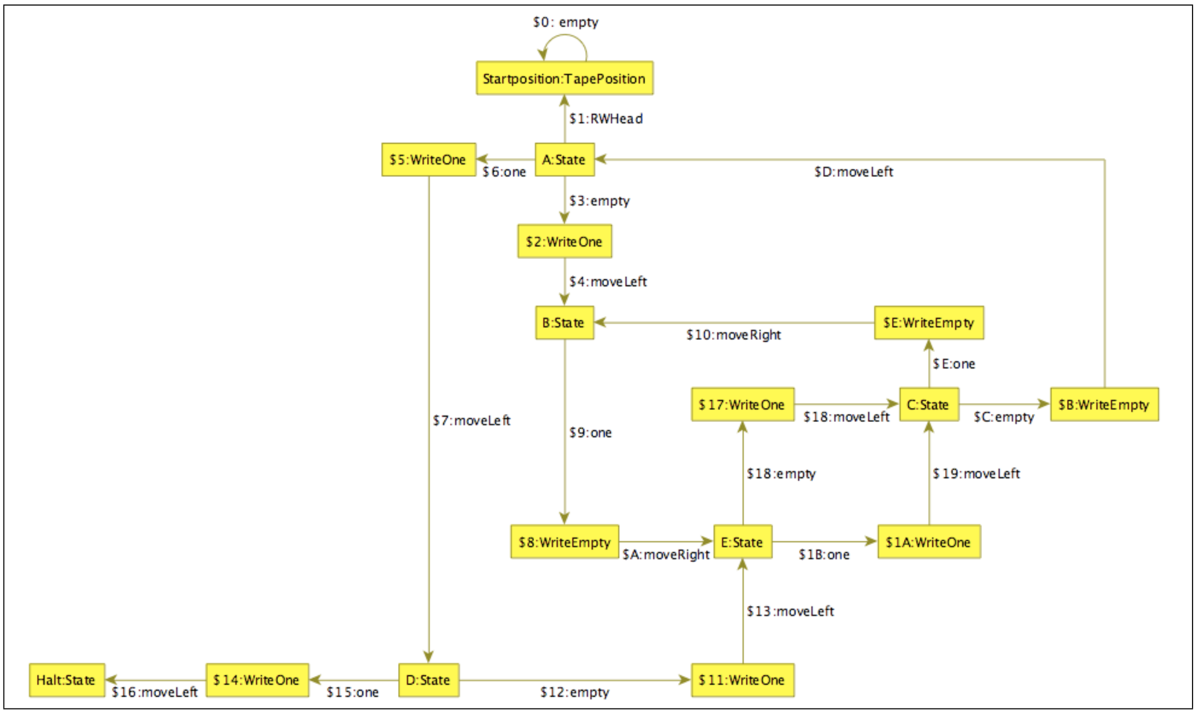
6.1.3 Rule Execution with GrShell

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

```
select backend "lgspBackend.dll"
new graph "../lib/lgsp-TuringModel.dll" "Busy␣Beaver"
select actions "../lib/lgsp-TuringActions.dll"

# Initialize tape
new tp:TapePosition($="Startposition")

# States
new sA:State($="A")
new sB:State($="B")
new sC:State($="C")
new sD:State($="D")
new sE:State($="E")
new sH:State($ = "Halt")

new sA -:RWHead-> tp

# Transitions: three lines per state for
#   - updating cell value
#   - moving read-write-head
# respectively

new sA_0: WriteOne
```

```
new sA -:empty-> sA_0
new sA_0 -:moveLeft-> sB

new sA_1: WriteOne
new sA -:one ->sA_1
new sA_1 -:moveLeft->sD

new sB_0: WriteOne
new sB -:empty-> sB_0
new sB_0 -:moveRight-> sC

new sB_1: WriteEmpty
new sB -:one-> sB_1
new sB_1 -:moveRight-> sE

new sC_0: WriteEmpty
new sC -:empty ->sC_0
new sC_0 -:moveLeft->sA

new sC_1: WriteEmpty
new sC -:one-> sC_1
new sC_1 -:moveRight-> sB

new sD_0: WriteOne
new sD -:empty ->sD_0
new sD_0 -:moveLeft->sE

new sD_1: WriteOne
new sD -:one-> sD_1
new sD_1 -:moveLeft-> sH

new sE_0: WriteOne
new sE -:empty ->sE_0
new sE_0 -:moveLeft->sC

new sE_1: WriteOne
new sE -:one-> sE_1
new sE_1 -:moveLeft-> sC
}
```

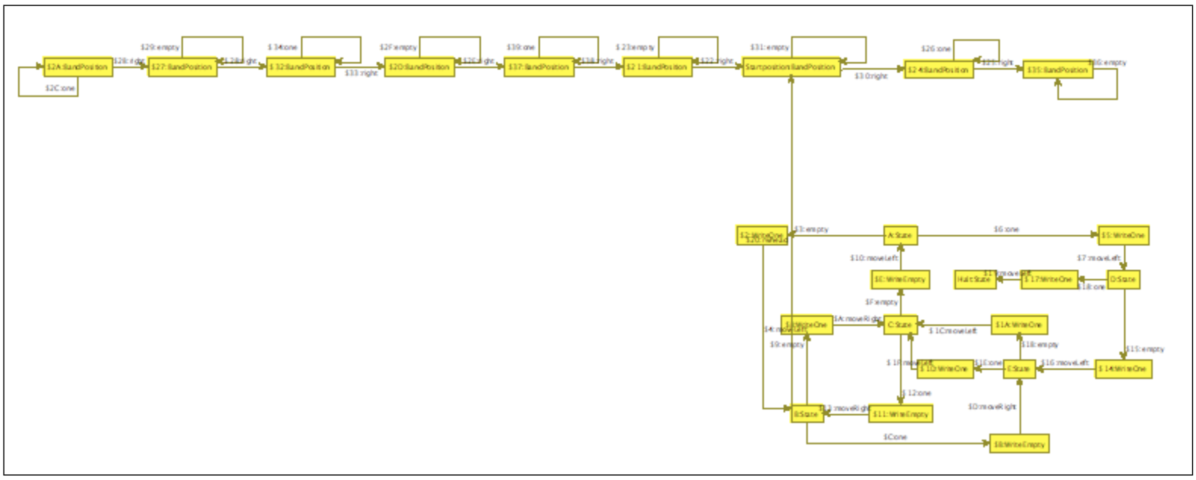
Our busy beaver looks like this:



The graph rewriting sequence is quite straight forward and generic to the turing graph model. Note that for each state the "*...Empty...* — *...One...*" selection is unambiguous.

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

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



In order to improve the performance we generate better search plans.

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

Let the beaver run:

```
grs ((readOneRule | readEmptyRule) ; (writeOneRule | writeEmptyRule) ;
    (ensureMoveLeftValidRule | ensureMoveRightValidRule) ; (moveLeftRule |
    moveRightRule))*
```

6.2 Fractals

BIBLIOGRAPHY

[1] R. Geiß et al.: GrGen*: A Fast SPO-Based Graph Rewriting Tool* in Graph Transformations, number 4178 in LNCS, pages 383-397, Springer, 2006

[2] M. Kroll: *Portierung des C-Anteils des Graphersetzungssystems* GrGen *nach C# mit Erweiterungen*, Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2007

[3] S. Hack: *Graphersetzung für Optimierung in der Codeerzeugung* Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2003

[4] D. Grund: *Negative Anwendungsbedingungen für den Graphersetzer* GrGen Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2004

[5] A. Szalkowski: *Negative Anwendungsbedingungen für das suchprogrammbasierte Backend von GrGen* Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2005

[6] G. Batz: *Graphersetzung für eine Zwischendarstellung im Übersetzerbau* Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2005

[7] K. Jensen, N. Wirth: *Pascal User Manual and Report* Springer, ^{4}1991

[8] A. Dewdney: *A computer trap for the Busy Beaver, the hardest-working machine* Scientific American, 251(2), pages 10-12, 16, 17, August 1984

[9] H. Marxen, J. Buntrock: *Old list of record TMs.*
http://www.drb.insel.de/ heiner/BB/index.html. Version: August 2000