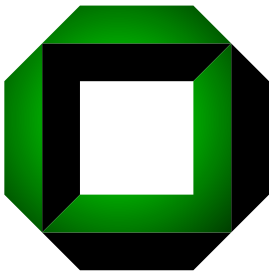


Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

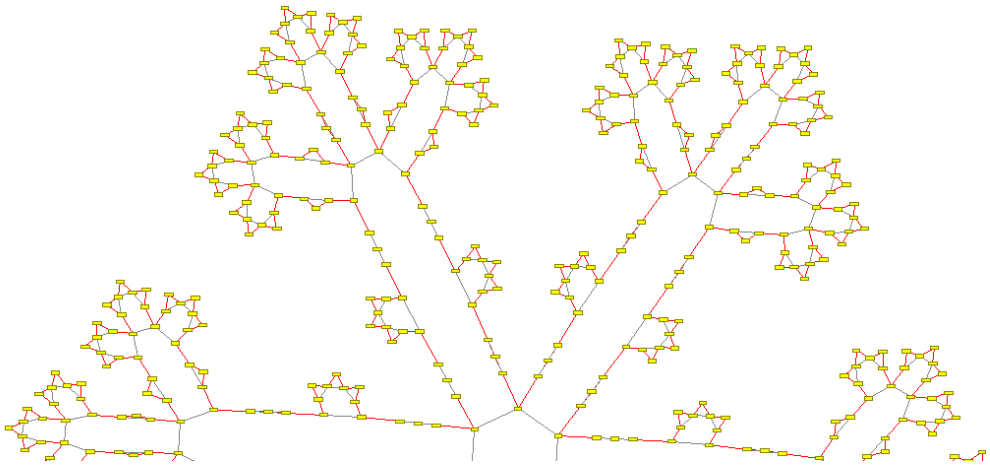
Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Goos

The GrGen.NET User Manual

Refers to GrGen.NET Release 4.0alpha

—DRAFT—

www.grgen.net

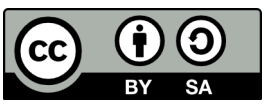


Jakob Blomer Rubino Geiß
Edgar Jakumeit

March 29, 2013

ABSTRACT

GrGen.NET: transformation of structures made easy – with languages for graph modeling, pattern matching, and rewriting, as well as rule control; brought to life by a compiler and a rapid prototyping environment offering graphical and stepwise debugging. The Graph Rewrite Generator is a tool enabling elegant and convenient development of graph rewriting applications with comparable performance to conventionally developed ones. This user manual contains both, normative statements in the sense of a reference manual as well as an informal guide to the features and usage of GrGen.NET.



FOREWORD FOR RELEASE 1.4

First of all a word about the term "graph rewriting". Some would rather say "graph transformation"; some even think there is a difference between these two. We don't see such differences and use graph rewriting for consistency.

The GrGen project started in spring 2003 with the diploma thesis of Sebastian Hack under supervision of Rubino Geiß. At that time we needed a tool to find patterns in graph based intermediate representations used in compiler construction. We imagined a tool that is fast, expressive, and easy to integrate into our compiler infrastructure. So far Optimix[Ass00] was the only tool that brought together the areas of compiler construction and graph rewriting. However its approach is to feature many provable properties of the system per se, such as termination, confluence of derivations, and complete coverage of graphs. This is achieved by restricting the expressiveness of the whole formalism below Turing-completeness. Our tool GrGen in contrast should be Turing-complete. Thus GrGen.NET provides the user with strong expressiveness but leaves the task of proving such properties to the user.

To get a prototype quickly, we delegated the costly task of subgraph matching to a relational database system [Hac03]. Albeit the performance of this implementation could be improved substantially over the years, we believed that there was more to come. Inspired by the PhD thesis of Heiko Dörr [Dör95] we reimplemented our tool to use search plan driven graph pattern matching of its own. This matching algorithm evolved over time [Sza05, Bat05b, Bat05a, Bat06a, BKG08] and has been ported from C to C# [KG07, Kro07]. In the year 2005 Varró [VVF06] independently proposed a similar search plan based approach.

Though we started four years ago to facilitate some compiler construction problems, in the meantime GrGen.NET has grown into a general purpose tool for graph rewriting.

We want to thank all co-workers and students that helped during the design and implementation of GrGen.NET as well as the writing of this manual. Especially we want to thank Dr. Sebastian Hack, G. Veit Batz, Michael Beck, Tom Gelhausen, Moritz Kroll, Dr. Andreas Ludwig, and Dr. Markus Noga. Finally, all this would not happened without the productive atmosphere and the generous support that Prof. Goos provides at his chair.

We wish all readers of the manual—and especially all users of GrGen.NET—a pleasant graph rewrite experience. We hope you enjoy using GrGen.NET as much as we enjoy developing it.

Thank you for using GrGen.NET.

Karlsruhe in July 2007, Rubino Geiß on behalf of the GrGen.NET-Team

FOREWORD

Since the last version of this manual which was written for GrGen.NET v1.4 a lot has happened, as can be seen quite easily in the fact that this manual describes GrGen.NET v3.6. The porting of C to C# [Kro07] allowed for a faster pace of development, which yielded alternatives and subpatterns allowing for structural recursion [Jak08, HJG08], undirected edge support plus fine grain pattern conditions [Buc08], a data model that is more user friendly at the API, support for visited flags, and an prototypical implementation of an embedding of GrGen.NET as a domain specific language into C# [Kro] – resulting in GrGen.NET v2.0.

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

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

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

Regarding questions please contact the GrGen.NET-Team via email to `grgen` at the host given by `ipd.info.uni-karlsruhe.de`.

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

Thank you for using GrGen.NET.

Karlsruhe in March 2013, Edgar Jakumeit on behalf of the GrGen.NET-Team

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 What Is GrGen.NET?

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

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

GrGen.NET is the successor of the GrGen tool presented at ICGT 2006 [GBG+06]. The ".NET" postfix of the new name indicates that GrGen has been reimplemented in C# for the Microsoft .NET or Mono environment [Mic07, Tea07]; it is open source licensed under LGPL3(`www.gnu.org/licenses/lgpl.html`) and available for download at `www.grgen.net`. Starting as a compiler construction helper tool it has grown into a software development tool for general purpose graph transformation, which offers the highest combined speed of development and execution for graph based algorithms through its declarative languages with automatic optimization.

1.2 When to Use GrGen.NET and When Not

You may be interested in using GrGen.NET if you have to tackle the task of transforming meshes of massively linked objects, i.e. graph-like data structures, as is the case in e.g. model transformation, computer linguistics, or modern compiler construction (any time there is more than one relation of interest in between the data entities your algorithm operates upon). These tasks are traditionally handled by pointer structures and pointer structure navigation-, search-, and replacement routines written by hand – this low-level, pointer-fiddling code can be generated automatically for you by GrGen.NET. You specify your transformation task on a higher level of nodes connected by edges, and rewrite rules of patterns to be searched plus modifications to be carried out, and then let GrGen.NET generate the algorithmic core of your application.

There is nothing to gain from GrGen.NET if scalars and lists are sufficient to model your domain, which is the case for a lot of tasks in computing indeed; but which is not the case for others which would be better modeled with trees and especially graphs, but aren't because of the cost of maintaining pointer structures by hand. The graph rewrite generator is not the right tool for you if you're searching for a visual environment to teach children programming – it's a tool for software engineers. Neither is it what you need if your graph structured data is to be interactively edited by an end user instead of being automatically transformed by rules (the editor generator DiaGen[Dia] may be of interest in this case).

1.3 Features of GrGen.NET

The process of graph rewriting can be divided into four steps: Representing a graph according to a model (creating an instance graph), searching a pattern aka finding a match, performing changes to the matched spot in the host graph, and, finally, selecting which rule(s) to apply where next. We have organized the presentation of the features of the GrGen.NET languages according to this breakdown of graph rewriting:

- The graph model (meta-model) language supports:
  - Typed nodes and edges, with multiple inheritance on types
  - Directed multigraphs (including multiple edges of the same type)
  - Undirected and arbitrarily directed edges
  - Node and edge types can be equipped with typed attributes (like structs) including generic set, map, and array types
  - Connection assertions to restrict the "shape" of graphs
  - Turing complete language for checking complex conditions
- The pattern language supports:
  - Plain isomorphic subgraph matching (injective mapping)
  - Homomorphic matching for a selectable set of pattern elements, so that they may match (non-injectively) the same graph element
  - Attribute conditions (e.g. arithmetic-,boolean-,string- or set-expressions on the attributes)
  - Type conditions (including powerful instanceof-like type expressions)
  - Nested patterns, specifying negative and positive application conditions as well as iterated, optional, or alternative structures
  - Subpatterns for pattern reuse, allowing via recursion to match substructures of arbitraty depth (e.g. iterated paths) and breadth (e.g multinodes)
  - Parameter passing to rules and subpatterns
- The rewrite language supports:
  - Keeping, adding and deleting graph elements according to the SPO approach
  - Choosing out of three additional rule application semantics: DPO or exact patterns only or induced subgraphs only
  - Attribute re-/calculation (assigning the result of e.g. arithmetic expressions to the attributes)
  - Retyping of nodes/edges (a more general version of casts known from common programming languages)
  - Creation of new nodes/edges of only dynamically known types or as exact copies of other nodes/edges

  - Two modes of specification: A rule can either express changes to be made to the match or replace the whole match
  - A rewrite part for the nested patterns and subpatterns, so that complex structures can not only get matched (parsing), but also get rewritten (transduction)
  - Embedded graph rewrite sequences capable of calling other rules (with access to the nodes/edges of the rule)
  - Emitting user-defined text to `stdout` or files during the rewrite steps
  - Visited flags and storages to communicate between rule applications using state
  - Parameter passing out of rules and subpatterns

- The rule application control language (grs: graph rewrite sequences) supports:
  - Rule execution
  - Logical and sequential connectives
  - Loops
  - Variable handling
  - Basic graph querying and manipulation
  - Visited flags and storages management
  - Extended control (e.g decisions, transactions, backtracking, indeterministic choice)
  - Sequence definitions (procedural abstraction)
  - These constructs combined allow to program a state space enumeration

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

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

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

1.4 System Overview

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

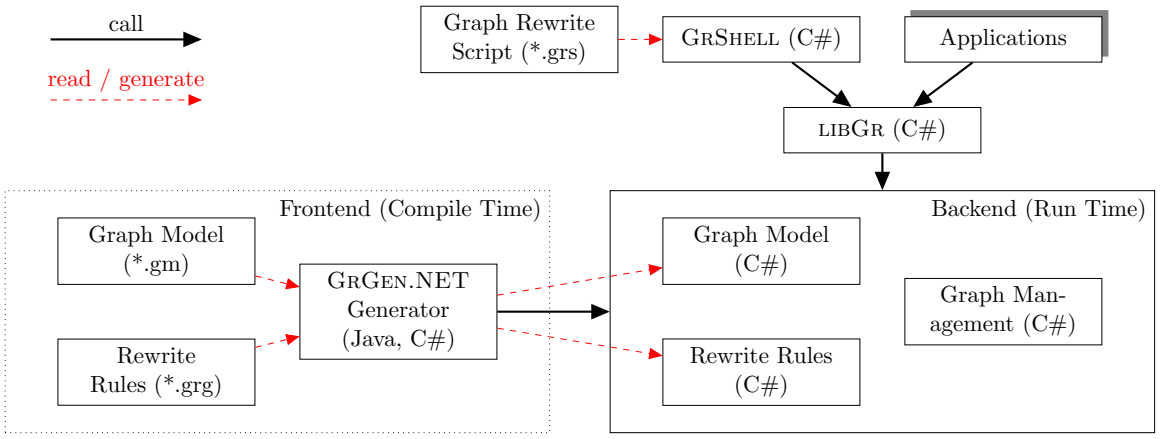


Figure 1.1: GrGen.NET system components [Kro07]

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

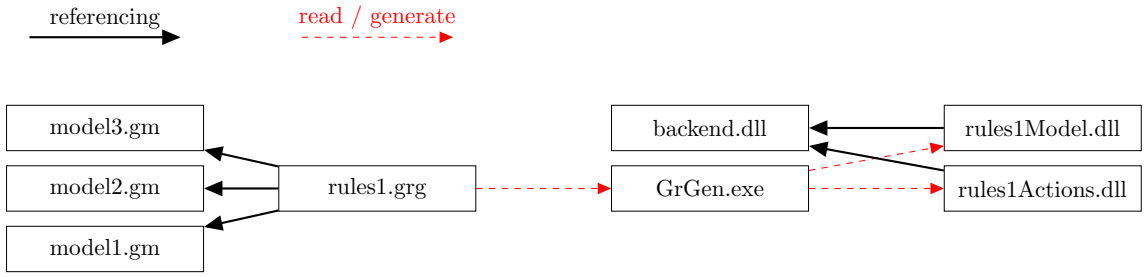


Figure 1.2: Generating a graph rewrite system

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

Generating a GrGen.NET graph rewrite system may be considered a preliminary task. The actual process of rewriting as well as dealing with host graphs is performed by GrGen.NET's backend. GrGen.NET provides a backend API in two versions — the named and typed entities which get generated plus the name string and object based interface offered by the .NET library libGr. For most issues—in particular for experimental purposes—you might rather want to work with the GrShell because of its rapid prototyping support. However, GrShell does not provide the full power of the libGr; see also note 16 on page 38.

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

1.5 What Is Graph Rewriting?

The notion of graph rewriting as understood by GrGen.NET is a method for declaratively specifying "changes" to a graph. This is comparable to the well-known term rewriting. Normally you use one or more *graph rewrite rules* to accomplish a certain task. GrGen.NET implements an SPO-based approach (as default). In the simplest case such a graph rewrite rule consists of a tuple $L \rightarrow R$, whereas L—the *left hand side* of the rule—is called *pattern graph* and R—the *right hand side* of the rule—is the *replacement graph*.

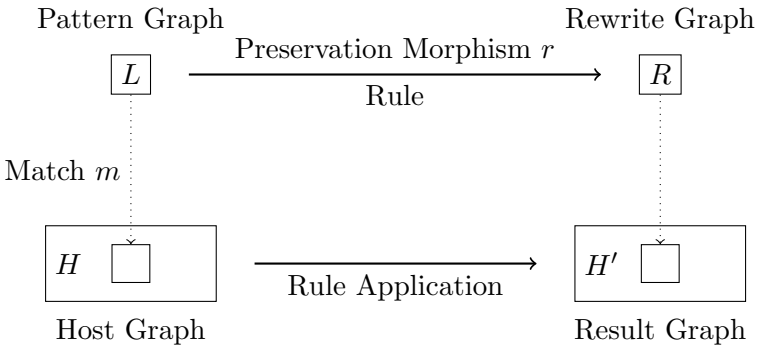


Figure 1.3: Basic Idea of Graph Rewriting

Moreover we need to identify graph elements (nodes or edges) of L and R for preserving them during rewrite. This is done by a *preservation morphism* r mapping elements from L to R; the morphism r is injective, but needs to be neither surjective nor total. Together with a rule name p we have $p : L \xrightarrow{r} R$.

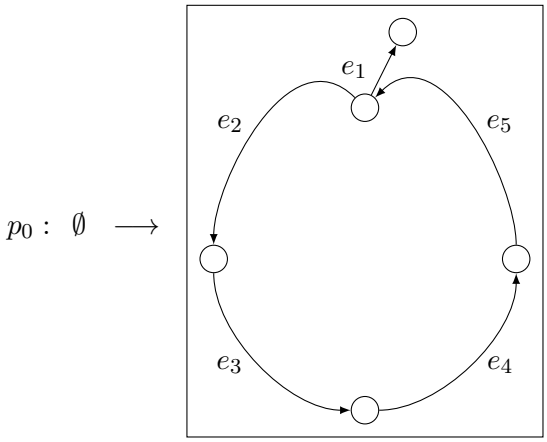
The transformation is done by *application* of a rule to a *host graph* H. To do so, we have to find an occurrence of the pattern graph in the host graph. Mathematically speaking, such a *match* m is an isomorphism from L to a subgraph of H. This morphism may not be unique, i.e. there may be several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

1.6 An Example

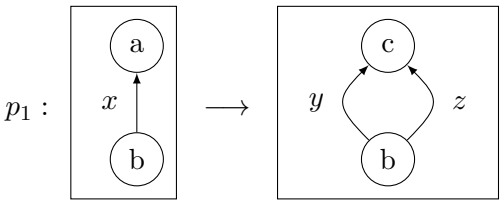
We'll have a look at a small example. Graph elements (nodes and edges) are labeled with and identifier. If a type is necessary then it is stated after a colon. We start using a special case to construct our host graph: an empty pattern always produces exactly one[2] match

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

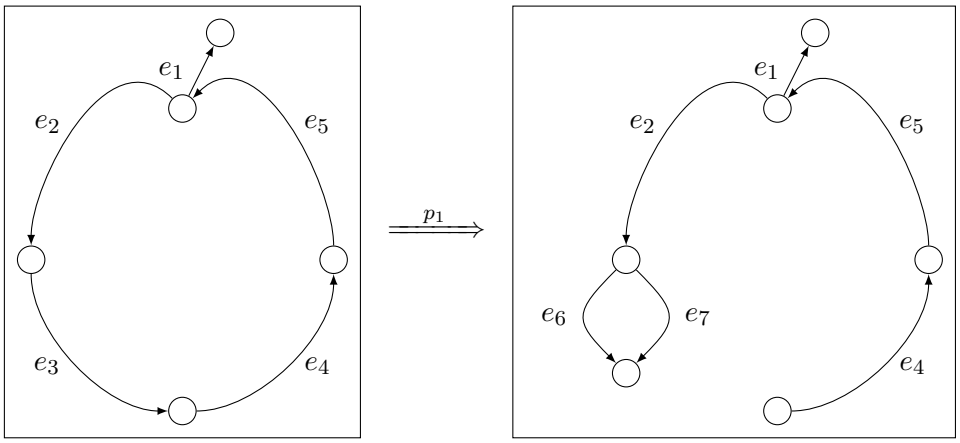
(independent of the host graph). So we construct an apple by applying



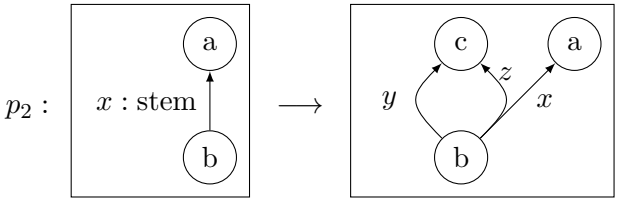
to the empty host graph. As the result we get an apple as new host graph H. Now we want to rewrite our apple with stem to an apple with a leaflet. So we apply



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

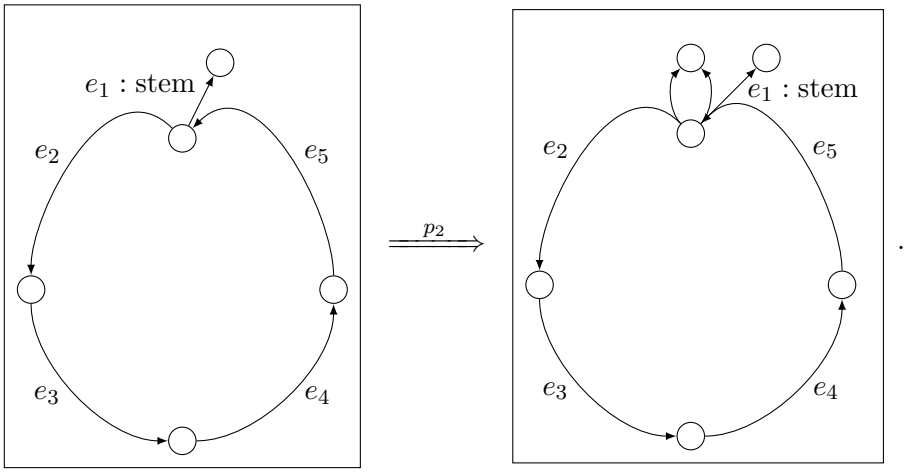


What happened? GrGen.NET has arbitrarily chosen one match out of the set of possible matches, because x matches edge e_3 as well as e_1. A correct solution could make use of edge type information. We have to change rule p_0 to generate the edge e_1 with a special type "stem". And this time we will even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



.

If we apply p_2 to the modified H_1 this leads to



1.7 The Tools

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

1.7.1 `GrGen.exe`

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

Usage

```
[mono] GrGen.exe [-keep [<dest-dir>]] [-use <existing-dir>] [-debug]
                 [-b <backend-dll>] [-o <output-dir>] [-r <assembly-path>]
                 [-lazynic] [-noinline]
                 <rule-set>
```

rule-set is a file containing a rule set specification according to Chapter 4. Usually such a file has the suffix `.grg`. The suffix `.grg` may be omitted. By default GrGen.NET tries to write the compiled assemblies into the same directory as the rule set file. This can be changed by the optional parameter *output-dir*.

Options

`-keep` Keep the generated C# source files. If *dest-dir* is omitted, a subdirectory `tmpgrgen`*n*[3] within the current directory will be created. The destination directory contains:

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

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

`-debug` Compile the assemblies with debug information.

`-lazynic` Negatives, Independents, and Conditions are only executed at the end of matching (normally asap).

`-noinline` Subpattern usages are never inlined.

`-b` Use the backend library *backend-dll* (default is LGSPBackend).

`-r` Link the assembly *assembly-path* as reference to the compilation result.

`-o` Store generated assemblies in *output-dir*.

Requires

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

NOTE (1)

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

NOTE (2)

The grgen compiler consists of a Java frontend used by the C# backend `grgen.exe`. The java frontend can be executed itself to get a visualization of the model and the rewrite rules, in the form of a dump of the compiler IR as a .vcg file:

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

NOTE (3)

If you run into `Unable to process specification: The system cannot find the file specified` errors, you may need to install a JDK to a non system path and add the bin folder of the JDK to the path variable. (Normally just installing a JRE is sufficient.)

[3] *n* is an increasing number.

1.7.2 GrShell.exe

The `GrShell.exe` is a shell application on top of the libGr. GrShell is capable of creating, manipulating, and dumping graphs as well as performing graph rewriting with graphical debug support. For further information about the GrShell language see Chapter 17.

Usage

`[mono] grShell.exe [-N] [-C "<commands>"] <grshell-script>*`
Opens the interactive shell. The GrShell will include and execute the commands in the optional list of *grshell-scripts* (usually `*.grs` files) in the given order. The `grs` suffixes may be omitted. GrShell returns 0 on successful execution, or in non-interactive mode -1 if the specified shell script could not be executed, or -2 if a `validate` with `exitonfailure` failed.

Options

`-N` Enables non-debug non-gui mode which exits on error with an error code instead of waiting for user input.

`-C` Execute the quoted GrShell commands immediately (before the first script file). Instead of a line break use a double semicolon `;;` to separate commands.

Requires

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

1.7.3 LibGr.dll

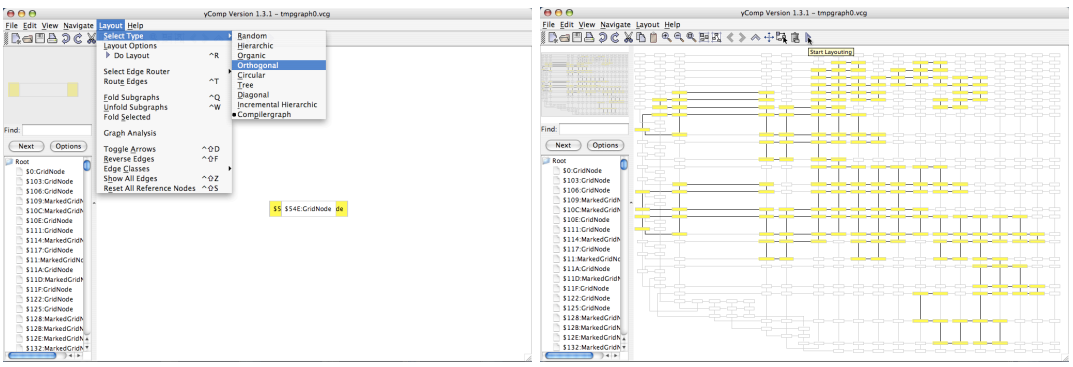
The libGr is a .NET assembly implementing GrGen.NET's API. See the extracted HTML documentation for interface descriptions at http://www.grgen.net/doc/API_3_5/; a short introduction is given in chaper 20.

1.7.4 lgspBackend.dll

The lgspBackend is a .NET assembly containing the libGr SearchPlan backend, the only backend supported by GrGen.NET as of now, implementing together with the generated assemblies the API offered by libGr. It allows to analyze the graph and to regenerate the matcher programs at runtime, on user request, see 17.3.2. For a more detailed introduction have a look at chapter 22.

1.7.5 yComp.jar

yComp [KBG+07] is a graph visualization tool based on yFiles [yWo07]. It is well integrated and shipped with GrGen.NET, but it's not a part of GrGen.NET. yComp implements several graph layout algorithms and has file format support for VCG, GML and YGF among others.



Usage

Usually yComp will be loaded by the GrShell. You might want to open yComp manually by typing

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

Or by executing the batch file `ycomp` under Linux / `ycomp.bat` under Windows, which will start yComp on the given file with increased heap space. The *graph-file* may be any graph file in a supported format. yComp will open this file on startup.

Hints

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

Requires

Java Runtime Environment 1.5 (or above).

1.8 Development Goals

The development goals of GrGen.NET were

Expressiveness

is achieved by powerful and declarative specification languages for pattern matching and rewriting by rewrite rules, builing upon a rich graph model language. In addition to the unmatched expressiveness of the basic actions, the rule language now offers nested and subpatterns which allow to handle substructures of arbitrary depth and arbitrary breadth declaratively within the rules, by now even surpassing the capabilities of the VIATRA2[VB07, VHV08] graph rewriting tool, the strongest competitor in rule expressiveness. The rules can be combined by graph rewrite sequences, a rule application control language with variables and logical as well as iterative control flow; it was recently extended by storages as pioneered by the VMTS[LLMC05] graph rewriting tool, allowing for more concise and faster solutions. You may have a look at the GrGen.NET solution of the Hello World! case [BJ11b] (or the other cases [JB11] and [BJ11a]) of TTC 2011 highlighting the conciseness achieved by the expressiveness of the many language constructs.

Performance

i.e. high speed at modest memory consumption, is needed to tackle real world problems. It is achieved by typing, easening the life of the programmer by eliminating large classes or errors as well as speeding up the pattern matcher, by the generative approach compiling the rules into executable code, and by the heuristic optimizations of search state space stepping and the host graph sensitive search plans. In order to accelerate the matching step, we internally introduce *search plans* to represent different *matching strategies* and equip these search plans with a cost model, taking the present host graph into account. The task of selecting a good search plan is then considered as an optimization problem [BKG08, Bat06a]. In contrast to systems like Fujaba[Fuj07, NNZ00], our strongest competitor regarding performance, our pattern matching algorithm is fully automatic and does not need to be tuned nor to be implemented in parts by hand. According to Varró's benchmark[VSV05], it is at least one order of magnitude faster than any other tool known to us.

Understandability and Learnability

was taken care by evaluating for each language construct several options, prefering

constructs already known from programming languages — the ones which seemed most clean and intuitive while satisfying the other constraints were chosen. This can be noted in comparison with the languages of the (otherwise well-engineered) graph rewrite/-database system GReTL [HE11], which may be pleasing to someone from the realm of formal specification, but are not to the mind of a programmer which just wants to process his graphs on a higher level of abstraction, with declarative pattern matching and rewriting, on a visualization of the network of objects, instead of low level pointer structure fiddling, chasing objects by following references in the debugger. As we know that even the best designed language is not self explaining we put an emphasize on the user manual currently read by you.

Development Convenience
is gained by interactive and graphical debugging of the rule application, capable of visualizing the matched pattern, the rewrite which will be applied, and the currently active rule out of the rewrite sequence. A further point easening development is the application programming interface of the generated code, which offers access to named, statically typed entities, catching most errors before runtime and allowing the code completion mechanisms of modern IDEs to excel. In addition a generic interface operating on name strings and .NET objects is available for applications where the rules may change at runtime (as e.g. the GrShell). There's one convenience not offered you might expect: a visual rule language and an editor. This brings a clear benefit – graph transformation specifications to be processed by GrGen.NET can be easily generated – but especially is a good deal cheaper to implement. Given the limited resources of an university project this is an important point, as can be seen with the AGG[ERT99] tool, offering a nice graphical editor but delivering performance only up to simple toy examples (causing the wrong impression that graph rewriting is notoriously inefficient).

Well Founded Semantics
to ease formal, but especially human reasoning. The semantics of GrGen.NET are specified in [Gei08], based upon graph homomorphisms, denotational evaluation functions and category theory. The GrGen.NET-rewrite step is based by default on the *single-pushout approach* (SPO, for explanation see [EHK+99]), with the *double-pushout approach* (DPO, for explanation see [CMR+99]) available on request, too. The semantics of the recursive rules introduced in version 2.0 are given in [Jak08], utilizing pair star graph grammars on the meta level to assemble the rules of the object level. The formal semantics are not as complete as for the graph programming language GP[Plu09] though, mainly due to the large amount of features — the convenience at using the language had priority over the convenience at reasoning formally about the language.

Platform Independence
is achived by using languages compiled to byte code for virtual machines backed by large, standardized libraries, specifically: Java and C#. This should prevent the fate of the grandfather of all graph rewrite systems, PROGRES[SWZ99], which achieved a high level of sophistication, but is difficult to run by now, or simply: outdated.

General Purpose Graph Transformation
in contrast to special purpose graph transformation. A lot of other graph based tools are geared towards special purpose applications, e.g. verification (GROOVE [Ren04]), or biology (XL [KK07], or model transformation (VIATRA2[VB07]). This means that design decisions were taken to ease uses in this application areas at the cost of rendering uses in other domains more difficult. And that features were added in a way which just satisfies the needs of the domain at hand instead of striving for a more general solution (which would have caused higher costs at designing and implementing this features). While the old GrGen was built as a special purpose compiler construction tool for

internal use (optimizations on the graph based compiler intermediate representation FIRM – see www.libfirm.org), the new GrGen.NET was built from the beginning as a general purpose graph transformation tool for external use – to be used in areas as diverse as computer linguistics, software engineering, computational biology or sociology – for reasoning with semantic nets, transformation of natural language to UML models, model transformation, processing of program graphs, genome simulation, or pattern matching in social nets. Several of them are worked on, you may have a look at [BG09] or [GDG08] or [SGS09].

CHAPTER 2

QUICKSTART

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

2.1 Downloading & Installing

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

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

If you're using Linux: Unpack the package to a directory of your choice, for example into `/opt/grgen`:

```
mkdir /opt/grgen
tar xvfj GrGenNET-V1_3_1-2007-12-06.tar.bz2
mv GrGenNET-V1_3_1-2007-12-06/* /opt/grgen/
rmdir GrGenNET-V1_3_1-2007-12-06
```

Add the `/opt/grgen/bin` directory to your search paths, for instance if you use `bash` add a line to your `/home/.profile` file.

```
export PATH=/opt/grgen/bin:$PATH
```

Furthermore we create a directory for our GrGen.NET data, for instance by `mkdir /home/grgen`.

If you're using Microsoft Windows: Extract the .zip archive to a directory of your choice and add the `bin` subdirectory to your search path via *control panel* → *system properties* / environment variables. Execute the GrGen.NET assemblies from a command line window (*Start* → *Run...* → `cmd`). For MS .NET the `mono` prefix is neither applicable nor needed.

NOTE (4)

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

2.2 Creating a Graph Model

In the directory `/home/grgen` we create a text file `StateMachine.gm` that contains the graph meta model for our state machine[1]. By graph meta model we mean a set of node types and edge types which are available for building state machine graphs (see Chapter 3). Figure 2.1 shows the meta model. What have we done? You can see two base types, `State` for

```
node class State {
    id: int;
}

abstract node class SpecialState extends State;
node class StartState extends SpecialState;
node class FinalState extends SpecialState;
node class StartFinalState extends StartState, FinalState;

edge class Transition {
    Trigger: string;
}

const edge class EpsilonTransition extends Transition;
```

Figure 2.1: Meta Model for State Machines

state nodes and `Transition` for transition edges that will connect the state nodes. `State` has an integer attribute `id` and `Transition` has a string attribute `Trigger` which indicates the character sequence for switching from the source state node to the destination state node. For the rest of the types we use inheritance (keyword `extends`) which works more or less like inheritance in object oriented languages. Accordingly the `abstract` modifier for `SpecialState` means that you cannot create a node of that precise type, but you might create nodes of non-abstract subtypes. As you can see GrGen.NET supports multiple inheritance, and with `StartFinalState` we have constructed a "diamond" type hierarchy.

2.3 Creating Graphs

Let's test our graph meta model by creating a state machine graph. We will use the GrShell (see Chapter 17) and—for visualization—yComp. To get everything working we need a rule set file, too. For the moment we just create an almost empty file `removeEpsilons.grg` in the `/home/grgen` directory, containing only the line

```
using StateMachine;
```

Now, we could start by launching the GrShell and typing the commands interactively. This is, however, in most of the cases not the preferred way. We rather create a GrShell script, say `removeEpsilons.grs`, in the `/home/grgen` directory. Figure 2.2 shows this script. Run the script by executing `grshell removeEpsilons.grs`. The first time you execute the script, it might take a while because GrGen.NET has to compile the meta model and the rule set into .NET assemblies. The graph viewer yComp opens and after clicking the blue "layout graph" button on the very right side of the button bar, you get a window similiar to figure 2.3 (see also Section 1.7.5). The graph looks still a bit confusing. In fact it is quite normal that yComp's automatic layout algorithm needs manual adjustments. Quit yComp and exit the GrShell by typing `exit`.

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

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

Figure 2.2: Constructing a state machine graph in GrShell

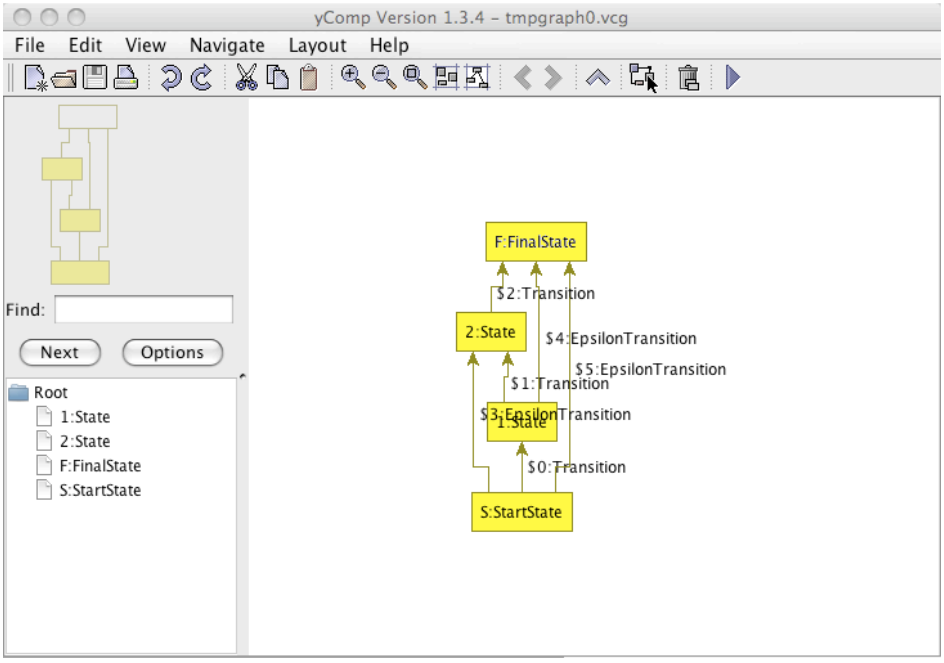


Figure 2.3: A first state machine

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

2.4 The Rewrite Rules

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

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

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

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

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

```
using StateMachine;

test checkStartState {
    x:StartState;
    negative {
        x;
        y:StartState;
    }
}

test checkDoublettes {
    negative {
        x:State -e:Transition-> y:State;
        hom(x,y);
        x -doublette:Transition-> y;
        if {typeof(doublette) == typeof(e);}
        if { ((typeof(e) == EpsilonTransition) || (e.Trigger == doublette.Trigger)); }
    }
}

rule forwardTransition {
    x:State -:EpsilonTransition-> y:State -e:Transition-> z:State;
    hom(x,y,z);
    negative {
        x -exists:Transition-> z;
        if {typeof(exists) == typeof(e);}
        if { ((typeof(e) == EpsilonTransition) || (e.Trigger == exists.Trigger)); }
    }
    modify {
        x -forward:typeof(e)-> z;
        eval {forward.Trigger = e.Trigger;}
    }
}

rule addStartFinalState {
    x:StartState -:EpsilonTransition-> :FinalState;
    modify {
        y:StartFinalState<x>;
        emit("Start␣state␣(" + x.id +  ")␣mutated␣into␣a␣start-and-final␣state");
    }
}

rule addFinalState {
    x:State -:EpsilonTransition-> :FinalState;
    if {typeof(x) < SpecialState;}
    modify {
        y:FinalState<x>;
    }
}

rule removeEpsilonTransition {
    -:EpsilonTransition->;
    replace {}
}
```

Figure 2.4: Rule set for removing ε-transitions

instance we don't want to forward just one ε-transition as `forwardTransition` would do; we want to forward them all. Such a rule composing is called *graph rewrite sequence* (see Chapter 8). We add the following line to our shell script `removeEpsilons.grs`:

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

This looks like a boolean expression and in fact it behaves similar. The whole expression is evaluated from left to right. A rule is successfully evaluated if a match could be found. We first check for a valid state machine, i.e. if the host graph contains exactly one start state and no redundant transitions. Thereafter we do the actual rewriting. These three steps are connected by lazy-evaluation-ands (`&&`), i.e. if one of them fails the evaluation will be canceled. We continue by disjunctively connected rules (connected by `|`). The angle brackets (`<>`) around the transformation rules indicate transactional processing: If the enclosed sequence returns `false` for some reason, all the already performed graph operations will be rolled back. That means not all of the rules must find a match. The `*` is used to apply the rule repeatedly as long as a match can be found. This includes applying the rule zero times. Even in this case `Rule*` is still successful.

2.5 Debugging and Output

If you execute the modified GrShell script, GrGen.NET starts its debugger. This way you can follow the evaluation of the graph rewrite sequence step by step in yComp. Just play around with the keys `d`, `s`, and `r` in GrShell: the `d` key lets you follow a single rewrite operation in multiple steps; the `s` key jumps to the next rule; and the `r` key runs to the end of the graph rewrite sequence. Finally you should get a graph like the one in figure 2.5

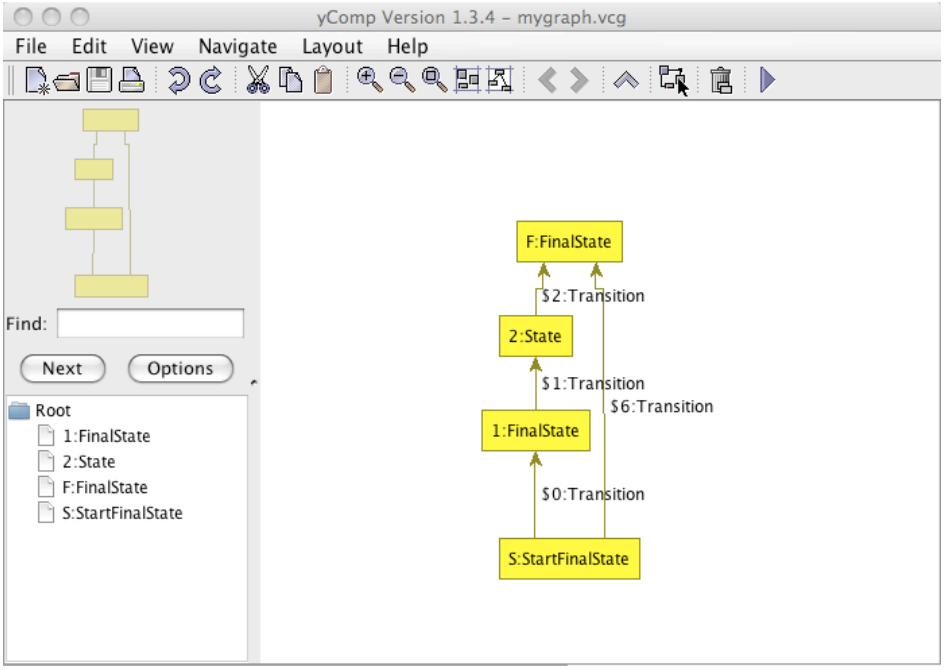


Figure 2.5: A state machine without ε-transitions

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

CHAPTER 3

GRAPH MODEL LANGUAGE

The key features of GrGen.NET *graph models* as described by Geiß et al. [GBG+06, KG07] are given below:

Types
: Nodes and edges are typed. This is similar to classes in common programming languages, except for the concept of methods that GrGen.NET nodes and edges don't support. GrGen.NET edge types can be directed and undirected.

Attributes
: Nodes and edges may possess attributes. The set of attributes assigned to a node or edge is determined by its type. The attributes themselves are typed, too.

Inheritance
: Node and edge types (classes) can be composed by multiple inheritance. `Node` and `Edge` are built-in root types of node and edge types, respectively. Inheritance eases the specification of attributes because subtypes inherit the attributes of their super types. Note that GrGen.NET lacks a concept of overwriting. On a path in the type hierarchy graph from a type up to the built-in root type there must be exactly one declaration for each attribute identifier. Furthermore, if multiple paths from a type up to the built-in root type exist, the declaring types for an attribute identifier must be the same on all such paths.

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

In this chapter as well as in Chapter 17 (GrShell) we use excerpts of Example 1 (the `Map` model) for illustration purposes.

3.1 Building Blocks

NOTE (5)

The following syntax specifications make heavy use of *syntax diagrams* (also known as rail diagrams). Syntax diagrams provide a visualization of EBNF[a] grammars. Follow a path along the arrows through a diagram to get a valid sentence (or subsentence) of the language. Ellipses represent terminals whereas rectangles represent non-terminals. For further information on syntax diagrams see [MMJW91].

[a] Extended Backus–Naur Form.

EXAMPLE (1)

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

```
enum Resident {VILLAGE = 500, TOWN = 5000, CITY = 50000}

node class Sight;

node class City {
  Size:Resident;
}

const node class Metropolis extends City {
  River:String;
}

abstract node class AbandonedCity extends City;
node class GhostTown extends AbandonedCity;

edge class Street;
edge class Trail extends Street;
edge class Highway extends Street
    connect Metropolis[+] --> Metropolis[+]
{
    Jam:boolean = false;
}
```

Basic elements of the GrGen.NET graph model language are identifiers to denominate nodes, edges, and attributes. The model's name itself is given by its file name. The GrGen.NET graph model language is case sensitive.

Ident, *IdentDecl*
A non-empty character sequence of arbitrary length consisting of letters, digits, or underscores. The first character must not be a digit. *Ident* and *IdentDecl* differ in their role: While *IdentDecl* is a *defining* occurrence of an identifier, *Ident* is a *using* occurrence. An *IdentDecl* non-terminal may be annotated. See Section 21.7 for annotations of declarations.

NOTE (6)

The GrGen.NET model language does not distinguish between declarations and definitions. More precisely, every declaration is also a definition. For instance, the following C-like pseudo GrGen.NET model language code is illegal:

```
node class t_node;
node class t_node {
  ...
}
```

Using an identifier before defining it is allowed. Every used identifier has to be defined exactly once.

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

3.1.1 Base Types

The GrGen.NET model language has built-in types for nodes and edges. All nodes have the attribute-less, built-in type `Node` as their ancestor. All edges have the abstract (see Section 3.2), attribute-less, built-in type `AEdge` as their ancestor. The `AEdge` has two non-abstract built-in children: `UEdge` as base type for undirected edges and `Edge` as base type for directed edges. The direction for `AEdge` and its anchestors that do not inherit from `Edge` or `UEdge` is undefined or *arbitrary*. Because there is the "magic of direction" linked to the edge base types, its recommended to use the keywords `directed`, `undirected`, and `arbitrary` in order to specify inheritance (see Section 3.2). As soon as you decided for directed or undirected edge classes within your type hierarchie, you are not able to let anchestor classes inherited from a contradicting base type, of course. That is, no edge may be directed *and* undirected. This is an exception of the concept multi-inheritance. Figure 3.1 shows the edge type hierarchy.

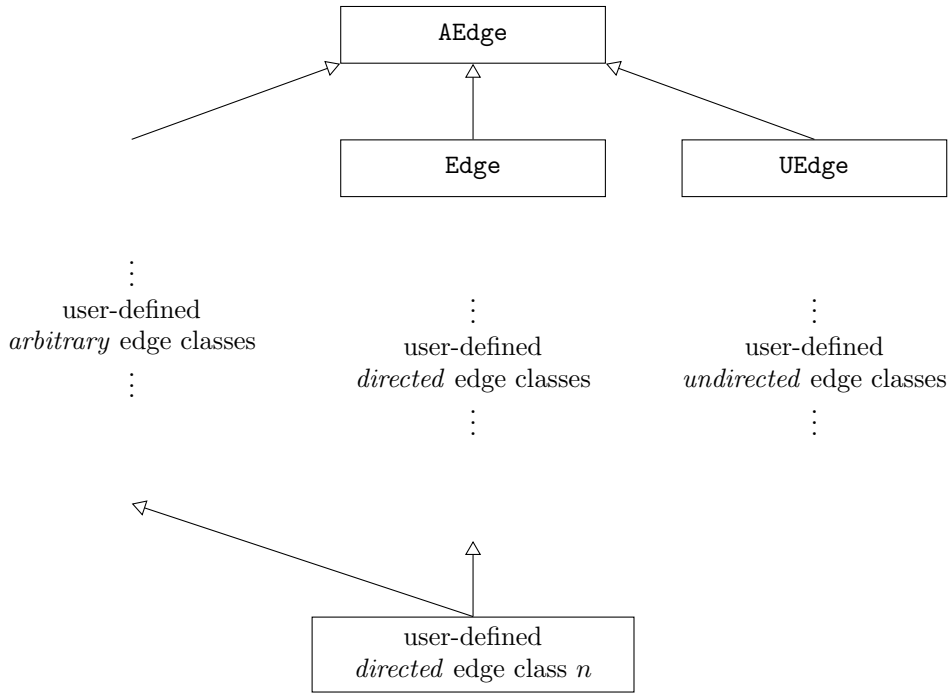
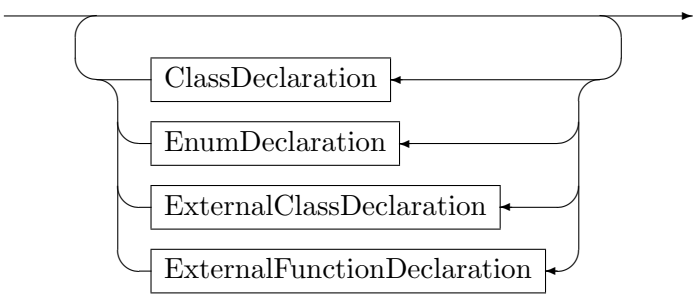


Figure 3.1: Type Hierarchy of GrGen.NET Edges

3.2 Type Declarations

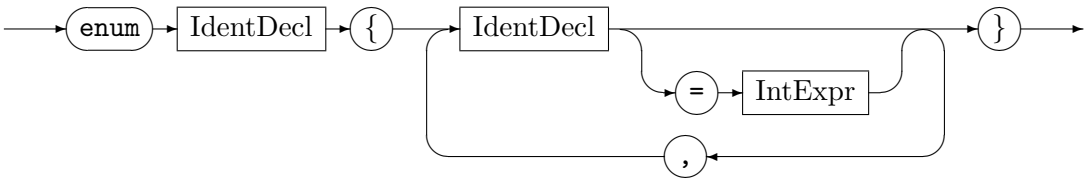
GraphModel



The graph model consists of zero or multiple type declarations. Whereas *ClassDeclaration* defines a node type or an edge type, *EnumDeclaration* defines an enum type to be used as a type for a attributes of nodes or edges. Like all identifier definitions, types do not need to be declared before they are used. The *ExternalClassDeclaration* registers an external class, including its subtype hierarchy, exluding any attributes with GrGen.NET which can subsequently be used in attribute computations with external function calls.

3.2.1 Enumeration Types

EnumDeclaration



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

NOTE (7)

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

NOTE (8)

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

EXAMPLE (2)

```
1 enum Color {RED, GREEN, BLUE}
2 enum Resident {VILLAGE = 500, TOWN = 5000, CITY = 50000}
3 enum AsInC {A = 2, B, C = 1, D, e = (int)Resident::VILLAGE + C}
```

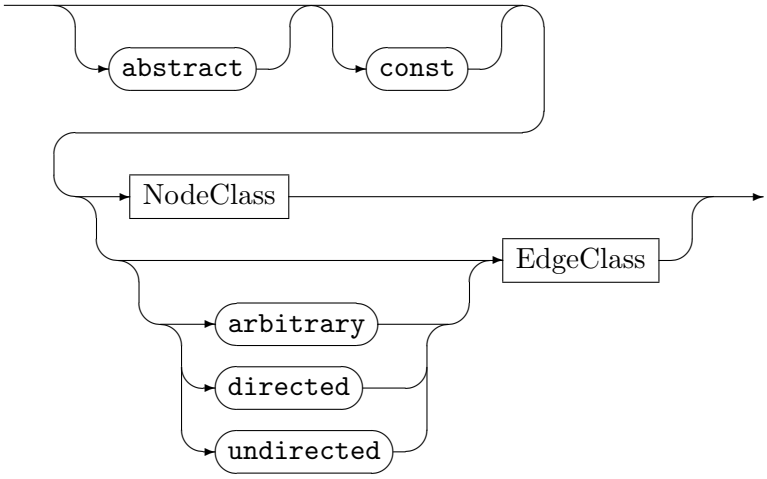
Consider, e.g., the declaration of the enum item `e`: By implicit casts of `Resident::VILLAGE` and `C` to `int` we get the `int` value 501, which is assigned to `E`. Moreover, the semantics is as in C [SAI+90]. So, the following holds: `RED` = 0, `GREEN` = 1, `BLUE` = 2, `A` = 2, `B` = 3, `C` = 1, `D` = 2, and `E` = 501.

NOTE (9)

The C-like semantics of enum item declarations implies, that multiple items of one enum type can be associated with the same same `int` value. Moreover, it implies, that an enum item must not be used *before* its definition. This also holds for items of other enum types, meaning that the items of another enum type can only be used in the definition of an enum item, when the other enum type is defined *before* the enum type currently defined.

3.2.2 Node and Edge Types

ClassDeclaration



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

EXAMPLE (3)

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

```
1 abstract node class City {
2   Size:int;
3 }
4 abstract node class AbandonedCity extends City;
5 node class GhostTown extends AbandonedCity;
```

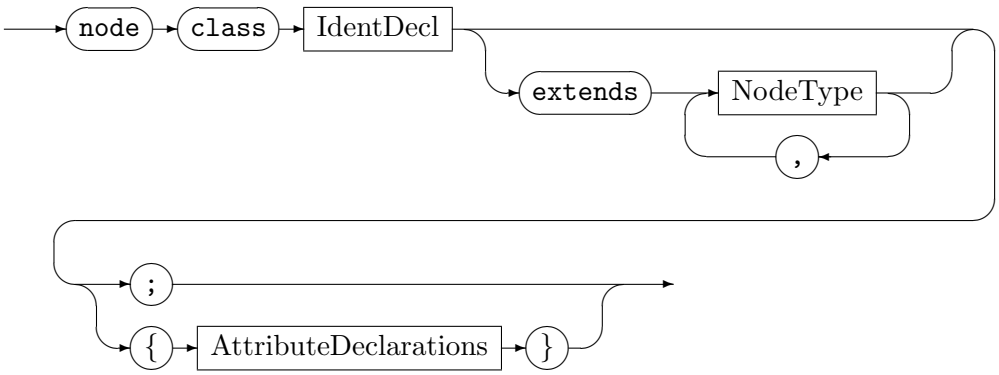
You will be able to create nodes of type `GhostTown`, but not of type `City` or `AbandonedCity`. However, nodes of type `GhostTown` are also of type `AbandonedCity` as well as of type `City` and they have the attribute `Size`, hence.

The keyword `const` indicates that rules may not write to attributes (see also Section 4.4, `eval`). However, such attributes are still writable by LIBGR and GRSHELL directly. This property applies to attributes defined in the current class, only. It does not apply to inherited attributes. The `const` property will not be inherited by subclasses, either. If you want a subclass to have the `const` property, you have to set the `const` modifier explicitly.

The keywords `arbitrary`, `directed`, and `undirected` specify the direction "attribute" of an edge class and thus its inheritance. An `arbitrary` edge inherits from `AEdge`, it is always abstract and neither directed nor undirected. A `directed` edge inherits from `Edge`.

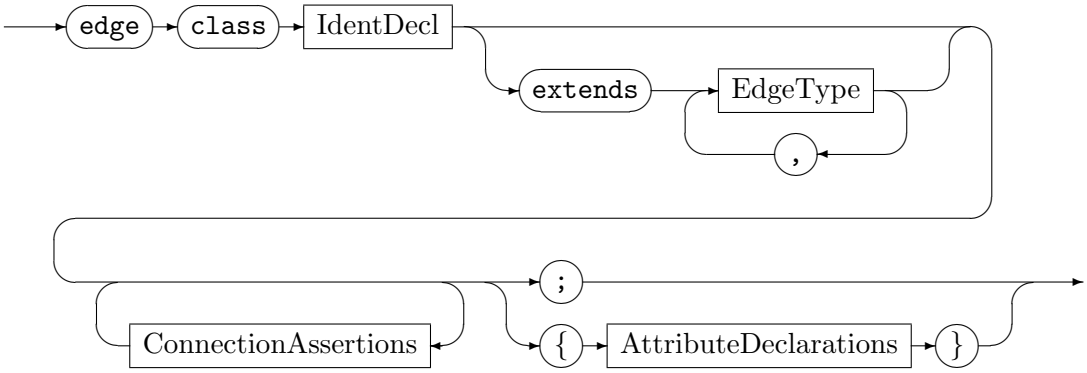
An `undirected` edge inherits from `UEdge`. If you do not specify any of those keywords, a `directed` edge is choosen by default. See also Section 3.1.1

NodeClass



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

EdgeClass

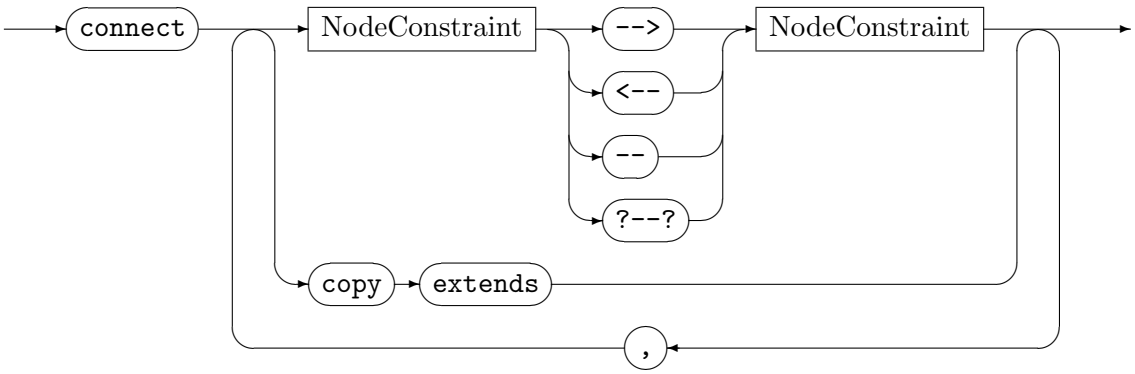


Defines a new edge type. Edge types can inherit from other edge types defined within the same file. If the `extends` clause is omitted, *EdgeType* will inherit from the built-in type `Edge`. Optionally edges can possess attributes. A *connection assertion* specifies that certain edge types should only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

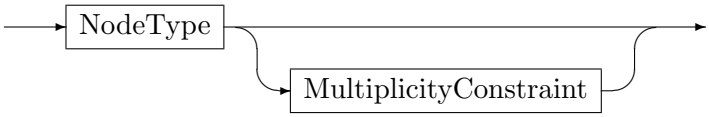
NOTE (10)

It is not forbidden to create graphs that are invalid according to connection assertions. GrGen.NET just enables you to check, whether a graph is valid or not. See also Section 17.2.2, `validate`.

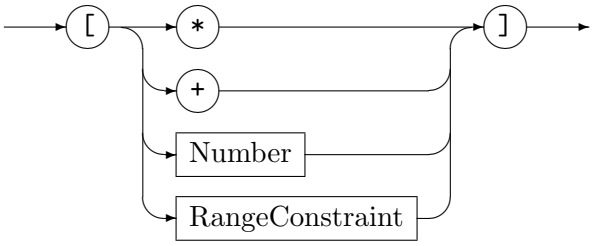
ConnectionAssertions



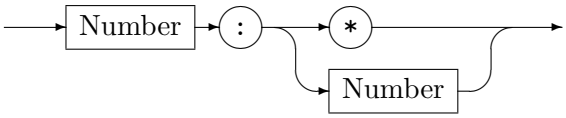
NodeConstraint



MultiplicityConstraint



RangeConstraint



A *connection assertion* is denoted as a pair of node types, optionally with their multiplicities. It allows you to specify the node types an edge may connect and the multiplicities with which such edges appear on the nodes. It is best understood as a simple pattern of the form(cf. 4.1.1) `:SourceNodeType -:EdgeType-> :TargetNodeType`, of which every occurrence in the graph is searched. In contrast to a real such pattern and the node types only edges of exactly the given edge type are taken into account. Per node of `SourceNodeType` (or a subtype) it is counted how often it was covered by a match of the pattern starting at it, and per node of `TargetNodeType` (or a subtype) it is counted how often it was covered by a match of the pattern ending at it. The numbers must be in the range specified at the `SourceNodeType` and the `TargetNodeType` for the connection assertion to be fulfilled. Giving no multiplicity constraint is equivalent to `[*]`, i.e. $[0, \infty[$, i.e. unconstrained. Please take care of non-disjoint source/target types/subtypes in the case of undirected and especially arbitrary edges. In the case of multiple connection assertions all are checked and errors for each one reported; for strict validation to succeed at least one of them must match. It might happen that none of the connection assertions of an `EdgeType` are matching an edge of this type in the graph. This is accepted in the case of normal validation (throwing connection assertions without multiplicities effectively back to nops); but as you normally want to see *only* the specified connections occuring in the graph, there is the additional mode of strict validation: if an edge is not covered by a single matching connection, validation fails. Furtheron there is the strict-only-specified mode, which only does strict validation of the edges for which connection assertions are given. See Section 17.2.2, `validate`, for an example.

The arrow syntax is based on the GrGen.NET graphlet specification (see Section 4.1.1). The different kinds of arrows distinguish between directed, undirected, and arbitrary edges. The `-->` arrow means a directed edge aiming towards a node of the target node type (or one of its subtypes). The `A<--B` connection assertion is equivalent to the `B-->A` connection assertion. The `--` arrow is used for undirected edges. The `?--?` arrow means an arbitrary edge, i.e. directed as well as undirected possible (fixed by the concrete type inheriting from it); in case of an directed edge the connection pattern gets matched in both directions. *Number* is an `int` constant as defined in Chapter 5. Table 3.1 describes the multiplicity definitions.

In order to apply the connection assertions of the supertypes to an `EdgeType`, you may use the keywords `copy extends`. The `copy extends` assertion "imports" the connection assertions of the *direct* ancestors of the declaring edge. This is a purely syntactical simplification,

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

Table 3.1: GrGen.NET node constraint multiplicities

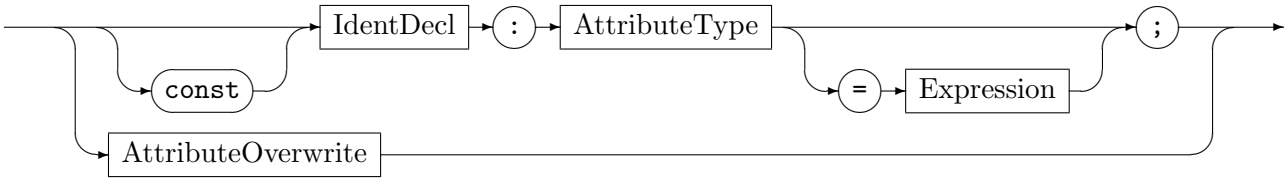
i. e. the effect of using `copy extends` is the same as copying the connection assertions from the direct ancestors by hand.

NOTE (11)

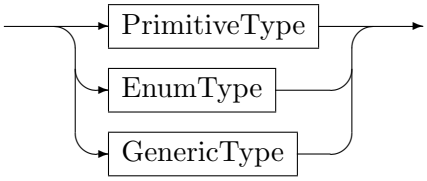
GrGen.NET supports multiple inheritance on nodes and edges – use it!
• Your nodes have something in common? Then factor it out into some base class. This works very well because of the support for multiple inheritance; you don't have to decide what is the primary hierarchy, forcing you to fall back to alternative patterns in the situations you need a different classification. Fine grain type hierarchies not only allow for concise matching patterns and rules, but deliver good matching performance, too (the search space is reduced early).
• Your edges have something in common? Then just do the same, edges are first class citizens.

3.2.3 Attributes and Attribute Types

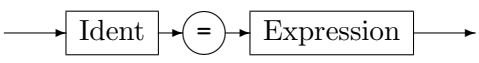
AttributeDeclaration



AttributeType



AttributeOverwrite



Defines a node or edge attribute. Possible types are enumeration types (`enum`) and primitive types or generic types. See Section 5.1 for a list of built-in primitive and Section 13.1 for a list of built-in generic types. Optionally attributes may be initialized with a *constant* expression. The expression has to be of a compatible type of the declared attribute. See Chapter 5 for the GrGen.NET types and expressions reference. The *AttributeOverwrite* clause lets you overwrite initialization values for attributes of super classes. The initialization values are evaluated in the order as they appear in the rule set file.

NOTE (12)

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

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

NOTE (13)

If you need to mark nodes in the graph, but only very few of them at the same time, think of reflexive edges of a special type. If the marking is sparse, they are the most efficient solution, before attributes, even before visited flags.

CHAPTER 4

RULE SET LANGUAGE

The rule set language forms the core of GrGen.NET. Rule files refer to zero [1] or more graph models and specify a set of rewrite rules. The rule language covers the pattern specification and the rewrite specification in the form of the `replace` or the `modify` block. Attributes of graph elements can be re-evaluated during an application of a rule. The following rewrite rule mentioned in Geiß et al. [GBG+06] gives a rough picture of the language:

EXAMPLE (4)

```
using SomeModel;

rule SomeRule {
  n1:NodeTypeA;
  n2:NodeTypeA;
  hom(n1, n2);
  n1 --> n2;
  n3:NodeTypeB;
  negative {
    n3 -e1:EdgeTypeA-> n1;
    if {n3.a1 == 42*n2.a1;}
  }
  negative {
    n4:Node\(NodeTypeB);
    n3 -e1:EdgeTypeB-> n4;
    if {typeof(e1) >= EdgeTypeA;}
  }
  replace {
    n5:NodeTypeC<n1>;
    n3 -e1:EdgeTypeB-> n5;
    eval {n5.a3 = n3.a1*n1.a2;}
  }
}
```

In this chapter we explain the basics of Example 4 (`SomeRule`), the more advanced constructs are illustrated in chapter 9. The nested `negative`s which specify a pattern which must not be available in the host graph are described in Chapter 6

4.1 Building Blocks

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

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

Ident, *IdentDecl*
A non-empty character sequence of arbitrary length consisting of letters, digits, or underscores. The first character must not be a digit. *Ident* may be an identifier defined in a graph model (see Section 3.1). *Ident* and *IdentDecl* differ in their role: While *IdentDecl* is a *defining* occurrence of an identifier, *Ident* is a *using* occurrence. An *IdentDecl* non-terminal can be annotated. See Section 21.7 for annotations of declarations.

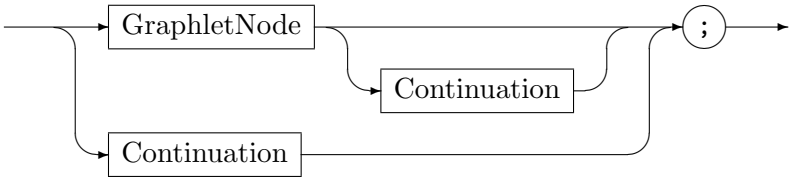
NOTE (14)

As in the GrGen.NET model language (see note 6) every declaration is also a definition. Using an identifier before defining it is allowed. Every used identifier has to be defined exactly once.

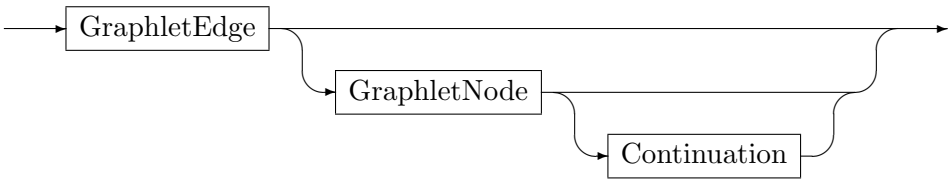
ModelIdent, *TypeIdent*, *NodeType*, *EdgeType*
These are (semantic) specializations of *Ident*. *TypeIdent* matches every type identifier, i.e. a node type, an edge type, an enumeration type or a primitive type. All the type identifiers are actually type *expressions*. See Section 5.7 for the use of type expressions.

4.1.1 Graphlets

Graphlet



Continuation



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

All the graph elements of a graphlet have *names*. The name is either user-assigned or a unique internal, non-accessible name. In the second case the graph element is called *anonymous*. For illustration purposes we use a `$<number>` notation to denote anonymous graph elements in this document. For example the graphlet `n1 --> n2` contains an anonymous edge; thus can be understood as `n1 -$1:Edge-> n2`. Names must not be redefined; once defined, a name is *bound* to a graph element. We use the term "binding of names" because a name not only denotes a graph element of a graphlet but also denotes the mapping of the abstract graph element of a graphlet to a concrete graph element of a host graph. So graph elements of different names are pair wise distinct except for homomorphically matched graph elements (see Section 4.3). For instance `v:NodeType1 -e:EdgeType-> w:NodeType2` selects some node of type `NodeType1` that is connected to a node of type `NodeType2` by an edge

of type `EdgeType` and binds the names `v`, `w`, and `e`. If `v` and `w` are not explicitly marked as homomorphic, the graph elements they bind to are distinct. Binding of names allows for splitting a single graphlet into multiple graphlets as well as defining cyclic structures.

EXAMPLE (5)

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

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

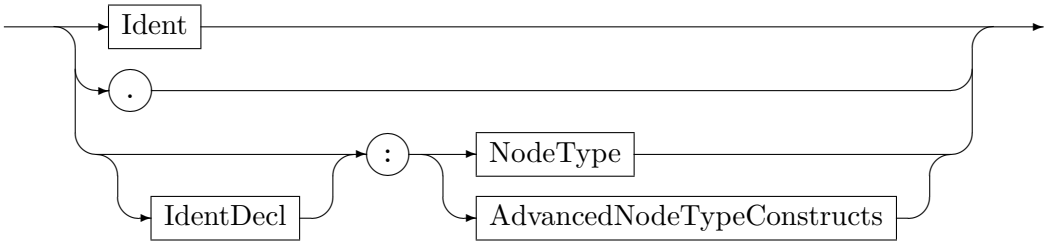
is equivalent to

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

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

The visibility of names is determined by scopes. Scopes can be nested. Names of surrounding scopes are visible in inner scopes. Usually a scope is defined by { and }. In Example 4, lines 13 to 17, the negative condition uses `n3` from the surrounding scope and defines `n4` and `e1`. We may safely reuse the variable name `e1` in the replace part.

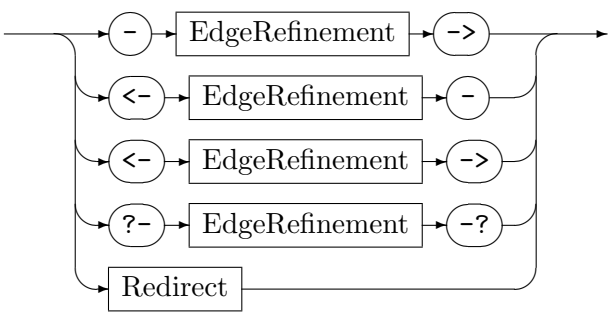
GraphletNode



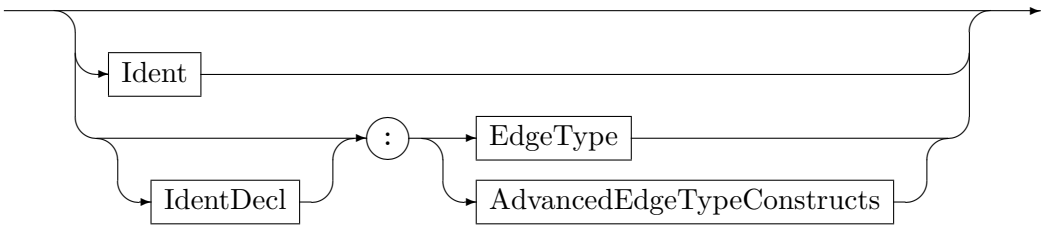
Specifies a node of type *NodeType*; the alternative advanced node constructs are explained in chapter 9. The `.` is an anonymous node of the base type `Node`; remember that every node type has `Node` as super type.

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

GraphletEdge



EdgeRefinement



A *GraphletEdge* specifies an edge. Anonymous edges are specified by an empty *EdgeRefinement* clause, i. e. `-->`, `<--`, `<-->`, `--`, `?--?` or `-:T->`, `<-:T-`, ... for an edge type `T`, respectively. A non-empty *EdgeRefinement* clause allows for detailed edge type specification. The alternative advanced edge constructs as well as the *Redirect* clause are explained in chapter 9.

The different kind of arrow tips distinguish between directed, undirected, and arbitrary edges (see also Section 3.1.1). The arrows `-->` and `<--` are used for directed edges with a defined source and target. The arrow `--` is used for undirected edges. The pattern part allows for further arrow tips, namely `?--?` for arbitrary edges and `<-->` for directed edges with undefined direction. Note that `<-->` is *not* equivalent to the `--> ; <-- ;` statements. In order to produce a match for the arrow `<-->`, it is sufficient that one of the statements `-->`, `<--` matches. If an edge type is specified (through the *EdgeRefinement* clause), this type has to correspond to the arrow tips, of course.

Graphlet	**Meaning**
`-e:EdgeType-> ;`	The name `e` is bound to an edge of type `EdgeType` or one of its subtypes.
`-:EdgeType-> ;`	`-$1:EdgeType-> ;`
`--> ;`	`-$1:Edge-> ;`
`<--> ;`	`-$1:Edge-> ;` or `<-$1:Edge- ;`
`-- ;`	`-$1:UEdge-> ;`
`?--? ;`	`-$1:AEdge-> ;`
`-e-> ;`	The edge, `e` is bound to.

As the above table shows, edges can be defined and used separately, i.e. without their incident nodes. Accidentally "redirecting" an edge is prevented by compiler checks (you must explicitely use the *Redirect* clause to achieve this): The graphlets

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

are illegal, because the edge `e` would have two destinations: an anonymous node and `y`. However, the graphlets

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

are allowed, but the first graphlet `-e->` is superfluous. In particular this graphlet does not identify or create any "copies", neither if the graphlet occurs in the pattern part nor if it occurs in the replace part.

EXAMPLE (6)

Some attempts to specify a loop edge:

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

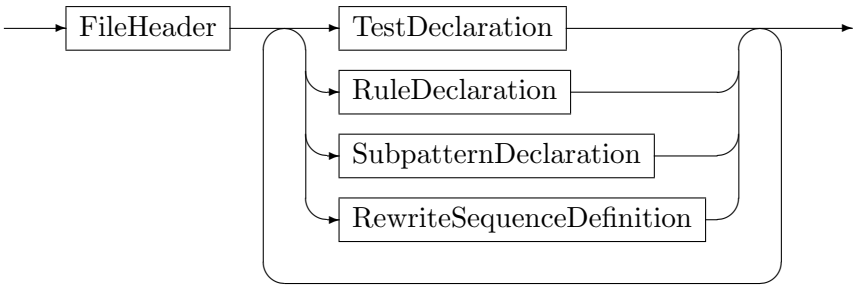
NOTE (15)

Although both, the pattern part and the replace/modify part use graphlets, there are subtle differences between them. Most of the differences can be seen in chapter 9 where the advanced constructs are explained .

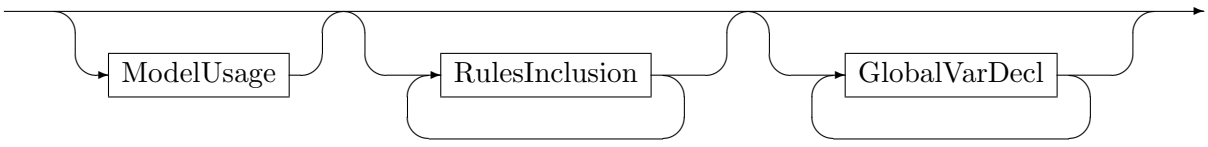
4.2 Rules and Tests

The structure of a rule set file is as follows:

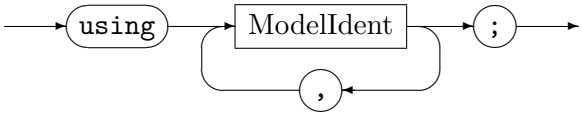
RuleSet



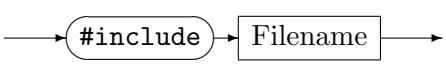
FileHeader



ModelUsage

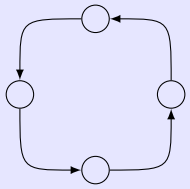


RulesInclusion

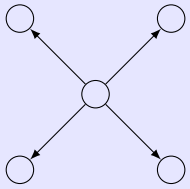


EXAMPLE (7)

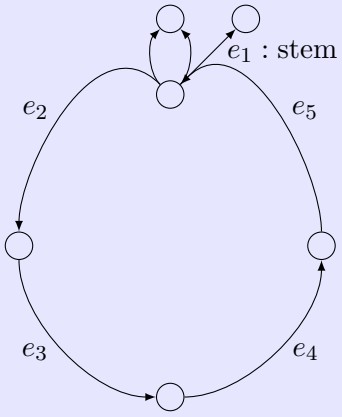
Some graphlets:



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

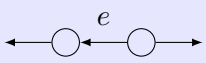


```
. <-- x:Node --> .;
. <-- x --> .;
```




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

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

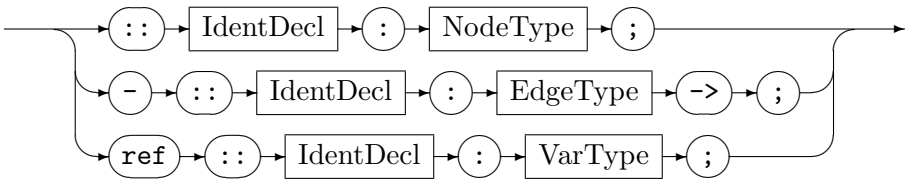



```
-e:Edge->
<-- . <-e- . --> ;
```

And some illegal graphlets:

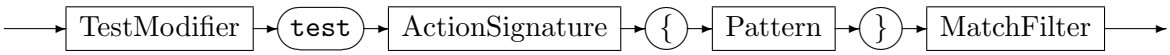
`. -e:Edge-> .; . -e-> .;`	Would affect redirecting of edge `e`.
`x -e:T-> y; x -e-> x;`	Would affect redirecting of edge `e`.
`<-- --> ;`	There must be at least a node between the edges.

GlobalVarDecl

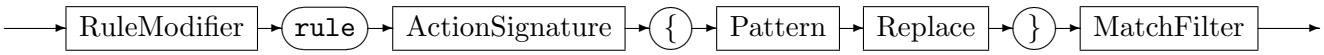


A rule set consists of the underlying graph models and several rewrite rules and tests (subpatterns will be introduced in 7.1, rewrite sequence definitions in 15.1). (As a hint regarding the syntax diagrams: please note that the bottom rail in the *RuleSet* diagram departs before the end and joins in after the *FileHeader*, i.e. it denotes a looping back edge (a fast forward edge would have split and join points at the same positions but of the opposite direction)). Additionally you may `include` further rule set files (without `using` directives, we prefer to suffix them with `.gri` in this case). Furthermore, you may declare graph global variables; this is a pure declaration that they will exist with the given type during execution. It renders them accessible in the rules (esp. the attribute condition and attribute evaluation), but you must define and assign them before rule execution outside of the rules. See 8 for more on this. In case of multiple graph models, GrGen.NET uses the union of these models. In this case beware of conflicting declarations. There is (unfortunately) no built-in conflict resolution mechanism for models like packages or namespaces. If necessary you must use prefixes as you might do in C.

TestDeclaration



RuleDeclaration



Declares a single rewrite rule such as `SomeRule`. It consists of a pattern part (see Section 4.3) in conjunction with its rewrite/modify part (see Section 4.4). A *test* has no rewrite specification. It's intended to check whether (and maybe how many times) a pattern occurs (see example 8). For an explanation of the available modifiers see Section 9.1, for an explanation of the external match filters see Section 21.3.

EXAMPLE (8)

We define a test `SomeCond`

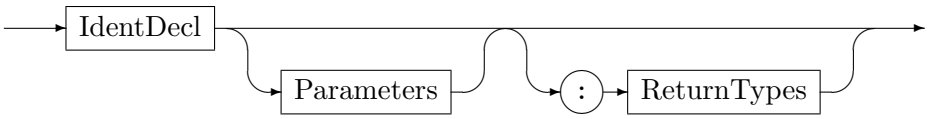
```
1 test SomeCond {
2   n:SeldomNodeType;
3 }
```

and execute in GrShell:

```
1   exec SomeCond & SomeRule
```

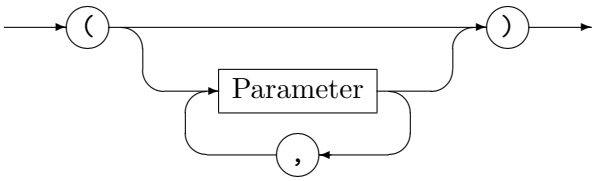
SomeRule will only be executed, if a node of type `SeldomNodeType` exists. For graph rewrite sequences in GrShell see Section 17.2.8.

ActionSignature

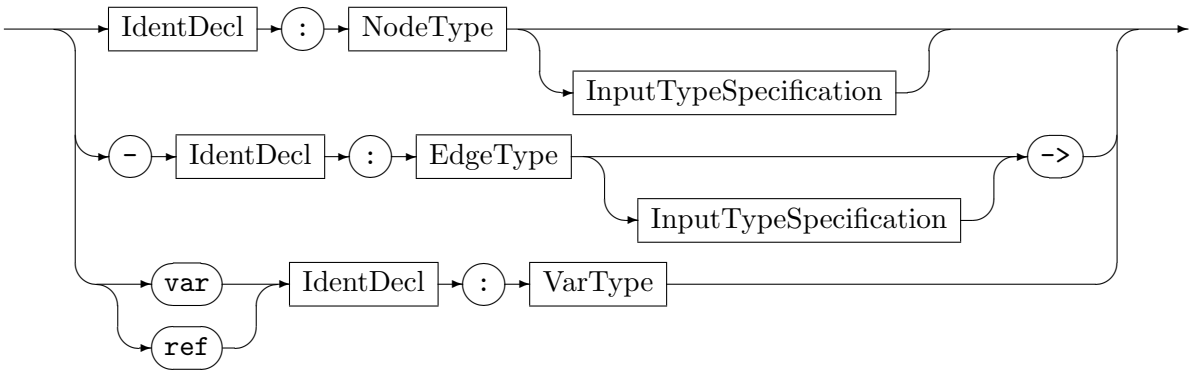


The signature sets the name of a rewrite rule to *IdentDecl* and optionally names and types of formal parameters as well as a list of return types. Parameters and return types provide users with the ability to exchange graph elements between rules, similar to parameters of procedural languages. This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

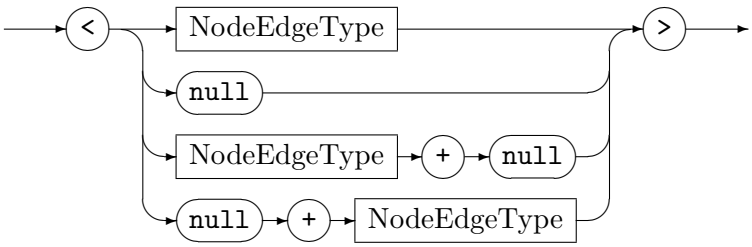
Parameters



Parameter



InputTypeSpecification



Within a rule, graph element parameters are treated as graph elements of the pattern - just predefined. But in contrast to pervious versions it is the task of the user to ensure the elements handed in satisfy the interface, i.e. parameters must not be null and must be of the type specified or a subtype of the type specified. If you need more flexibility and want to call the rule with parameters not fullfilling the interface you can append an input type specification to the relevant parameters, which consists of the type to accept at the action interface, or null, or both, enclosed in left and right angles. If the input type specification type is given, but the more specific pattern element type is not satisfied, matching simply fails. If null is declared in the input type specification and given at runtime, the element is searched in the host graph. Don't use null parameters unless you need them, because every null parameter doubles the number of matcher routines which get generated. Non-graph element parameters must be prefixed by the `var` or `ref`-keyword; VarType is one of the attribute types supported by GrGen.NET (cf. 5.1 and 13.1). The primitive types require the `var` prefix and are handed in by-value; the generic types require the `ref` prefix and are handed in by-reference. Please note that the effect of assigning to a var/ref parameter in `eval` (see 4.4) is undefined (concerning the parameters as well as the argument); they are only available for reading, the by-ref parameters additionally for set/map-addition and removal (cf. 10)

EXAMPLE (9)

The test `t` that checks whether node `n1` is adjacent to `n2` (connected by an undirected edge or incoming directed edge or outgoing directed edge)

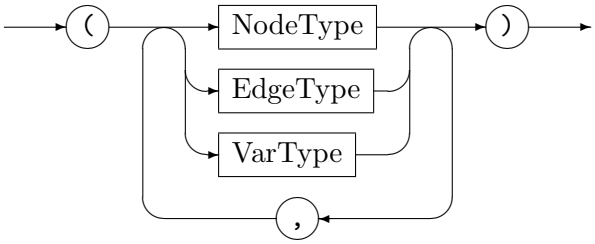
```
1  test t(n1:Node<null>, n2:Node<null>) {
2    n1 ?--? n2;
3  }
```

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

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

So if both parameters are not defined, `t4` is chosen, which succeeds as soon as there are two distinct nodes in the graph connected by some edge.

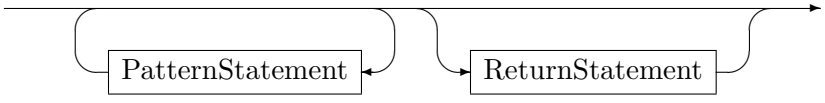
ReturnTypes



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

4.3 Pattern Part

Pattern



A pattern consists of zero or more pattern statements and, (only) in case of a test, an optional return statement. All the pattern statements must be fulfilled by a subgraph of the host graph in order to form a match. An empty pattern always produces exactly one (empty) match. This is caused by the uniqueness of the total and totally undefined function. For an explanation of the pattern modifiers `dpo`, `identification`, `dangling`, `induced`, and `exact` see Section 9.1.

EXAMPLE (10)

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

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

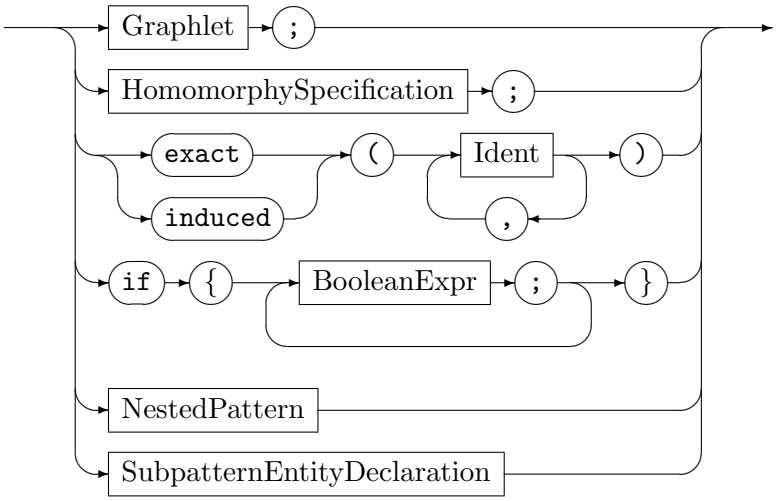
We do not define `varnode` within the pattern part because this is already covered by the parameter specification itself.

Names defined for graph elements may be used by other pattern statements as well as by replace/modify statements. Like all identifier definitions, such names may be used before their declaration. See Section 4.1 for a detailed explanation of names and graphlets.

NOTE (16)

The application of a rule is not deterministic (remember the example of the introduction in Section 1.6); in particular there may be more than one subgraph that matches the pattern. Whereas the GrShell selects one of them arbitrarily (without further abilities to control the selection), the underlying libGr provides a mechanism to deal with such ambiguities. Also notice that graph rewrite *sequences* introduce a further variant of non-determinism on rule application level: The `$<op>` flag marks the operator `<op>` as commutative, i.e. the execution order of its operands (rules) is non-deterministic. See Chapter 8 for further information on graph rewrite sequences.

PatternStatement



The semantics of the various pattern statements are given below:

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

Isomorphic/Homomorphic Matching
: See Subsection 4.3.1 for a discussion of this.

Attribute Conditions
: The Java-like attribute conditions (keyword `if`) in the pattern part allow for further restriction of the applicability of a rule. The pattern can only match if the *BooleanExpression* (see chapter 5) is evaluated to `true`.

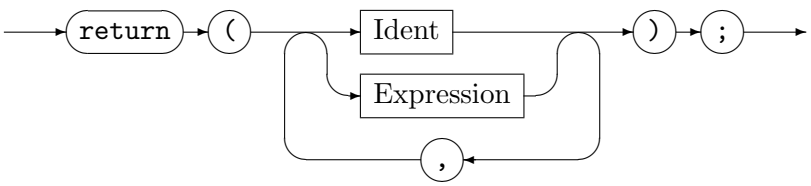
Pattern Modifiers
: Additionally to modifiers that apply to a pattern as a whole, you may also specify pattern modifiers for a specific set of nodes. Accordingly the list of identifiers for a pattern modifier must not contain any edge identifier. See Section 9.1 for an explanation of the `exact` and `induced` modifiers.

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

SubpatternEntityDeclaration
: will be explained in 7.1.

Keep in mind that using type constraints or the `typeof` operator might be helpful. See Section 5.7 for further information.

ReturnStatement



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

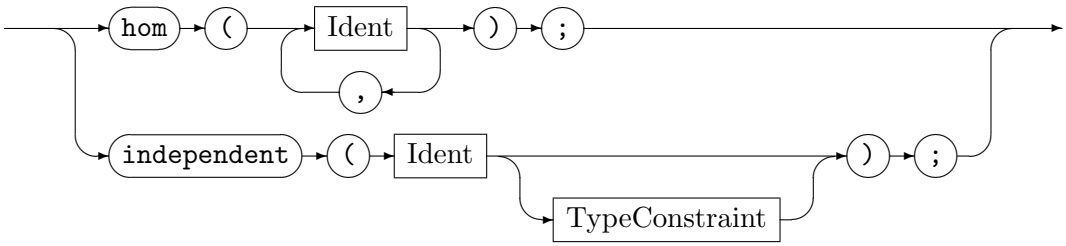
NOTE (17)

If you are using a graph at the API level without shell-provided names accessible by the `nameof`-operator, you may want to number the graph elements for dumping like this:

```
1  rule numberNode(var id:int) : (int)
2  {
3    n:NodeWithIntId;
4    if { n.id == 0; }
5
6    modify {
7      eval {
8        n.id = id;
9      }
10     return (id + 1);
11   }
12 }
```

4.3.1 Isomorphic and Homomorphic Matching

HomomorphySpecification



The matching of pattern elements to host graph elements in GrGen.NET is isomorphic (injective) by default, i.e. two pattern elements can *not* be bound to the same host graph element. The `hom` operator specifies the nodes or edges that may be matched homomorphically. In contrast to the default isomorphic matching, the specified graph elements *may* be mapped to the same graph element in the host graph. Note that the graph elements must have a common subtype. Several homomorphically matched graph elements will be mapped to a graph element of a common subtype. In Example 4 nodes `n1` and `n2` may be the same node. This is possible because they are of the same type (`NodeTypeA`). Inside a NAC/PAC the `hom` operator may only operate on graph elements that are either defined or used in the NAC/PAC (cf. 6.1/6.2). Nested `negative`/`independent` blocks inherit the `hom` declarations of their nesting pattern. In contrast to previous versions of GrGen `hom` delarations are non-transitive, i.e `hom(a,b)` and `hom(b,c)` don't cause `hom(a,c)` unless specified.

The `independent` operator specifies the node or edge given within the parentheses to be homomorphic to all the other pattern elements. With the constraint clause following, exceptions can be given defining the pattern elements it must be distinct to. Thus we got a specification mode requesting homomorphic matching with additional isomorphy exceptions in contrast to the default mode, requesting isomorphic matching with additional homomorphy exception. It is recommended to *not* use the `independent` operator, it it potentially dangerous allowing to carry out conflicting rewrites, with an element `a` to be deleted, element `b` to be kept, and element `c` to be retyped, all mapping to the same graph element (you will experience funny effects and/or crashes in this case; the `hom` operator offers some static checks against this). The operator is available as a last resort for some situations in matching complex structures with iterated and subpatterns, in which it is unfeasible or not possible to explicitely give elements the pattern element may be homomorphic to, because they were

matched in a pattern at an arbitrary distance in the derivation path which only dynamically called the pattern of interest, i.e. they can't be referenced by name, cf. 7.4.

4.4 Replace/Modify Part

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

4.4.1 Implicit Definition of the Preservation Morphism r

In theory the transformation specification is done by defining the preservation morphism r. In GrGen.NET the preservation morphism r is defined implicitly by using names both in

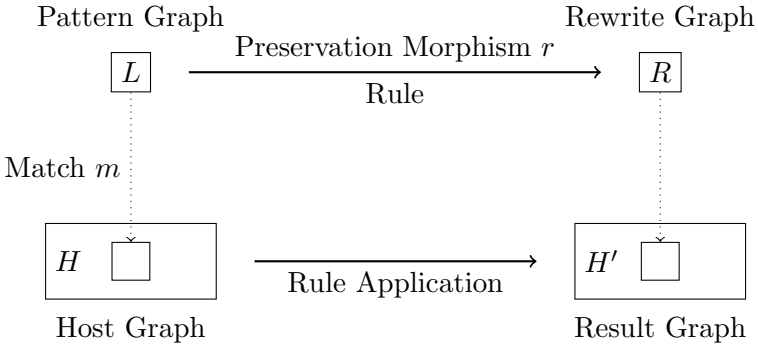


Figure 4.1: Process of Graph Transformation

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

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

Table 4.1: Definition of the preservation morphism r

4.4.2 Specification Modes for Graph Transformation

For the task of rewriting, GrGen.NET provides two different modes: A *replace mode* and a *modify mode*.

Replace mode

The semantics of this mode is to delete every graph element of the pattern that is not used (occur) in the replace part, keep every graph element that is used, and create every additionally defined graph elements. "Using" means that the name of a graph element occurs in a replace graphlet. Attribute calculations are no using occurrences. In Example 10 the nodes `varnode` and `n3` will be kept. The node `n1` is replaced by the node `n5` preserving `n1`'s edges. The anonymous edge instance between `n1` and `n2` only occurs in the pattern and therefore gets deleted.
See Section 4.4.1 for a detailed explanation of the transformation semantics.

Modify mode

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

NOTE (18)

In general modify mode should be prefered as it allows to read the rewrite part as a diff of the changes to be made to the pattern part, whereas replace mode requires comparing the LHS and RHS pattern while reading to find out about the changes. Only if most of the pattern is to be deleted replace mode is advantageous, pinpointing what should stay. (Furthermore it might be simpler to generate code for, just dumping both patterns.)

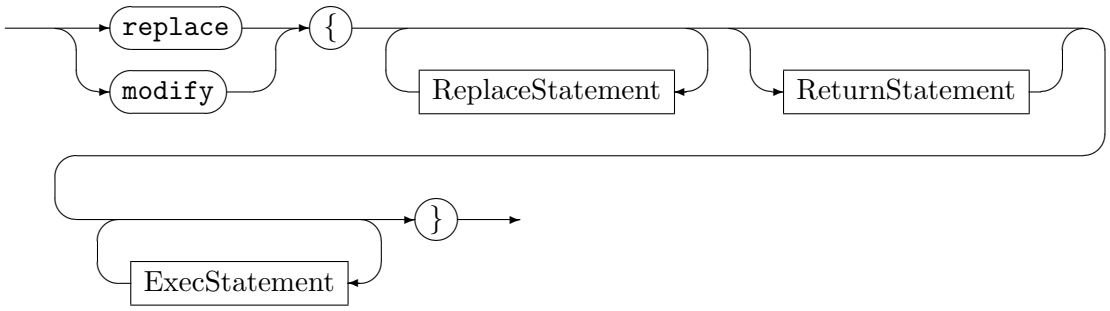
EXAMPLE (11)

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

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

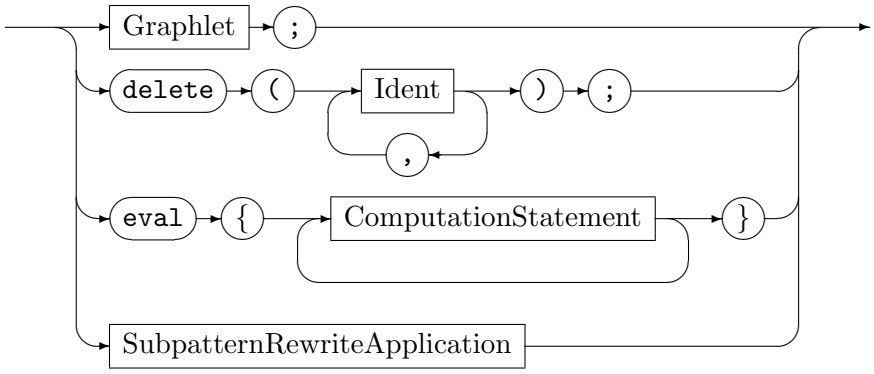
4.4.3 Syntax

Replace



Selects whether the replace mode or the modify mode is used. Several replace statements describe the transformation from the pattern subgraph to the destination subgraph. The *ReturnStatement* was already introduced, for tests it can appear in the pattern part. Regarding rules it can only be given in the rewrite part. The *ExecStatement* will be introduced in chapter 10.

ReplaceStatement



The semantics of the various replace statements are given below:

Graphlet
: Analogous to a pattern graphlet; a specification of a connected subgraph. Its graph elements are either kept because they are elements of the pattern or added otherwise. Names defined in the pattern part must not be redefined in the replace graphlet. See Section 4.1 for a detailed explanation of graphlets.

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

Computation (Attribute Evaluation)
: If a rule is applied, then the attributes of matched and inserted graph elements will be recalculated according to the `eval` statements. Besides attribute evaluations, further computations may be executed and side effects applied, see Chapter 11 for more on this.

SubpatternRewriteApplication
: will be explained in 7.2.

Several evaluation parts are allowed within the replace part. Multiple evaluation statements will be internally concatenated, preserving their order. Evaluation takes place before any graph element specified to be deleted by the rule gets deleted and after all the new elements (according to the rule rewrite part) have been created. You may read (and write, although this is pointless) attributes of graph elements to be deleted.

EXAMPLE (12)

```
1  ...
2  modify {
3    ...
4    eval { y.i = 40; }
5    eval { y.j = 0;  }
6    x:IJNode;
7    y:IJNode;
8    delete(x);
9    eval {
10     x.i = 1;
11     y.j = x.i;
12     x.i = x.i + 1;
13     y.i = y.i + x.i;
14   }
15 }
```

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

CHAPTER 5

BASIC TYPES AND EXPRESSIONS

5.1 Built-In Types

Besides user-defined node types, edge types, and enumeration types (as introduced in Chapter 3), GrGen.NET supports the built-in primitive types in Table 5.1 (and built-in generic types, cf. 12). The exact type format is backend specific. The LGSPBackend maps the GrGen.NET primitive types to the corresponding C# primitive types.

`boolean`	Covers the values `true` and `false`
`byte, short, int, long`	A signed integer, with 8 bits, with 16 bits, with 32 bits, with 64 bits
`float, double`	A floating-point number, with single precision, with double precision
`string`	A character sequence of arbitrary length
`object`	Contains a .NET object

Table 5.1: GrGen.NET built-in primitive types

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

Table 5.2: GrGen.NET type casts

Table 5.2 lists GrGen.NET's implicit type casts and the allowed explicit type casts. Of course you are free to express an implicit type cast by an explicit type cast as well as "cast" a type to itself. The `int` is the default integer type, in the table it stands for all the integer types. The `double` is the default floating point type, in the table it stands for all the floating points types. The integer types are implicitely casted upwards from smaller to larger types (`byte < short < int < long`), whereas a downcast requires an explicit cast. The `byte` and `short` types are not used in computations, they are casted up to `int` (or `long` if required by the context.) The floating point types are implicitely casted upwards (`float < double`), and require an explicit cast downwards, too. As specified by the table, integer numbers are automatically casted to floating point numbers, and castable with an explicit cast vice versa. According to the table neither implicit nor explicit casts from `int` to any enum type are allowed. This is because the range of an enum type is very sparse in general. For the same reason implicit and explicit casts between enum types are also forbidden. Thus, enum values can only be assigned to attributes having the same enum type. A cast of an enum value to

a string value will return the declared name of the enum value. A cast of an object value to a string value will return "null" or it will call the `toString()` method of the .NET object. Everything is implicitely casted to `string` to enable concise text output without the need for boilerplate casting. Be careful with assignments of objects: GrGen.NET does not know your .NET type hierarchy and therefore it cannot check two objects for type compatibility. Objects of type object are not very useful for GrGen.NET processing and the im/exporters can't handle them, but they can be used on the API level.

EXAMPLE (13)

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

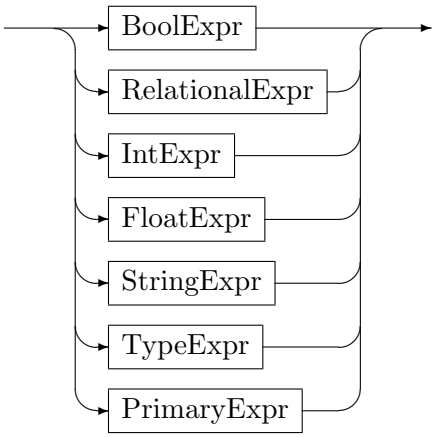
NOTE (19)

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

5.2 Expressions

GrGen.NET supports numerous operations on the entities of the types introduced above, which are organized into left associative expressions. In the following they will be explained with their semantics and relative priorities one type after another in the order of the rail diagram below.

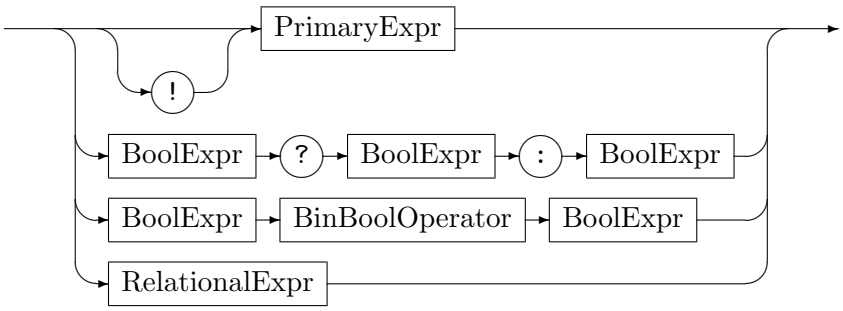
Expression



5.3 Boolean Expressions

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

BoolExpr



The unary `!` operator negates a Boolean. The binary *BinBoolOperator* is one of the operators in Table 5.3. The ternary `?` operator is a simple if-then-else: If the first *BoolExpr* is evaluated

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

Table 5.3: Binary Boolean operators, in ascending order of precedence

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

5.4 Relational Expressions

The relational expressions compare enitites of different kinds, mapping them to the type boolean. They are a middle between the boolean type of decisions and the other types. The bind stronger than the boolean expressions but weaker than all the other non-boolean expressions.

RelationalExpr

The *CompareOperator* is one of the following operators:

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

Their semantics are type dependent.

For arithmetic expressions on `int` and `float` or `double` types the semantics is given by Table 5.4 (by implicit casting they can also by used with all enum types).

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

Table 5.4: Compare operators on arithmetic expressions

`String` types, `boolean` types, and `object` types support only the `==` and the `!=` operators; for strings they denote whether the strings are the same or not, on boolean values they denote

equivalence and antivalence, and on object types they tell whether the references are the same, thus the objects identical.

For type expressions the semantics of compare operators are given by table 5.5, the rule to remember is: types grow larger with extension/refinement. An example is given in 5.7.

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

Table 5.5: Compare operators on type expressions

NOTE (20)

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

NOTE (21)

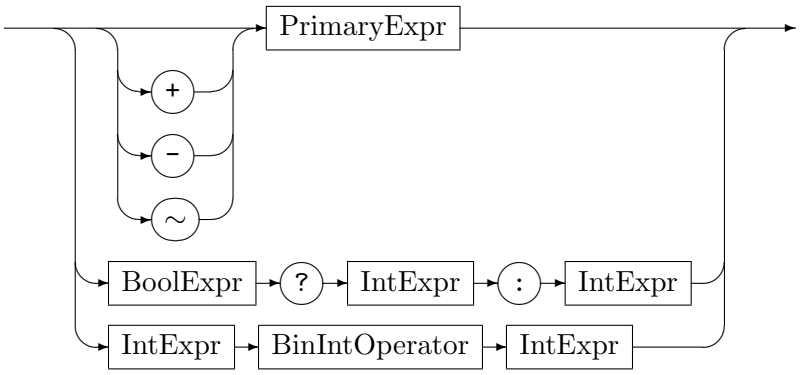
`Node` and `Edge` are the least specific, thus bottom elements $\bot$ of the type hierarchy, i.e. the following holds:

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

5.5 Arithmetic and Bitwise Expressions

The arithmetic and bitwise expressions combine integer and floating point values with the arithmetic operations usually available in programming languages and integer values with bitwise logical operations (interpreting integer values as bit-vectors).

IntExpr



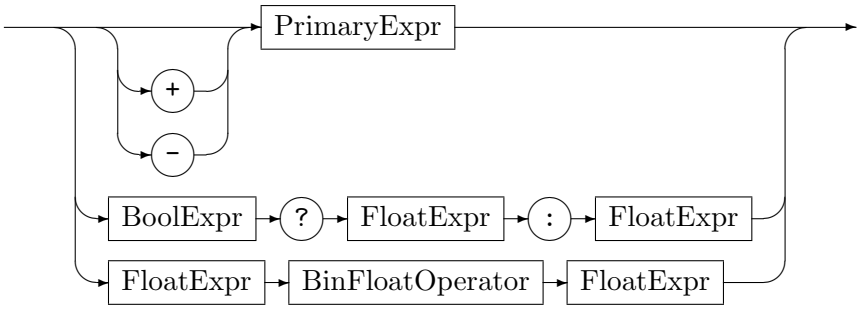
The $\sim$ operator is the bitwise complement. That means every bit of an integer value will be flipped. The `?` operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *IntExpr*, otherwise it returns the second *IntExpr*. The *BinIntOperator* is one of the operators in Table 5.6.

^ & |	Bitwise XOR, AND and OR
<< >> >>>	Bitwise shift left, bitwise shift right and bitwise shift right prepending zero bits (unsigned mode)
+ -	Addition and subtraction
* / %	Multiplication, integer division, and modulo

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

If one operand is `long` the operation is carried out with 64 Bits, otherwise the operation is carried out with 32 Bits, i.e. `int`-sized — even if all the operands are of type `byte` or `short`.

FloatExpr



The `?` operator is a simple if-then-else: If the *BoolExpr* is evaluated to `true`, the operator returns the first *FloatExpr*, otherwise it returns the second *FloatExpr*. The *BinFloatOperator* is one of the operators in Table 5.7.

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

Table 5.7: Binary float operators, in ascending order of precedence

NOTE (22)

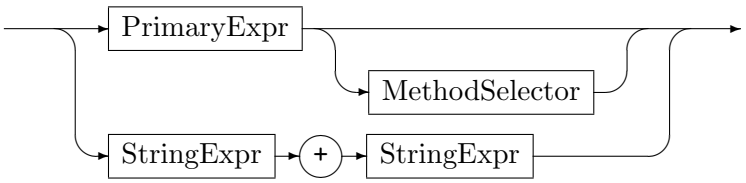
The `%` operator on float values works analogous to the integer modulo operator. For instance `4.5 % 2.3 == 2.2`.

If one operand is `double` the operation is carried out with 64 Bits, otherwise the operation is carried out with 32 Bits, i.e. `float`-sized.

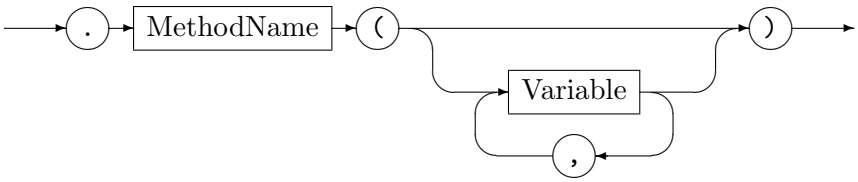
5.6 String Expressions

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

StringExpr



MethodSelector



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

.length()
: returns length of string, as `int`

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

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

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

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

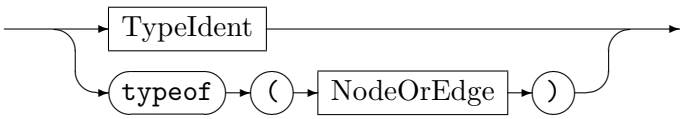
EXAMPLE (14)

For `n.str == "foo bar foo"` the operations yield

```
n.str.length()==11
n.str.indexOf("foo")==0
n.str.lastIndexOf("foo")==8
n.str.substring(4,3)=="bar"
n.str.replace(4,3,"foo")=="foo foo foo"
```

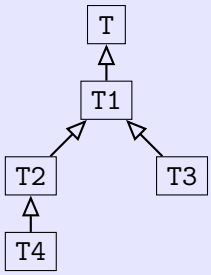
5.7 Type Expressions

TypeExpr



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

EXAMPLE (15)

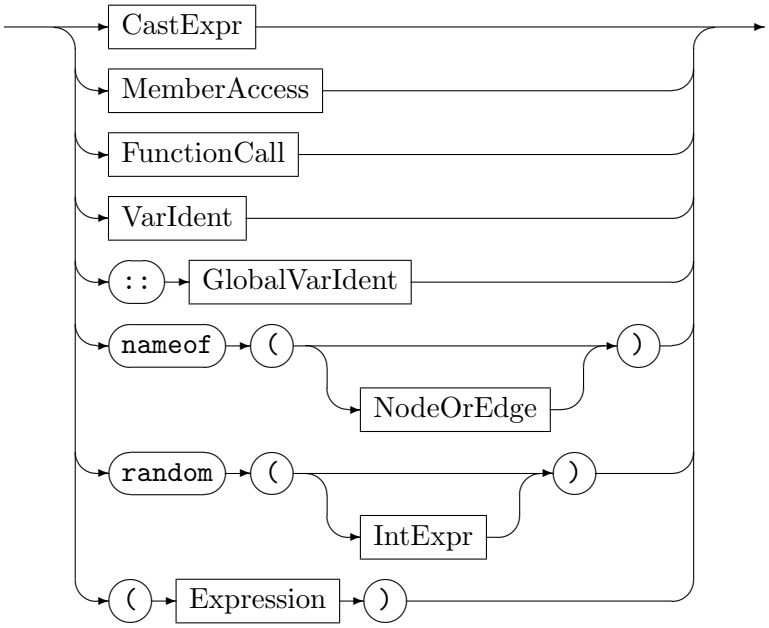


The expression `typeof(x)<=T2` applied to the type hierarchy on the left side yields `true` if `x` is a graph element of type `T` or `T1` or `T2`. The expression `typeof(x)>T2` only yields `true` for `x` being a graph element of type `T4`. The expression `T1<T3` always yields `true`.

5.8 Primary Expressions

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

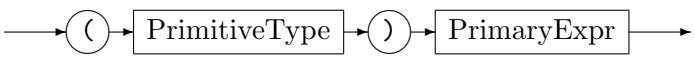
PrimaryExpr



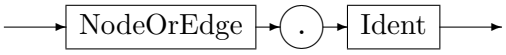
The `nameof` query returns the name (persistent name, see example 83) of the given graph element as `string` (graphs elements of pure `LGSPGraph`s bear no name, then the query fails, but normally you are using `LGSPNamedGraph`s). If no graph element is given, the name of the graph is returned.

The `random` function returns a double random value in between 0.0 and 1.0 if called without an argument, or, if an integer argument value is given, an integer random value in between 0 and the value given, excluding the value itself.

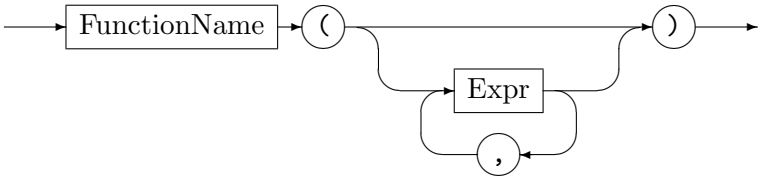
CastExpr



MemberAccess



FunctionCall



The cast expression returns the original value casted to the new prefixed type. The member access `n.a` returns the value of the attribute `a` of the graph element `n`. A function call employs an external (attribute evaluation) function (cf. 21.2) or one of the following built-in functions:

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

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

abs(.)
: returns the absolute value of the argument, which must be of numeric type (i.e. `byte` or `short` or `int` or `long` or `float` or `double`)

pow(.,.)
: returns the first argument value to the power of the second value; both must be of type `double`

pow(.)
: returns e to the power of the argument value, which must be of type `double`

log(.,.)
: returns the logarithm of the first argument value regarding the base given by the second value; both must be of type `double`

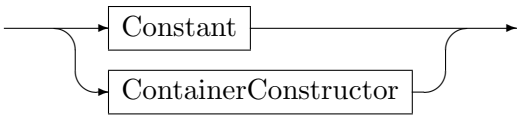
log(.,.)
: returns the logarithm of the argument value that must be of type `double` regarding the base e

sin(.)
: returns the sinus of the argument, which must be of type `double`

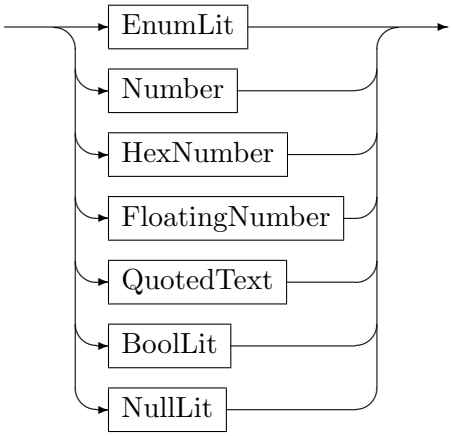
cos(.)
: returns the cosinus of the argument, which must be of type `double`

tan(.)
: returns the tangens of the argument, which must be of type `double`

Literal



Constant



The Constants are:

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

Number
: Is a `byte` or `short` or `int` or `long` number in decimal notation without decimal point, postfixed by `y` or `Y` for `byte`, postfixed by `s` or `S` for `short`, postfixed by `l` or `L` for `long`, or of `int` type if not postfixed.

HexNumber
: Is a `byte` or `short` or `int` or `long` number in hexadecimal notation starting with `0x`, the different types are distinguished by the suffix as for a decimal notation number.

FloatingNumber
: Is a `float` or `double` number in decimal notation with decimal point, postfixed by `f` or `F` for `float`, maybe postfixed by `d` or `D` for `double` .

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

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

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

EXAMPLE (16)

Some examples of literals:

```
1  Apple::ToffeeApple // an enum literal
2  42y // an integer number in decimal notation of byte type
3  42s // an integer number in decimal notation of short type
4  42 // an integer number in decimal notation of int type
5  42L // an integer number in decimal notation of long type
6  0x7eadbeef // an integer number in hexadecimal notation of int type
7  0xdeadbeefL // an integer number in hexadecimal notation of long type
8  3.14159 // a double number
9  3.14159f // a float number
10 "ve␣rule␣and␣0wn␣ze␣world" // a text literal
11 true // a bool literal
12 null // the object literal
```

5.9 Operator Priorities

The priorities of all available operators are shown in ascending order in the table below, the dots mark the positions of the operands, the commas separate the operators available on the respective priority level.

01	`. ? . : .`
02	`. || .`
03	`. && .`
04	`. | .`
05	`. ^ .`
06	`. & .`
07	`. \ .`
08	`. ==,!= .`
09	`. <,<=,>,>=,in .`
10	`. <<,>>,>>> .`
11	`. +,- .`
12	`. *,%,/ .`
13	`~,!,-,+ .`

Table 5.8: All operators, in ascending order of precedence

CHAPTER 6

NESTED PATTERNS

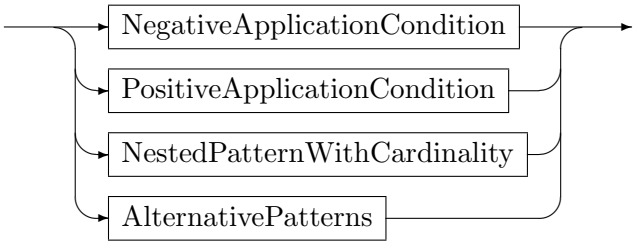
The extension of the rule set language described in the chapter 4 by nested patterns in this chapter 6 and the following chapter 7 greatly enhances the flexibility and expressiveness of pattern matching and rewriting. The following patterns to match a simplified abstract syntax tree give a rough picture of the language of nested and subpatterns:

EXAMPLE (17)

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

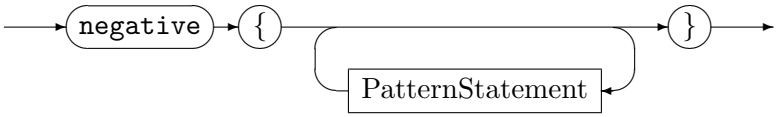
Until now we have seen rules and tests with one left hand side static pattern specification in a direct 1:1 correspondence with its dynamic match in the host graph on a successful application. From now on we will increase the expressiveness of the pattern language, and dependent on it the rewrite language, to describe much finer and more flexible what patterns to accept. This will be done by pattern specifications built up from multiple static pattern piece specifications, where the pieces may be matched dynamically zero, one, or multiple times, or are forbidden to exists for the entire pattern to be matched. These rule set language constructs can be split into nested patterns (negative application condition, positive application condition, nested pattern with cardinality, alternative patterns) and subpatterns (subpattern declaration and subpattern entity declaration, subrule declaration and usage), in this chapter we will focus on the nested patterns:

NestedPattern



6.1 Negative Application Condition (NAC)

NegativeApplicationCondition



With negative application conditions (keyword `negative`) we can specify graph patterns which forbid the application of a rule if any of them is present in the host graph (cf. [Sza05]). NACs possess a scope of their own, i.e. names defined within a NAC do not exist outside the NAC. Identifiers from surrounding scopes must not be redefined. If they are not explicitely mentioned, the NAC gets matched independent from them, i.e. elements inside a negative are `hom(everything from the outside)` by default. But referencing the element from the outside within the negative pattern causes it to get matched isomorphically/distinct to the other elements in the negative pattern. This is a bit unintuitive if you think of extending the pattern by negative elements, but cleaner and more powerful: just think of NACs to simply specify a pattern which should not be in the graph, with the possibility of forcing elements to be the same as in the enclosing pattern by name equality.

EXAMPLE (18)

We specify a variable which is not badly typed, i.e. a node `x` of type `Variable` which must not be target of an edge of type type with a source node of type `BadType`:

```
1  x:Variable;
2  negative {
3    x <-:type- :BadType;
4  }
```

Because NACs have their "own" binding, using NACs leads to specifications which might look a bit redundant.

EXAMPLE (19)

Let's check the singleton condition, meaning there's exactly one node of the type to check, for the type `RootNamespace`. The following specification is *wrong* (it will never return a match):

```
x:RootNamespace;
negative {
  y:RootNamespace;
}
```

Instead we have to specify the *complete* forbidden pattern inside the NAC. This is done by:

```
x:RootNamespace;
negative {
  x;
  y:RootNamespace; // now it is ensured that y must be different from x
}
```

Btw: the `x;` is not a special construct, it's a normal graphlet (cf. 4.1.1).

If there are several patterns which should not occur, use several negatives. If there are exceptions to the forbidden pattern, use nested negatives. As a straight-forward generalization of negatives within positive patterns, negatives may get nested to an arbitrary depth. Matching of the nested negative pattern causes the matching of the nesting pattern to fail.

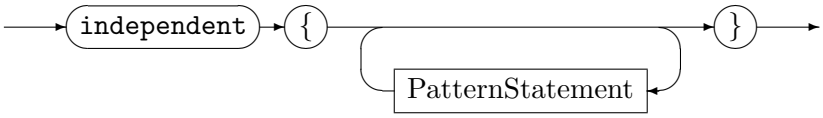
EXAMPLE (20)

A fabricated example using parallel as well as nested `negatives`:

```
test onlyOneChildOrAllChildrenHaveExactlyOneCommonChild
{
  root:Class;
  negative {
    root -:extending-> :Class; // root does not extend another class
  }
  root <-:extending- c1:Class; // a class c1 extends root
  negative {
    c1;
    root <-:extending- c2:Class; // there is no c2 which extends root
    negative {
      c1 <-:extending- child:Class -:extending-> c2; // except c1 and c2 have a common child
      negative { // and c1 has no further children
        child;
        c1 <-:extending- :Class;
      }
      negative { // and c2 has no further children
        child;
        c2 <-:extending- :Class;
      }
    }
  }
}
```

6.2 Positive Application Condition (PAC)

PositiveApplicationCondition



With positive application conditions (keyword `independent`) we can specify graph patterns which, in contrast to negative application conditions, must be present in the host graph to cause the matching of the enclosing pattern to succeed. Together with NACs they share the property of opening a scope, with elements being independent from the surrounding scope (i.e. a host graph element can easily get matched to a pattern element and a PAC element with a different name, unless the pattern element is referenced in the PAC). They are used to improve the logical structure of rules by separating a pure condition from the main pattern of the rule amenable to rewriting. They are used when all matches of a pattern are wanted, and a part of that pattern is available in the graph multiple times, but should not cause combinatorically additional matches; then the `independent` can be used to check only for the existence of that part, limiting the all-matching to the core pattern. They are really needed if subpatterns want to match elements which were already matched during the subpattern derivation.

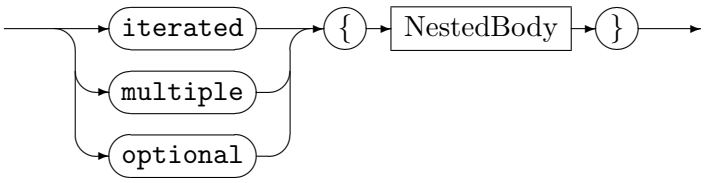
EXAMPLE (21)

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

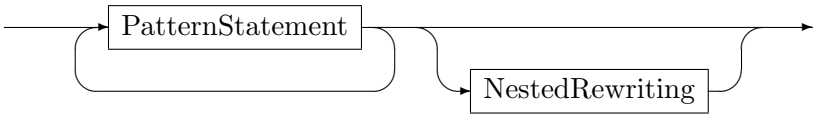
```
1  rule moveMethod
2  {
3    c:Class --> m:Method;
4    csub -:extending-> c;
5    csub:Class -e:Edge-> msub:Method;
6
7    independent {
8      // a complicated pattern to find out that m and msub have same signatures
9    }
10   independent {
11     // a complicated pattern to find out that msub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

6.3 Pattern Cardinality (iterated / multiple / optional)

NestedPatternWithCardinality



NestedBody



The blocks allow to specify how often the nested pattern – opening a scope – is to be matched. Matching will be carried out eagerly, i.e. if the construct is not limiting the number of matches and a further match is possible it will be done. (The nested body will be explained in Section 6.5.)

The Iterated Block

The iterated block is matching the contained subpattern as often as possible, succeeding even in the case the contained pattern is not available (thus it will never fail). It was included in the language to allow for matching breadth-splitting structures, as in capturing all methods of a class in a program graph.

EXAMPLE (22)

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

The Multiple Block

The multiple block is working like the iterated block, but expects the contained subpattern to be available at least once; if it is not, matching of the multiple block and thus its enclosing pattern fails.

EXAMPLE (23)

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

The Optional Block

The optional block is working like the iterated block, but matches the contained subpattern at most once; further occurences of the subpattern are left unmatched. If the nested pattern is available, it will get matched, otherwise it won't; matching of the optional block will succeed either way.

EXAMPLE (24)

```
test variableMaybeInitialized
{
  v:Variable; // match variable
  optional { // and an initialization with a different one if available
    v <-- otherV:Variable;
  }
}
```

Iteration Breaking

If an application condition inside an iteration block fails, then that potential match of the iterated pattern is thrown away and matching continues trying to find further matches. Sometimes a different behaviour is wanted, with an application condition terminating the iteration and causing it to fail. This would allow to check with a single rule that "every pattern X must also satisfy Y" holds. This behaviour is supported with the `break` keyword prepended to an application condition, transforming it into an iteration breaking condition.

EXAMPLE (25)

If the `negative` matches, not only the current iteration instance is prevented from matching, but the entire `iterated` (and thus the `test`) is failing to match:

```
test forEachXMustNotBeTheCaseY
{
  iterated {
    <X>;
    break negative {
      <Y>;
    }
  }
}
```

If the `independent` does not match, not only the current iteration instance is prevented from matching, but the entire `iterated` (and thus the `test`) is failing to match:

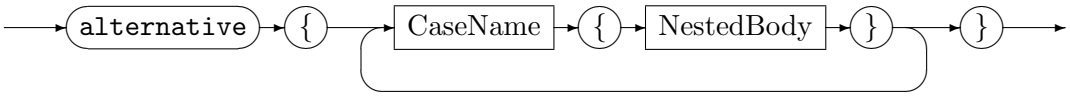
```
test forEachXMustBeTheCaseY
{
  iterated {
    <X>;
    break independent {
      <Y>;
    }
  }
}
```

NOTE (23)

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

6.4 Alternative Patterns

AlternativePatterns



With the alternative block you can specify several nested alternative patterns. One of them must get matched for the matching of the alternative (and thus its directly nesting pattern) to succeed, and only one of them is matched per match of the alternative / overall pattern. The order of matching the alternative patterns is unspecified, especially it is not guaranteed that a case gets matched before the case textually following – if you want to ensure that a case cannot get matched if another case could be matched, you must explicitely prevent that from happening by adding negatives to the cases. In contrast to the iterated which locally matches everything available and inserts this combined match into the current match, the alternative decides for one case match which it inserts into the current match tree, ignoring other possible matches by other cases.

EXAMPLE (26)

```
1  test feature(c:Class)
2  {
3    alternative // a feature of the class is either
4    {
5      FeatureMethod { // a method
6        c --> :Method;
7      }
8      FeatureVariable { // or a variable
9        c --> :Variable;
10     }
11     FeatureConstant { // or a constant
12       c ---> :Constant;
13     }
14   }
15 }
```

EXAMPLE (27)

```
1  test variableMaybeInitialized
2  {
3    v:Variable; // match variable
4    alternative { // and an initialization with a different one if available
5      Empty {
6        // the empty pattern matches always
7        negative { // so prevent it to match if initialization is available
8          v <-- otherV:Variable;
9        }
10     }
11     Initialized { // initialization
12       v <-- otherV:Variable;
13     }
14   }
15 }
```

EXAMPLE (28)

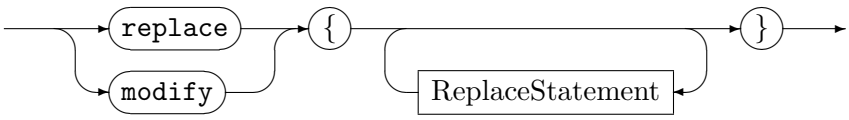
When working with the subtyping hierarchy one may be interested in matching in a first step an abstract base class, specifying the rewriting behaviour for this base class once, and in a refinement step in an alternative the different possible subtypes, then being able to access their specific attributes and being able of giving different additional rewrite parts.

```
1  test refineFeature
2  {
3    f:Feature; // match abstract base class Feature
4
5    alternative {
6      Variable {
7        v:Variable<f>; // try to cast to concrete Variable, if succeeds we can access the
             Variable attributes
8      }
9      Method {
10       m:Method<f>; // try to cast to a concrete Method, if succeeds we can access the Method
             attributes
11     }
12   }
13
14   modify {
15     // do stuff common to a Feature here
16   }
17 }
```

6.5 Nested Pattern Rewriting

Until now we focused on the pattern matching of nested and subpatterns – but we're not only interested in finding patterns combined from several pattern pieces, we want to rewrite the pattern pieces, too. So we will extend the language of the structure parser introduced so far into a language for a structure transducer. This does not hold for the application conditions, which are pure conditions, but for all the other language constructs introduced in this chapter.

NestedRewriting



Syntactically the rewrite is specified by a modify or replace clause nested directly within the scope of each nested pattern; in addition to the rewrite clause nested within the top level pattern. Semantically for every instance of a pattern piece matched its dependent rewrite is applied. So in the same manner the complete pattern is assembled from pattern pieces, the complete rewrite gets assembled from rewrite pieces (or operationally: rewriting is done along the match tree by rewriting one pattern piece after the other). Note that `return` statements are not available as in the top level rewrite part of a rule, and the `exec` statements are slightly different.

For a static pattern specification like the iterated block yielding dynamically a combined match of zero to many pattern matches, every submatch is rewritten, according to the rewrite specification applied to the host graph elements of the match bound to the pattern elements (if the pattern was matched zero times, no dependent rewrite will be triggered - but note that zero matches still means success for an iterated, so the dependent rewrite piece of the enclosing pattern will be applied). This allows e.g. for reversing all edges in the iterated-example (denoting containment in the class), as it is shown in the first of the following two examples. For the alternative construct the rewrite is specified directly at every nested pattern, i.e. alternative case as shown in the second of the following two examples); the rewrite of the matched case will be applied.

Nodes and edges from the pattern containing the nested pattern containing the nested rewrite are only available for deletion or retyping inside the nested rewrite if it can be statically determined this is unambigous, i.e. only happening once. So only the rewrites of alternative cases, optional patterns or subpatterns may contain deletions or retypings of elements not declared in their pattern (in contrast to iterated and multiple pattern rewrites).

EXAMPLE (29)

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

EXAMPLE (30)

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

NOTE (24)

In case you got a `rule` or `pattern` with an empty `modify` clause, with all the real work going on in an `alternative` or an `iterated`, you can omit the empty `modify` clause. This is a small syntactic convenience reducing noise which is strictly restricted to the top level pattern — omitting rewrite parts of nested patterns specifies the entire pattern to be match-only (like a `test`; this must be consistent for all nested patterns).

EXAMPLE (31)

This is an example which shows how to decide with an alternative on the target type of a retyping depending on the context. Please note the omitted rewrite (cf. 24).

```
1  rule alternativeRelabeling
2  {
3    m:Method;
4
5    alternative {
6      private {
7        if { m.access == Access::private; }
8
9        modify {
10         pm:PrivateMethod<m>;
11       }
12     }
13     static {
14       negative {
15         m <-- c;
16       }
17
18       modify {
19         sm:StaticMethod<m>;
20       }
21     }
22   }
23 }
```

CHAPTER 7

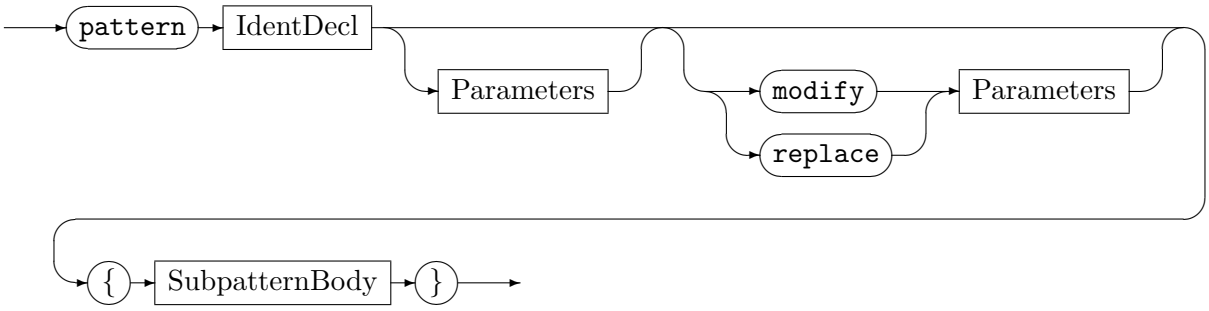
SUBPATTERNS AND YIELDING

After the introduction of the nested patterns in chapter 6 we will now have a look at the subpatterns, with the split into subpattern declaration plus subpattern entity declaration and subrule declaration plus usage, the other means to greatly enhances the flexibility and expressiveness of pattern matching and rewriting.

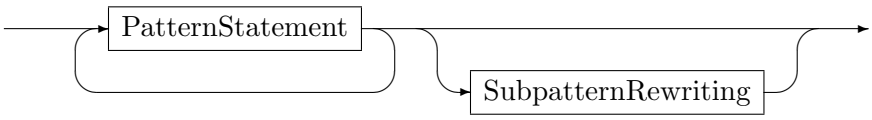
7.1 Subpattern Declaration and Subpattern Entity Declaration

Subpatterns were introduced to factor out a common recurring pattern – a shape – into a named subpattern type, ready to be reused at points the pattern should get matched. The common recurring pattern is specified in a subpattern declaration and used by a subpattern entity declaration.

SubpatternDeclaration

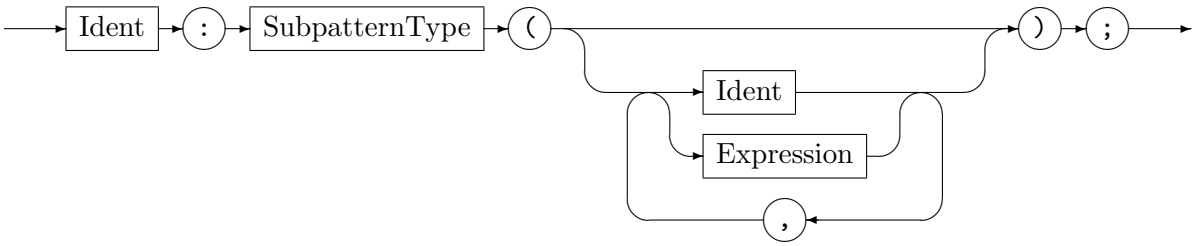


SubpatternBody



Subpattern declarations define a subpattern type denoting the specified shape in the global namespace; the parameters specify some context elements the pattern may refer to, but which are not part of the pattern itself. So they are only syntactically the same as test/rule-parameters (which are members of the pattern part). A further difference is the lack of ReturnTypes; they are not actions, just a helper in constructing complex patterns. In order to get values out they employ the language construct of def entites which are yielded to (cf. 7.3 later in this chapter). Subpatterns can receive additional rewrite parameters in contrast to the actions; they can be used to hand in nodes which are created in the rewrite part of the action or subpattern which contains the subpattern entity. (The nested body will be explained in Section 7.2.)

SubpatternEntityDeclaration



Subpattern entity declarations instantiate an entity of the subpattern type (i.e. specified shape), which means the subpattern must get matched for the matching of the enclosing pattern to succeed. The arguments given are bound to the corresponding parameters of the subpattern. If you prefer a syntactical point of view, you may see the subpattern entity as a placeholder, which gets substituted in place by the textual body of the subpattern declaration under renaming of the parameters to the arguments. If you prefer an operational point of view, you may see the subpattern entity as a call to the matcher routine searching for the specified pattern from the given arguments on.

EXAMPLE (32)

```
1  pattern TwoParameters(mp:Method)
2  {
3    mp <-- :Variable;
4    mp <-- :Variable;
5  }
6  test methodAndFurther
7  {
8    m:Method <-- n:Name;
9    tp:TwoParameters(m);
10 }
```

In the given example a subpattern `TwoParameters` is declared, connecting the context element `mp` via two edges to two variable nodes. The test `methodAndFurther` is using the subpattern via the declaration of the entity `tp` of type `TwoParameters`, binding the context element to its local node `m`. The resulting test after subpattern derivation is equivalent to the test `methodWithTwoParameters`.

```
1  test methodWithTwoParameters
2  {
3    m:Method <-- n:Name;
4    m <-- :Variable;
5    m <-- :Variable;
6  }
```

7.1.1 Recursive Patterns

Subpatterns can be combined with alternative patterns or the cardinality patterns into recursive subpatterns, i.e. subpatterns which may contain themselves. Subpatterns containing themselves alone – directly or indirectly – would never yield a match as an infinite pattern can't be found in a limited graph.

EXAMPLE (33)

```
test iteratedPath
{
  root:Assign;
  negative { --> root; }
  :IteratedPath(root); // match iterated path = assignment list
}

pattern IteratedPath(prev:Node)
{
  optional { // nothing or a linked assignment and again a list
    prev --> a:Assign; // assignment node
    :IteratedPath(a); // next one, plz
  }
}
```

The code above searches an iterated path from the root node on, here an assignment list. The iterated path with the optional is equivalent to the code below. Note the negative which ensures you get a longest match – without it the empty case may be chosen lazily just in the beginning. Please note that if you only need to check for the existence of such a simple iterated path you can use the `reachable` function introduced in 13.2.3.

```
pattern IteratedPath(prev:Node)
{
  alternative {
    Empty {
      negative {
        prev --> a:Assign;
      }
    }
    Further {
      prev --> a:Assign;
      :IteratedPath(a);
    }
  }
}
```

The code below searches an iterated path like the code above, just that it stops when a maximum length is reached. The bounded iterated path is realized by calling a pattern with the requested maximum depth, counting the depth parameter down with each recursion step, until the limit checked by a condition is reached.

```
test boundedIteratedPath(root:Assign)
{
  :BoundedIteratedPath(root, 3); // match iterated path of depth 3
}

pattern BoundedIteratedPath(prev:Node, var depth:int)
{
  optional { // nothing or a linked assignment and again a list
    prev --> a:Assign; // assignment node
    :IteratedPath(a, depth-1); // next one, plz
    if{ depth >= 1; } // stop when we have reached max depth
  }
}
```

EXAMPLE (34)

```
1  rule removeMiddleAssignment
2  {
3    a1:Assign --> a2:Assign --> a3:Assign;
4    independent {
5      :IteratedPath(a1,a3)
6    }
7
8    replace {
9      a1; a3;
10   }
11 }
12
13 pattern IteratedPath(begin:Assign, end:Assign)
14 {
15   alternative { // an iterated path from begin to end is either
16     Found { // the begin assignment directly linked to the end assignment (base case)
17       begin --> end;
18     }
19     Further { // or an iterated path from the node after begin to end (recursive case)
20       begin --> intermediate:Assign;
21       :IteratedPath(intermediate, end);
22     }
23   }
24 }
```

This is once more a fabricated example, for an iterated path from a source node to a distinctive target node, and an example for the interplay of subpatterns and positive application conditions to chech complex conditions independent from the pattern already matched. Here, three nodes `a1`,`a2`,`a3` of type `Assign` forming a list connected by edges are searched, and if found, `a2` gets deleted, but only if there is an iterated path of directed edges from `a1` to `a3`. The path may contain the host graph node matched to `a2` again. Without the `independent` this would not be possible, as all pattern elements – including the ones originating from subpatterns – get matched isomorphically. The same path specified in the pattern of the rule – not in the independent – would not get matched if it would go through the host graph node matched to b, as it is locked by the isomorphy constraint.

With recursive subpatterns you can already capture neatly structures extending into depth (as iterated paths) and also structures extending into breadth (as forking patterns, although the cardinality statements are often much better suited to this task). But combined with an iterated block, you may even match structures extending into breadth and depth, like e.g. an inheritance hierarchy of classes in our example domain of program graphs, see example 35, i.e. you can match a spanning tree in the graph. This gives you a very powerful and flexible notation to capture large, complex patterns built up in a structured way from simple, connected pieces (as e.g. abstract syntax trees of programming languages).

NOTE (25)

If you are working with hierarchic structures like that, you might be interested in the capablities of GrShell/yComp for nested layout as described and shown in 18.1/48).

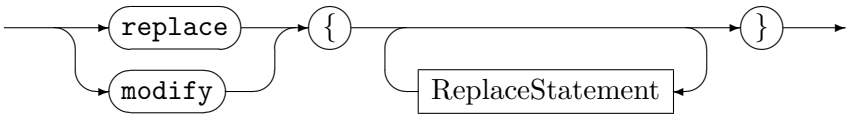
EXAMPLE (35)

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

7.2 Subpattern Rewriting

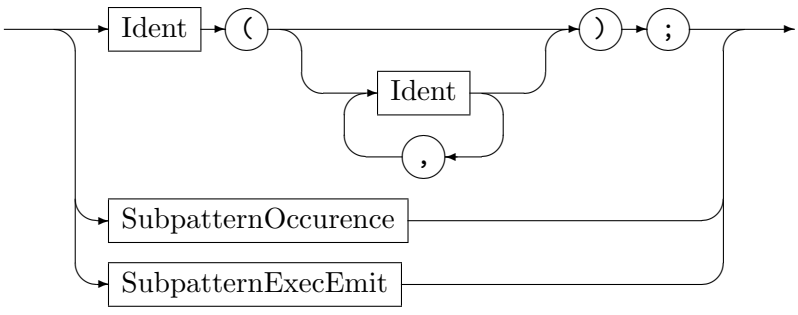
Alongside the separation into subpattern declaration and subpattern entity declaration, subpattern rewriting is separated into a nested rewrite specification given within the subpattern declaration defining how the rewrite looks like and a subpattern rewrite application given within the rewrite part of the pattern containing the subpattern entity declaration requesting the rewrite to be actually applied.

SubpatternRewriting



The subpattern rewriting specifications within the subpattern declaration looks like a nested rewriting specification, but additionally there may be rewrite parameters given in the subpattern header (cf. 7.1) which can be referenced in the rewrite body. (Most elements can be handed in with normal parameters, but elements created in the rewrite part of the user of the subpattern can only be handed in at rewrite time.)

SubpatternRewriteApplication



The *SubpatternRewriteApplication* is part of the *ReplaceStatement* already introduced (cf. 4.4.3). The subpattern rewrite application is given within the rewrite part of the pattern containing the subpattern entity declaration, in call notation on the declared subpattern identifier. It causes the rewrite part of the subpattern to get used; if you leave it out, the subpattern is simply kept untouched. The *SubpatternOccurence* is explained in the next subsection 7.2.1. The *SubpatternExecEmit* is explained in chapter 10.

EXAMPLE (36)

This is an example for a subpattern rewrite application.

```
pattern TwoParametersAddDelete(mp:Method)
{
  mp <-- v1:Variable;
  mp <-- :Variable;

  modify {
    delete(v1);
    mp <-- :Variable;
  }
}
rule methodAndFurtherAddDelete
{
  m:Method <-- n:Name;
  tp:TwoParametersAddDelete(m);

  modify {
    tp(); // trigger rewriting of the TwoParametersAddDelete instance
  }
}
```

EXAMPLE (37)

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

```
pattern IteratedPathReverse(prev:Node)
{
  optional {
    prev --> next:Node;
    ipr:IteratedPathReverse(next);

    replace {
      prev <-- next;
      ipr();
    }
  }

  replace {
  }
}
```

EXAMPLE (38)

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

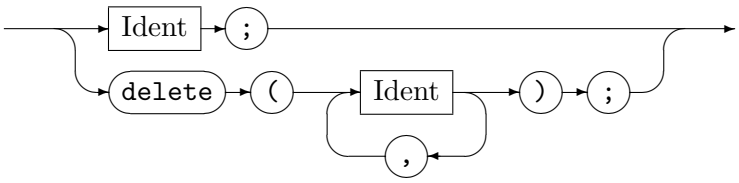
```
pattern ChainFromToReverseToCommon(from:Node, to:Node) replace(common:Node)
{
  alternative {
    rec {
      from --> intermediate:Node;
      cftrtc:ChainFromToReverseToCommon(intermediate, to);

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

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

  replace {
    from; to;
  }
}
```

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

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

7.2.1 Deletion and Preservation of Subpatterns

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

SubpatternOccurence



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

EXAMPLE (39)

```
rule R {
  m1:Method; m2:Method;
  tp1:TwoParameters(m1);
  tp2:TwoParameters(m2);

  replace {
    tp1; // is kept
    // tp2 not included here - will be deleted
    // tp1(); or tp2(); -- would apply dependent replacement
    m1; m2;
  }
}
```

NOTE (26)

You may even give a SubpatternEntityDeclaration within a rewrite part which causes the subpattern to be created; but this employment has several issues which can only be overcome by introducing explicit creation-only subpatterns – so you better only use it if you think it should obviously work (examples for the issues are alternatives – which case to instantiate? – and abstract node or edge types – what concrete type to choose?).

```
pattern ForCreationOnly(mp:Method)
{
  // some complex pattern you want to instantiate several times
  // connecting it to the mp handed in
}
rule createSubpattern
{
  m:Method;

  modify {
    :ForCreationOnly(m); // instantiate pattern ForCreationOnly
  }
}
```

7.3 Local Variables, Ordered Evaluation, and Yielding Outwards

Local Variables and Ordered Evaluation

Sometimes attribute evaluation becomes easier with temporary variables; such local variables can be introduced on a right hand side employing the known variable syntax `var name:type`, prefixed with the `def` keyword. From then on they can be read and assigned to in eval statements of the RHS, or used as variable parameters in subpattern rewrite calls. In addition, on their introduction an initializing expression may be given.

EXAMPLE (40)

```
rule R {
  n1:N; n2:N; n3:N; n4:N; n5:N;

  modify {
    def var mean:double = (n1.v + n2.v + n3.v + n4.v + n5.v)/5;
    eval {
      n1.variance = (n1.v - mean)*(n1.v - mean);
      n2.variance = (n2.v - mean)*(n2.v - mean);
      n3.variance = (n3.v - mean)*(n3.v - mean);
      n4.variance = (n4.v - mean)*(n4.v - mean);
      n5.variance = (n5.v - mean)*(n5.v - mean);
    }
  }
}
```

Normally the rewrite order is as given in table 7.1:

1.	Extract elements needed from match
2.	Create new nodes
3.	Call rewrite code of used subpatterns *and more...*
4.	Call rewrite code of nested iterateds
5.	Call rewrite code of nested alternatives
6.	Redirect edges
7.	Retype (and merge) nodes
8.	Create new edges
9.	Retype edges
10.	Create subpatterns
11.	Attribute reevaluation
12.	Remove edges
13.	Remove nodes
14.	Remove subpatterns
15.	Emit / Exec
16.	Return

and more... at `3.` are `evalhere`, `emithere`, `alternative` Name, `iterated` Name

Table 7.1: Execution order rewriting

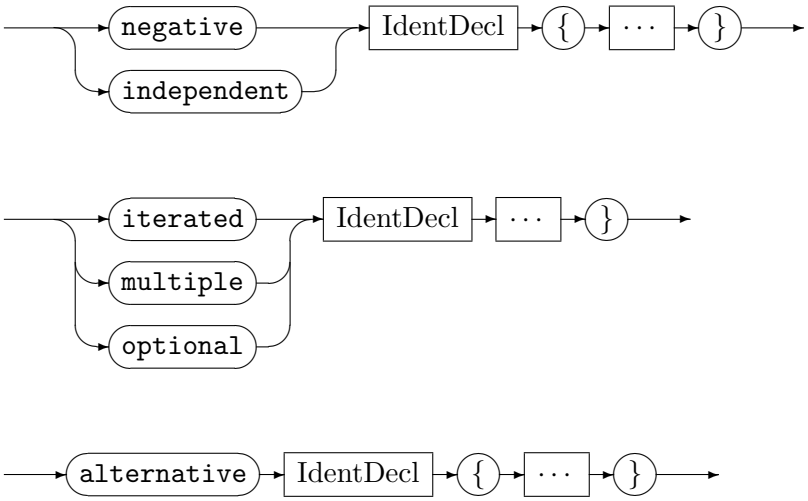
So first the subpatterns rewrites, then the iterated rewrites, then the alternative rewrits are executed, and finally the local eval statements (computations). This might be sufficient in some cases, but in other cases when you want to compute an attributation over a tree/a graph, you want to have local computations influenced by attributes in nested/called children or its siblings, and attributes in nested/called children influenced by its parents or siblings. So

we need a language device which allows us to intermingle attribute computations in between the rewrite part executions of nested patterns and subpattern rewrite calls. And a language device which allows us to give the execution oder of the alternative and iterated statements relative to the subpattern rewrite calls and attribute evaluations.

To achieve attribute evaluation in a defined order in between the subpattern rewrite calls, we use ordered evaluation statements, introduced with the keyword `evalhere`; they get executed in the order in which they are given syntactically (a further statement executed in order is `emithere`, introduced in 10.2).

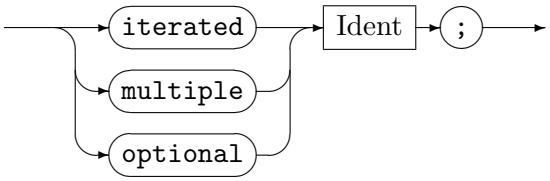
To achieve iterated/alternative execution in order, we allow names to be given to nested patterns, and reuse this name in a nested pattern rewrite order specification. Naming nested patterns is done with the following syntax, as the already introduced syntax remains valid, on aggregate we extend the nested patterns with optional names to form named nested pattern.

NamedNestedPattern

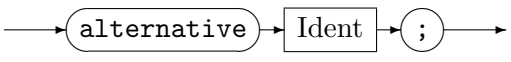


Alternatives and iterateds named this way can then be referenced in the rewrite part with an alternative rewrite order specification or an iterated rewrite order specification.

CardinalityRewriteUsage



AlternativeRewriteUsage



What we've seen so far is applied in the following example. A `yield` prefix is needed whenever a `def` variable is written to (as assignment target as well as a subpattern output argument).

EXAMPLE (41)

```
rule R {
  iterated foo { .; modify { ..read i.. } }
  alternative bar { case { modify { ..read i.. } } }
  sub1:Subpattern1();
  sub2:Subpattern2();

  modify {
    def var i:int = 0; // initializes i to 0
    evalhere { yield i = i + 1; } // afterwards i==1
    sub1(i); // input 1 to subpattern rewrite
    evalhere { yield i = i + 1; } // afterwards i==2
    iterated foo; // nested iterated reads i==2
    evalhere { yield i = i + 1; } // afterwards i==3
    alternative bar; // nested alternative reads i==3
    evalhere { yield i = i + 1; } // afterwards i==4
    sub2(i); // input 4 to subpattern rewrite
    evalhere { yield i = i + 1; } // afterwards i==5
    eval { yield j = i + 1; } // assign 6 to j
  }
}
```

NOTE (27)

For rewriting the execution order of the parts can be defined, to allow programming attribute evaluation orders of interest, defining when to descend into which part and defining glueing/local computations in between. (A depth first run with a defined order in between the siblings, comparable to an LAG/RAG run in compiler construction, but with an explicitely defined sequence of children visits, instead of a temperal succession implicitely induced by the syntactical left-to-right ordering). In contrast to rewriting, the *matching* order of the pattern parts can *not* be defined, to allow the compiler/the runtime to use the evaluation order it estimates to be the best. So we can't access attributes from sibling elements, we can only compute attributes top down from local elements or elments handed in on matching, and later on bottom up from local elements or elements bubbleing up at match object tree construction. Top down attribute evaluation operates on the already matched elements and attribute values or the ones received as inputs, which are handed down implicitely into nested patterns or explicitely via subpattern parameters into subpattern instances. (A depth first run too, but without a defined order in between the siblings, comparable to an IAG run in compiler construction computed during matching while descending). Bottom up attribute evalutaion operates on the matched elements and attribute values locally available or the ones received into def elements yielded implicitely upwards from the nested patterns or explicitely accumulating iterated results or with assigning out parameters of subpatterns. (The same depth first run, but with attributes computed while ascending, comparable to an SAG run in compiler construction.)

Yielding Outwards During Rewriting

Sometimes one needs to bring something matched within a nested or subpattern to an outer pattern containing it (nested patterns) or calling it (subpatterns). So that one can do there (in the using pattern) operations on it, e.g. attaching a further edge to an end node of a chain matched with recursive patterns (thus modularizing the graph rewrite specification into chain

matching patterns and patterns using chains doing things on the chain ends), or summing attributes matched in iterated pattern instances.

The first thing one needs to bring something outwards is a target in a nesting or calling pattern. This is achieved by nodes, edges, and variables declared with the `def` keyword in a rewrite part, marking them as output entities; variables were already introduced in previous paragraphs, but in addition to them nodes and edges are allowed, too. Furthermore subpattern rewrite parameters may be declared as def parameters, marking them as output parameters. These elements are then yielded to from within `eval` or `evalhere` statements, subpattern rewrite usages, and `exec` statements. While the latter will be covered in chapter 10, the former will be explained in the following.

Yielding is specified by prepending the `yield` keyword to the entity yielded to, in an assignment to a variable or a method call on a variable, inside an `eval` or `evalhere`-statement, or a change assignment; the target of the assignment may be a node or edge (if declared as output variable). The yield must be prepended to the argument for a subpattern def rewrite parameter, too.

EXAMPLE (42)

```
1  pattern Chain(begin:Node) modify(def end:Node)
2  {
3    alternative {
4      Further {
5        begin --> intermediate:Node;
6        c:Chain(intermediate);
7
8        modify {
9          c(yield end);
10       }
11     }
12     Done {
13       negative {
14         begin --> ;
15       }
16
17       modify {
18         eval {
19           yield end = begin;
20         }
21       }
22     }
23   }
24
25   modify { }
26 }
27
28 rule R(begin:Node) : (Node) {
29   c:Chain(begin);
30
31   modify {
32     def end:Node;
33     c(yield end); // end is filled with chain end
34     return(end);
35   }
36 }
```

First example for RHS yielding: returning the end node of a chain.

EXAMPLE (43)

```
rule outCount(head:Node) : (int)
{
  iterated {
    head --> .;
    modify {
      eval { yield count = count + 1; }
    }
  }

  modify {
    def var count:int = 0;
    return (count);
  }
}
```

Second example for RHS yielding: counting the number of edges matched with an iterated.

Yielding Outwards During Match Object Construction

Bubbeling up the elements from nested patterns and called patterns during rewriting might be too late or inconvenient. Luckily it can be done before, at the end pattern matching when the match object tree gets constructed.

As for RHS yielding the targets of the yielding must be nodes, edges, or variables declared with the `def` keyword prepended, marking them as output entities; but this time in the pattern part. Furthermore subpattern parameters may be declared as def parameters in the subpattern definition header, marking them as output parameters.

These elements can then be yielded to from within `eval` statements inside a `yield` block (maybe with iterated accumulation) and subpattern usages. A `yield` block is a constrained `eval` block which can be given in the pattern part; it does not allow to assign to or change non-`def` variables, or carry out graph-changing commands. Yielding is specified by prepending the `yield` keyword to the entity yielded to, in the assignment or method call. The `yield` must be prepended to the argument for a subpattern def parameter, too.

NOTE (28)

A def entity from the pattern part can't be yielded to from the rewrite part, they are constant after matching.

Let's have a look at two examples for yielding:

EXAMPLE (44)

```
pattern Chain(begin:Node, def end:Node)
{
   alternative {
      further {
         begin --> next:Node;
         :Chain(next, yield end);
      }
      done {
        negative {
          begin --> ;
        }
         yield {
            yield end = begin;
         }
      }
   }
}

pattern LinkChainTo(begin:Node) modify(n:Node)
{
   alternative {
      further {
         begin --> next:Node;
         o:LinkChainTo(next);

         modify {
           next --> n;
           o(n);
         }
      }
      done {
        negative {
          begin --> ;
        }

         modify {
         }
      }
   }

   modify { }
}

rule linkChainEndToStartIndependent(begin:Node) : (Node)
{
   def end:Node;

   independent {
      c:Chain(begin, yield end);
   }
  o:LinkChainTo(begin);

   modify {
     o(end);
      return(end);
   }
}
```

The first example for LHS yielding follows within an independent a chain piece by piece to some a priori unknown end node, and yields this end node chain piece by chain piece again outwards to the chain start. There it is used as input to another chain (maybe the same chain, maybe overlapping due to the independent), linking all the nodes of this chain to the end node of the former.

When yielding from an iterated pattern there's the problem that each yielding assignment from an iterated instance would overwrite the one def variable from outside the iterated, while one is interested most of the time in some accumulation of the values, e.g. summing integers or concatenating strings. This can be achieved with a `for` loop iterating a def variable inside an iterated for all the matches of the iterated pattern referenced by name, allowing to assign to an outside def variable a value computed from the def variable and the value of the iterated def variable.

This is shown in the second example for LHS yielding, summing the integer attribute `a` of nodes of type `N` adjacent to a start node, matched with an iterated.

EXAMPLE (45)

```
1  test sumOfWeight(start:Node) : (int,int)
2  {
3     def var sum:int = 0;
4     def var v:int = 0;
5
6     iterated it {
7        def var i:int;
8
9        start --> n:N; // node class N { a:int; }
10
11       yield {
12          yield i = n.a;
13          yield v = 42; // v is assigned 42 multiple times
14       }
15    }
16
17    yield {
18       for{i in it; yield sum = sum + i};
19    }
20
21    return (sum,v);
22 }
```

7.4 Regular Expression Syntax and Locking

In addition to the already introduced syntax for the nested patterns with the keywords `negative`, `independent`, `alternative`, `iterated`, `multiple` and `optional`, there is a more lightweight syntax resembling regular expressions; using it together with the subpatterns gives graph rewrite specifications which look like EBNF-grammars with embedded actions. Exceeding the more verbose syntax they offer constructs for matching the pattern a bounded number of times (same notation as the one for the bounded iteration in the xgrs).

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

Table 7.2: Map of nested patterns in keyword syntax to regular expression syntax

EXAMPLE (46)

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

Isomorphy Locking

When matching a program graph as in the introductory example 17 one might be satisfied with matching a tree structure. But on other occasions one wants to match *backlinks* and especially the targets of the backlinks, too, from *uses* nested somewhere in the syntax graph to *definitions* whose nodes were already matched earlier in the subpattern derivation (subpatterns can be seen as an equivalent of grammar rules known from parser generators). Unfortunately these elements are already matched and thus isormorphy locked following the default semantics of isomorphic matching. And unfortunately these elements can't be declared `homomorphic` as they are unknown in the nested subpattern. Handing them in as parameters and then declaring them `homomorphic` is only possible if they are of a statically fixed number (as the number of parameters is fixed at compile time), which is normally not the case for e.g. the attributes of a class in a syntax graph. In order to handle this case the `independent` *operator* (cf. 4.3.1) was added to the rule language — when you declare the backlink target node `n` as `independent(n)` it can be matched once again. Thus it is possible to match e.g. a class attribute definition node which was already matched when collecting the attributes of the class again later on in a subpattern when matching an expression containing a usage of that attribute, allowing to e.g. add further edges to it.

Patternpath Locking

As stated in the sections on the negative and independent constructs (6.1, 6.2), they get matched homomorphically to all already matched elements. By referencing an element from outside you can isomorphy lock that element to prevent it to get matched again.

Maybe you want to lock all elements from the directly enclosing pattern, in this case you can just insert `pattern;` in the position of a graphlet into the NAC or PAC.

Maybe you want to lock all elements from the patterns dynamically containing the NAC/-PAC of interest, i.e. all subpattern usages and nesting patterns on the path leading to the NAC/PAC of interest (but not their siblings). In this case you can insert `patternpath;` in the position of a graphlet into the NAC or PAC. You might be interested in this construct when matching a piecewise constructed pattern, e.g. a chain, which requires to check for another chain (iterated path) which is not allowed to cross (include an element of) the original one.

CHAPTER 8

RULE APPLICATION CONTROL LANGUAGE (XGRS)

Graph rewrite sequences (GRS), better extended graph rewrite sequences XGRS, to distinguish them from the older graph rewrite sequences, are a domain specific GrGen.NET language used for controlling the application of graph rewrite rules. They are available

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

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

Graph rewrite sequences are built from a pure *rule control* language, written down in a syntax similar to boolean and regular expressions, with rule applications as atoms, and a *computations* sublanguage, noted down as a sequence of assignments, function calls, or procedure calls. A computation is given as an atom of the rule control language, nested in curly braces. (The syntactical separation into two sublanguages was carried out in version 3.5, you might be interested in the `-deprecated` flag available for the generator and the shell in version 3.0.1 in case you got old code; when the flag is set the sequence constructs which were moved to the computations sublanguage employed in your solution are reported.)

The graph rewrite sequences are a means of composing complex graph transformations out of single graph rewrite rules and further computations. The control flow in the rule control language is determined by the evaluation order of the operands. Graph rewrite sequences have a boolean return value; for a single rule, `true` means the rule was successfully applied to the host graph. A `false` return value means that the pattern was not found in the host graph.

In order to store and reuse return values of rewrite sequences and most importantly, for passing return values of rules to other rules, *variables* can be defined. A variable is an arbitrary identifier which can hold a graph element or a value of one of the attribute or value types GrGen.NET knows. There are two kinds of variables available in GrGen.NET, i) graph global variables and ii) sequence local variables. A variable is alive from its first declaration on: graph global variables are implicitely declared upon first usage of their name, sequence local variables are explicitely declared with a typed variable declaration of the form `name:type`. Graph global variables must be prefixed with a double colon ::, local variables are referenced just with their name. Graph global variables are untyped; their values are typed, though, so type errors cause an exception at runtime. They belong to and are stored in the graph processing environment – if you save the graph in GrShell then the variables are saved, too, and restored next time you load the saved graph. Further on, they are nulled if the graph element assigned to them gets deleted (even if this happens due to a transaction rollback), thus saving one from debugging problems due to zombie elements (you may use the `def()` operator to check during execution if this happened). Sequence local variables are typed, so type errors are caught at compile time (parsing time for the interpreted sequences); an assignment of an untyped variable to a typed variable is checked at runtime. They belong

to the sequence they appear in, their life ends when the sequence finishes execution (so there is no persistency available for them as for the graph global variables; neither do they get nulled on element deletion as the graph does not know about them).

If used in some rule, i.e. within an `exec`, named graph elements of the enclosing rule are available as read-only variables.

Note that we have two kinds of return values in graph rewrite sequences. Every rewrite sequence returns a boolean value, indicating whether the rewriting could be successfully processed, i.e. denoting success or failure. Additionally rules may return graph elements. These return values can be assigned to variables on the fly (see example 47).

EXAMPLE (47)

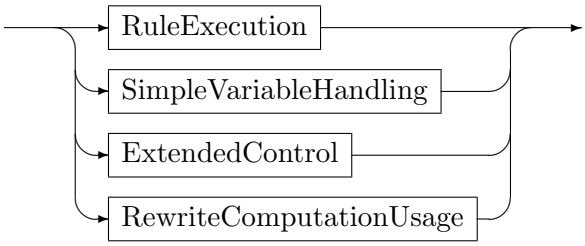
The graph rewrite sequences

```
1 (b,c)=R(x,y,z)=>a
2 a = ((b,c)=R(x,y,z))
```

assign the two returned graph elements from rule `R` to variables `b` and `c` and the information whether `R` mached or not to variable `a`. The first version is recommended.

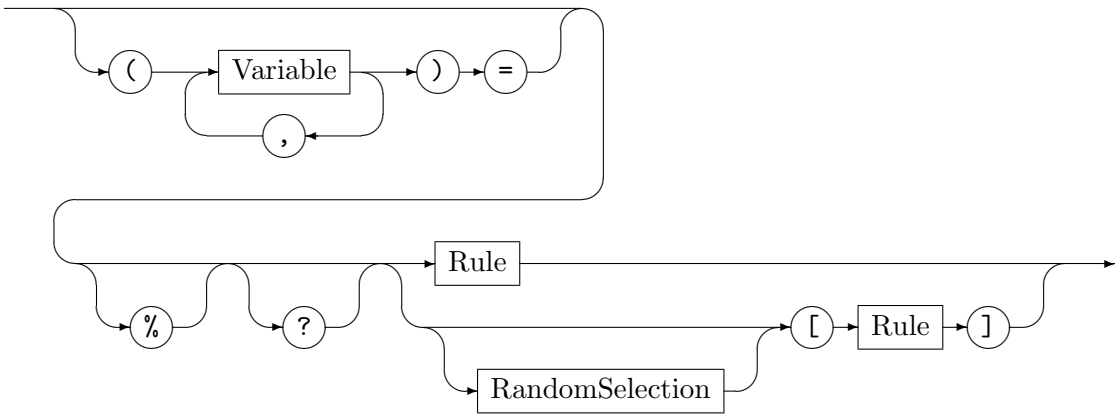
8.1 Rule Application

RewriteFactor

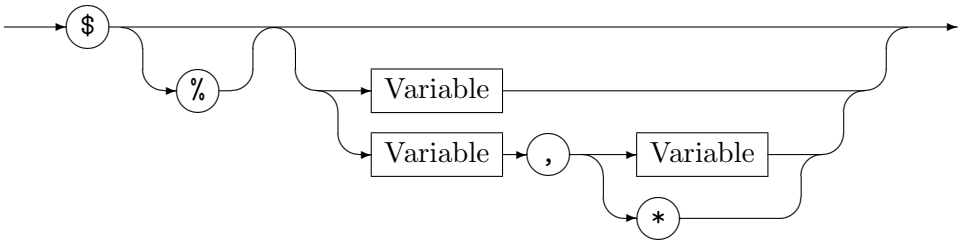


Rewrite factors are the building blocks of graph rewrite sequences. They are split into four major areas: rule (and sequence) application, simple variable handling, extended control, and sequence computation usages. Here we start with the most import one, applying rules. In section 8.3 we visit simple variable handling. In chapter 15 we have a look at advanced control 15.2 and sequence computations 14.1.

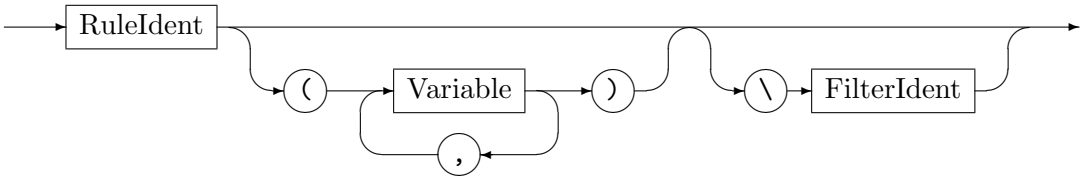
RuleExecution



RandomSelection



Rule



The *RuleExecution* clause applies a single rule or test. In case of a rule, the first found pattern match will be rewritten. Application will fail in case no match was found and succeed otherwise. Variables and named graph elements can be passed into the rule. The returned graph elements can be assigned to variables again. The operator `?` switches the rule to a test, i.e. the rule application does not perform the rewrite part of the rule but only tests if a match exists. The operator `%` is a multi-purpose flag. In the GrShell (see Chapter 17) it dumps the matched graph elements to `stdout`; in `debug`-mode (see Chapter 18) it acts as a break point (which is its main use in fact); you are also able to use this flag for your own purposes, when using GrGen.NET via its API interface (see Section 1.7.3). The match filter application (which allows e.g. to filter symmetric matches) is explained in 21.3.

The *RuleExecution* clause can be applied to a defined sequence (cf. 15.1), or an external sequence (cf. 21.4), too. Application will succeed or fail depending on the result of the body of the sequence definition called. In case of success, the output variables of the sequence definition are written to the destination variables of the assignment. In case of failure, no assignment takes place, so sequence calls behave the same as rule calls. The break point `%` can be applied to a sequence call, but neither the `?` operator nor all braces (`[]`).

The sqare braces (`[]`) introduce a special kind of multiple rule application: Every pattern match produced by the rule will be rewritten; if at least one was found, rule application will succeed, otherwise it will fail. Attention: This all bracketing is **not** equal to `Rule*`. Instead this operator collects all the matches first before starting to rewrite. So if one rewrite destroys other matches or creates new match opportunities the semantics differ; in particular the semantics is unsafe, i.e. one needs to avoid deleting or retyping a graph element that is bound by another match (will be deleted/retyped there). On the other hand this version is more efficient and allows one to get along without marking already handled situations (to prevent a rule matching again and again because the match situation is still there after the rewrite; normally you would need some match preventing device like a negative or visited flags to handle such a situation). If *Rule* returns values, the values of *one* of the executed rules will be returned.

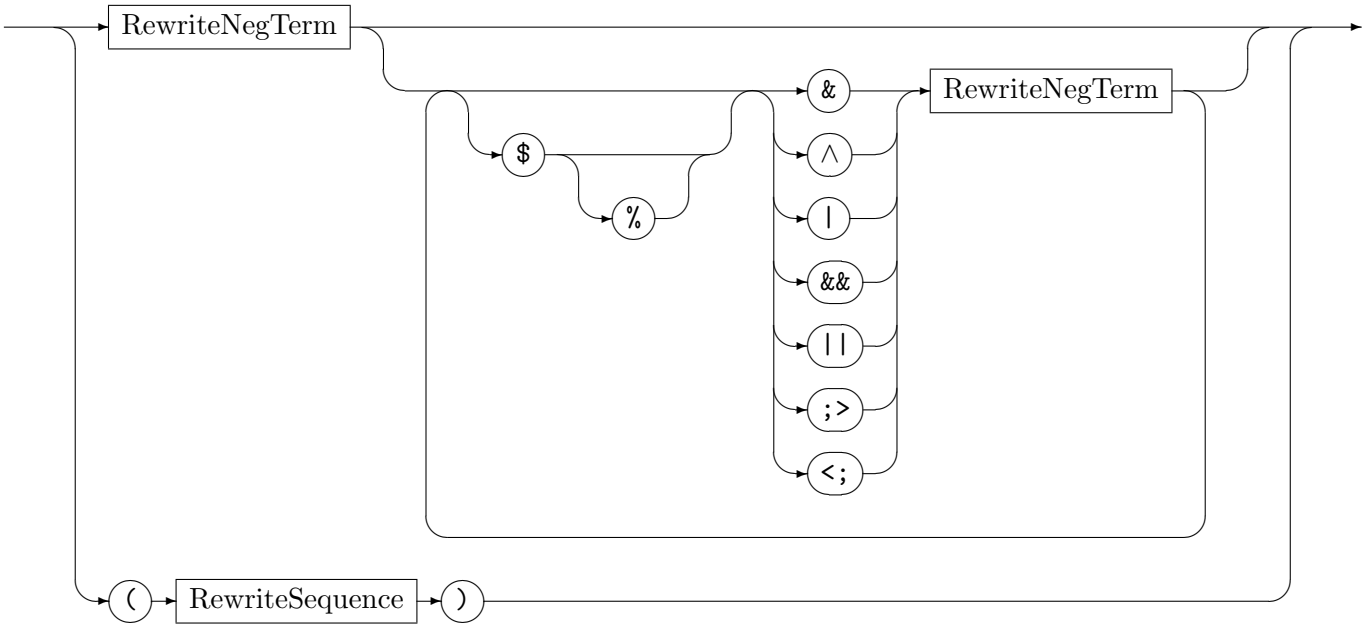
The random match selector `$v` searches for all matches and then randomly selects `v` of them to be rewritten (but at most as much as are available), with `$[r1]` being equivalent to `anonymousTempVar=1 & $anonymousTempVar[r1]`. Rule application will fail in case no match was found and succeed otherwise. You may change the lower bound for success by giving a variable containing the value to apply before the comma-separated upper bound variable. In case a lower bound is given the upper bound may be set to unlimited with the `*`. An `%` appended to the `$` denotes a choice point allowing the user to choose the match to be applied from the available ones in the debugger (see Chapter 18).

EXAMPLE (48)

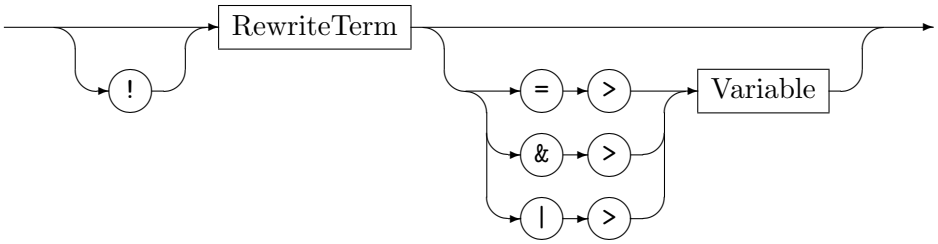
The sequence `(u,v)=r(x,y)` applies the rule `r` with input from the variables `x` and `y` on the host graph and assigns the return elements from the rule to the variables `u` and `v` (the parenthesis around the out variables are always needed, even if there's only one variable assigned to). The sequence `$[t]` determines all matches of the parameterless rule `t` on the host graph, then one of the matches is randomly chosen and executed.

8.2 Logical and Sequential Connectives

RewriteSequence



RewriteNegTerm



A graph rewrite sequence consists of several rewrite terms linked by operators. Table 8.1 gives the priorities and semantics of the operators, priorities in ascending order. Forcing execution order against the priorities can be achived by parentheses. The modifier `$` changes the semantics of the following operator to randomly execute the left or the right operand first (i.e. flags the operator to act commutative); usually operands are executed / evaluated from left to right if not altered by bracketing. In contrast the sequences `s`, `t`, `u` in `s $<op> t $<op> u` are executed / evaluated in arbitrary order. The modifier `%` appended to the `$` overrides the random selection by a user selection (cf. see Chapter 18, choice points).

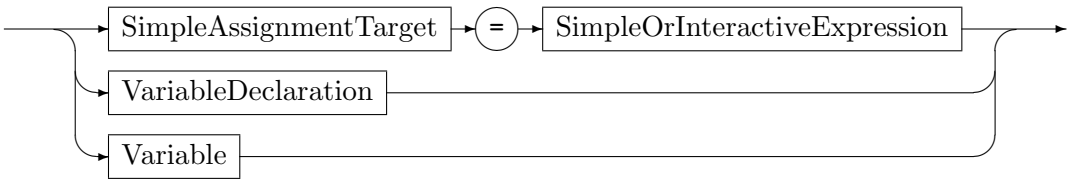
The assign-to operator `=>` optionally available at the end of the *RewriteNegTerm* assigns the (negated in case of `!`) result of the *RewriteTerm* execution to the given variable; the and-to `&>` operator assigns the conjunction and the or-to `|>` operator assigns the disjunction of the original value of the variable with the sequence result to the variable.

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

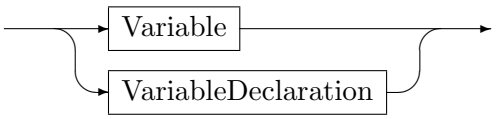
Table 8.1: Semantics and priorities of rewrite sequence operators

8.3 Simple Variable Handling

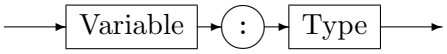
SimpleVariableHandling



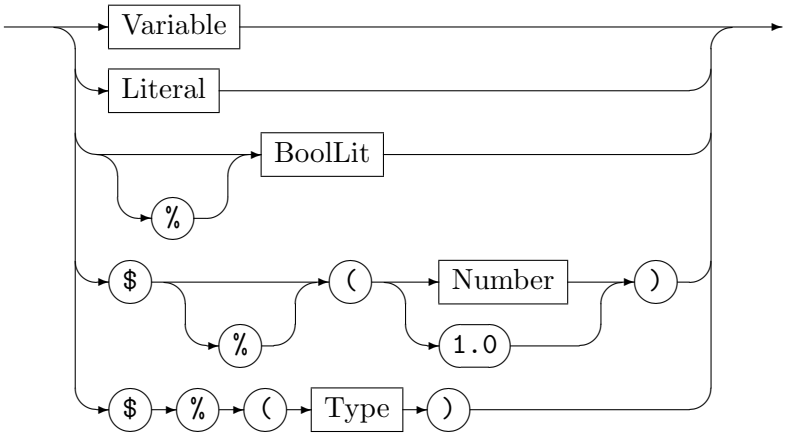
SimpleAssignmentTarget



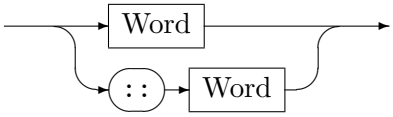
VariableDeclaration



SimpleOrInteractiveExpression



Variable

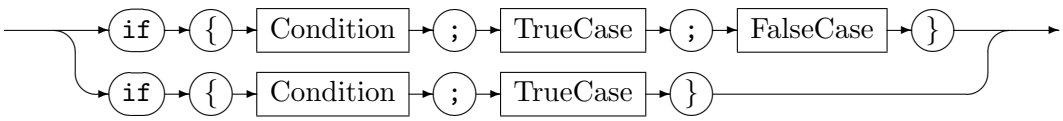


The simple variable handling in the sequences allows to assign a variable or a constant to a variable, to interactively query for an element of a given type or a number and assign it to a variable, or to declare a local variable; these constructs always result in true/success. In addition, a boolean variable may be used as a predicate; using such a variable predicate together with the sequence result assignment allows to directly transmit execution results from one part of the sequence to another one. Furtheron, a boolean constant may be used as a predicate. These sequence constants being one of the boolean literals `true` or `false` come in handy if a sequence is to be evaluated but its result must be a predefined value; furtheron a break point may be attached to them.

Variables can hold graph elements, or values of value/attribute types, including booleans. The typed explicit declaration (which may be given at an assignment, rendering that assignment into an initialization) introduces a sequence local variable, the name alone references a sequence local variable. A global variable is accessed with the double colon prefix, it gets implicitely declared if not existing yet (you can't declare a graph global variable). The random number assignment `v=$(42)` assigns an integer random number in between 0 and 41 (42 excluded) to the variable `v`. The random number assignment `v=$(1.0)` assigns a double random number in between 0.0 and 1.0 exclusive to the variable `v` (here you can't change the upper bound as with the integer assignment). Appending a `%` changes random selection to user selection (defining a choice point). The user input assignment `v=$%(string)` queries the user for a string value – this only works in the GrShell. The user input assignment `v=$%(Node)` queries the user for a node from the host graph – this only works in the GrShell in debug mode. The non simple variable handling is given in 14.1, even further variable handling constructs are given in 14.3.

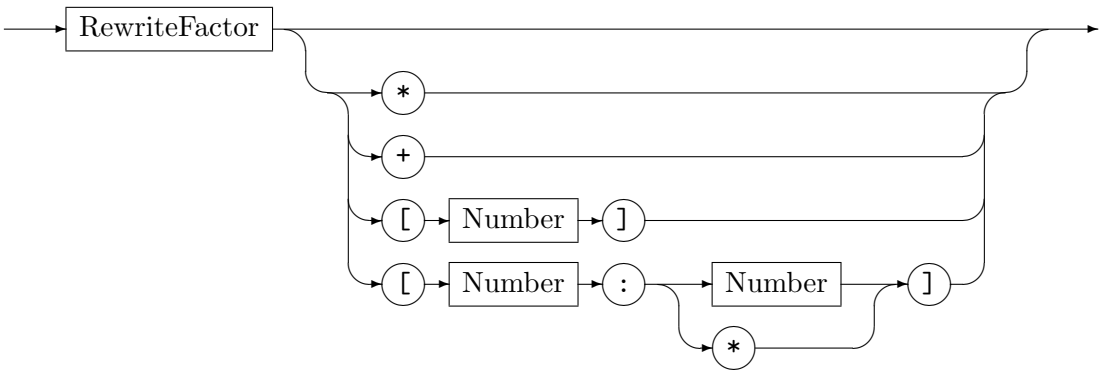
8.4 Decisions and Loops

ExtendedControl



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

RewriteTerm



A rewrite term consists of a rewrite factor which can be executed multiple times. The star (`*`) executes a sequence repeatedly as long as its execution does not fail. Such a sequence always returns `true`. A sequence `s+` is equivalent to `s && s*`. The brackets (`[m]`) execute a sequence repeatedly as long as its execution does not fail but m times at most; the min-max-brackets (`[n:m]`) additionally fail if the minimum amount n of iterations was not reached.

NOTE (29)

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

8.5 Quick reference table

Table 8.2 lists the basic operations of the graph rewrite sequences at a glance.

`s ;> t`	Execute `s` then `t`. Success if `t` succeeded.
`s <; t`	Execute `s` then `t`. Success if `s` succeeded.
`s | t`	Execute `s` then `t`. Success if `s` or `t` succeeded.
`s || t`	The same as `s | t` but with lazy evaluation, i.e. if `s` is successful, `t` will not be executed.
`s & t`	Execute `s` then `t`. Success if `s` and `t` succeeded.
`s && t`	The same as `s & t` but with lazy evaluation, i.e. if `s` fails, `t` will not be executed.
`s ^ t`	Execute `s` then `t`. Success if `s` or `t` succeeded, but not both.
`if{r;s;t}`	Execute `r`. If `r` succeeded, execute `s` and return the result of `s`. Otherwise execute `t` and return the result of `t`.
`if{r;s}`	Same as `if{r;s;true}`
`!s`	Switch the result of `s` from successful to fail and vice versa.
`$<op>`	Use random instead of left-associative execution order for `<op>`.
`s*`	Execute `s` repeatedly as long as its execution does not fail.
`s+`	Same as `s && s*`.
`s[n]`	Execute `s` repeatedly as long as its execution does not fail but `n` times at most.
`s[m:n]`	Same as `s[n]` but fails if executed less than m times.
`s[m:*]`	Same as `s*` but fails if executed less than m times.
`?`*`Rule`*	Switches *Rule* to a test.
`%`*`Rule`*	This is the multi-purpose flag when accessed from LIBGR. Also used for graph dumping and break points.
`[`*`Rule`*`]`	Rewrite every pattern match produced by the action *Rule*.
`true`	A constant acting as a successful match.
`false`	A constant acting as a failed match.
`v`	A variable acting as a predicate.

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

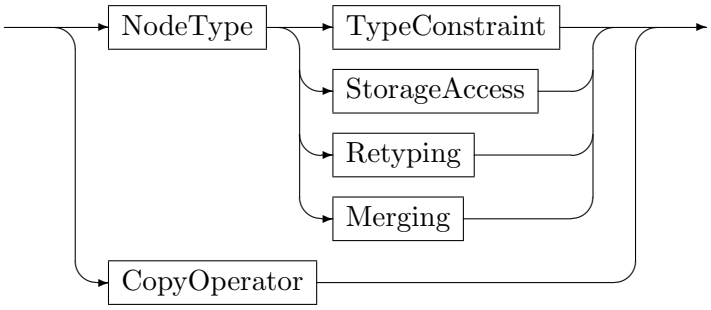
Table 8.2: Sequences at a glance

CHAPTER 9

ADVANCED MATCHING AND REWRITING

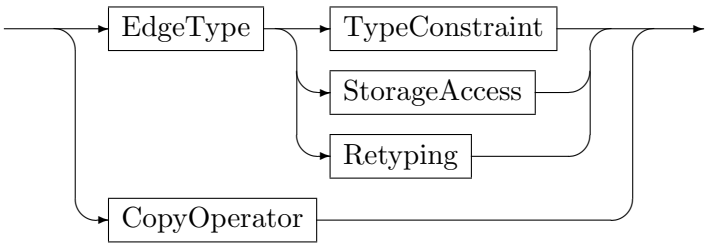
The advanced *modifiers* introduced in the following section allow to annotate patterns or actions with keywords which restrict what graph patterns are accepted as matches (some of them independent of the rewrite part, some of them depending on the rewrite specification). But first the advanced *matching* constructs are introduced in this section, before they are elaborated on in a later section: they allow to request more fine grain or more dynamically what types to match, as well as allowing to specify a match from a storage. Followed by the advanced *rewrite* constructs which are handled in the same way (introduction here, then elaboration in a later section); these enable the specification of retyping (relabeling) and copying, as well as node merging and edge redirection.

AdvancedNodeTypeConstructs



Specifies a node of type *NodeType*, constrained in type with a *TypeConstraint* (see Section 9.2, *TypeConstraint*), or bound by a storage access (see 12.6, *StorageAccess*), or retyped with a *Reytping* (see Section 9.4, *Retyping*), or merged with a *Merging* (see Section 9.6, *Merging*). Alternatively it may define a node having the same type and bearing the same attributes as another matched node (see Section 9.5, *CopyOperator*). Type constraints are allowed in the pattern part only. The CopyOperator and the Merging clause are allowed in the replace/modify part only. The Retyping clause is a chimera which restricts the type of an already matched node when used on the LHS, and casts to the target type when used on the RHS, which is its primary use.

AdvancedEdgeTypeConstructs

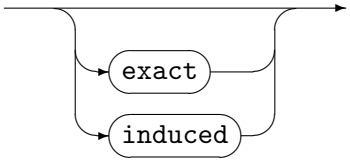


The *AdvancedEdgeTypeConstructs* specify an edge of type *EdgeType* or a copy of an edge. Type constraints are allowed in the pattern part only (see Section 9.2, *TypeConstraint*); the

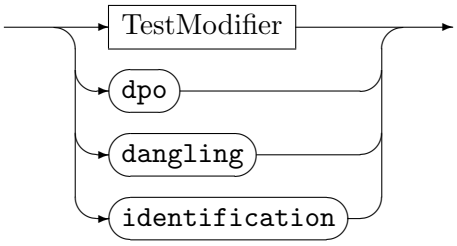
same holds for the storage access (see 12.6, *StorageAccess*). The CopyOperator and the Redirect clause are allowed in the replace/modify part only (see Section 9.5, *CopyOperator*, see Section 9.7, *Redirect*). The Retyping clause is a chimera which restricts the type of an already matched edge when used on the LHS, and casts to the target type when used on the RHS (its primary use). Furthermore edges may be redirected, this is shown in Section 9.7, *Redirection*.

9.1 Rule and Pattern Modifiers

TestModifier



RuleModifier



By default GrGen.NET performs rewriting according to SPO semantics as explained in Section 4.4.1. This behaviour can be changed with *pattern modifiers* and *rule modifiers* (and the other advanced rewrite constructs introduced in the following sections which spoil the theoretical foundation but are highly useful in practice). Such modifiers add certain conditions to the applicability of a pattern. The idea is to match only parts of the host graph that look more or less exactly like the pattern. The level of "exactness" depends on the choosen modifier. A pattern modifier in front of the `rule`/`test`-keyword is equivalent to one modifier-statement inside the pattern containing all the specified nodes (including anonymous nodes). Table 9.1 lists the pattern modifiers with their semantics, table 9.2 lists the rule only modifiers with their semantics. Example 49 explains the modifiers by small toy-graphs.

Modifier	**Meaning**
`exact`	Switches to the most restricitive mode. An exactly-matched node is matched, if all its incident edges in the host graph are specified in the pattern.
`induced`	Switches to the induced-mode, where nodes contained in the same `induced` statement require their induced subgraph within the host graph to be specified, in order to be matched. In particular this means that in general `induced(a,b,c)` differs from `induced(a,b); induced(b,c)`.

Table 9.1: Semantics of pattern modifiers

Modifier	Meaning
`dpo`	Switches to DPO semantics . This modifier affects only nodes that are to be deleted during the rewrite. DPO says—roughly spoken—that implicit deletions must not occur by all means. To ensure this the dangling condition (see `dangling` below) and the identification condition (see `identification` below) get enforced (i.e. `dpo = dangling + identification`). In contrast to `exact` and `induced` this modifier applies neither to a pattern as such (can't be used with a `test`) nor to a single statement but only to an entire rule. See Corradini et al.[CMR+99] for a DPO reference.
`dangling`	Ensures the dangling condition . This modifier affects only nodes that are to be deleted during the rewrite. Nodes going to be deleted due to the rewrite part have to be specified exactly (with all their incident edges, `exact` semantics) in order to be matched. As with `dpo`, this modifier applies only to rules.
`identification`	Ensures the identification condition . This modifier affects only nodes that are to be deleted during the rewrite. If you specify two pattern graph elements to be homomorphically matched but only one of them is subject to deletion during rewrite, those pattern graph elements will never actually be matched to the same host graph element. As with `dpo`, this modifier applies only to rules.

Table 9.2: Semantics of rule only modifiers

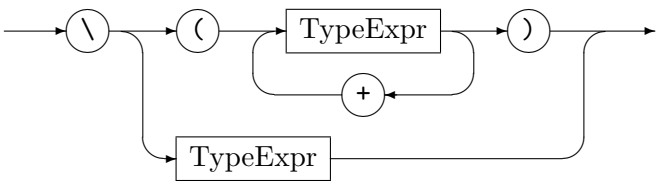
NOTE (30)

Internally all the modifier-annotated rules are resolved into equivalent rules in standard SPO semantics. The semantics of the modifiers is mostly implemented by NACs. In particular you might want to use such modifiers in order to avoid writing a bunch of NACs yourself. The number of internally created NACs is bounded by $\mathcal{O}(n)$ for `exact` and `dpo` and by $\mathcal{O}(n^2)$ for `induced` respectively, where n is the number of specified nodes.

9.2 Static Type Constraint

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

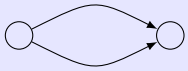
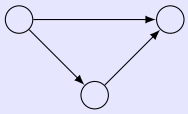
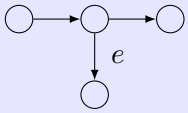
TypeConstraint



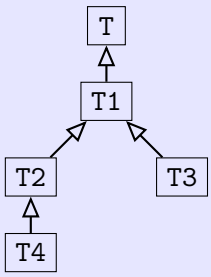
A type constraint is used to exclude parts of the type hierarchy. The operator `+` is used to create a union of its operand types. So the following pattern statements are identical:

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

EXAMPLE (49)

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

EXAMPLE (50)

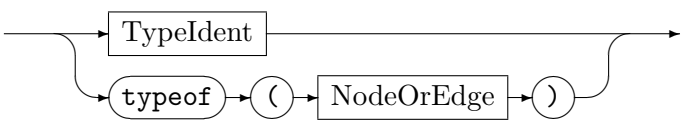


The type constraint `T\(T2+T3)` applied to the type hierarchy on the left side yields only the types `T` and `T1` as valid.

9.3 Exact Dynamic Type

In the following section we'll have a look at a language construct which allows to require an element to be typed the same as another element or to create an element with the same exact dynamic type as another element.

Type



The type of a graph element may be given by a type identifier, or a typeof denoting the exact dynamic type of a matched graph element. The element declaration `el:typeof(x)` introduces a graph element of the type the host graph element `x` is actually bound to. It may appear in the pattern or in the rewrite part. If it is given in the pattern, the element to match must be of the same exact dynamic type as the element referenced in the `typeof`,

otherwise matching will fail. If is is given in the rewrite part, the element to create is created with the same exact dynamic type as the element referenced in the `typeof`; have a look at the next section for the big brother of this language construct, the `copy` operator, which is only applicable in the rewrite part.

EXAMPLE (51)

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

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

Remember that we have several subtypes of `street`. By the aid of the `typeof` operator, the reverse edge will be automatically typed correctly (the same type as the one-way edge). This behavior is not possible without the `typeof` operator.

9.4 Retyping

In addition to graph rewriting, GrGen.NET allows graph relabeling[LMS99], too; we prefer to call it retyping. Nodes as well as edges may be retyped to a different type; attributes common to the initial and final type are kept. The target type does not need to be a subtype or supertype of the original type. Retyping is useful for rewriting a node but keeping its incident edges; without it you'd need to remember and restore those. Syntactically it is specified by giving the original node enclosed in left and right angles.

Retyping

Pattern (LHS)	**Replace (RHS)**	$r : L \longrightarrow R$	**Meaning**
`x:N1;`	`y:N2<x>;`	$r : \text{lhs}.x \mapsto \text{rhs}.x$	Match `x`, then retype `x` from `N1` to `N2` and bind name `y` to retyped version of `x`.
`e:E1;`	`f:E2<e>;`	$r : \text{lhs}.e \mapsto \text{rhs}.e$	Match `e`, then retype `e` from `E1` to `E2` and bind name `f` to the retyped version of `e`.

Table 9.3: Retyping of matched nodes and edges

Retyping enables us to keep all adjacent nodes and all attributes stemming from common super types of a graph element while changing its type (table 9.3 shows how retyping can be expressed both for nodes and edges). Retyping differs from a type cast: During replacement both of the graph elements are alive. Specifically both of them are available for evaluation, a respective evaluation could, e.g., look like this:

```
eval {
  y.b = ( 2*x.i == 42 );
```

```
  f.a = e.a;
}
```

Furthermore the source and destination types need *not* to be on a path in the directed type hierarchy graph, rather their relation can be arbitrary. However, if source and destination type have one ore more common super types, then the respective attribute values are adopted by the retyped version of the respective node (or edge). The edge specification as well as *ReplaceNode* supports retyping. In Example 4 node `n5` is a retyped node stemming from node `n1`. Note, that—conceptually—the retyping is performed *after* the SPO conforming rewrite.

EXAMPLE (52)

The following rule will promote the matched city `x` from a `City` to a `Metropolis` keeping all its incident edges/streets, with exception of the matched street `y`, which will get promoted from `Street` to `Highway`, keeping all its adjacent nodes/cities.

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

The following rule will retype the matched city `x` to the exacty dynamic type of `z` (which might be e.g. `Metropolis`).

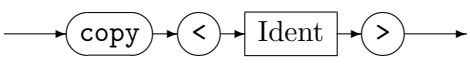
```
1 rule retypeToTypeof {
2   x:City; z:City;
3
4   modify {
5     x_rt:typeof(z)<x>;
6   }
7 }
```

The retyping clause `new:type<old>` can be used on the LHS, too. When it appears on the left hand side, the node(/edge) `old` is not changed in any way, instead a further node(/edge) `new` is made available for querying, being identical to `old` regarding the object reference, but additionally giving access to the attributes known to the `type` – if matching was successful. The construct tries to cast `old` to `new`, if successful `new` allows to access `old` as `type`, otherwise matching fails for this `old` (and binding another graph element to `old` is tried out). Please have a look at example 28 for more on this.

9.5 Copy

The copy operator allows to create a node or edge of the type of another node/edge, bearing the same attributes as that other node. It can be seen as an extended version of the `typeof` construct not only copying the exact dynamic type but also the attributes of the matched graph element. Together with the `iterated` construct it allows to simulate node replacement grammars or to copy entire structures, see 16 and **??**.

CopyOperator



EXAMPLE (53)

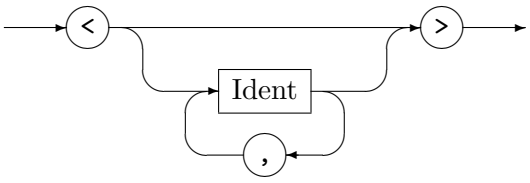
The following rule will add a reverse edge to a one-way street, of the exact dynamic subtype of `street`, bearing the same attribute values as the original street.

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

  replace {
    a -x-> y;
    y -:copy<x>-> a;
  }
}
```

9.6 Node Merging

The retyping construct for nodes can be extended into a node merging construct, which internally redirects edges.

Merging



Merging enables us to fuse several nodes into one node. Syntactically it is given by a retyping clause which not only mentiones the original node inside angle brackets, but several original nodes. Semantically the first node in the clause is retyped, then all edges of the other original nodes are redirected to the retyped node, and finally the other original nodes are deleted. As the type of the merging clause can be set to `typeof(first original node)`, a pure merging without retyping can be achieved.

EXAMPLE (54)

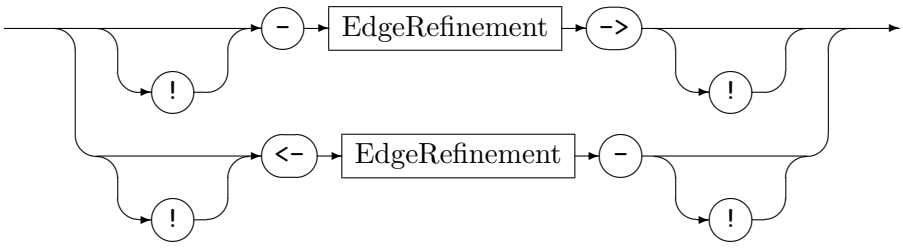
The following rule will match two `State`s and merge them. Every edge incident to `b` before and every edge incident to `w` before will be incident to the merged successor state `bw` afterwards; edges connecting the two `State`s become reflexive edges.

```
rule merge {
  b:State;
  w:State;
  if { b.name == "Baden" && w.name == "Wuerttemberg"; }

  modify {
    bw:typeof(b)<b,w>;
    eval { bw.name = b.name + w.name; }
  }
}
```

9.7 Edge Redirection

The redirect statement allows to exchange the source or the target node of a directed edge (or both) with a different node; it can be seen as syntactic sugar for removing one edge and creating a new one with the source/target node being replaced by a different node, with the additional effect of keeping edge identity.

Redirect



Redirection is specified with an exclamation mark at the end to be redirected as seen from the edge center; the exclamation mark enforces the redirection which would normally be rejected by the compiler "This is different from what was matched, but that's intentionally, make it happen!"

EXAMPLE (55)

The following rule will reverse the one-way street in between `a` and `y` by rewriting the old source to the new target and the old target to the new source.

```
1 rule oneway {
2   a:Node -x:street-> y:Node;
3
4   modify {
5     a !<-x-! y;
6   }
7 }
```

CHAPTER 10

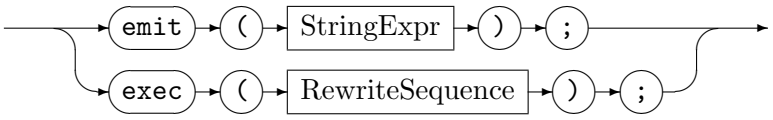
EMBEDDED SEQUENCES AND TEXTUAL OUTPUT

In this chapter we'll have a look at language constructs which allow to emit text from rules/-subpatterns and which allow to execute a graph rewrite sequence at the end of a rule invocation. The ability to execute a sequence at the end of a rule invocation allows to combine rules and build complex rules.

10.1 Exec and Emit in Rules

The following syntax diagram gives an extensions to the syntax diagrams of the Rule Set Language chapter 4:

ExecStatement



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

XGRS Execution

The `exec` statement executes a graph rewrite sequence, which is a composition of graph rewrite rules. Graph elements may be yielded to def variables in the RHS pattern. See Chapter 8 for a description of graph rewrite sequences. The `exec` statement is one of the means available in GrGen.NET to build complex rules and split work into several parts, see 16 for a discussion of this topic.

Text Output

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

EXAMPLE (56)

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

```
rule AddRedundancy
{
  s: SourceNode;
  t: DestinationNode;
  modify {
    emit ("Source node is " + s.name + ". Destination node is " + t.name + ".");
    exec ( (x:SourceNode) = DuplicateNode(s) & ConnectNeighbors(s, x)* );
    exec ( [DuplicateCriticalEdge] );
    exec ( MaxCapacityIncidentEdge(t)* );
    emit ("Redundancy added");
  }
}
```

EXAMPLE (57)

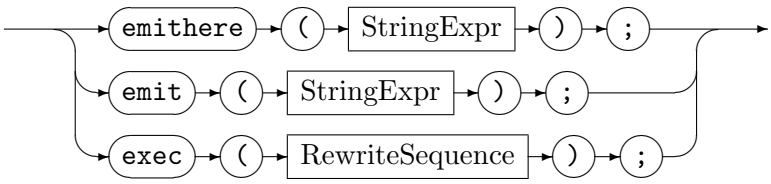
This is an example for returning elements yielded from an `exec` statement. The results of the rule `bar` are written to the variables `a` and `b`; The `yield` is a prefix to an assignment showing that the target is from the outside.

```
rule foo : (A,B)
{
  modify {
    def u:A; def v:B;
    exec( (a,b)=bar ;> yield u=a ;> yield v=b) );
    return(u,v);
  }
}
```

10.2 Deferred Exec and Emithere in Nested and Subpatterns

The following syntax diagram gives an extensions to the syntax diagrams of the Subpatterns chapter 7:

SubpatternExecEmit



The statements `emit`, `emithere` and `exec` enhance the declarative nested (and subpattern) rewrite part by imperative clauses. The `emit` and `emithere` statements get executed during rewriting before the `exec` statements; the `emithere`-statements get executed before the `emit` statements, in the order in between the subpattern rewrite applications they are specified syntactically (see 7.3 for more on this). The `exec` statements are executed i) after the rule which used the pattern they are contained in was executed and ii) in the order as they

appear within the rule. They are a slightly different version of the `exec`-statements from the *ExecStatement* introduced in 4.4.3, only available in the rewrite parts of subpatterns or nested alternatives/iterateds (but not in the rewrite part of rules as the original embedded sequences). They are executed after the original rule calling them was executed, so they can't get extended by `yield`s, as the containing rule is not available any more when they get executed.

NOTE (31)

The embedded sequences are executed after the top-level rule which contains them (in a nested pattern or in a used subpattern) was executed; they are *not* executed during subpattern rewriting. They allow you to put work you can't do while executing the rule proper (e.g. because an element was already matched and is now locked due to the isomorphy constraint) to a waiting queue which gets processed afterwards — with access to the elements of the rule and contained parts which are available when the rule gets executed. Or to just split the work into several parts, reusing already available functionality, see 16 for a discussion on this topic.

NOTE (32)

And again — the embedded sequences are executed *after* the rule containing them; thus rule execution is split into two parts, a declartive of parts a) and b), and an imperative. The declarative is split into two subparts: First the rule including all its nested and subpatterns is matched. Then the rule modifications are applied, including all nested and subpattern modification.
After this declarative step, containing only the changes of the rule and its nested and used subpatterns, the deferred execs which were spawned during the main rewriting are executed in a second, imperative step; during this, a rule called from the sequence to execute may do other nifty things, using further own sequences, even calling itself recursively with them. First all sequences from a called rule are executed, before the current sequences is continued or other sequences of its parent rule get executed (depth first).
Note: all changes from such dynamically nested sequences are rolled back if a transaction/a backtrack enclosing a parent rule is to be rolled back (but no pending sequences of a parent of this parent).

EXAMPLE (58)

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

```
rule caller
{
  n:Node;
  sub:Subpattern();

  modify {
    sub();
    exec(r(n));
  }
}
pattern Subpattern
{
  n:Node;
  modify {
    exec(s(n));
  }
}
```

EXAMPLE (59)

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

```
pattern BinaryExpression(root:Expr)
{
  root --> l:Expr; le:Expression(l);
  root --> r:Expr; re:Expression(r);
  root <-- binOp:Operator;

  modify {
    le(); // rewrites and emits the left expression
    emithere(binOp.name); // emits the operator symbol in between the left tree and the
        right tree
    re(); // rewrites and emits the right expression
  }
}
```

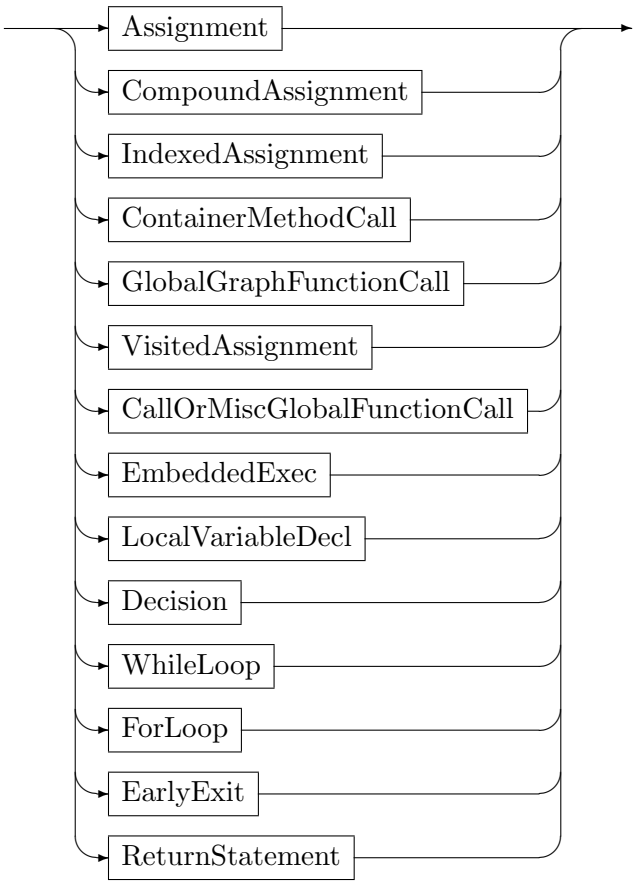
CHAPTER 11

COMPUTATIONS (EXTENDED ATTRIBUTE EVALUATION)

In this chapter we'll have a look at the computations available in the rule language. Their most important construct is the attribute assignment to assign new values to graph element attributes. Besides this, further types of assignments are available, and several control flow constructs as known from imperative programming languages of the C-family. Furthermore, container methods may be called (more on this in the following chapter 12), and global graph functions may be called (more on this in the following chapter 13).

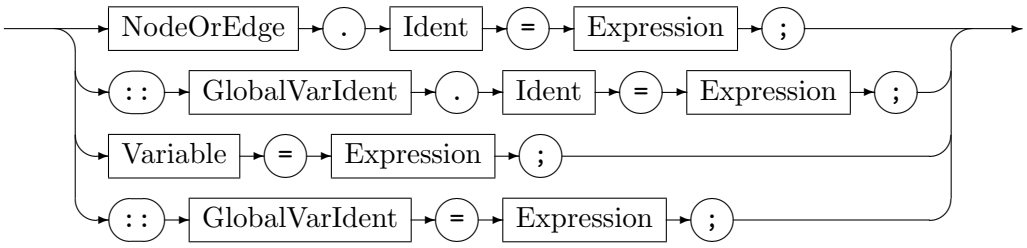
An example implementing depth-first search and breadth-first search as computations of the rule language can be found in `test/should_pass`, in `/DfsBfsSearch.grg`; you may have a look at the `/computations_*.grg` files in that folder for further (but fabricated) examples.

Statement



11.1 Assignments

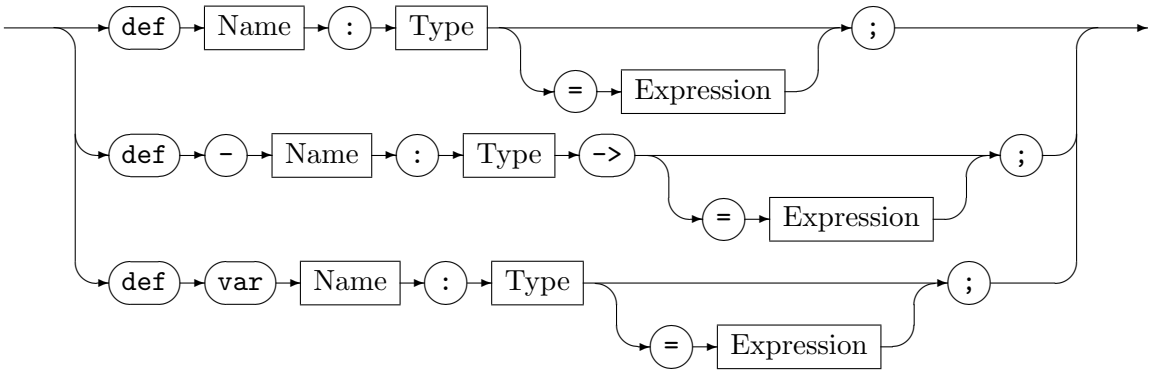
Assignment



Evaluation statements have imperative semantics. In particular, GrGen.NET does not care about data dependencies (you must care for them!). Assignment is carried out using value semantics, even for entities of container or `string` type. The only exception is the type `object`, there reference semantics is used. You can find out about the available *Expressions* in chapter 5.

11.2 Local Variable Declarations

LocalVariableDecl



Local variables can be defined with the syntax known for local def pattern variables, as already introduced in 7.3, i.e. `def var name:type` for variables, or `def name:type` for nodes, or `def -name:type->` for edges. At their definition, an initializing expression may be given.

11.3 Calls and Misc. Global Functions

You may call any of the functions from the expressions that were already introduced or that will be introduced just for its side effects, throwing away the result. Besides there are functions to emit text or record graph changes available in the computations:

`emit(.)`
: writes the argument value, typically a test string, to stdout, or to a file if output was redirected.

`record(.)`
: writes the argument value, typically a text string, to the graph change record.

11.4 Embedded Exec

As in the rules (cf. chapter 10), `exec` statements may be embedded. They are executing the graph rewrite sequence (see chapter 8) contained, with access to the variables defined

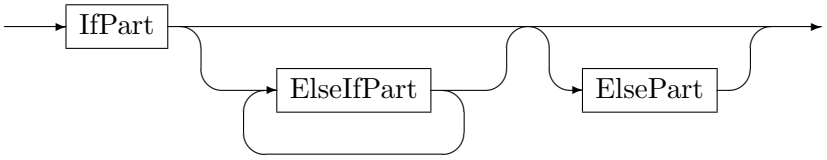
outside. This means for one reading this variables, but for the other assigning to that variables with `yield`. The embedded `exec`s allow to apply rules from the computations, while the sequence computations may call defined computations, mixing and merging declarative rule-based graph rewriting (pattern matching and dependent modifications) with traditional graph programming; in addition to the computations as such, which are already going into that direction, enriching rules with general graph programming.

`exec(.)`
: executes the graph rewrite sequence given.

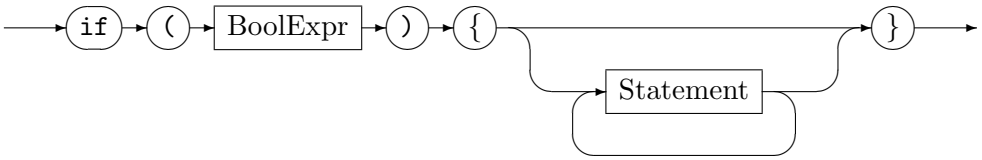
11.5 Control flow

Based on the value of a boolean expression different computations can be executed conditionally with the `if` and `else` statements. Please not that the braces are mandatory.

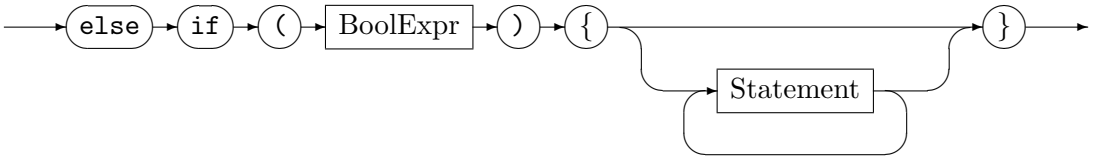
Decision



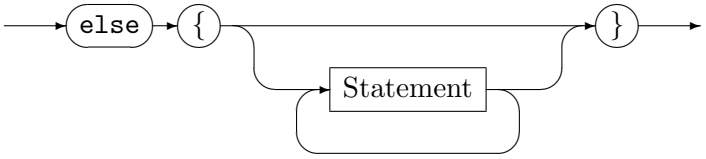
IfPart



ElseIfPart

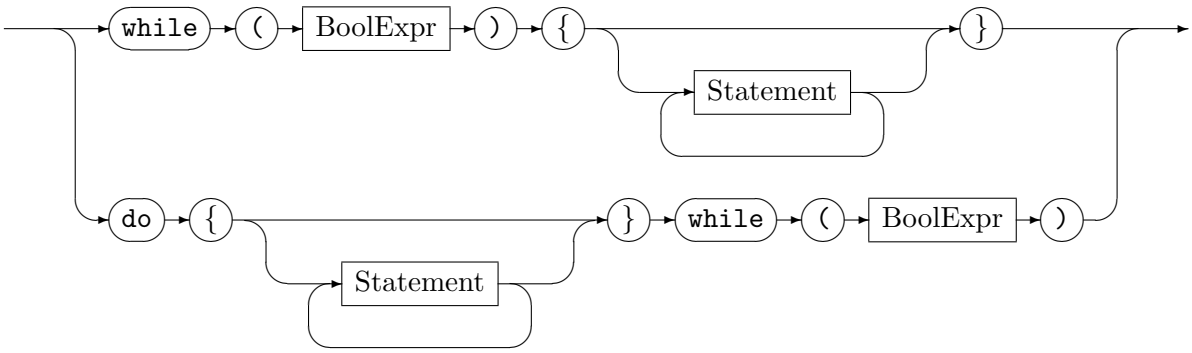


ElsePart



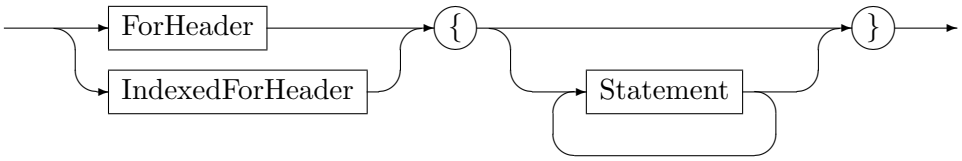
Computations may be executed repeatedly by the `while` and `do-while` statements. Please note that the braces are mandatory and that the `do-while` loop does not need a terminating semicolon.

WhileLoop



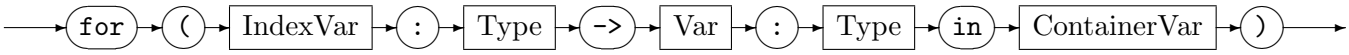
The containers introduced in chapter 12 may be iterated over with a for loop, one assigning the current value to a variable, the other assigning the current index and the current value to a variable.

ForLoop



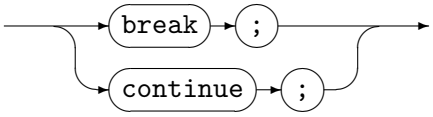
ForHeader

IndexedForHeader



Loop body execution may be exited early or continued with the next iteration with the `break` and `continue` statements. Such a statement must not occur outside of a loop.

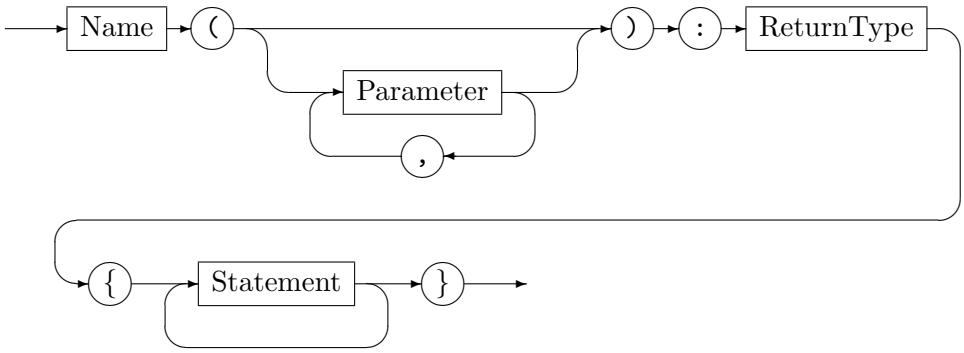
EarlyExit



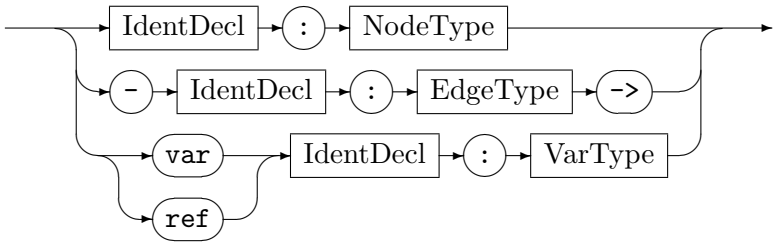
11.6 Computation definition

Computations that are occuring frequently may be factored out into a computation definition given in the header of a rule file.

ComputationDefinition



Parameter

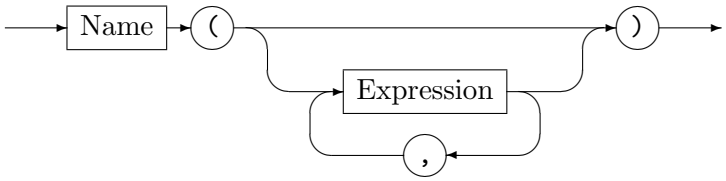


The computation definition must return a value with a return statement (so there must be at least one return statement). The type of the one return value must be compatible to the return type specified in the computation definition. A return statement must not occur outside of a computation definition.

ReturnStatement

A such defined computation may then be called as an expression atom from anywhere in the rule language file where an expression is required; or even from the sequence computations.

CallExpr



CHAPTER 12

CONTAINER TYPES AND COMPUTATIONS

12.1 Built-In Types and Concept of Containers

Besides the types already introduced, GrGen.NET supports the built-in generic types in Table 12.1. The exact type format is backend specific. The LGSPBackend maps the generic types to generic C#-Dictionaries (i.e. hashmaps) or generic C#-Lists (misnamed dynamic arrays) or a GrGen.NET supplied generic C#-Deque of their corresponding primitive types, with `de.unika.ipd.grGen.libGr.SetValueType` as target type for sets, only used with the value `null`.

`set<T>`	A (mathematical) set of type `T`, where `T` may be an enumeration type or one of the primitive types from 5.1; it may even be a node or edge type, then we speak of storages.
`map<S,T>`	A (mathematical) map from type `S` to type `T`, where `S` and `T` may be enumeration types or one of the primitive types from 5.1; it may even be a node or edge type, then we speak of storages.
`array<T>`	An array of type `T`, where `T` may be an enumeration type or one of the primitive types from 5.1; it may even be a node or edge type, then we speak of storages. Shares some similarities with `map<int,T>`.
`deque<T>`	A deque of type `T`, where `T` may be an enumeration type or one of the primitive types from 5.1; it may even be a node or edge type, then we speak of storages. Shares some similarities with `array<T>`.

Table 12.1: GrGen.NET built-in generic types

The four container types supported by GrGen share a lot of conceptual similarities and can be accessed in a similar way. They support multiple methods to update them: `add` to add an element to the container, `rem` to remove an element from the container, and `clear` to remove all elements from the container. Furthermore, they support multiple methods to query them: `size` to return the count of elements in the container, `empty` to return whether the container is empty or not, and `peek` to return an element from the container.

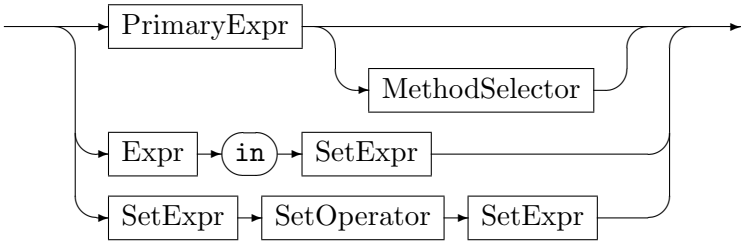
In addition to those common methods there is special syntax support available. Left associative binary expressions allow to compute a new container from two input containers, or to compare two containers. Indexed access returns an element at an index position, and indexed assignment overwrites an element at a specified position with another one. Container typed variables as such may be assigned a container, employing value semantics. Compound assignments combine a binary expression with an assignment. The `in` operator allows to query for containment. Constructor literals may be used to create and initialize containers. And finally iteration over the elements in the container is supported with a `for` loop. These operations are available in the rule language, as an extension of the expressions from 5 and the computation statements from 11; most of them are available in the sequence computations language, too (14.3 will tell about the differences compared to the rule language). In the `if` attribute evaluation clause only side-effect free container queries are allowed. In the following

the container operations will be explained in detail for one type after another.

12.2 Set Operations

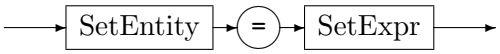
Set expressions consist of the known mathematical set operations, plus some operations in method call notation.

SetExpr

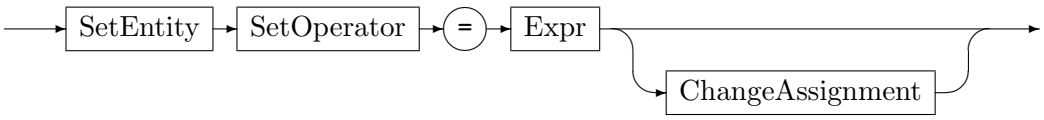


MethodCall

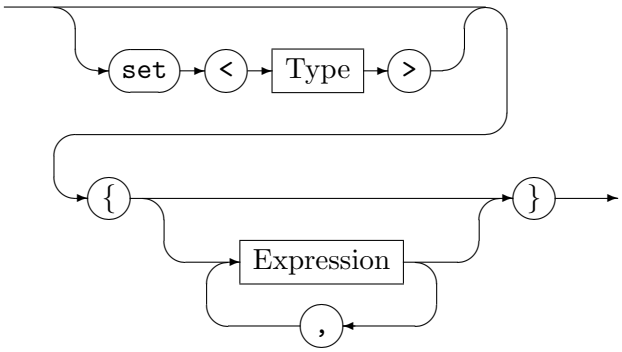
Assignment



CompoundAssignment



SetConstructor



The query method calls on sets are:

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

`.empty()`
: returns whether the set is empty, as `boolean`

`.peek(num)`
: returns the element which comes at position `num:int` in the sequence of enumeration, as `T` for `set<T>`; the higher the number, the longer retrieval takes

The update method calls on sets are:

Set addition:
`s.add(v)` adds the value `v` to the set `s`.

Set removal:
`s.rem(v)` removes the value `v` from the set `s`.

Set clearing:
`s.clear()` removes all values from the set `s`.

The binary set operators are:

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

Table 12.2: Binary set operators, in ascending order of precedence

The binary set operators require the left and right operands to be of identical type `set<T>`. The operator `x in s` denotes set membership $x \in s$, returning whether the set contains the given element, as `boolean`. Furthermore, the container may be iterated over with a `for` loop, as introduced in 11.5. The set only allows for non-indexed iteration.

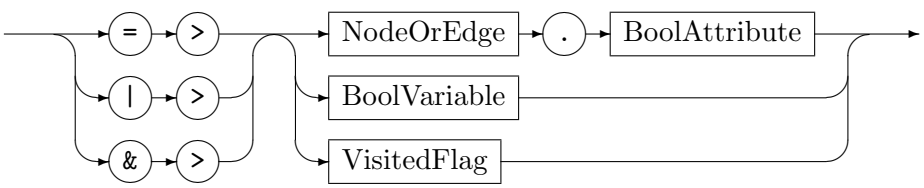
The relational expressions (already introduced in 5.4) used to compare enitites of different kinds, mapping them to the type boolean, are extended to sets according to table 12.3: Some set `A` is a subset of `B` iff all elements in `A` are contained in `B`, too.

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

Table 12.3: Binary set operators for comparison

The assignments implement the computation constructs introduced in 11. The pure assignment overwrites the target set with the source set, commonly with value semantics, creating a copy of the source set. Only if a local variable (i.e. not an attribute) is assigned to a local variable, is reference semantics used (i.e. both variables point afterwards to the same set). Compound assignments are assignments which use the first source as target, too, only adapting the target value instead of computing a new value and overwriting the target with it. For scalars this is not supported, but for container valued entities it is offered due to the potential for massive computational cost savings. The compound assignment statements on sets are a set union `|=`, intersection `&=` and difference `\=` assignment.

ChangeAssignment



The compound assignments on sets and maps may be enhanced with a change assignment declaration. The change value is `true` in case the target collection changes and `false` in case the target collection is not altered. The assign-to operator `=>` assigns the change value to the specified target, the or-to operator `|>` assigns the boolean disjunction of the change value

target with the change value to the change value target, the and-to operator `&>` assigns the boolean conjunction of the change value target with the change value to the change value target. This addition allows for efficient data flow computations not needing to check for a change by set comparison, see 16.6.

The *SetConstructor* extends the *Literal* from 5.8 (as a refinement of the *ContainerConstructor* there). It is constant if only primitive type literals, enum literals, or constant expressions are used; this is required for container initializations in the model. It is non-constant if it contains nodes/edges/or member accesses, which may be the case if used in the rules. If the type of the container is given before the constructor, the elements given in the type constructor are casted to the specified member types if needed and possible. Without the type prefix all elements given in the constructor must be of the same type.

NOTE (33)

To add a value to a set you may use set union with a single valued set constructor, to remove a value from a set you may use set difference with a single valued set constructor.

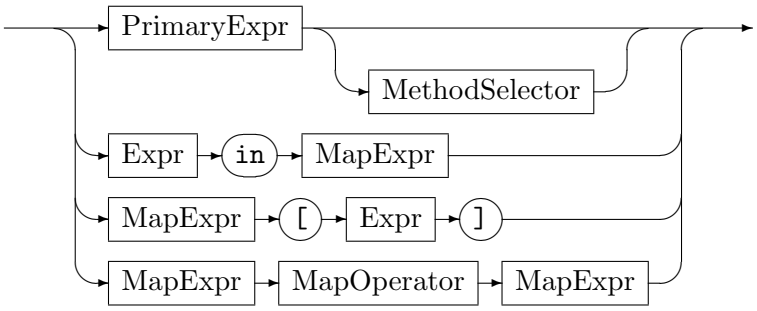
```
1 s | { "foo" }
2 s \ { n.a }
```

Used in this way they get internally optimized to the imperative set addition `s.add(x)` and removal `s.rem(x)` methods available in the `eval` block and the XGRS.

12.3 Map Operations

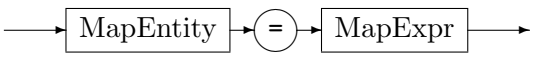
Map expressions consist of the known mathematical set operations extended to maps, and map value lookup, plus some operations in method call notation.

MapExpr

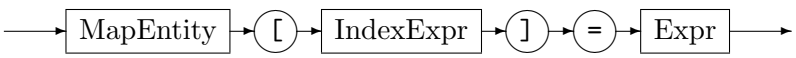


MethodCall

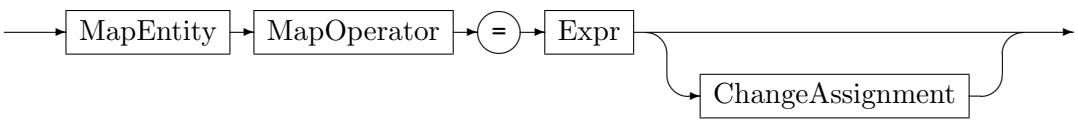
Assignment



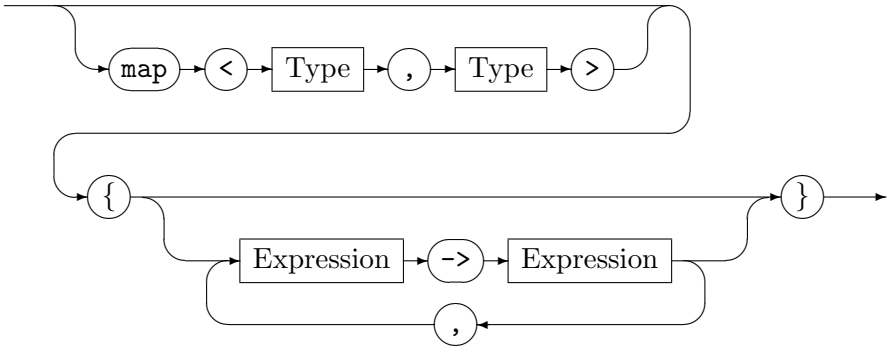
IndexedAssignment



CompoundAssignment



MapConstructor



The query method calls on maps are:

.size()
: returns the number of elements in the map, as `int`

.empty()
: returns whether the map is empty, as `boolean`

.peek(num)
: returns the key of the element which comes at position `num:int` in the sequence of enumeration, as `S` for `map<S,T>`; the higher the number, the longer retrieval takes

.domain()
: returns the set of elements in the domain of the map, as `set<S>` for `map<S,T>`

.range()
: returns the set of elements in the range of the map, as `set<T>` for `map<S,T>`

The update method calls on maps are:

Map addition:
: `m.add(k,v)` adds the pair (`k`,`v`) to the map `m`, overwrites the old value if a pair (`k`,unknown) was already existing.

Map removal:
: `m.rem(k)` removes the pair (`k`,unknown) from the map `m`.

Map clearing:
: `m.clear()` removes all values from the map `m`.

The binary map operators are:

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

Table 12.4: Binary map operators, in ascending order of precedence

The binary map operators require the left and right operands to be of identical type `map<S,T>`, with one exception for map difference, this operator accepts for a left operand

of type `map<S,T>` a right operand of type `set<S>`, too. The operator `x in m` denotes map domain membership $x \in dom(m)$, returning whether the domain of the map contains the given element, as `boolean`. The operator `m[x]` denotes map lookup, i.e. it returns the value `y` which is stored in the map `m` for the value `x` (domain value `x` is mapped by the mapping `m` to range value `y`). The value `x` *must* be in the map, i.e. `x in m` must hold. Furthermore, the container may be iterated over with a `for` loop, as introduced in 11.5. The map only allows for indexed iteration, with the key getting assigned to the index variable and the corresponding value getting assigned to the value variable.

The relational expressions (already introduced in 5.4) used to compare enitites of different kinds, mapping them to the type boolean, are extended to sets according to table 12.5: A map `A` is a submap of `B` iff all key-value pairs of `A` are contained in `B`, too. If they have a key in common but map to a different value, they count as not identical.

`A == B`	True, iff A and B are identical.
`A != B`	True, iff A and B are not identical.
`A <  B`	True, iff A is a submap of B, but A and B are not identical.
`A >  B`	True, iff A is a supermap of B, but A and B are not identical.
`A <= B`	True, iff A is a submap of B or A and B are identical.
`A >= B`	True, iff A is a supermap of B or A and B are identical.

Table 12.5: Binary map operators for comparison

The assignments implement the computation constructs introduced in 11. The pure assignment overwrites the target map with the source map, commonly with value semantics, creating a copy of the source map. Only if a local variable (i.e. not an attribute) is assigned to a local variable, is reference semantics used (i.e. both variables point afterwards to the same map). The indexed assignment `m[i]=v` overwrites the old value at index `i` in the map `m` with the new value `v`. Compound assignments are assignments which use the first source as target, too, only adapting the target value instead of computing a new value and overwriting the target with it. For scalars this is not supported, but for container valued entities it is offered due to the potential for massive computational cost savings. The compound assignment statements on maps are a map union `|=`, intersection `&=` and difference `\=` assignment.

The *MapConstructor* extends the *Literal* from 5.8 (as a refinement of the *Container-Constructor* there). It is constant if only primitive type literals, enum literals, or constant expressions are used; this is required for container initializations in the model. It is non-constant if it contains nodes/edges/or member accesses, which may be the case if used in the rules. If the type of the container is given before the constructor, the elements given in the type constructor are casted to the specified member types if needed and possible. Without the type prefix all elements given in the constructor must be of the same type.

NOTE (34)

To add a (key,value)-pair to a map you may use map union with a single valued map constructor, to remove a value from a map you may use map difference with a single valued set or map constructor.

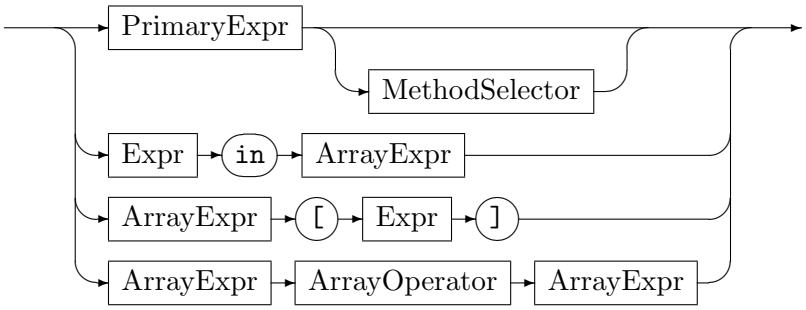
```
1 m | { "foo" -> 42 }
2 m \ { n.a -> n.b } or m \ { n.a }
```

Used in this way they get internally optimized to the imperative map addition `s.add(key,value)` and removal `s.rem(key)` methods available in the `eval` block and the XGRS.

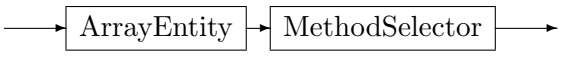
12.4 Array Operations

Array expressions consist of array membership checking, array value lookup, and array concatenation, plus some operations in method call notation.

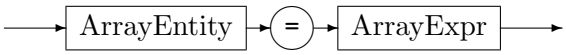
ArrayExpr



MethodCall

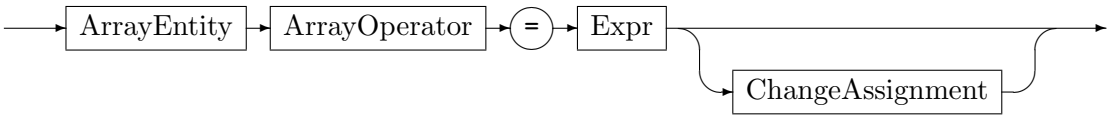


Assignment

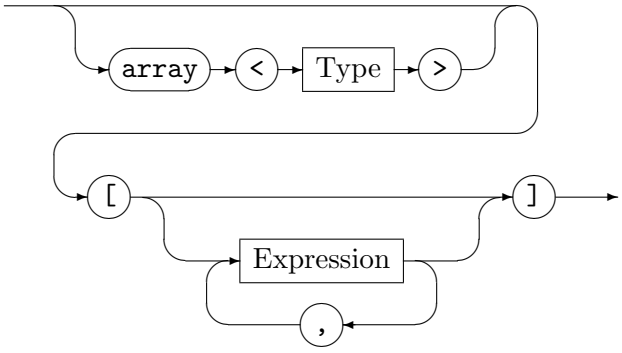


IndexedAssignment

CompoundAssignment



ArrayConstructor



The query method calls on arrays are:

`.size()`
: returns the number of elements in the array, as `int`

`.empty()`
: returns whether the array is empty, as `boolean`

`.peek(num)`
: returns the value stored in the array at position `num:int` in the sequence of enumeration, is equivalent to (and implemented by) `a[num]`); retrieval occurs in constant time.

`.peek()`
: returns the last value stored in the array; retrieval occurs in constant time.

`.indexOf(valueToSearchFor)`
: returns first position `valueToSearchFor:T` appears at, as `int`, or -1 if not found

`.lastIndexOf(valueToSearchFor)`
: returns last position `valueToSearchFor:T` appears at, as `int`, or -1 if not found

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

The update method calls on arrays are:

Array addition:
: `a.add(v)` adds the value `v` to the end of array `a`.

Array addition:
: `a.add(v,i)` inserts the value `v` at index `i` to array `a`.

Array removal:
: `a.rem()` removes the value at then end of the array `a`.

Array removal:
: `a.rem(i)` removes the value at index `i` from the array `a`.

Array clearing:
: `a.clear()` removes all values from the array `a`.

The binary array operators are:

`+`	Array concatenation: returns new array with the right appended to the left array the left and right operands must be of identical type `array<T>`

Table 12.6: Binary array operators, in ascending order of precedence

The operator `x in a` denotes array value membership, returning whether the array contains the given element, as `boolean`. The operator `a[x]` denotes array lookup, i.e. it returns the value `y` which is stored in the array `a` at the index `x`. The index `x` *must* be a valid array index. Furthermore, the container may be iterated over with a `for` loop, as introduced in 11.5. The array allows for non-indexed as well as indexed iteration; if non-indexed iteration is used the array values are iterated over, if indexed iteration is used the index is assigned to the index variable and the corrsponding value is assigned to the value variable.

The relational expressions (already introduced in 5.4) used to compare enitites of different kinds, mapping them to the type boolean, are extended to sets according to table 12.7: An array `A` is a subarray of `B` iff it is smaller or equal in size and the values at each common index are identical (lexicographic order as for strings).

`A == B`	True, iff A and B are identical.
`A != B`	True, iff A and B are not identical.
`A <  B`	True, iff A is a subarray of B, but A and B are not identical.
`A >  B`	True, iff A is a superarray of B, but A and B are not identical.
`A <= B`	True, iff A is a subarray of B or A and B are identical.
`A >= B`	True, iff A is a superarray of B or A and B are identical.

Table 12.7: Binary array operators for comparison

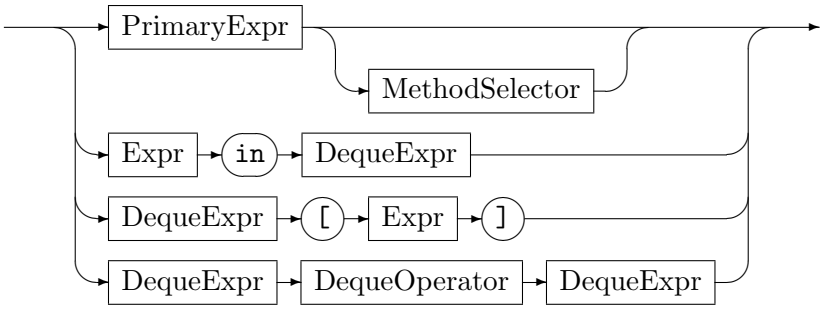
The assignments implement the computation constructs introduced in 11. The pure assignment overwrites the target array with the source array, commonly with value semantics, creating a copy of the source array. Only if a local variable (i.e. not an attribute) is assigned to a local variable, is reference semantics used (i.e. both variables point afterwards to the same array). The indexed assignment `a[i]=v` overwrites the old value at index `i` in the array `a` with the new value `v`. Compound assignments are assignments which use the first source as target, too, only adapting the target value instead of computing a new value and overwriting the target with it. For scalars this is not supported, but for container valued entities it is offered due to the potential for massive computational cost savings. The compound assignment statement on arrays is the concatenation assignment `+=`.

The *ArrayConstructor* extends the *Literal* from 5.8 (as a refinement of the *Container-Constructor* there). It is constant if only primitive type literals, enum literals, or constant expressions are used; this is required for container initializations in the model. It is non-constant if it contains nodes/edges/or member accesses, which may be the case if used in the rules. If the type of the container is given before the constructor, the elements given in the type constructor are casted to the specified member types if needed and possible. Without the type prefix all elements given in the constructor must be of the same type.

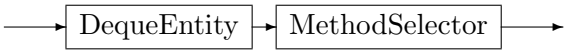
12.5 Deque Operations

Deque expressions consist of deque membership checking, deque value lookup, and deque concatenation, plus some operations in method call notation.

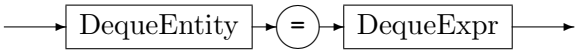
DequeExpr



MethodCall

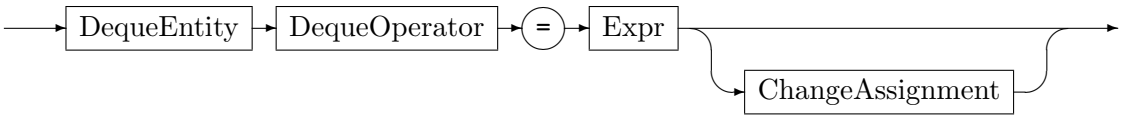


Assignment

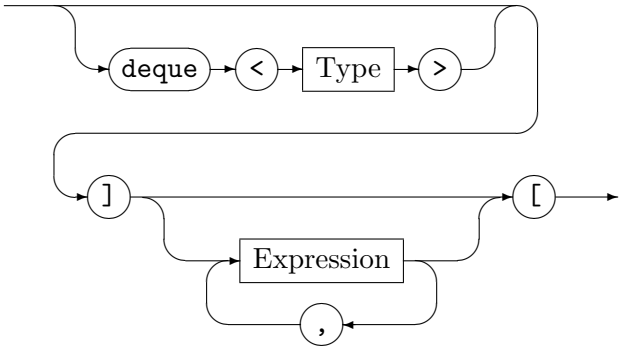


IndexedAssignment

CompoundAssignment



DequeConstructor



The query method calls on deques are:

.size()
: returns the number of elements in the deque, as `int`

.empty()
: returns whether the deque is empty, as `boolean`

.peek(num)
: returns the value stored in the deque at position `num:int` in the sequence of enumeration, is equivalent to (and implemented by) `a[num]`); retrieval occurs in constant time.

.peek()
: returns the first value stored in the deque; retrieval occurs in constant time.

.indexOf(valueToSearchFor)
: returns first position `valueToSearchFor:T` appears at, as `int`, or -1 if not found

.lastIndexOf(valueToSearchFor)
: returns last position `valueToSearchFor:T` appears at, as `int`, or -1 if not found

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

The update method calls on deques are:

Deque addition:
: `d.add(v)` adds the value `v` to the end of deque `d`.

Deque addition:
: `d.add(v,i)` inserts the value `v` at index `i` to deque `d`.

Deque removal:
: `d.rem()` removes the value at then begin of the deque `d`.

Deque removal:
: `d.rem(i)` removes the value at index `i` from the deque `d`.

Deque clearing:
: `d.clear()` removes all values from the deque `d`.

The binary deque operators are:

The operator `x in d` denotes deque value membership, returning whether the deque contains the given element, as `boolean`. The operator `d[x]` denotes deque lookup, i.e. it

`+`	Deque concatenation: returns new deque with the right appended to the left deque the left and right operands must be of identical type `deque<T>`

Table 12.8: Binary deque operators, in ascending order of precedence

returns the value `y` which is stored in the deque `d` at the index `x`. The index `x` *must* be a valid deque index. Furthermore, the container may be iterated over with a `for` loop, as introduced in 11.5. The deque allows for non-indexed as well as indexed iteration; if non-indexed iteration is used the deque values are iterated over, if indexed iteration is used the index is assigned to the index variable and the corrsponding value is assigned to the value variable.

The relational expressions (already introduced in 5.4) used to compare enitites of different kinds, mapping them to the type boolean, are extended to sets according to table 12.9: A deque `A` is a subdeque of `B` iff it is smaller or equal in size and the values at each common index are identical (lexicographic order as for strings).

`A == B`	True, iff A and B are identical.
`A != B`	True, iff A and B are not identical.
`A < B`	True, iff A is a subdeque of B, but A and B are not identical.
`A > B`	True, iff A is a superdeque of B, but A and B are not identical.
`A <= B`	True, iff A is a subdeque of B or A and B are identical.
`A >= B`	True, iff A is a superdeque of B or A and B are identical.

Table 12.9: Binary deque operators for comparison

The assignments implement the computation constructs introduced in 11. The pure assignment overwrites the target deque with the source deque, commonly with value semantics, creating a copy of the source deque. Only if a local variable (i.e. not an attribute) is assigned to a local variable, is reference semantics used (i.e. both variables point afterwards to the same deque). The indexed assignment `d[i]=v` overwrites the old value at index `i` in the deque `d` with the new value `v`. Compound assignments are assignments which use the first source as target, too, only adapting the target value instead of computing a new value and overwriting the target with it. For scalars this is not supported, but for container valued entities it is offered due to the potential for massive computational cost savings. The compound assignment statement on deques is the concatenation assignment `+=`.

The *DequeConstructor* extends the *Literal* from 5.8 (as a refinement of the *ContainerConstructor* there). It is constant if only primitive type literals, enum literals, or constant expressions are used; this is required for container initializations in the model. It is non-constant if it contains nodes/edges/or member accesses, which may be the case if used in the rules. If the type of the container is given before the constructor, the elements given in the type constructor are casted to the specified member types if needed and possible. Without the type prefix all elements given in the constructor must be of the same type.

NOTE (35)

The double ended queue allows for fast addition and removal both at the frond and at the back, in contrast to arrays that only support this at the end. It is implemented by a ringbuffer that is grown as needed, a lookup is slightly more expensive than an array lookup. The primary usage of the deque is as a queue, as needed for breadth first search, accessed FIFO. This is in contrast to the array that is suited to be employed as a stack, e.g. in a depth first search (unless that is programmed using the call stack), accessed LIFO.

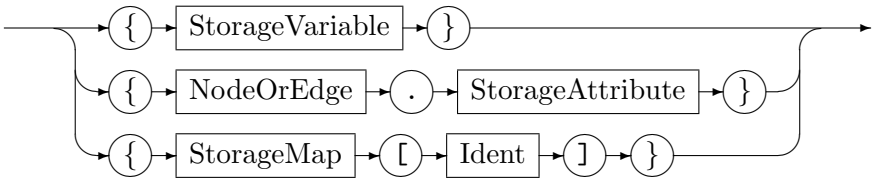
12.6 Storage Access in the Rules

Storages can be used in the rule application control language as introduced above 14.3, they can get filled or emptied in the rules as defined here 4.4.3, a discussion about their usage and examples are given here 16, here 16.5, and here 16.6. In the pattern part you may ask for an element to get bound to an element from a storage or a storage attribute; this is syntactically specified by giving the storage enclosed in left and right braces. You may ask for an element to get bound to the value element queried from a storagemap by a key graph element; this is syntactically specified by giving the storagemap indexed by the key graph element enclosed in left and right braces (this is not possible for storage map attributes due to internal limitations with the search planning). If the type of the element retrieved from the storage is not compatible to the type of the pattern element specified, or if the storage is empty, or if the key element is not contained in the storagemap, matching fails.

The advantage of this storage querying inside the rule over handing in a value from a for loop iterating the storage values outside the rule are: i) a more concise syntax, ii) the ability to access a storage attribute of an element just matched or to access a storage map with an element just matched in the same rule, which would require to break up the rule in two rules in the other case, and iii), a restriction of the iteration to the matching phase, so that at rewriting one can happily manipulate the storage without destroying the iterator/enumerator used in the loop which would be the case when using an outside loop.

The following syntax diagram gives an extensions to the syntax diagrams of the Rule Set Language chapter 4, pattern part:

StorageAccess



EXAMPLE (60)

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

```
test neighbour(ref startCities:set<City>) : City
{
    :City{startCities} -:Street-> n:City;
    return(n);
}
```

EXAMPLE (61)

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

```
test neighbourneighbour(ref neighbours:map<City, City>) : City
{
    someCity:City;
    nc:City{neighbours[someCity]} -:Street-> nnc:City;
    return(nnc);
}
```

12.7 Hints on container usage

EXAMPLE (62)

The container state change methods `add` and `rem` allow to add graph elements to storages or remove graph elements from storages, i.e. sets or maps or arrays or deques holding nodes and edges used for rewrite in the calling sequence (cf. 14.3). This way you can write transformations consisting of several passes with one pass operating on nodes/edges determined in a previous pass, without the need to mark the element in the graph by helper edges or visited flags.

```
1 rule foo(ref storage:set<Node>)
2 {
3   n:Node;
4   modify {
5     eval {
6       storage.add(n);
7     }
8   }
9 }
```

EXAMPLE (63)

Some examples of container literals:

```
1 { "foo", "bar" } // a constant set<string> constructor
2 map<string,int>{ (n.strVal+m.strVal)->(m.intVal+n.intVal), intVal->strVal, "fool"->42 } // a
      non-constant map constructor with type prefix
3 [ 1,2,3 ] // a constant array<int> constructor
4 ] 1,2,3 [ // a constant deque<int> constructor
```

CHAPTER 13

GRAPH TYPE AND COMPUTATIONS

13.1 Built-In Types

Besides the types already introduced, GrGen.NET offers supports for a `graph` type. If used explicitely, it denotes a subgraph of the host graph, which can be used for storing and comparing subgraphs of the current host graph (after subgraph extraction no further graph operations are possible (unless the API is used), because of the "there is only a single host graph"-design of GrGen).

But more important are the large number of global functions (with function call syntax as already inroduced in 5.8, semantically distinguished into *GlobalGraphFunctionCall*, cf. 11) that implicitely operate on the single host graph (and thus on the graph type). There are update functions available that allow to manipulate the graph available with `add`, `rem` and `retype`. The host graph may be queried for all `nodes` or `edges` of a given type. An edge may be queried for its `source` and `target` elements, while a node may be queried for its direct neighbourhood with all `incident` edges, or all `adjacent` nodes. Or even for its transitive neighbourhood with all `reachable` nodes or edges. In the form of functions returning `set`s of nodes or edges, or in the form of iterations with `for` loops, or in the form of boolean predicates to test the neighbourhood if two elements are given. The `induced` subgraph of a set of nodes or edges may be computed, or directly `insert`ed into the host graph.

Furthermore, `visited` flags may be used for marking already visited elements during graph walks or for partitioning a graph. These operations are available in the rule language, as an extension of the expressions from 5 and the computation statements from 11; most of them are available in the sequence computations language, too (14.1 and 14.1 will tell about the differences compared to the rule language).

13.2 Global Graph Functions

13.2.1 Graph Updates / Basic Graph Manipulation

There are functions to update the graph by adding or removing or retyping available, on nodes and edges:

`add(.)`
: adds a new node of the given node type to the host graph, returns the added node.

`add(.,.,.)`
: adds a new edge of the given edge type to the host graph, in between the source node specified as second argument and the target node specified as third argument, returns the added edge.

`rem(.)`
: removes the node or edge given from the host graph (no expression, does not return anything).

retype(.,.)
: retypes the node or edge given to the node type or edge type given as second argument, returns the retyped entity.

clear()
: clears the host graph.

Besidse those basic graph manipulation functions, the advanced rewriting operations are available as statements, too:

merge(.,.)
: merges the source node given as second argument into the target node given as first argument.

redirectSource(.,.)
: redirects the edge given as first argument to the new source node given as second argument.

redirectTarget(.,.)
: redirects the edge given as first argument to the new target node given as second argument.

redirectSourceAndTarget(.,.)
: redirects the edge given as first argument to the new source node given as second argument and the new target node given as third argument.

The versions introduced above are only available on named graphs, as they fetch the debug display name from the old element. If you want to use them on unnamed graphs you must supply an additional argument giving the name of the old element; in case of the `redirectSourceAndTarget` you must supply two additional arguments, first the string to use for the old source node name, then the string to use for the old target node name.

13.2.2 Graph Query by Types

There are functions to ask for all nodes or edges of a type available:

nodes(.)
: returns all nodes in the graph compatible to the given node type, as set.

nodes()
: returns all nodes in the graph, as set.

edges(.)
: returns all edges in the graph compatible to the given edge type, as set.

edges()
: returns all edges in the graph, as set.

The same functions can be used from for loops to iterate over the entities, omitting the filling of a set:

for(n:NodyType in nodes(NodeType)) {Statements}
: iterates over all nodes in the graph compatible to the given node type.

for(n:Node in nodes()) {Statements}
: iterates over all nodes in the graph.

for(e:EdgeType in edges(EdgeType)) {Statements}
: iterates over all edges in the graph compatible to the given edge type.

for(e:Edge in edges()) {Statements}
: iterates over all edges in the graph.

13.2.3 Graph Query by Neighbourhood

Multiple functions are available to query the neighbourhood of nodes and edges.

Edge Neighbourhood

The nodes incident to a given edge may be queried by the following functions:

source(.)
: returns the source node of the given edge.

target(.)
: returns the target node of the given edge.

opposite(.,.)
: returns the opposite node of the edge and the node (second argument) given.

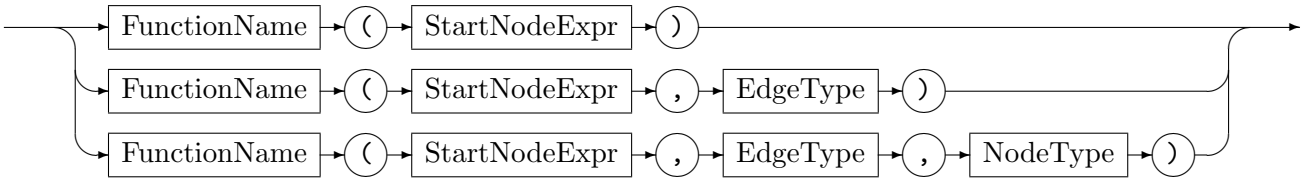
Node Neighbourhood Common Concepts

The edges `incident` or the nodes `adjacent` to a given node may be queried.

The neighbourhood query functions allow to additionally constrain the direction of the edges to `incoming` or `outgoing` edges, otherwise both directions are accepted.

The neighbourhood query functions furthermore allow to constrain the accepted situations by optional arguments. The type the incident edges must have to be accounted for can be specified. Or the type the incident edges must have to be accounted for and the type the adjacent nodes must have to be accounted for (cf. 13.2.3).

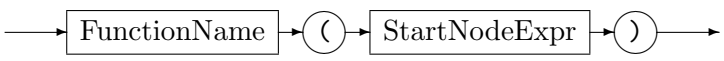
NeighbourhoodFunctionCall



The neighbourhood query function can be used in three possible ways:

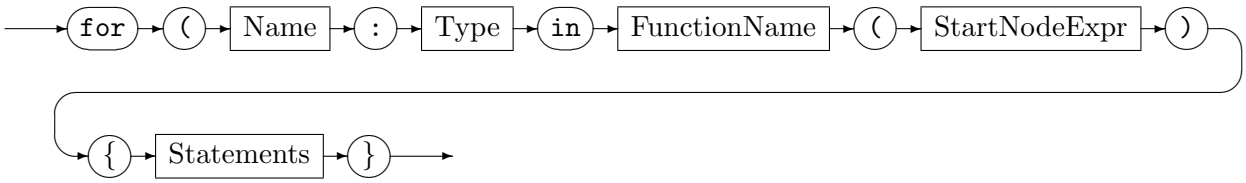
Set functions:
: The neighbourhood function returns a set of neighbouring entities of the start node. It builds a set that is likely thrown away thereafter, so esp. for large sets this functions is less efficent than the other versions.

  Expression



Iteration loops:
: The neighbourhood function is employed from a for loop that allows to iterate the neighbouring entities of the start node. No set needs to be built here. But if the source node has multiple edges to a target node, it might be iterated multiple times. And an edge may be iterated twice in case of the undirected functions.

ForLoop



Boolean functions:

The neighbourhood function is employed in a boolean predicate version that allows the check whether a second target entity is in the queried neighbourhood of the start node. The computation has the smallest internal processing overhead of the three options and stops as soon as a positive result is obtained.

Expression

The *FunctionNamePredicate* is built from the *FunctionName* by prepending `is` and switching the first charecter of *FunctionName* to upper case.

Direct Node Neighbourhood

Available are queries for the neighbouring edges:

`incident(.)`
: returns the set of the edges that are incident to the node given as argument value.

`incoming(.)`
: same as the incident above, but restricted to incoming edges.

`outgoing(.)`
: same as the incident above, but restricted to outgoing edges.

In the two argument version, only edges of the type given as second argument are contained. The three argument version behaves as the two argument version, but additionally only edges incident to an opposite node of the type given as third argument are contained.

EXAMPLE (64)

```
1  rule foo {
2      src:Node -e:Edge->;
3      if(!isIncoming(src, e));
4
5      modify {
6          eval {
7              if(incident(src, NiftyEdge, NiftyNode).size()>2)
8              {
9                   for(ne:NiftyEdge in outgoing(src, NiftyEdge))
10                  {
11                      ne.attr = 42;
12                  }
13              }
14         }
15     }
16 }
```

Some fabricated example showing how to use the incoming, outgoing, and incident functions in their boolean predicate, set function, and for iteration versions, with and without constraining the edge and node types.

Available are queries for the neighbouring nodes:

adjacent(.)
: returns the set of the nodes that are adjacent to the node given as argument value.

adjacentIncoming(.)
: same as the adjacent above, but restricted to nodes reachable via incoming edges.

adjacentOutgoing(.)
: same as the adjacent above, but restricted to nodes reachable via outgoing edges.

In the two argument version, nodes incident to an edge of the type given as second argument are contained. The three argument version behaves as the two argument version, but additionally only nodes of the node type given as third argument are contained.

Transitive Node Neighbourhood

Besides direct neighbourhood, transitive neighbourhood can be queried with the reachability functions.

Available are queries for the reachable edges:

reachableEdges(.)
: returns the set of the edges that are reachable from the node given as argument value.

reachableEdgesIncoming(.)
: same as the reachableEdges above, but restricted to incoming edges.

reachableEdgesOutgoing(.)
: same as the reachableEdges above, but restricted to outgoing edges.

In the two argument version, only edges of the type given as second argument are contained and followed. The three argument version behaves as the two argument version, but additionally only edges incident to an opposite node of the type given as third argument are contained and followed.

Available are queries for the reachable nodes:

reachable(.)
: returns the set of the nodes that are reachable from the node given as argument value.

reachableIncoming(.)
: same as any of the reachables above, but restricted to nodes reachable via incoming edges.

reachableOutgoing(.)
: same as any of the reachables above, but restricted to nodes reachable via outgoing edges.

In the two argument version, nodes incident to an edge of the type given as second argument are contained and followed. The three argument version behaves as the two argument version, but additionally only nodes of the node type given as third argument are contained and followed.

EXAMPLE (65)

```
rule bar {
    src:Node;
    tgt:Node;
    if(isReachableOutgoing(src, tgt, NiftyEdge));

    modify {
        eval {
            if(!(reachableIncoming(src) & reachableOutgoing(src)).empty())
            {
                for(ne:NiftyEdge in reachable(src))
                {
                    ne.attr = 42;
                }
            }
        }
    }
}
```

Some fabricated example showing how to use the `isReachableOutgoing` function to check for an iterated path between the `src` and `target` nodes, how to check for loops by interesecting the sets of nodes reachable by outgoing edges from a node and reachable by incoming edges to a node, and how to iterate with one loop over all edges reachable in either way from a node.
The isReachable functions give the most efficient and most convenient way to check for an iterated path in GrGen, if you need more elaborate checking than constraining the edge type to one type and the taget node type to one type you need to program the iterated path with subpattern recursion.
The reachable iteration is the most concise way to note down a depth first walk over a graph, visiting all elements reachable from a source node on.

13.2.4 Induced Subgraph Computation and Insertion to Host Graph

Functions that allow to compute (node-or-edge) induced subgraphs, and to insert clones of induced subgraphs are available. They are especially useful in state space enumeration, cf. 16.7.

inducedSubgraph(.)
: returns the induced subgraph (type: `graph`) of the host graph for the set of nodes given as argument value.

definedSubgraph(.)
: returns the defined (edge-induced) subgraph (type: `graph`) of the host graph for the set of edges given as argument value.

insertInduced(.,.)
: adds a clone of the subgraph induced by the set of nodes given as first argument to the host graph, returns the clone of the anchor node given as second argument.

insertDefined(.,.)
: adds a clone of the subgraph defined (edge-induced) by the set of edges given as first argument to the host graph, returns the clone of the anchor edge given as second argument.

13.3 Graph comparison

Here we extend the relational expressions already introduced in 5.4 (and already extended in 12 to include container types) with the (sub)graph type.

`A == B`	True, iff A is isomorphic to B.
`A != B`	True, iff A is not isomorphic to B.
`A ~~ B`	True, iff A is structurally the same as B but maybe different regarding the attributes.

Table 13.1: Compare operators on graph expressions

The `graph` type support the `==`, the `!=`, and the `~~` operators; on (sub)graph types they tell whether the (sub)graphs are isomorphic to each other (isomorphy checking/graph isomorphy checking) or not, including the attributes, or whether the (sub)graphs are isomorphic disregarding the attributes.

These operators consist just of two charactes, but don't underestimate their impact on performance: they do graph isomorphy checking, which is expensive. They are implemented in an early out style, i.e. the more different the graphs are, the earlier does the check return with the result they are not isomorphic. But if the graphs you are checking are isomorphic (which will happen easily if you use automorphic patterns), then you have to pay the full price for isomorphy checking; if this occurs often, your solution may become prohibitively costly (including an external graph canonization library or the `canonize` function may be of interest in that case).

Some notes on the early out implementation: first the number of nodes and edges per type are checked, if they are different the graphs can't be isomorphic. The numbers are directly supplied by the `lgspBackend`, refuting isomorphy based on them is extremely cheap. Then the V-Structures (see 22.3) used in computing better matchers at runtime are first computed and then compared; if they are different the graphs can't be isomorphic. They are a good deal less expensive to compute than trying to match the one graph in the other; on well typed graphs the V-Structure counts are highly discriminating.

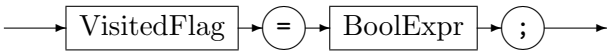
If these two pruning methods failed, a matcher is computed from one graph with search planning based on the V-Structure information just computed, and then applied on the other graph. The matchers are stored in the graphs from which they originated, so if you do repeated comparisons of a subgraph which does not change, take care to extract that subgraph only once, store it, e.g. as an attribute in the graph, and continue to compare against it. This will save you from the cost of repeated search planning; in addition, often-used isomorphy matchers get eventually compiled resulting in a further speed-up.

13.4 Visited Flags

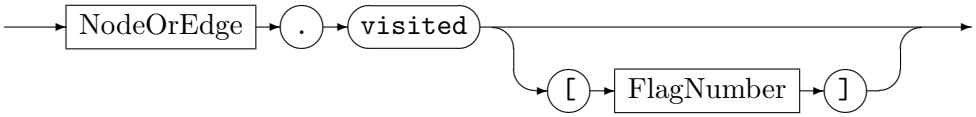
The boolean `visited` flags may be used for marking already visited graph elements during graph walks or for partitioning a graph. They may be queried with a function of the expressions, and set in the computations (extended attribute evaluation functions). The following syntax diagram gives an extensions to the *Expression* clause of chapter 5 and an extension to the computation *Statements* of chapter 11:

Expression

VisitedAssignment



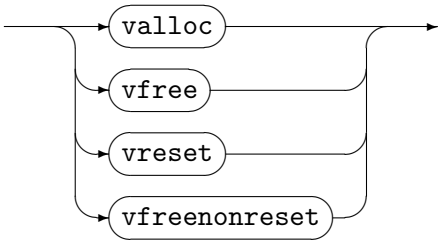
VisitedFlag



The `visited` flag expression returns the `boolean` status of the visited flag of the given number for the given graph element. The `visited` flag assignment sets the status of the visited flag of the given number for the given graph element to the value defined by the `boolean` expression. If no `int` flag number is given, the default number for the first visited flag of 0 is used.

The visited flags are stored in some excess bits of the graph elements, if this pool is exceeded they are stored in additional dictionaries, one per visited flag requested. Due to this flags must get allocated before they are used and deallocated afterwards, and all flag related operations require an integer number – the flag id – specifying the flag to operate on (with exception of the allocation operation returning this flag id). If you try to access a not previously allocated visited flag, an exception is thrown.

FunctionName



The operations managing the visited flags are:

Flag allocation:
: By `valloc()` – allocates space for a visited flag in the elements of the graph and returns the id of the visited flag (integer number), starting at 0. Afterwards, the visited flag of the id can be read and written by the `visited`-expression and the `visited`-assignment. The first visited flags are stored in some excess bits of the graph elements and are thus essentially for free, but if this implementation defined space is used up completely, the information is stored in graph element external dictionaries.

Flag deallocation:
: By `vfree` – frees the space previously allocated for the visited flag; afterwards you must

not access it anymore. The value passed in `vfree(IntExpr)` must be of integer type, stemming from a previous allocation. This function internally calls a `vreset` to ensure that no corresponding visited flag is set in the graph.

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

Flag deallocation without reset:
With `vfreenonreset` the space previously allocated for the visited flag is freed, too, but the implicit internal `vreset(id)` of `vfree` is not executed. It is your duty to ensure the flag is `false` in all graph elements – otherwise after a following allocation elements may start as being marked. This saves us an O(n) operation, but opens the door to nasty bugs if you can't design your algorithm in a way which renders unmarking trivial.

13.5 Transaction Handling

In addition to the functions implemented directly in the graph there are the transaction manager functions available, implemented in the graph processing environment. While a transaction is underway, an undo log is filled with commands to undo the changes that occured in the graph in the meantime. Those transaction handling functions are:

`startTransaction()`
starts a transaction and returns its transaction id (as number of type `int`).

`pauseTransaction()`
pauses transaction handling so changes are not recorded and can't be undone.

`resumeTransaction()`
resumes paused transaction handling.

`commitTransaction(.)`
keeps the changes of the transaction of the given id (of type `int`) in the graph, removing undo information.

`rollbackTransaction(.)`
reverts the changes of the transaction of the given id (of type `int`) in the graph by executing the undo log.

Please note that transactions may be nested; those functions are used in implementing the transaction and backtracking constructs explained in 15.2.

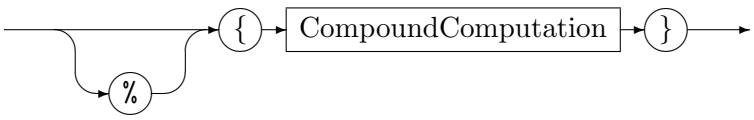
CHAPTER 14

SEQUENCE COMPUTATIONS

In this chapter we'll have a look at sequence computations, which are not concerned with directly controling rules, but with computing values or creating side effects.

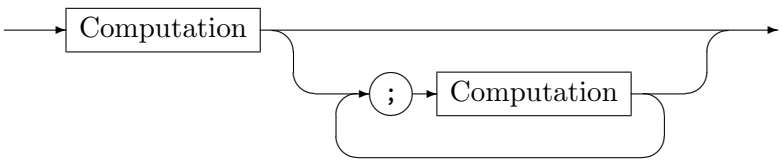
14.1 Sequence Computation

RewriteComputationUsage



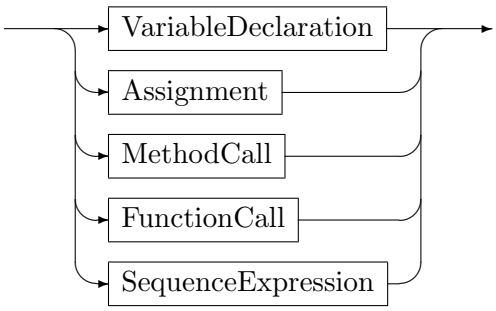
The non-computation constructs introduced before are used for executing rules, to determine which rule to execute next depending on success and failure of the previous rule applications, and where to apply it next by transmitting simple valued variables in between the rules. Sequence computations in contrast are used for manipulating complex valued variables, evaluating computational expressions, or for causing side effects like output or element markings. The computation will return a boolean value by comparing the return value of the compound computation to the default value of the corresponding type, and returning false if equal, or true if unequal; a computation without a return value always returns true. So just using a boolean variable as computation returns the value of the variable. A prepended `%` attaches a break point to the computation.

CompoundComputation

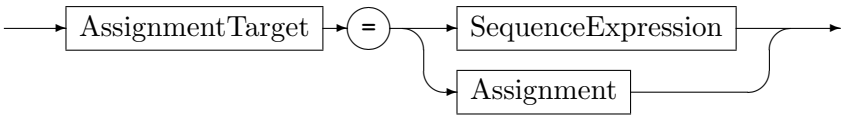


A compound computation consists of a computation followed by an optional list of computations separated by semicolons. The computations are executed from left to right; the value of the compound computation is the value of the last computation (so you must give an expression there in order to return a value, whereas it is pointless to specify an expression before).

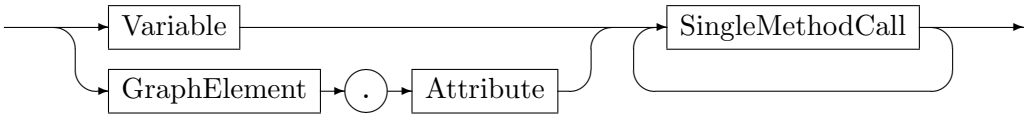
Computation



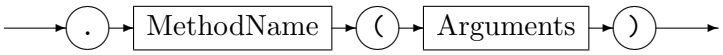
Assignment



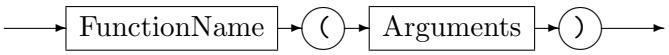
MethodCall



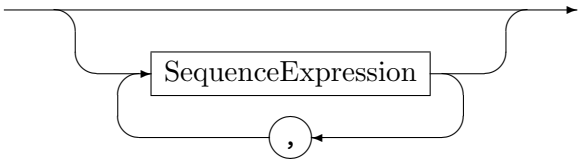
SingleMethodCall



FunctionCall



Arguments



A variable declaration declares a local variable in the same way as in the sequences. An assignment assigns the value of a sequence expression to an assignment target. It may be chained; such an assignment chain is executed from right to left, assigning the rightmost value to all the assignment targets given. The form of expressions and assignment targets will be specifed below.

A method call executes a (predefined) method on a variable, passing further arguments. It may be chained; such a method call chain is executed from left to right. This is possible with storage changing methods which return the variable again, better: which return the then altered variable. They were already explained in 14.3.

A function call executes a (predefined) function, passing further arguments. In addition to the visited flag functions which were already explained in more detail in 14.2, `emit`, `record`, and `export` function calls can be given here: the emit function writes a double quoted string or the value of a variable to the emit target (stdout as default, or a file specified with the shell command `redirect emit`). The record function writes a double quoted string or the value of a variable to the currently ongoing recordings (see 17.2.5). This feature allows to mark states reached during the transformation process in order to replay only interesting parts of an recording. It is recommended to write only comment label lines, i.e. `"#"`, some label,

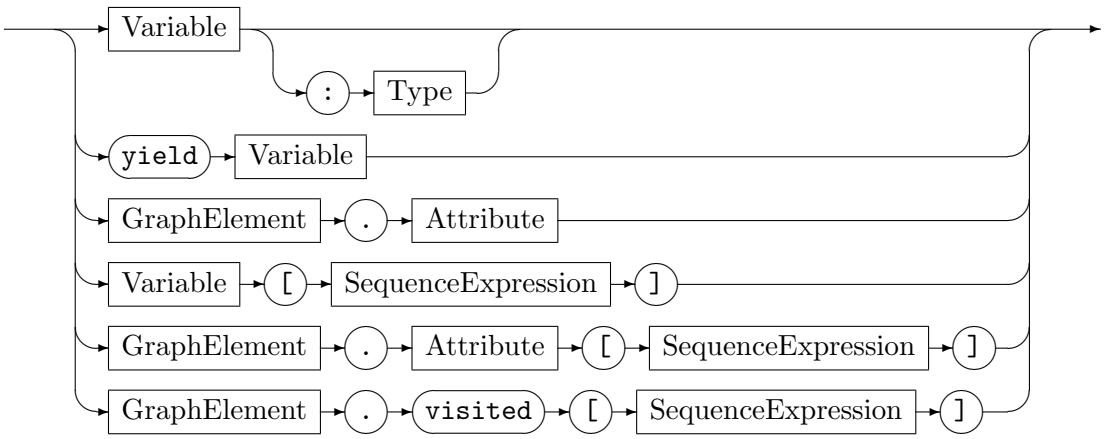
and `"\n"`. The export function exports the current graph to the path specified if called with one argument, or it exports the subgraph specified as first argument to the path specified as second argument. It behaves like the export command from the GrShell, see 17.2.4. Having it available in the sequences allows for programmed exporting, and exporting of parts of the graph, with the subgraph containment just computed.

Furthermore, a function `canonize(g:graph):string` is available, which is intended to provide a canonical string representation for any graph, but currently does not work for all graphs. The function currently uses the SMILES[Wei88] method of producing an equitable partition of graph nodes, not a canonical order; while not offering full fledged graph conanization this algorithm is sufficient for many purposes. It allows to reduce graph comparisons to string comparisons, at the price of computing the Weininger algorithm for equitable partitions of nodes.

Besides those predefined functions, you may call computation functions, defined in the rules file; cf. 11.6.

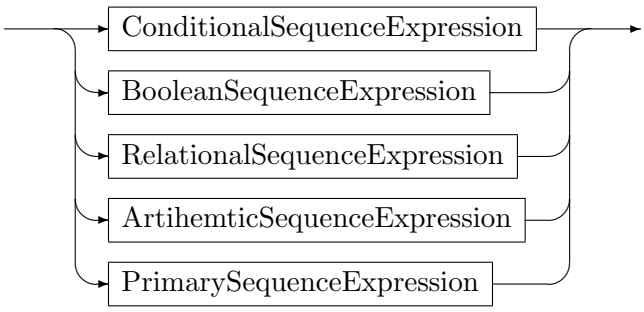
Finally, an expression (without side effects) can be evaluated, this allows to return a (boolean) value from a computation.

AssignmentTarget



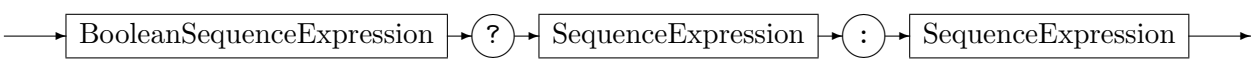
Possible targets of assignments are the variables and def-variables to be yielded to, as in the simple assignments of the sequences. A `yield` assignment writes the rhs variable value to the lhs variable which must be declared as a def-to-be-yielded-to variable (`def`-prefix) in the pattern containing the `exec` statement. Yielding is only possible from compiled sequences, it always succeeds. Further on, the attributes of graph elements may be written to, the values at given positions of array or deque or map variables may be written to, and the visited status of graph elements may be changed.

SequenceExpression



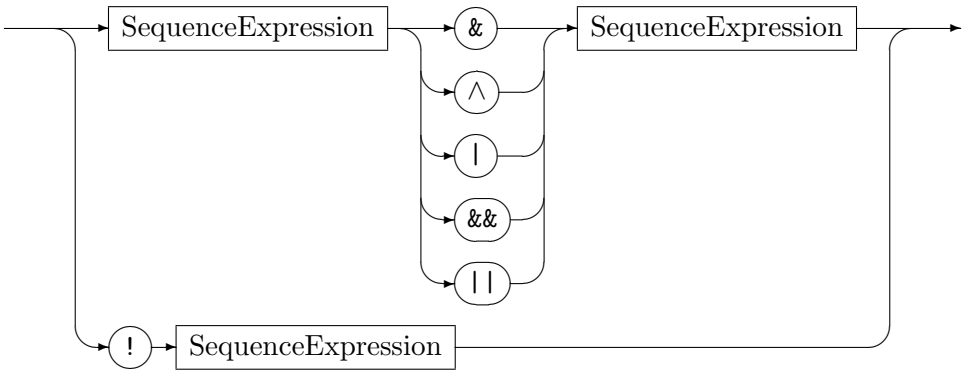
Sequence expressions are basically a subset of the expressions introduced in 5.2.

ConditionalSequenceExpression



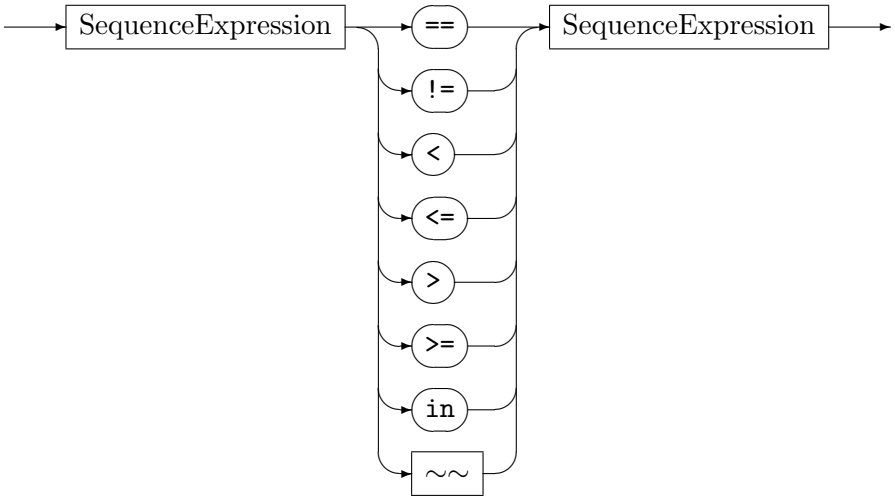
The conditional operator has lowest priority, if the condition evaluates to true the first expression is evaluated and returned, otherwise the second.

BooleanSequenceExpression



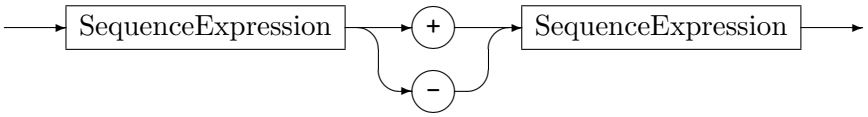
The boolean operators have the same semantics and same priority as in 5.2.

RelationalSequenceExpression



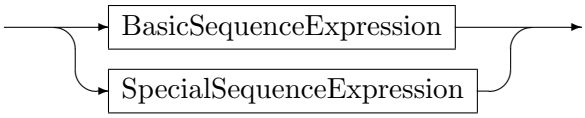
The equality operators work for every type and return whether the values to compare are equal or unequal. The relational operators on numerical, graph, and container types work as specified in 5.2 and 12.

ArithmeticSequenceExpression



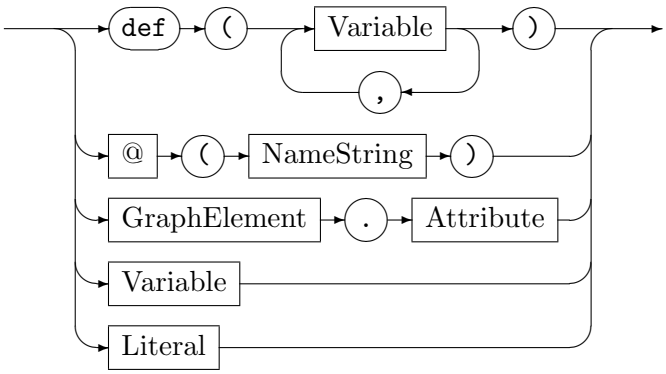
The arithemtic operator plus is used to denote addition of numerical values or string concatenation, the arithemtic operator minus is used to denote subtraction of numerical values.

PrimarySequenceExpression



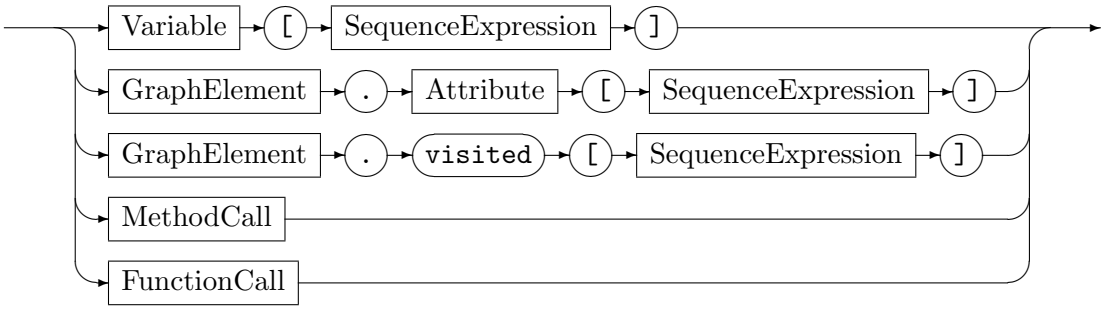
The atoms of the expressions are the basic and the special sequence expressions.

BasicSequenceExpression



The basic sequence expressions are the building blocks of the computation sequences. A `def` term is successful iff all the variables are defined (not null). The at operator allows to access a graph element by its persistent name. The attribute access clause returns the attribute value of the given graph element. The variable and literal basic expressions are the same as in the SimpleOrInteractiveExpression; this means esp. that a Variable may denote a graph global variable if prefixed with a double colon as in 8.3, here as well as in the AssignmentTarget.

SpecialSequenceExpression



The special sequence expressions are used for storage and visited flag handling, for random value queries, and for graph and subgraph handling.

The storage and visited flag oriented ones are used to check whether a value is marked, to access a storage, or to call a method on a storage (note: here it is not possible to build method call chains). They were explained in the chapters 12 and 13.

The random value function `random` behaves like the random function from the expressions, see 5.8; i.e. if noted down with an integer as argument it returns a random integer in between 0 and that upper bound, exclusive; if given without an argument it returns a random double in between 0.0 and 1.0, exclusive.

The graph and subgraph oriented ones can be separated into three groups.

Basic Graph Manipulation

TODO: reduce to diff compared to chapters before

The first group is built from basic graph manipulation operators used for adding or removing elements:

add(.)
: creates a node of the given type and adds it to the host graph.

add(.,.,.)
: creates an edge of the given type and adds it to the host graph, starting at the node given as second argument, ending at the node given as third argument.

rem(.)
removes the given node or edge from the graph.

clear()
clears the host graph.

Connectedness Queries

The second group is built from the operators querying primarily the connectedness of graph elements. See 13.2 for more on them; they were moved to an own chapter, because besides being available in the expressions of the sequence computations we take care of here, they are available in the expressions of the rules, too.

Subgraph Operations

The third group is defined by functions which operate on (sub-)graphs: It contains functions which allow to compute (node-or-edge) induced subgraphs, and to insert clones of induced subgraphs. They are especially useful in state space enumeration, cf. 16.7.

inducedSubgraph(.)
returns the induced subgraph (type: `graph`) of the host graph for the set of nodes given as argument value.

definedSubgraph(.)
returns the defined (edge-induced) subgraph (type: `graph`) of the host graph for the set of edges given as argument value.

insertInduced(.,.)
adds a clone of the subgraph induced by the set of nodes given as first argument to the host graph, returns the clone of the anchor node given as second argument.

insertDefined(.,.)
adds a clone of the subgraph defined (edge-induced) by the set of edges given as first argument to the host, returns the clone of the anchor edge given as second argument.

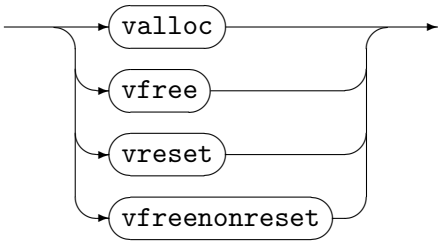
14.2 Visited Flag Handling in the Sequences

TODO: reduce to diff compared to chapters before

Visited flags are flags available for/in each graph element which can be set, reset, and queried in the rules and in the sequences and must be allocated and deallocated in the sequences; they allow to mark already visited elements during a run over the graph. The syntax diagrams given in the Rule Application Control Language chapter 8 already contained the visited flag constructs. Here we only give some refinements and explanations of the semantics.

The visited flags are stored in some excess bits of the graph elements, if this pool is exceeded they are stored in additional dictionaries, one per visited flag requested. Due to this flags must get allocated/deallocated, and all flag related operations require an integer number – the flag id – specifying the flag to operate on (with exception of the allocation operation returning this flag id). The operations always return true as sequence results (with exception of the operation reading the flag, it fails iff the visited flag is not set for the graph element); if you try to access a not previously allocated visited flag, an exception is thrown.

FunctionName



The operations managing the visited flags are:

Flag allocation:
By `valloc` – allocates space for a visited flag in the elements of the graph and returns the id of the visited flag (integer number), starting at 0. Afterwards, the visited flag of the id can be read and written within the rules by the `visited`-expression and the `visited`-assignment, as well as by the `visited` flag reading and writing rewrite factors. The first visited flags are stored in some excess bits of the graph elements and are thus essentially for free, but if this implementation defined space is used up completely, the information is stored in graph element external dictionaries. Visited flag allocation is only possible in sequence computations in the form of `var=valloc()`.

Flag deallocation:
By `vfree` – frees the space previously allocated for the visited flag; afterwards you must not access it anymore. The value passed in `vfree(var)` must be of integer type, stemming from a previous allocation.

Flag writing:
By `e.visited[f] = b` – sets the visited status of the flag given by the flag id variable `f` of the graph element `e` to the given boolean value `b`; visited flags are normally written by `eval` parts of the rule language.

Flag reading:
By `e.visited[f]` – returns the visited status of the flag given by the flag id variable `f` of the graph element `e`; visited flags are normally read by `if` conditions of the rule language.

Flag resetting:
By `vreset` – resets the visitor flag given by the flag id variable in all graph elements.

Flag deallocation without reset:
With `vfreenonreset` the space previously allocated for the visited flag is freed, too, but the implicit internal `vreset(id)` of `vfree` is not executed. It is your duty to ensure the flag is `false` in all graph elements – otherwise after a following allocation elements may start as being marked. This saves us an O(n) operation, but opens the door to nasty bugs if you can't design your algorithm in a way which renders unmarking trivial.

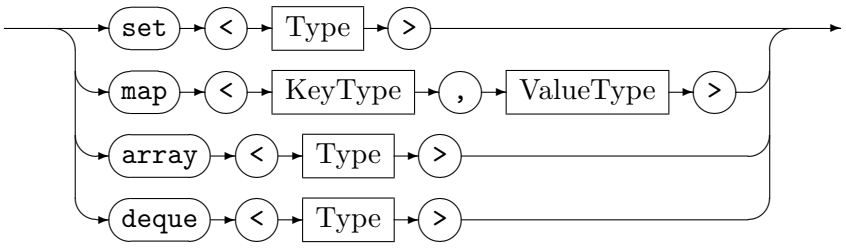
14.3 Storage Handling in the Sequences

TODO: reduce to diff compared to chapters before

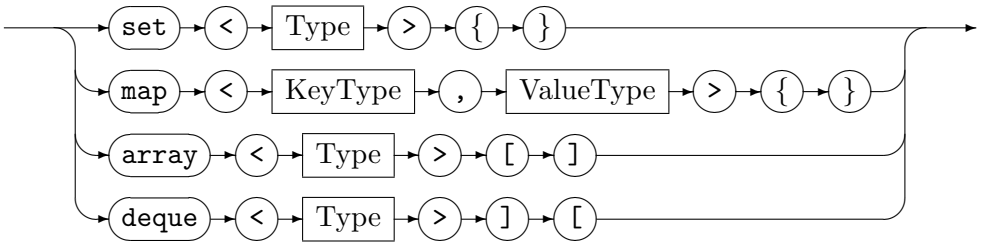
Storages are variables of container (set/map/array/deque) type (cf. 13.1) storing nodes or edges. They are primarily used in the sequences, from where they are handed in to the rules via `ref` parameters (but additionally container attributes in graph elements may be used as storages, esp. for doing data flow analyses, cf. 16.6). They allow to decouple processing phases: the first run collects all graph elements relevant for the second run which consists of a sequence executed for each graph element in the set. A difference of storage sets

and maps in the sequences to the sets and maps in the rewrite rules is that they only offer imperative addition and removal instead of union, intersection, difference and construction, and only empty constructors without initializing elements. The splitting of transformations into passes mediated by container valued global variables allows for subgraph copying without model pollution, cf. 16.5; please have a look at 16, 16.5 and 16.6 regarding a discussion on when to use which transformation combinators and for storage examples. The syntax diagrams given in the Rule Application Control Language chapter 8 already contained the storage constructs. Here we only give some refinements and explanations of the semantics.

Type



Literal



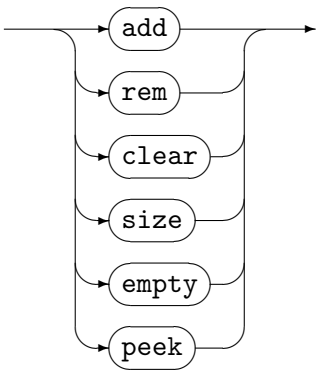
The Type used in a variable declaration may be set or map or array or deque. The Literals used in variable initialization may be empty sets or maps or arrays or deques. A container must be created and assigned to a variable before it can be used.

EXAMPLE (66)

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

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

MethodName



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

Set addition:
`s.add(v)` adds the value `v` to the set `s`, succeeds always.

Set removal:
`s.rem(v)` removes the value `v` from the set `s`, succeeds always.

Set clearing:
`s.clear()` removes all values from the set `s`, succeeds always.

Very similar operations are available on map variables:

Map addition:
`m.add(k,v)` adds the pair (`k`,`v`) to the map `m`, succeeds always.

Map removal:
`m.rem(k)` removes the pair (`k`,unknown) from the map `m`, succeeds always.

Map clearing:
`m.clear()` removes all key-value-pairs from the map `m`, succeeds always.

Similar operations are available on array variables:

Array addition:
`a.add(v)` adds the value `v` to the end of array `a`, succeeds always.

Array addition:
`a.add(v,i)` inserts the value `v` to the array `a` at index `i`, succeeds always.

Array removal:
`a.rem()` removes the value at the end of array `a`, succeeds always.

Array removal:
`a.rem(i)` removes the value at index `i` from the array `a`, succeeds always.

Array clearing:
`a.clear()` removes all values from the array `a`, succeeds always.

Very similar operations are available on deque variables:

Deque addition:
`d.add(v)` adds the value `v` to the end of deque `d`, succeeds always.

Deque addition:
`d.add(v,i)` inserts the value `v` to the deque `d` at index `i`, succeeds always.

Deque removal:
`d.rem()` removes the value at the begin of deque `d`, succeeds always.

Deque removal:
`d.rem(i)` removes the value at index `i` from the deque `d`, succeeds always.

Deque clearing:
`d.clear()` removes all values from the deque `d`, succeeds always.

There are further operations which are only available in the sequence expressions, too, not only in the sequence computations as the constructs before; but they can't be chained as they don't return the storage and are in that sense terminal:

Size assignment:
`v=w.size()` writes the number of entries in the container `w` to the variable `v`, succeeds always.

Emptyness assignment:
`v=w.empty()` writes to the variable `v` whether the container `w` is empty, succeeds always.

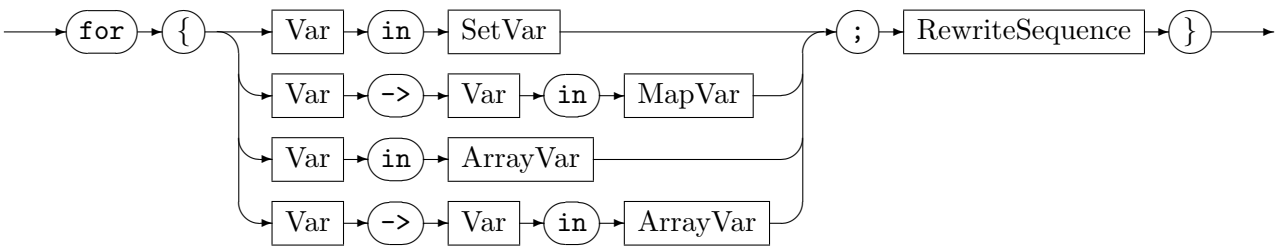
Peeking:
`v=w.peek(i)` writes to the variable `v` the value at the corresponding position `i` in the sequence of iteration, succeeds always. For arrays and deques that is the value at the corresponding index. A `w.peek()` on an array `w` yields the last element of the array, a `w.peek()` on a deque `w` yields the first element of the deque.

Lookup assignement:
`v=m[k]` assigns the result of the map lookup or array or deque indexed access to the variable `v`, succeeds iff `k` was contained in `m` (or was a valid index), fails otherwise, not touching the variable `v`.

Indexed assignement:
`a[i]=v` assigns the variable `v` to the array or deque or map `a` at the index `i`, overwriting the old value, succeeds iff `i` is a valid index into `a`, fails otherwise.

Handling of the storages is completed by the sequence expression operator `in` for membership query and the sequence loop for storage iteration.

RewriteFactor



The binary operator `v in w` checks for container membership; it returns true if `v` is contained in the set, or the domain of the map, or the array or deque `w`, otherwise false. It is an O(1) operation for sets and maps, and a O(n) operation for arrays or deques. The `for` command iterates over all elements in the set or array or deque, or all key-value pairs in the map or array or deque, and executes for each element / key-value pair the nested graph rewrite sequence; it completes successfully iff all sequences were executed successfully (an empty container causes immediate successful completion); the key in the key-value pair iteration of an array or deque is the integer typed index. (See 15.3 for another version of the `for` command.)

EXAMPLE (67)

The following XGRS is a typical storage usage. First an empty set `x` is created, which gets populated by an rule `t` executed iteratedly, returning a node which is written to the set. Then another rule is executed iteratedly for every member of the set doing the main work, and finally the set gets cleared to prevent memory leaks or later mistakes. If the graph should stay untouched during set filling you may need `visited` flags to prevent endless looping.

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

Handing in the storage to the rule, and using the set `add` method as introduced down below in 10 within the rule to fill the storage, allows to shorten the sequence to:

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

The for loop could be replaced by employing the storage access in the rule construct, cf. 12.6; this would be especially benefitial if the rule `r` inside the for loop would have to change the storage `x`, which would corrupt the iteration/enumeration variable.

NOTE (36)

The container over which the for loop iterates must stay untouched during iteration.

14.4 Quick Reference Table

Table 14.1 lists most of the operations of the graph rewrite computations at a glance.

`c;d`	Computes c then d; the value of the computation is d
`t=e`	Simple assignment of an expression value to an assignment target
`t=e=f`	Chained assignment
`v.m(e)`	Simple method call, with m e.g. being storage add
`v.m(e).m(e)`	Chained method call
`e ? f : g`	Returns f if e evaluates to true, otherwise g
`e op f`	For `op` being one of the boolean operators `||,|,&,&&,^`
`e op f`	For `op` being one of comparison operators `==,!=,<,<=,>,>=,in`
`e + f`	Numerical addition or string concatenation
`v`	Variable. Assignment target or expression.
`v.name`	Attribute of graph element. Assignment target or expression.
`@(name)`	Return graph element of given name.
`emit(v)`	Emits value of v to stdout.
`record(v)`	Records value of v to the replay log.
`export(filename)`	Exports the current graph to a file with the specified name.
`export(graph, path)`	Exports the (sub)graph given to the path given.
`def(`*`Parameters`*`)`	Check if all the variables are defined.
`random(upperBound)`	Returns random number from [0;upper bound[, if upper bound is missing from [0.0;1.0[.
`v=valloc()`	Allocates a visited-flag, assigns its id to v.
`vfree(e)`	Frees the visited-flag given.
`vreset(e)`	Resets the visited-flag given in all graph elements.
`u.visited[e]`	Visited flag e of u. Assignment target or expression.
`u=set<Node>{}`	Create storage set and assign to `u`.
`u.add(e)`	Add `e` to storage set `u`.
`u.rem(e)`	Remove `e` from storage set `u`.
`u.clear()`	Clears the storage set `u`.
`u.size()`	Returns the size of storage set `u`.
`u.empty()`	Returns whether storage set `u` is empty.
`u=map<N,Edge>{}`	Create storage map and assign to `u`. Operations are the same or similar to the operations of storage sets.
`u[e]`	Target value of `e` in `u`. Fails if `!(e in u)`. Assignment target or expression.
`add(T)`	Adds a node to the graph.
`add(T,src,tgt)`	Adds an edge to the graph.
`rem(e)`	Remove the node or edge `e` from the graph.
`clear()`	Clears the graph.
`incident(n)`	Returns the set of edges incident to `n`.
`adjacent(n)`	Returns the set of nodes adjacent to `n`.
`subgraph operations`	The subgraph operations allow to compute an induced or defined subgraph, or allow to replicate an induced or defined subgraph.

Let `c` and `d` be computations, `t` be an assignment target, `e`, `f`, `g` be expressions, `u`, `v`, `w` be variable identifiers

Table 14.1: Sequence computations at a glance

CHAPTER 15

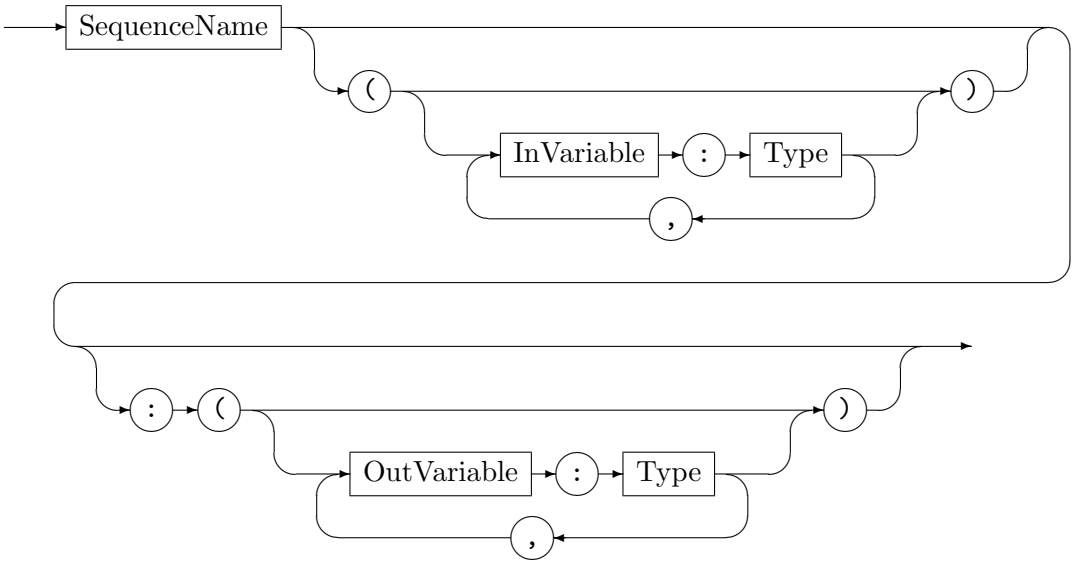
ADVANCED CONTROL

In this chapter we'll have a look at advanced graph rewrite sequence constructs, with the subsequences and the backtracking double angles as the central statements.

15.1 Sequence Definitions (Procedural Abstraction)

RewriteSequenceDefinition

RewriteSequenceSignature



If you want to use a sequence or sequence part at several locations, just factor it out into a sequence definition and reuse with its name as if it were a rule. A sequence definition declares input and output variables; when the sequence gets called the input variables are bound to the values it was called with. If and only if the sequences succeeds, the values from the output variables get assigned to the assignment target of the sequence call. Thus a sequence call behaves as a rule call, cf. 8.1.

A sequence definition may call itself recursively, as can be seen in example 68.

The compiled sequences must start with the `sequence` keyword in the rule file. The interpreted sequences in the shell must start with the `def` keyword; a shell sequences can be overwritten with another shell sequence in case the signature is identical. (Overwriting is needed in the shell to define direct or mutually recursive sequences as a sequence must be defined before it can get used; apart from that it allows for a more rapid-prototyping like style of development in the shell.)

EXAMPLE (68)

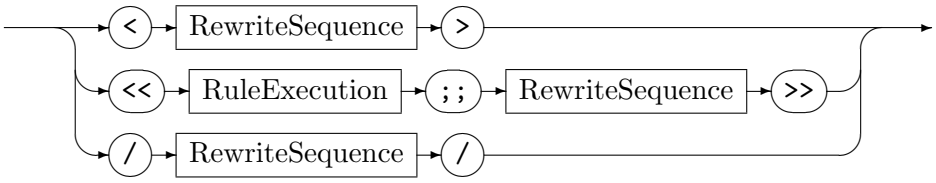
```
1 def rec(depth:int) {\
2   if{ {depth<::MAXDEPTH}; foo() ;> rec(depth+1); bar() }\
3 }
```

This example shows a sequence defined in the shell which is executing itself recursively. The host graph is transformed by applying `MAXDEPTH` times the rule `foo`, until finally the rule `bar` is executed. The result of the sequence is the result of `bar`, returned back from recursion step to recursion step while unwinding the sequence call stack.

15.2 Transactions, Backtracking, and Pause Insertions

The extended control constructs offer further rule application control in the form of transactions, backtracking, and pause insertions.

ExtendedControl



Graph rewrite sequences can be processed transactionally by using angle brackets (`<>`), i.e. if the return value of the nested sequence is `false`, all the changes carried out on the host graph will be rolled back. Nested transactions are supported, i.e. a transaction which was committed is rolled back again if an enclosing transaction fails.

Transactions as such are only helpful in a limited number of cases, but they are a key ingredient for backtracking, which is syntactically specified by double angle brackets (`<<r;;s>>`. The semantics of the construct are: First compute all matches for rule `r`, then start a transaction. For each match: execute the rewrite of the match, then execute `s`. If `s` failed then rollback and continue with the loop. If `s` succeeded then commit and break from the loop. On first sight this may not look very impressive, but this construct in combination with recursive sequences is the key operation for crawling through search spaces or for the unfolding of state spaces.

The backtracking double angles separate matching from rewriting: first all matches are found, but then only one after the other is applied, without interference of the other matches. The "without interference of other matches" statement is ensured by rolling back the changes of the application of the previous match, and much more, of the entire sequence which followed the rewriting of the previous match.

If you are just interested in the first goal state stumbled upon which satisfies your requirements during a search, then you only need to give a condition as last statement of the sequence which returns true if the goal was reached; the iteration stops exactly in the target state. When you are interested in finding all states which satisfy your requirements, or even in enumerating each and every state, just force backtracking by noting down the constant false as last element of the sequence.

The backtracking construct encodes a single decision point of a search, splitting into breadth along the different choices available at that point, and further continuing the search in the sequence. When this point of splitting into breadth is contained in a sequence, and this sequence calls itself again recursively on each branch of the descision taken, you get a recursion which is able to search into depth, continuing decision making on the resulting graph of the previous decision.

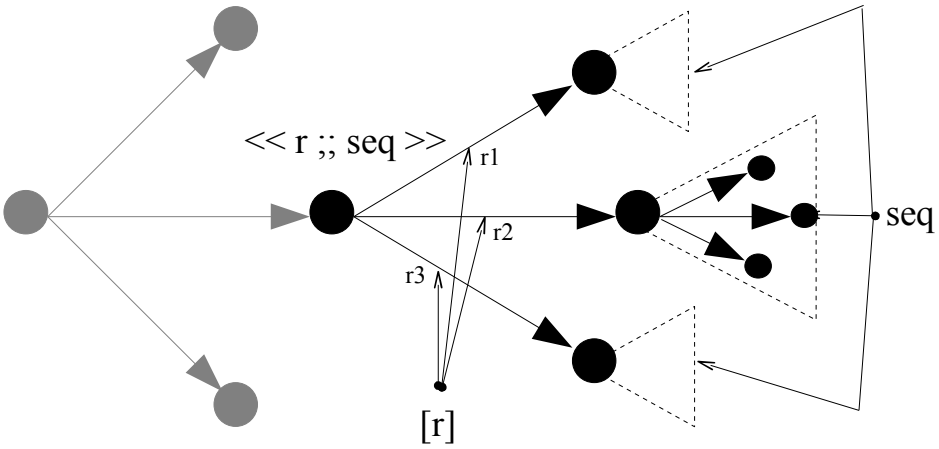


Figure 15.1: Search space illustration, a bullet stands for a graph

With each sequence call advancing one step into depth and each backtracking angle advancing into breadth, you receive a depth-first enumeration of an entire search space (as sketched in 15.1). Each state is visited in *temporal succession*, with only the most recent state being available in the graph. But maybe you want to keep each state visited, because you are interested in viewing all results at once, or because you want to compare the different states. As there is only one host graph in GrGen.NET, keeping each visited state requires a partition of the host graph into separate subgraphs, each denoting a state.

After you changed the modeling from a host graph to a state space graph consisting of multiple subgraphs, each representing one of the graphs you normally work with, you can materialize the search space visited in temporal succession into a state space graph, by copying the subgraphs (which are normally only existing at one point in time) during pause insertions out into space. When subgraphs would be copied without pause insertions, they would be rolled back during backtracking; but effects applied on the graph from `/ in between here /` are bypassing the recording of the transaction undo log and thus stay in the graph, even if the transaction fails and is rolled back.

When you have switched from a depth-first search over one single current graph to the unfolding of a state space graph containing all the subgraphs reached, you may compare each subgraph which gets enumerated with all the already available subgraphs, and if the new subgraph already exists (i.e. is isomorph to another already generated subgraph), you may refrain from inserting it. This symmetry reduction allows to save the space and time needed for storing and computing equivalent branches otherwise generated from the equivalent states. But please note that the `==` operator on graphs is optimized for returning early when the graphs are different; when the graphs are isomorphic you have to pay the full price of graph isomorphy checking. This will happen steadily with automorphic patterns and then degrade performance. To counter this filter the matches which cover the same spot in different ways, see 21.3 on how to do this. Merging states with already computed ones yields a DAG-formed state space, instead of the always tree like search space. Have a look at the transformation

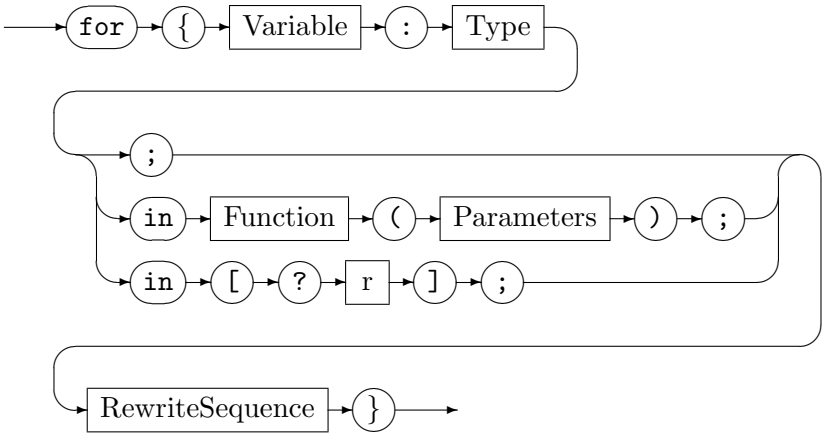
techniques chapter for more on state space enumeration 16.7 and copying 16.5. One caveat of the transactions and backtracking must be mentioned: rollback might lead to an incorrect graph visualization when employed from the debugger. This holds especially when using grouping nodes to visualize subgraph containment (18.1). You must be aware that you can't rely on the online display as much as you can normally, and that you maybe need to fall back to an offline display by opening a `.vcg`-dump of the graph written in a situation when the online graph looked suspicious; a dump can be written easily in a situation of doubt from the debugger pressing the `p` key.

NOTE (37)

While a transaction or a backtrack is pending, all changes to the graph are recorded into some kind of undo log, which is used to reverse the effects on the graph in the case of rollback (and is thrown away when the nesting root gets committed). So these constructs are not horribly inefficient, but they do have their price — if you need them, use them, but evaluate first if you really do.

15.3 For Loops and Indeterministic Choice

ExtendedControl



The `for` loop without containment is iterating over all the elements in the current host graph which are compatible to the type given. The iteration variable is bound to the currently enumerated graph element, then the sequence in the body is executed. The type of the iteration variable must be statically known to be of a node or edge type.

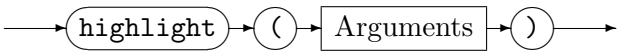
If you iterate a node type from a graph, you may be interested in iterating its incident edges or its adjacent nodes. This can be achieved with a for neighbouring elements loop, which binds the iteration variable to an edge in case the *Function* is one of `incoming`, `outgoing`, or `incident`. Or which binds the iteration variable to a node in case the *Function* is one of `adjacentIncoming`, `adjacentOutgoing`, or `adjacent`. The admissible *Parameters* are the source node, or the source node plus the incident edge type, or the source node plus the incident edge type, plus the adjacent node type — that's the same as for the sequence expression functions explained in 13.2/Connectedness queries. In contrast to these set returning functions, this loop contained functions enumerate nodes/edges multiple times in case of reflexive or multi edges.

The third `for` loop introduced here, the for matches loop, allows to iterate through the matches found for an all-bracketed rule reduced to a test; i.e. the rule is not applied, we only iterate its matches. The loop variable must be of a statically known `match<r>` type with `r` being the name of the rule matched. The elements (esp. the nodes and edges) of the pattern

of the matched rule can then be accessed by applying the .-operator on the loop variable, giving the name of the element of interest after the dot. Note: the elements must be assigned to a variable in order to access their attributes, a direct attribute access after the match access is not possible. Note: the match object allows only to access the top level nodes, edges, or variables. If you use subpatterns or nested patterns and want to access elements found by them, you have to `yield`(7.3) them out to the top-level pattern.

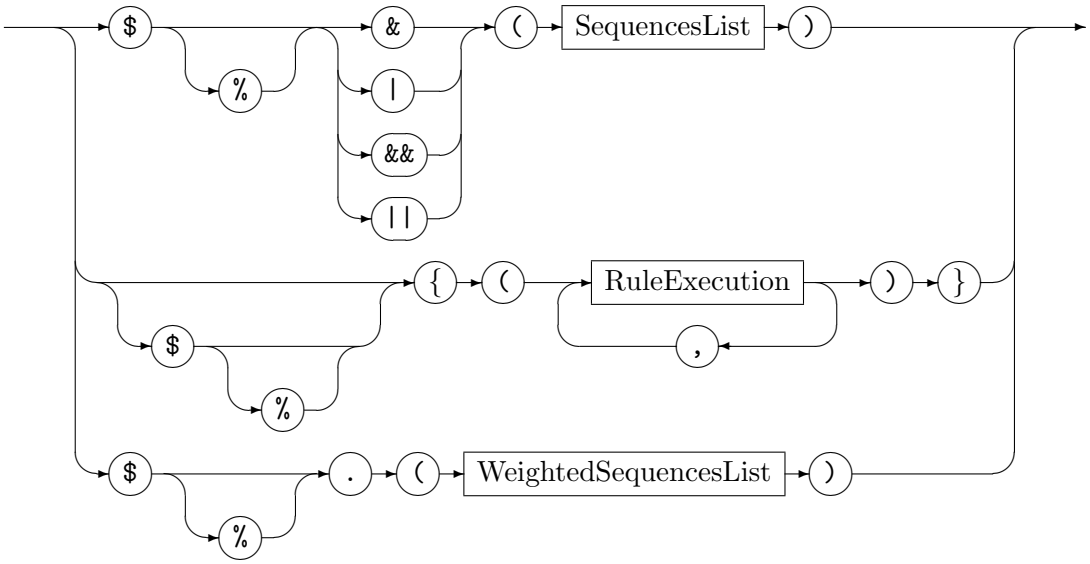
The most important `for` loop, the one iterating a container, for enumerating the elements contained in storages, was already introduced here: 14.3. All `for` loops fail if one of the sequence executions failed, and succeed otherwise.

ExtendedControl

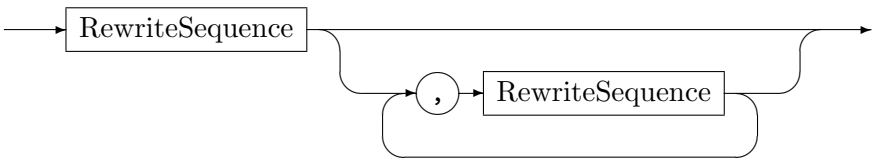


The `highlight` sequence highlights the arguments given as a quoted text in the graph; it does what the `(h)ighlight` command does in the debugger, see 18.4, just programmed from the sequences. Arguments is a comma-separated list of variable names or visited flag ids, the graph elements contained in the variables are highlighted, as are the graph elements marked by the visited flag.

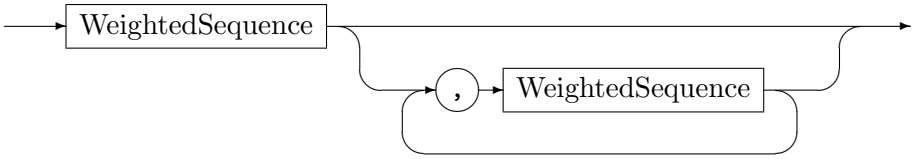
ExtendedControl



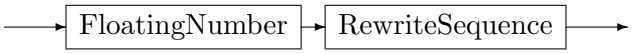
SequencesList



WeightedSequencesList



WeightedSequence



The indeterministic choice operators execute chosen elements from a sets of rules or sequences. The random-all-of operators given in function call notation with the dollar sign plus operator symbol as name have the following semantics: The strict operators `|` and `&` evaluate all their subsequences in random order returning the disjunction resp. conjunction of their truth values. The lazy operators `||` and `&&` evaluate the subsequences in random order as long as the outcome is not fixed or every subsequence was executed (which holds for the disjunction as long as there was no succeeding rule and for the conjunction as long as there was no failing rule). A choice point may be used to define the subsequence to be executed next.

The some-of-set braces `{(r,[s],$[t])}` matches all contained rules and then executes the ones which matched. The one-of-set braces `${(r,[s],$[t])}` (some-of-set with random choice applied) matches all contained rules and then executes at random one of the rules which matched (i.e. the one match of a rule, all matches of an all bracketed rule, or one randomly chosen match of an all bracketed rule with random choice). The one/some-of-set is true if at least one rule matched and false if no contained rule matched. A choice point may be used on the one-of-set; it allows you to inspect the matches available graphically before deciding on the one to apply.

The weighted one operator `$.(w1 s1, ..., wn sn)` is executed like this: the weights `w1-wn` (numbers of type double) are added into a series of intervals, then a random number (uniform distribution) is drawn in between `0.0` and `w1+...+wn`, the subsequence of the interval the number falls into is executed, the result of the sequence is the result of the chosen subsequence.

15.4 Quick Reference Table

Table 15.1 lists most of the operations of the graph rewrite sequences at a glance.

`(w)=s(w)`	Calls a sequence `s` handing in `w` as input and writing its output to `w`; defined e.g. with `sequence s(u:Node):(v:Node) { v=u }`.
`<s>`	Execute `s` transactionally (rollback on failure).
`<<r;;s>>`	Backtracking: try the matches of rule `r` until `s` succeeds.
`/ s /`	Pause insetion: execute `s` outside of the enclosing transactions and sequences, i.e. the changes of `s` are not rolled back.
`highlight(vars)`	Highlights the content of the variables in the graph.
`${(r1,[r2],$[r3])}`	Tries to match all contained rules, then rewrites indeterministically one of the rules which matched. True if at least one matched.
`for{v in u; t}`	Execute `t` for every `v` in storage set `u`. One `t` failing pins the execution result to failure.
`for{v->w in u; t}`	Execute `t` for every pair (`v`,`w` in storage map `u`. One `t` failing pins the execution result to failure.
`for{v:match<r> in [?r]; t}`	Execute `t` for every match `v` from rule `r`. One `t` failing pins the execution result to failure.
`for{v:T; s}`	Execute `s` for every `v` of type `T` available in the graph. One `s` failing pins the execution result to failure.
`for{v in func(w); s}`	Execute `s` for every edge/node incident/adjacent to `w` . One `s` failing pins the execution result to failure.
`{comp}`	An unspecified sequence computation (see table 14.1).

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

Table 15.1: Sequences at a glance

CHAPTER 16

TRANSFORMATION TECHNIQUES

In this chapter we'll have a look at transformation techniques to solve common graph rewriting tasks utilizing the constructs introduces so far. They could be offered directly by some dedicated operators, but these would need so much customization to be useful in the different situations one needs them, that we decided against dedicated operators; instead it is on you to program the version you need yourself by combining language constructs and rules.

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

nested and subpatterns
: which allow to match and rewrite complex patterns built in a structured way piece by piece. With different pieces connected together, pieces to decide in between, and pieces which appear repeatedly.

embedded graph rewrite sequences
: for deferred rule execution to do work later on you can't do while executing the rule proper (e.g. because an element was already matched and is now locked due to the isomorphy constraint); or for splitting work into several parts, reusing already available functionality.

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

In the following we'll employ them to merge and split nodes, emulate node replacement grammars, execute transformations in a narrow sense, copy substructures, compute flow equations over the graph structure with sets in the nodes, and enumerate state spaces. Further examples can be found in [BJ11a] employing two storagesets switched in between for computing a wavefront running over a compiler graph and [JB11] using the nested and subpatterns for elegantly extracting a state machine model from a program graph.

NOTE (38)

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

Furthermore we'll have a look at language constructs which allow to build transformations by introducing state as a communications means between several rule applications.

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

NOTE (39)

The storage sets are more efficient in case the count of elements of interest is a good deal smaller than the number elements in the graph; they are looked up in the set, in contrast to the visited flags, which are used by enumerating all available graph elements, filtering according to visited state. The visited flags on the other hand are extremely memory efficient (for free as long as you use only a few of them at the same time).

16.1 Merge and Split Nodes

Merging a node `m` into a node `n` means transferring all edges from node `m` to node `n`, then deleting node `m`. Splitting a node `m` off from a node `n` means creating a node `m` and transferring some edges from node `n` to `m`.

In both cases there are a lot of different ways how to handle the operation exactly: Maybe only incoming or only outgoing edges, or only edges of a certain type `T` or only edges not of type `T`; maybe the node `n` is to be retyped, maybe the edges are to be retyped. But common is the transferring of edges; this can be handled succintly by an `iterated` statement and the `copy` operator. In case the node opposite to an edge may be incident to several such edges, one must use an `exec` instead (or the `independent` operator 4.3.1), as every iteration locks the matched entities, so they can't get matched twice. Not needing the opposite node one could simply leave it unmentioned in the pattern, only referencing node `n` or `m` and the edge, but unfortunately we need the opposite node so we can connect the edge copy to it.

NOTE (40)

In case a simple node merging without edge retyping is sufficient the retype'n'merge clause introduced in 9.6 offers a much simpler alternative for merging.

Now we'll have a look at an example for node merging: T1-T2 analysis from compiler construction is used to find out whether a control flow graph of a subroutine is reducible, i.e. all loops are natural loops. All loops being natural loops is a very useful property for many analyses and optimizations. The analysis is split into two steps, T1 removes reflexive edges, T2 merges a control flow successor into its predecessor iff there is only one predecessor available. These two steps are iterated until the entire graph is collapsed into one node which means the control flow is reducible, or execution gets stuck before, in which case the control flow graph is irreducible. The analysis is defined on simple graphs, i.e. if two control flow edges between two basic block nodes appear because of merging they are seen as one, i.e. they are automatically fused into one. As GrGen.NET is built on multigraphs we have to explicitely do the edge fusion in a further step T3.

First let us have a look at T1 and T3, which are rather boring ... ehm, straight forward:

EXAMPLE (69)

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

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

EXAMPLE (70)

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

In case a control flow graph would be a multi-graph, with several control flow edges between two nodes, one would have to use an `exec` with an all-bracketed rule instead of the `iterated`, to be able to match a multi-`cf`-edge target of `succ` multiple times (which is prevented in the `iterated` version by the isomorphy constraint locking the target after the first match).

This is the second version using exec instead, capable of handling multi edges:

EXAMPLE (71)

```
rule T2exec
{
  pred:BB -e:cf-> succ:BB;
  negative {
    -e->;
    -:cf-> succ; // if succ has only one predecessor
  }

  modify { // then merge succ into this predecessor
    exec([copyToPred(pred, succ)] ;> delSucc(succ));
  }
}

rule copyToPred(pred:BB, succ:BB)
{
  succ -e:cf-> n:BB;

  modify {
    pred -:copy<e>-> n;
  }
}

rule delSucc(succ:BB)
{
  modify {
    delete(succ);
  }
}
```

Natural loops are so advantageous that one transforms irreducible graphs (which only occur by using wild gotos) into reducible ones, instead of bothering with them in the analyses and optimizations. An irreducible graph can be made reducible by node splitting, which amounts to code duplication (in the program behind the control flow graph). In a stuck situation after T1-T2 analysis, a `BB` node with multiple control flow predecessors is split into as many nodes as there are control flow predecessors, every one having the same control flow successors as the original node. (Choosing the `cf` edges and `BB` nodes which yield the smallest amount of code duplication is another problem which we happily ignore here.)

EXAMPLE (72)

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

```
rule split(succ:BB)
{
  pred:BB -first:cf-> succ;
  multiple {
    otherpred:BB -other:cf-> succ;

    modify {
      otherpred -newe:cf-> newsucc:copy<succ>;
      delete(other);
      exec(copyCfSuccFromTo(succ, newsucc));
    }
  }

  modify {
  }
}

rule copyCfSuccFromTo(pred:BB, newpred:BB)
{
  iterated {
    pred -e:cf-> succ:BB;

    modify {
      newpred -:copy<e>-> succ;
    }
  }

  modify {
  }
}
```

The examples given can be found in the `tests/mergeSplit/` directory including the control scripts and test graphs; you may add `debug` prefixes to the `exec` or `xgrs` statements in the graph rewrite script files and call GrShell with e.g. `mergeSplit/split.grs` as argument from the `tests` directory to watch execution.

16.2 Node Replacement Grammars

With node replacement grammars we mean edNCE grammars [ER97], which stands for edge label directed node controlled embedding. In this context free graph grammar formalism, every rule describes how a node with a nonterminal type is replaced by a subgraph containing terminal and nonterminal nodes and terminal edges. The nodes in the instantiated graph get connected to the nodes that were adjacent to the initial nonterminal node, by connection instructions which tell which edges of what direction and what type are to be created for which original edges of what direction and what type, going to a node of what type.

This kind of grammars can be encoded in GrGen.NET by rules with a left hand side consisting of a node with a type denoting a nonterminal and iterateds matching the edges and opposite nodes it is connected to of interest; "of interest" amounts to the type and direction of the edges and the type of the opposite node. The right hand side deletes the original node (thus implicitely the incident edges), creates the replacement subgraph, and tells in the

rewrite part of the iterateds what new edges of what directedness and type are to be created, from the newly created nodes to the nodes adjacent to the original node. (Multiple edges between two nodes are not allowed in the node replacement formalism, in case you want to handle them you've to use `exec` as shown in the merge/split example above.)

The following example directly follows this encoding:

EXAMPLE (73)

This is an example rule replacing a nonterminal node `n:NT` by a 3-clique. For the outgoing `E1` edges of the original node, the new node `x` receives incoming `E2` edges. And for incoming `E2` edges of the original node, the new nodes `y` and `z` receive edges of the same type, `y` with reversed direction and `z` of the exact dynamic subtype bearing the same values as the original edges.

```
rule example
{
  n:NT;

  iterated {
    n -:E1-> m:T;

    modify {
      x <-:E2- m;
    }
  }

  iterated {
    n <-e2:E2- m:T;

    modify {
      y -:E2-> m;
      z <-:copy<e2>- m;
    }
  }

  modify {
    delete(n);
    x:T -- y:T -- z:T -- x;
  }
}
```

As another example for node replacement grammars we encode the two rules needed for the generation of the completely connected graphs (cliques) in two GrGen.NET rules. The first replaces the nonterminal node by a new nonterminal node linked to a new terminal node, connecting both new nodes to all the nodes the original nonterminal node was adjacent to. The second replaces the nonterminal node by a terminal node, connecting the new terminal node to all the nodes the original nonterminal node was adjacent to. This "we want to preserve the original edges" can be handled more succintly and efficiently by retyping which we gladly use instead of the iteration.

EXAMPLE (74)

```
rule cliqueStep
{
  nt:NT;

  iterated {
    nt -- neighbour:T;

    modify {
      t -- neighbour;
      nnt -- neighbour;
    }
  }

  modify {
    delete(nt);
    t:T -- nnt:NT;
  }
}

rule cliqueTerminal
{
  nt:NT;

  modify {
    :T<nt>;
  }
}
```

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

16.3 Transformation in a Rewriting Tool

GrGen.NET is a *rewrite* tool, i.e. you modify *one* graph. This is in contrast to *transformation* tools, which are used for specifying mappings in between *two* (or more) graphs. (Graphs are typically called models in this kind of tools. Don't confuse this with the notion of graph model in GrGen.NETas introduced in chapter 3. In model transformation parlance our graph model denotes a meta model, with a model adhering to a meta model, which in turn adheres to a meta meta model, climbing a ladder of bullshit abstraction leading right into the land of the hot-air merchants, eh, real software engineers.)) The graph model might be a union of multiple graph models, and the graph may consist of multiple unconnected components, so you don't loose anything regarding expressiveness here.

What you loose is the automatic construction of the tracebility links which allow to retrieve the source node from a mapped node or the mapped node from a source node, which is normally carried out by a real transformation tool in the background. And there are no built-in annotations by which you specify the model in which to match the elements. But again neither means a loss in expressiveness.

You can construct the tracebality links on your own. Either by using simple edges between the nodes (this is not possible in a transformation tool as edges can only exist within a graph/model). Or by using maps of node type to node type (or from edge type to edge type) which you must fill manually: when you map a source node to a target node (i.e. for a matched source node you create a target node), you fill a global map from source to target nodes with this pair, and/or a global map from target to source nodes. When you need this

information, you look it up easily with `target=map[source]` or `source=map[target]`. The highlight command 18.4 of the debugger can visualize this storage map neatly by marking the contained nodes and adding directed edges in between the source and domain elements. In some situations you are even better off: if the retype operator is sufficient, you can easily transform the graph in-place. A given, named graph element then denotes the source and the target, just at different points in time.

The lack of built-in model annotations can be overcome by several ways of partitioning the graph. Often the types of the source and the target model are disjoint. This is the most convenient case, as you save any extra notational effort, the types alone define the source and target. In cases they are not disjoint (because you are forced to use certain non-disjoint graph models), you can tell the elements from different graphs apart by either using visited flags — then each "graph" is marked to be visited with its own flag. Or you may use anchor nodes representing subgraphs, with an edge from such an anchor node to all nodes contained in its subgraph. Thus all nodes of a subgraph are adjacent to an anchor node, and the subgraph consists exactly of the induced subgraph of these adjacent nodes (see 16.5.1 for more on this kind of modelling).

16.4 Comparing Structures

Structures are compared in three steps, the first is to collect all nodes of interest in a storage set, the second is to compute an induced subgraph from that set, and the third is to compare the subgraphs with the built in graph comparison operators.

The first step typically consists of covering the nodes of the structure one wants to compare with iterated subpatterns, i.e. subpatterns which match from a root node on with iterateds along the incident edges into breadth, employing a subpattern again on the node adjacent to the root node to match into depth (see Fig. 16.1 for an illustration of how example 75 matches the DAG reachable from the root node `$1` on via outgoing edges). In case the structure to compare consists of multiple connected components or is difficult to capture with iterated and subpatterns, you can collect it step by step with a sequence of rule applications building up the node set.

EXAMPLE (75)

```
1  pattern MatchSubgraph(root:Node, ref nodes:set<Node>)
2  {
3    iterated { // match spanning tree of graph from root on
4      root --> ch:Node;
5      ms:MatchSubgraph(ch, nodes);
6
7      modify {
8        ms();
9      }
10   }
11
12   modify {
13     eval { nodes.add(root); } // build node set inducing graph
14   }
15 }
```

The second step is to compute an induced subgraph from the node set collected, that can be done easily with the `inducedSubgraph` subgraph operation introduced in 14.1:
`exec sub:graph=inducedSubgraph(nodes)`.

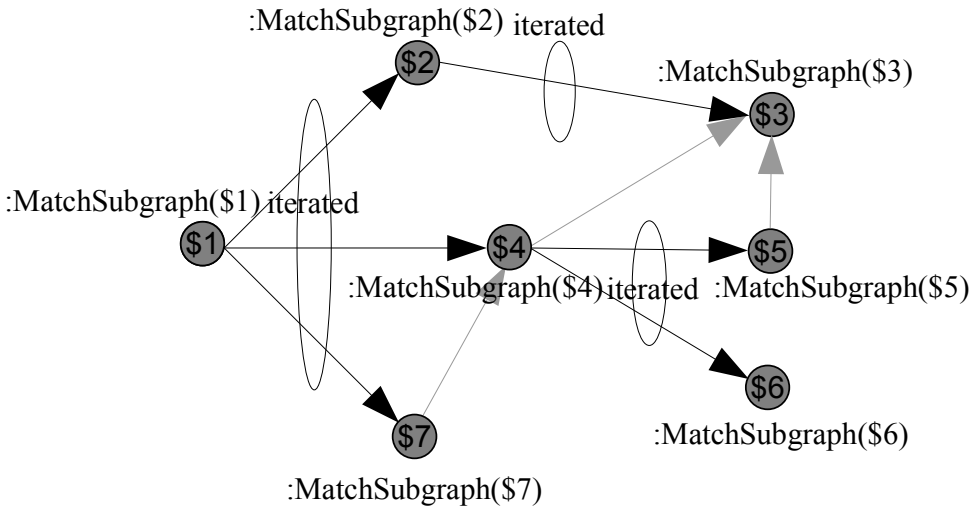


Figure 16.1: Matching a spanning tree in a graph

The third step consists of comparing the subgraphs we filleted out of the common host graph with the graph comparison operators introduced in table 13.1 (of section **??**):
`exec if{sub1==sub2; doIso; doNonIso}`.

In case each such subgraph needs to be compared more than once it is recommended to add an attribute of type `graph` (see chapter 12) to the subgraph starting anchor nodes, to assign the induced subgraph to this attribute, and to compare the subgraphs stored in these attribute. This saves us the cost of computing the attributes and allows for further internal optimizations which may result in huge speedups.

NOTE (41)

The subgraph is not adapted automatically when the host graph changes, if this happens you must compute the stored subgraph anew manually.

The comparison is then typically done with an idiom as such:
`exec if{ for{other:AnchorNodeType; {curSub!=other.sub}} ; doNonIso(curSub)}` or such:
`exec if{ for{other:graph in setOfCandidates; {curSub!=other}} ; doNonIso(curSub)}`.

16.5 Copying Structures

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

The first pass consists of covering the nodes of the structure one wants to copy with iterated subpatterns, in the same way as already introduced in 16.4, i.e. subpatterns which match from a root node on with iterateds along the incident edges into breadth, employing a subpattern again on the node opposite to the root node to match into depth. In the

example we match the entire subgraph from a root node on, if one wants to copy a more constrained subgraph one can simple constrain the types, directions, and structures in the iterated subpattern covering the nodes. The nodes are copied with the `copy` operators and a storagemap is filled, storing for every node copied its copy.

The second pass is started after the structure matching ended by executing the deferred execs which were issued for every node handled. Each `exec` copies all outgoing edges (one could process all incoming edges instead) of a node: for each edge leaving the original node towards another original node a copy is created in between the copy of the original node and the copy of the other node. The copies are looked up with the original nodes from the storage map (which fails for target nodes outside of the subgraph of interest). Here too one could constrain the subgraph copied by filtering certain edges. In case of undirected edges one would have to prevent that edges get copied twice (once for every incident node). This would require a visited flag for marking the already copied edges or a storage receiving them, queried in the edge copying pattern and set/filled in the edge copying rewrite part.

EXAMPLE (76)

The example shows very generally how a subgraph reachable from a root node by incident edges can get copied, collecting and copying the nodes along a spanning tree from the root node on, then copying the edges in between the nodes in a second run afterwards. The edges get connected to the correct node copies via a mapping from the old to the new nodes remembered in a storage-map (tracebility map).

```
1  pattern CopySubgraph(root:Node, ref oldToNew:map<Node, Node>)
2  {
3    iterated { // match spanning tree of graph from root on
4      root <--> ch:Node;
5      cs:CopySubgraph(ch, oldToNew);
6
7      modify {
8        cs();
9      }
10   }
11
12   modify {
13     newroot:copy<root>; // copy nodes
14     eval { oldToNew.add(root, newroot); }
15     exec( [CopyOutgoingEdge(root, oldToNew)] ); // deferred copy edges
16   }
17 }
18
19 rule CopyOutgoingEdge(n:Node, ref oldToNew:map<Node, Node>)
20 {
21   n -e:Edge-> m:Node;
22   hom(n,m); // reflexive edges
23   nn:Node{oldToNew[n]}; nm:Node{oldToNew[m]};
24   hom(nn,nm); // reflexive edges
25
26   modify {
27     nn -ee:copy<e>-> nm;
28   }
29 }
```

The example can be found in the `tests/copyStructure` directory. Without storagemaps one would have to pollute the graph model with helper edges linking the original to the copied

nodes.

16.5.1 Built-In Graph Copying

In contrast to the just introduced general copying of an entire graph, you may choose a limited form of copying subgraphs. This is enabled by the `insertInduced` and `insertDefined` operations introduced in 14.1, which add a clone of the subgraph induced by the set of nodes or set of edges given as first argument to the host graph. The clone of the second argument node or edge which was inserted into the host graph is returned as anchor element for further operations.

Before you can apply these operations you must collect the nodes or edges which are used to compute the induced subgraph. Besides building the sets element-by-element with `add`-methods you may use set returning functions, e.g. `adjacent` introduced in 13.2, which returns the set of all the nodes adjacent to the node given as (first) argument.

Employing `insertInduced(adjacent())` is a common idiom for copying a subgraph when the host graph is partitioned into multiple subgraphs in the following way: A special node type `Graph` and a special edge type `contains` are introduced into the graph model. Every subgraph is represented by a `Graph` anchor node. From these anchor nodes on `contains` edges lead to all the non-subgraph nodes contained in the corresponding subgraph. With this modelling, all nodes in a subgraph are easily reachable from one anchor node via `adjacent`, and the entire subgraph can be easily retrieved by using `inducedSubgraph` or copied by `insertInduced`. Besides this simplified manipulation, the graph can be easily visualized as being partitioned into multiple subgraphs with a `dump add node Graph group by contains` graph visualization command (see 18.1 for more on this).

16.6 Data Flow Analysis for Computing Reachability

In compiler construction, given a program graph, one wants to compute non-local properties in order to transform the program graph. This is normally handled within the framework of data flow analysis, which employs flow equations telling how property values of a node are influenced by property values of the predecessor or successor nodes in addition to the node's local share on the overall information, with the predecessor or successor nodes being again influenced by their predecessor or successor nodes. Property values are modeled as sets; the information is propagated around the graph until a fix point is reached (the operations must be monotone in order for a fix point to exist on the finite domain of discourse). You might be interested in the transparencies under http://www2.imm.dtu.dk/~riis/PPA/slides2.pdf for some reading on this topic; especially as this is a general method to compute non-local informations over graphs not limited to compiler construction.

We'll apply a backward may analysis (with only *gen* but no *kill* information) to compute for each node the nodes which can be reached from this node. Reachability is an interesting property if you have to do a lot of iterated path checks: instead of computing the itererated path with a recursive pattern each and every time you must check for it, compute it once and just look it up from then on. If you need to check several paths which must be disjoint you won't get around employing recursive subpatterns with one locking the elements for the other; but even in this case the precomputed information should be valuable (unless the graph is heavily connected), constraining the search to source and target nodes between which a path does exist, eliminating nodes which are not connected.

The reachability information will be stored in a storage set per node of the graph (indeed, we trade memory space for execution speed):

EXAMPLE (77)

```
node class N
{
   reachable:set<N>;
}
```

The analysis begins with initializing all the storage sets with the local information about the direct successors by employing the following rule on all possible matches:

EXAMPLE (78)

```
rule directReachability
{
  hom(n,m);
  n:N --> m:N;

  modify {
    eval { n.reachable.add(m); }
  }
}
```

The analysis works by keeping a global todo-set `todo` containing all the nodes which need to be (re-)visited, because the information in one of their successors changed; this set is initialized with all nodes available using the sequence `[addAllNodesToWorkset(todo)]`).

From then on in each iteration step a node `n` is removed with `(n)=pickAndRemove(todo)`, until the todo-set becomes empty, signaling the termination of the analysis; the node is processed by determining its successors `[successors(n, succs)]`, adding the reachability information available in each successor to `n` controled by the sequence
`for{s in succs; (changed)=propagateBackwards(n,s,changed)}`.
If the information in `n` changed due to this, the predecessors of the node are added to the workset via `if{changed;[addPredecessors(n,todo)]}`.

EXAMPLE (79)

This is the core rule of the dataflow analysis: the reachability information from the successor node `s` in its `reachable` storage attribute is added to the storage attribute of the node `n` of interest; if the storage set changes this is written to the returned variable.

```
rule propagateBackwards(n:N, s:N, var changed:boolean) : (boolean)
{
  modify {
    eval { n.reachable |= s.reachable |> changed; }
    return(changed);
  }
}
```

The example can be found in the `tests/dataFlowAnalysis` directory, just add `debug` before the `exec` (or `xgrs`) in `dataFlowAnalysisForReachability.grs` and watch it run. A sample situation showing a propagation step is given in 16.2. The subgraph at the top-left is already handled as you can see by the reachable set displayed in each node.

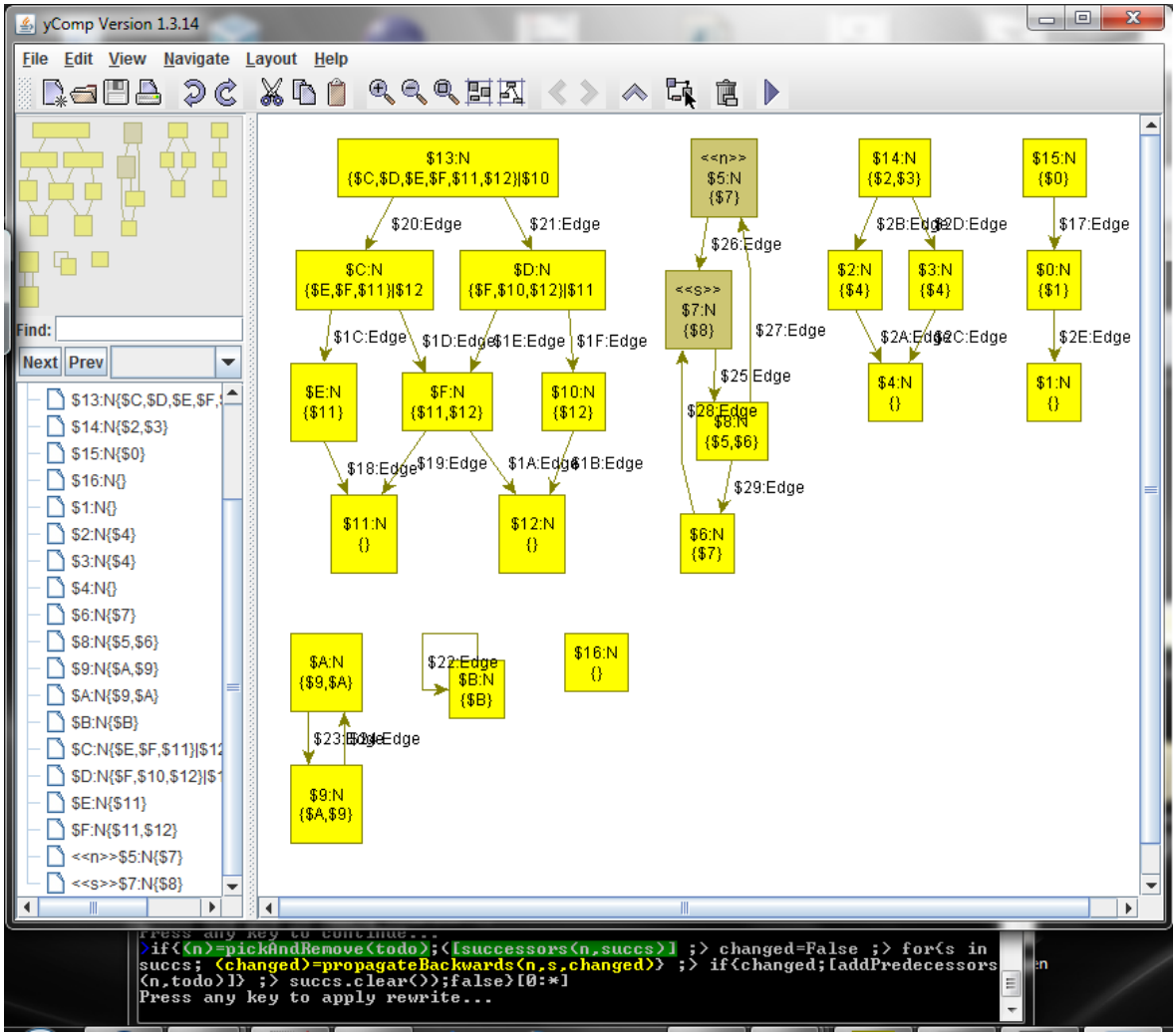


Figure 16.2: Situation from dataflow analysis

Worklist Based Data Flow Analysis

The approach introduced above implements the basics but will not scale well to large graphs – even medium sized graphs – due to the random order the nodes are visited. What is used in practice instead is a version employing a worklist built in postorder, so that a node is only visited after all its successor nodes have been processed. For graphs without backedges, i.e. loops for program graphs, this gives an analysis which visits every node exactly once in the propagation phase. For graphs with loops some nodes will be visited multiple times, but due to the ordering the analysis still terminates very fast.

The worklist is implemented directly in the graph by additional edges of the special type `then` between the nodes, and a special node for the list start; the `todo` set is kept, to allow for a fast "is the node already contained in the worklist"-check, used to save us from adding nodes again which are already contained (thus will be visited in the future anyway); i.e. the abstract worklist concept is implemented by the todo-set and the list added invasively to the graph.

EXAMPLE (80)

```
1 edge class then; // for building worklist of nodes to be handled
```

The initial todo-set population of the simple approach is replaced by worklist constructing, successively advancing the last node of the worklist given by the `last` variable; it starts with all nodes having no successor:
`(last)=addFinalNodesToWorklist(last, todo)*`
Then iteratively all nodes which lead to them get added:
`( (last)=addFurther(pos, last, todo)* ;> (pos)=switchToNextWorklistPosition(pos) )*`
In case of loops without terminal nodes we pick an arbitrary node from them:
`(last)=addNotYetVisitedNodeToWorklist(last, todo)`
and add everything what leads to them, until every node was added to the worklist.

Now we can start the analysis, which works like the simple one does, utilizing the very same propagation rule,but follows the worklist instead of randomly picking from a todo-set, shrinking and growing the worklist along the way.

EXAMPLE (81)

An example rule for worklist handling, adding a not yet contained node to the worklist; please note the quick check for containment via the set membership query.

```
1  rule addToWorklist(p:N, ref todo:set<N>, last:N) : (N)
2  {
3    if{ !(p in todo); }
4
5    modify {
6      last -:then-> p;
7      eval { todo.add(p); }
8      return(p);
9    }
10 }
```

EXAMPLE (82)

An exmaple rule for worklist handling, removing the by-then processed node `pos` from the worklist.

```
1  rule nextWorklistPosition(pos:N, ref todo:set<N>) : (N)
2  {
3    pos -t:then-> next:N;
4
5    modify {
6      delete(t);
7      eval { todo.rem(pos); }
8      return(next);
9    }
10 }
```

The example can be found in the `tests/dataFlowAnalysis` directory, just add `debug` before the `exec` (or `xgrs`) in `dataFlowAnalysisForReachabilityWorklist.grs` and watch it run. A sample situation showing a worklist building step is given in 16.3. The subgraph at the top-left is already handled as you can see by the reachable set displayed in each node.

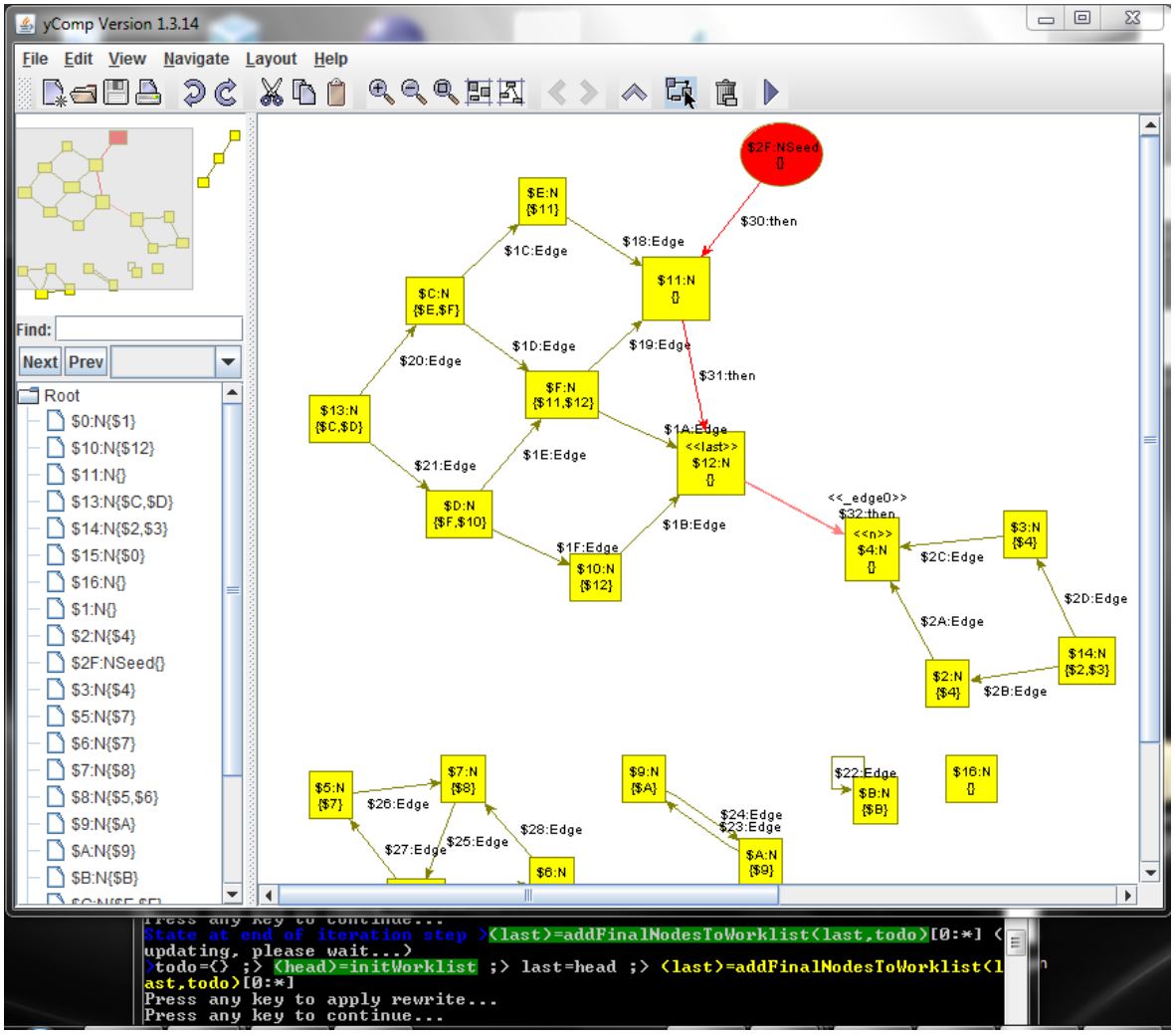


Figure 16.3: Situation from worklist building

16.7 State Space Enumeration

State space enumeration can be programmed in GrGen.NET utilizing the sequence constructs introduced up so far. GrGen.NET always operates on one host graph, over one combined model, applying actions from one combined rule set — so everything is always within reach. That is by far the most simple approach to graph transformation, and for a tool which is not specially geared towards enumerating state spaces what makes most sense. Thus state space enumeration (or transformation between different graphs/models) must be realized within the one host graph. This can be achieved by virtually partitioning the host graph into smaller graphs or subgraphs, following a convention like this one: There are nodes of type `Graph` which are representing a state of the statespace; each such representative is an anchor node for a subgraph. The containment in the subgraph is denoted by a `contains` edge from a node of type `Graph` to all the nodes contained in this (sub)graph. The edges between the `Graph` nodes give the relationship between the subgraphs, i.e. successorship between the states of the statespace. The edges between the non-`Graph` nodes are the "normal" graph edges inside the subgraphs.

The key ingredients for state space enumeration then are

- The aforementioned modelling with anchor nodes linking to the subgraphs, i.e. states
- Backtracking angles — they allow to recursively enumerate and apply the matches of a rule found, exhaustively stepping through the state space, rolling back the effects on the graph of the previous match tried before carrying out the next step (see section 15.2)
- Pause insertions — they allow to write the interesting subgraphs found during state space search out into the host graph while effects recording of backtracking supervision is paused, so that they are not rolled back during backtracking (see section 15.2)
- Recursive sequences — they allow to nest backtracking angles dynamically, so it becomes possible to enumerate an entire state space with a sequence just implementing one backtracking step (see section 15.1)
- The capability to create a copy of a subgraph and insert it into the host graph (see section 16.5, esp. subsection 16.5.1)
- The ability to compare subgraphs (see section 16.4). Implemented in an optimized way with a subgraph attribute in the anchor nodes which contains a copy of the subgraph in the host graph, ready to be compared to the current graph, in order to decide whether that one should be inserted into the state space or purged for being an isomorphic copy of an already available instance.
- Adjacency and induced functions to compute the subgraphs from the host graph following the `contains` edges from the anchor nodes (see section 13.2, Connectedness Queries and section 14.1, Subgraph Operations)
- The `auto` filters from 21.3 finally allow to optimize performance by filtering symmetric matches stemming from automorphic patterns. This is a lot more efficient than creating states which are for sure isomorphic and filtering them later on by comparison with all the other states.

The modelling given above allows to employ the ability of GrGen.NET/yComp to visualize nested graphs from a flat graph by interpreting node containment along edges of a special type (cf. 18.1), ballooning the anchor node up to a subgraph.

An example implementation of this approach with a variant utilizing isomorphy checking and a variant without isomorphy checking can be found in the `tests/statespace` folder.

Feel free to drop in some `debug` prefixes before the `exec` (/`xgrs`) used to watch the assembling of the state space graph. Another example implementation can be found in the `tests/statespaceChemistry` folder. It shows how to enumerate all possible reaction results derivable from a set of start molecules according to some reaction rules. The molecules are just accumulated in the host graph, it is taken care by storage sets that each step only processes the ones available at the start of the step. At the end, each resulting molecule (i.e. connected component) is exported as a `.grsi` file.

CHAPTER 17

GRSHELL LANGUAGE

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

17.1 Building Blocks

GrShell is case sensitive. A line may be empty, may contain a shell command, or may contain a comment. A comment starts with a `#` and is terminated by end-of-line or end-of-file. The following items are required for representing text, numbers, and rule parameters.

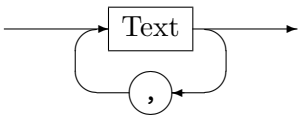
Text
May be one of the following:

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

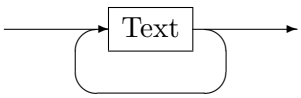
Due to the chosen parser generator shell keywords are not allowed for type names, attribute values and other entities (even if they are legal in the rule language). If this hits you, you can enclose the identifier by single or double quotes, i.e. Text can be used everywhere an identifier is required.

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

Parameters



SpacedParameters



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

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

Variable
: Identifier of a (graph global) variable that contains a graph element or a value. A double colon prefix as required in the sequences may be given, but as the shell only knows graph global variables, it is optional.

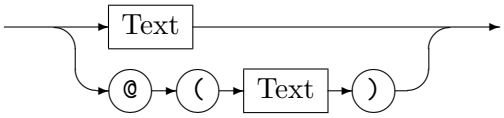
NodeType, EdgeType
: Identifier of a node type resp. edge type defined in the model of the current graph.

AttributeName
: Identifier of an attribute.

Graph
: Identifies a graph by its name.

Action
: Identifies a rule by its name.

GraphElement



The elements of a graph (nodes and edges) can be accessed both by their (graph global) variable identifier and by their *persistent name* specified through a constructor (see Section 17.2.6). The specializations *Node* and *Edge* of *GraphElement* require the corresponding graph element to be a node or an edge respectively.

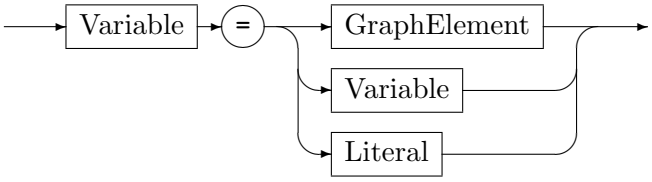
EXAMPLE (83)

We insert a node, anonymously and with a constructor (see also Section 17.2.6):

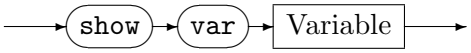
```
1  > new graph "../lib/lgsp-TuringModel.dll" G
2  New graph "G" of model "Turing" created.
3
4  # insert an anonymous node...
5  # it will get a persistent pseudo name
6  > new :State
7  New node "$0" of type "State" has been created.
8  > delete node @("$0")
9
10 # and now with constructor
11 >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

NOTE (42)

Persistent names will be saved (`save graph...`, see Section 17.2.4) and exported, and, if you visualize a graph (`dump graph...`, see Section 17.2.4), graph elements will be labeled with their persistent names. Persistent names have to be unique for a graph (the graph they belong to).



Assigns the variable or persistent name *GraphElement* or literal to *Variable*. If *Variable* has not been defined yet, it will be defined implicitly. As usual for scripting languages, variables have neither static types nor declarations. The variables known to GrShell are the graph global variables (see 8 for the distinction between graph global and sequence local variables).

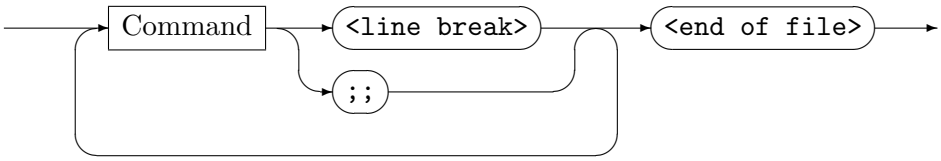


Prints the content of the specified variable.

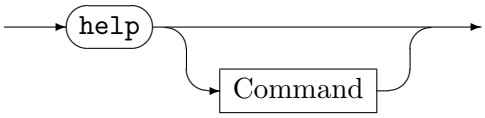
17.2 GrShell Commands

This section describes the GrShell commands. Commands are assembled from basic elements. As stated before commands are terminated by line breaks. Alternatively commands can be terminated by the `;;` symbol. Like an operating system shell, the GrShell allows you to span a single command over n lines by terminating the first $n-1$ lines with a backslash.

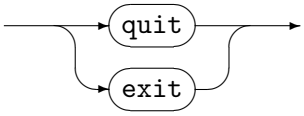
Script



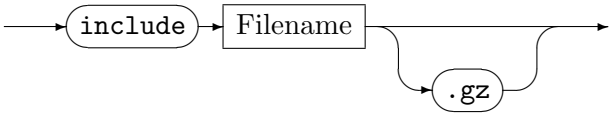
17.2.1 Common Commands



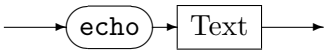
Displays an information message describing all the supported commands. A command `Command` displayed with `...` has further help available, which can be displayed with `help Command`.



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



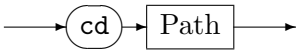
Executes the GrShell script *Filename* (which might be zipped). A GrShell script is just a plain text file containing GrShell commands. They are treated as they would be entered interactively, except for parser errors. If a parser error occurs, execution of the script will stop immediately.



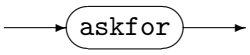
Prints *Text* onto the GrShell command prompt.

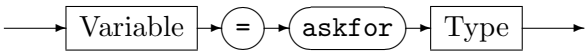
Prints the path to the current working directory.

Lists the directories and files in the current working directory, files relevant to GrGen are printed highlighted.



Changes the current working directory to the path given.





The `askfor` command just waits until the user presses enter. The `askfor` assignment interactively asks the user for a value of the specified type. The entered value is type checked against the expected type, and assigned to the given variable in case it matches. If the type is a value type, the user is prompted to enter a value literal with the keyboard. If the type is a graph element type, the user is prompted to enter the graph element by double clicking in yComp. Note that in this case the debug mode must have been enabled before. (The command is equivalent to `debug exec Variable=$%(Type)`.)

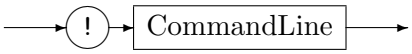
EXAMPLE (84)

```
x = askfor int
```

asks the user to enter an integer value; pressing 4 then 2 then enter will do fine.

```
x = askfor Node
```

asks the user to select a graph element in yComp; double clicking any node will do fine.

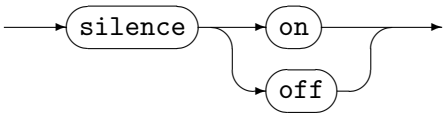


CommandLine is an arbitrary text, the operating system attempts to execute.

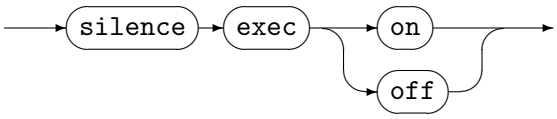
EXAMPLE (85)

On a Linux machine you might execute

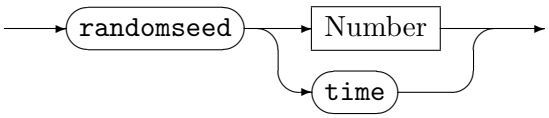
```
1 !sh -c "ls␣|␣grep␣stuff"
```



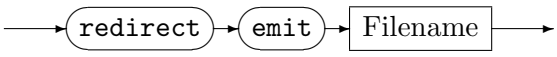
Switches the new node / edge created / deleted messages on(default) or off. Switching them off allows for much faster execution of scripts containing a lot of creation commands.



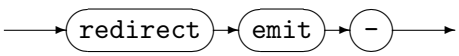
During non-debug sequence execution every second match statistics are printed to the console; they allow to assess the progress of long-running transformations. With this command they can be disabled (or enabled again). Switching them off may be of interest if own debug messages printed via emit from the sequences (or rules) should not be disturbed.



Sets the random seed to the given number for reproducible results when using the $-operator-prefix or the random-match-selector, whereas time sets the random seed to the current time in ms.



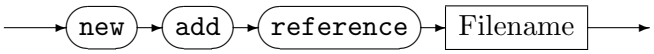
Redirects the output of the emit-statements in the rules from stdout to the given file.



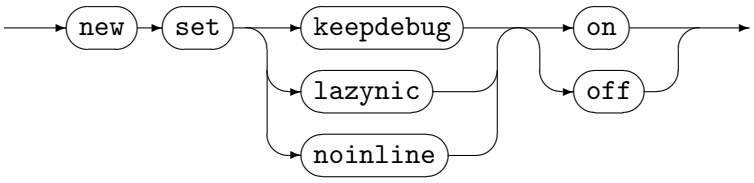
Redirects the output of the emit-statements in the rules to stdout (again).

17.2.2 Graph Commands

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

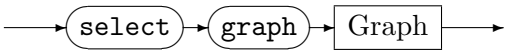


Configures a reference to an external assembly *Filename* to be linked into the generated assemblies, maps to the `-r` option of `grgen.exe` (cf. 1.7.1).

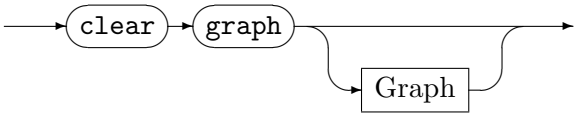


Configures the compilation of the generated assemblies to keep the generated files and to add debug symbols, or configures the generation of the matchers to execute negatives, independents, and conditions only at the end of matching (normally asap), or configures the generation of the matchers to never inline subpatterns. Maps to the `-keep` and the `-debug` options or to the `-lazynic` or to the `-noinline` option of `grgen.exe` (cf. 1.7.1).

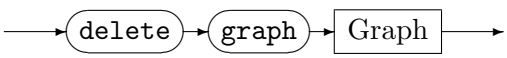
Displays a list of currently available graphs.



Selects the current working graph. This graph acts as *host graph* for graph rewrite sequences (see also Sections 1.5 and 17.2.8). Though you can define multiple graphs, only one graph can be the active "working graph".



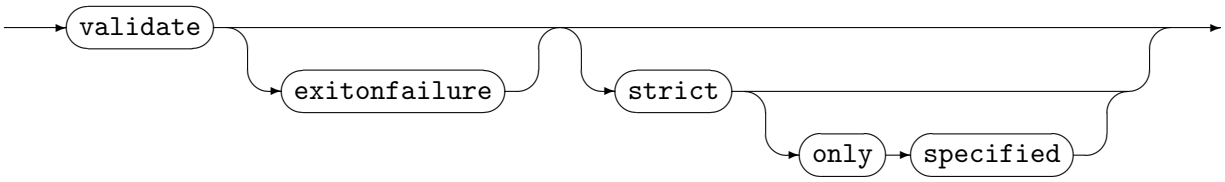
Deletes all graph elements of the current working graph resp. the graph *Graph*.



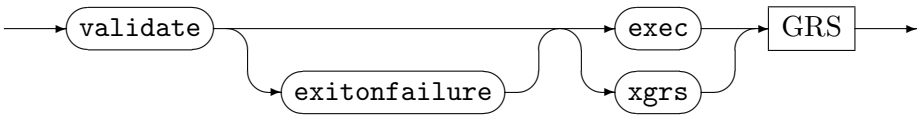
Deletes the graph *Graph* from the backend storage.

17.2.3 Validation Commands

GrGen.NET offers two different graph validation mechanisms, the first checks against the connection assertions specified in the model, the second checks against an arbitrary graph rewrite sequence containing arbitrary tests and rules.



Validates if the current working graph fulfills the connection assertions specified in the corresponding graph model (cf. 3.2.2). Validate without the strict modifier checks the multiplicities of the connections it finds in the host graph, it ignores node-edge-node connections which are available in the host graph but have not been specified in the model. The *strict* mode additionally requires that all the edges available in the host graph must have been specified in the model. This requirement is too harsh for models where only certain parts are considered critical enough to be checked or might be a too big step in tightening the level of structural checking in an already existing large model. So some form of selective strict checking is supported: The *strict only specified* mode requires strict matching (i.e. that all edges are covered) only of the edges for which connection assertions have been specified in the model.



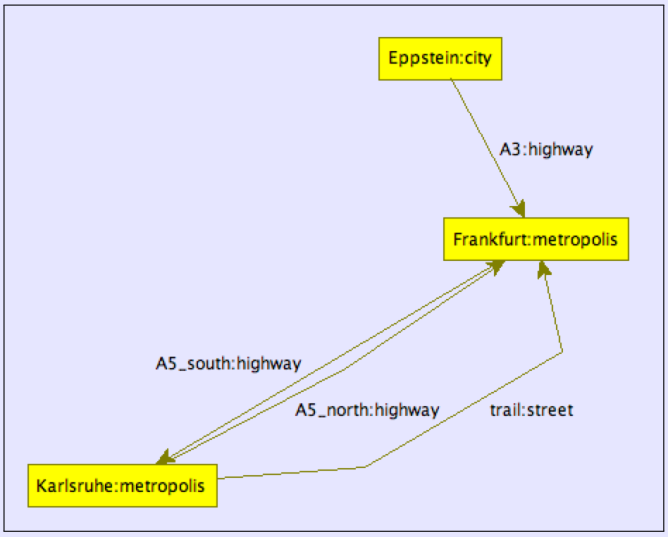
Validates if the current working graph satisfies the graph rewrite sequence given. Before the graph rewrite sequence is executed, the instance graph gets cloned; the sequence operates on the clone, allowing you to change the graph as you want to, without influence on the host graph. Validation fails iff the sequence fails. This gives a rather costly but extremely flexible and powerful mechanism to specify graph constraints. The GrShell is exited with an error code if `exitonfailure` is specified and the validation fails.

EXAMPLE (86)

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

```
model Map;

node class city;
node class metropolis;

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

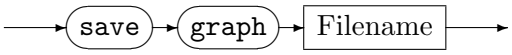
The node constraint on *highway* requires all the metropolises to be connected by highways. Now have a look at the following graph:



This graph is valid but not strict valid.

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

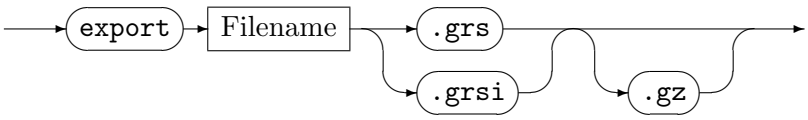
17.2.4 Graph Input and Output Commands



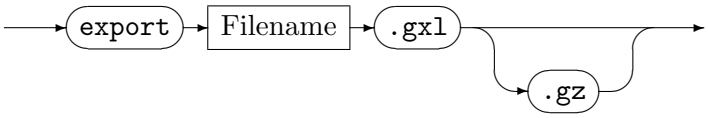
Dumps the current graph as GrShell script into *Filename*. The created script includes

- selecting the backend
- creating a new graph with all nodes and edges (including their persistent names)
- restoring the (graph global) variables
- restoring the visualisation styles

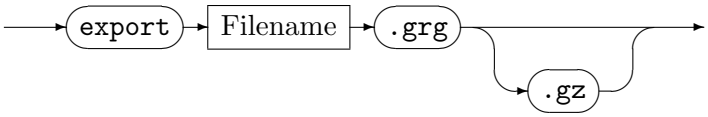
but not necessarily using the same commands you typed in during construction. Such a script can be loaded and executed by the `include` command (see Section 17.2.1).



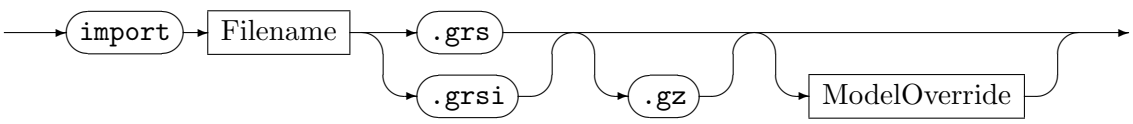
Exports an instance graph in GRS (.grs/.grsi) format, which is a reduced GrShell script (it can get imported and exported on API level20.4 without using the GrShell). This is the recommended standard format. It contains the `new graph` command, followed by `new node` commands, followed by `new edge` commands. If the `.gz` suffix is given the graph is saved zipped. The export is only complete with the model of the graph given in the `.gm` file. Exporting fails if the graph model contains attributes of `object`-type. The `save` command is for saving about a GrShell session including graph global variables and visualization commands, the goal of the `export` command is basic graph rewrite system interoperability.



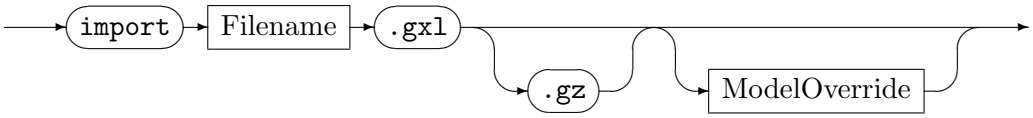
Exports an instance graph and a graph model in GXL format [WKR02, HSESW05], which is somewhat of a standard format for graphs of graph rewrite systems, but suffers from the well-known XML problems – it is barely human-readable and bloated. (Besides tools claiming to support it often can't import the export of the other.) Exporting fails if the graph model contains attributes of container or `object`-type. If the `.gz` suffix is given the graph is saved zipped.



Exports an instance graph in GRG format, i.e. as one GrGen rule with an empty pattern and a large modify part. There is no importer existing, this format is not for normal use! If the `.gz` suffix is given the graph is saved zipped.



Imports the specified graph instance in GRS (.grs/.grsi) format (the *reduced* GrShell script, a saved graph can only be imported by `include` (but an exported graph can be imported by `include`, too)). The graph model referenced in the .grs/.grsi must be available as `.gm`-file. If a model override of the form `Filename.gm` is specified, the given model will be used instead of the model referenced in the GRS file. If a model override of the form `Filename.grg` is specified, the model(s) of the given rule file will be used instead of the model in the GRS file. If the `.gz` suffix is given the graph is expected to be zipped.



Imports the specified graph instance and model in GXL format. If a model override of the form `Filename.gm` is specified, the given model will be used instead of the model in the GXL file. If a model override of the form `Filename.grg` is specified(s), the model of the given rule file will be used instead of the model in the GXL file. The `.gxl`-graph must be compatible to the `.gm`-model/`.grg`-model. If the `.gz` suffix is given the graph is expected to be zipped.

NOTE (43)

Normally you are not only interested in importing a GXL graph (and viewing it), but you want to execute actions on it. The problem is that the actions are model dependent. So, in order to apply actions, you must use a model override, which works this way:

1. `new graph "YourName.grg"`
   This creates the model library lgsp-YourNameModel.dll and the actions library lgsp-YourNameActions.dll (which depends on the model library generated from the `"using YourName;"`).

2. `import InstanceGraphOnly.gxl YourName.gm`
   This imports the instance graph from the .gxl but uses the model specified in YourName.gm (it must fit to the model in the .gxl in order to work).

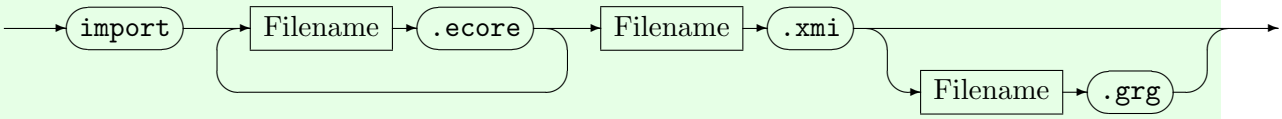
3. `select actions lgsp-YourNameActions.dll`
   This loads the actions from the actions library in addition to the already loaded model and instance graph (cf. 17.2.8).

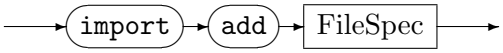
4. Now you are ready to use the actions.

As of version 3.0beta you can specify a `.grg` as model override; basically it does what the given enumeration does.

NOTE (44)

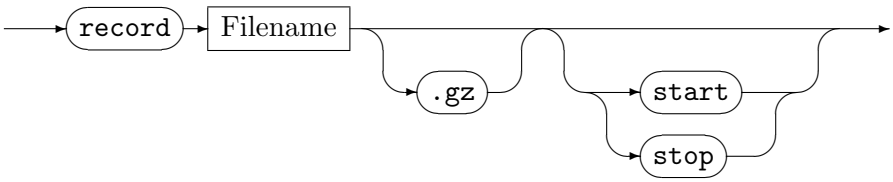
Further formats available for import are `.ecore` plus `.xmi` of Eclipse Modeling Framework (EMF). These are formats common to the model transformation community which are not directly geared towards graphs, so they can't be imported directly. Instead during the import process an intermediate `.gm` is written which is equivalent to the `.ecore` given – you may inspect it to see how the content gets mapped (the importer maps classes to GrGen node classes, their attributes to corresponding GrGen attributes, and their references to GrGen edge classes; inheritance is transferred one-to-one, and enumerations are mapped to GrGen enums; class names are prefixed by the names of the packages they are contained in and edge type names are prefixed by the names of the node types they originate from, to prevent name clashes). After this metamodel transformation the instance graph XMI adhering to the Ecore model thus adhering to the just generated equivalent GrGen graph model gets imported. Furthermore you can give specify a `.grg` containing the rules to apply (using further rule and model files). An export is not available – we coded the export we needed for the GraBaTs/TTC challenges the importer was written for with emit statements.



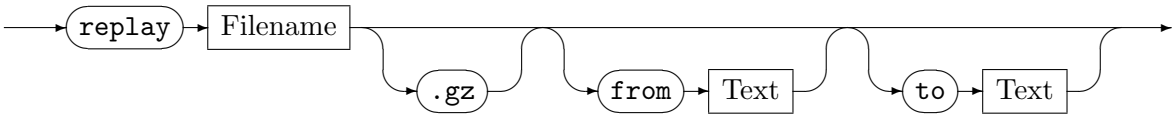


Imports the graph in the specified file and adds it to the current graph (instead of overwriting the old graph with the new graph). The `FileSpec` is of the same format as the file specification in the other two imports.

17.2.5 Graph Change Recording and Replaying



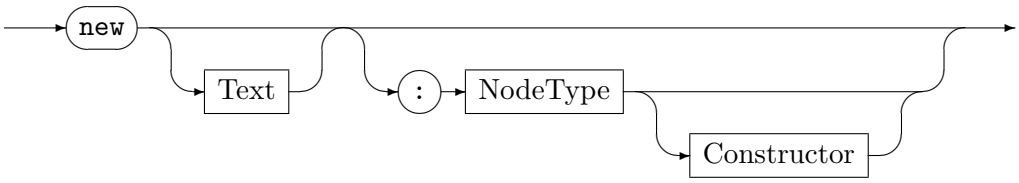
The record command starts or stops recording of graph changes to the specified file. If neither start nor stop are given, recording to the specified file is toggled (i.e. started if no recording to the file is underway or stopped if the file is already recorded to). Recording starts with an export (cf. 17.2.4) of the instance graph in GRS (.grs/.grsi) format, afterwards the command returns but all changes to the instance graph are recorded to the file until the recording stop command is issued. Furthermore the values given in the `record` statements (cf. 14.1) from the sequences are written to the recording (this allows you to mark states). If the `.gz` suffix is given the recording is saved zipped. You may start and stop recordings to different files at different times, every file receives the graph changes and records statements occuring during the time of the recording. Note: As a debugging help a recording does not only contain graph manipulation commands (cf. 17.2.6) but also comments telling about the rewrites and transaction events which occured (whose effects were recorded).



The replay command plays a recording back: the graph at the time the recording was started is recreated, then the changes which occured are carried out again, so you end up with the graph at the time the recording was stopped. Instead of replaying the entire GRS file you may restrict replaying to parts of the file by giving the line to start at and/or the line to stop at. Lines are specified by their textual content which is searched in the file. If a *from* line is given, all lines from file begin on including this line are skipped, then replay starts. If a *to* line is given, only the lines from the starting point on, until-excluding this one are executed (i.e. all lines from-including this one until file end are skipped). Normally you reference with `from` and `to` comment lines you write with the `record` statement (cf. 14.1) in the sequences, marking relevant states during a transformation process. An example for record and replay is given in `tests/recordreplay`.

17.2.6 Graph Manipulation Commands

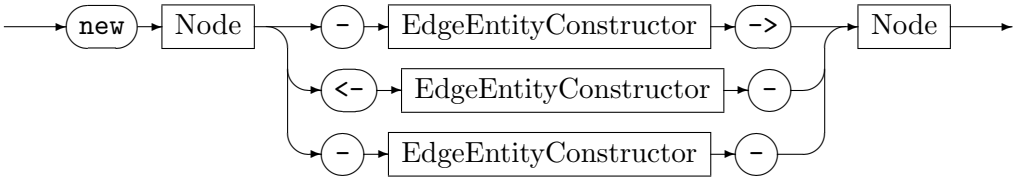
Graph manipulation commands alter existing graphs; they allow to create, retype and delete graph elements and change attributes. These are tasks which are or at least should be carried out by the rules of the rule language in the first place. On shell level they are available and mainly used as elementary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



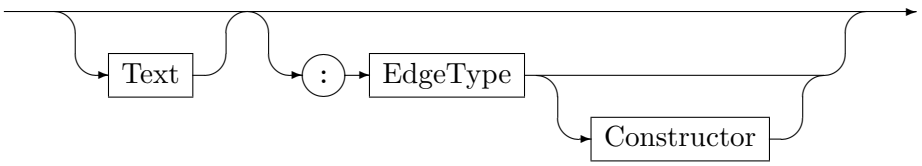
Creates a new node within the current graph. Optionally a variable *Text* is assigned to the new node. If *NodeType* is supplied, the new node will be of type *NodeType* and attributes can be initialized by a constructor. Otherwise the node will be of the base node class type *Node*.

NOTE (45)

The GrShell can reassign variables. This is in contrast to the rule language (Chapter 4), where we mainly use *names* (with exception of var and ref input variables and def entities).

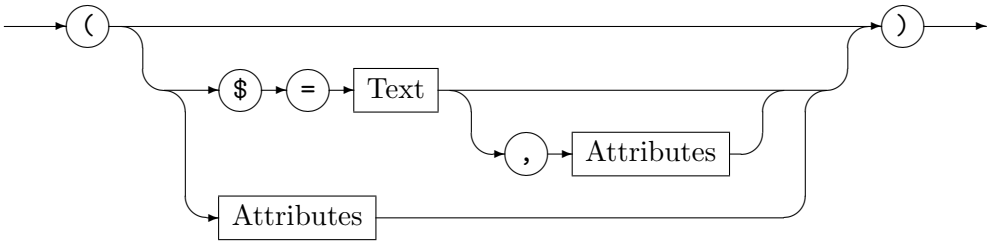


EdgeEntityConstructor

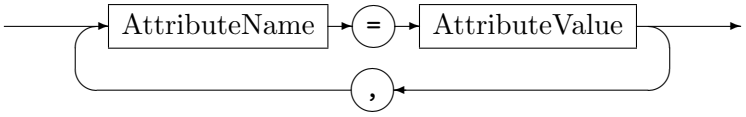


Creates a new edge within the current graph between the specified nodes, directed from the first to the second *Node* in the case of `-->`, directed from the second to the first *Node* in the case of `<--`, or undirected in the case of `--`. Optionally a variable *Text* is assigned to the new edge. If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

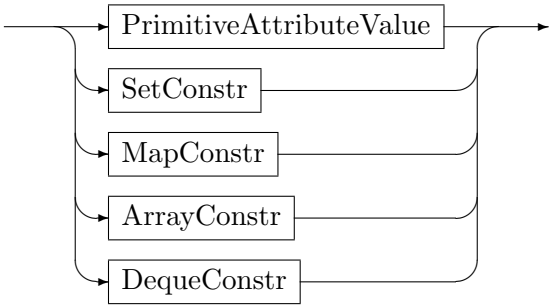
Constructor



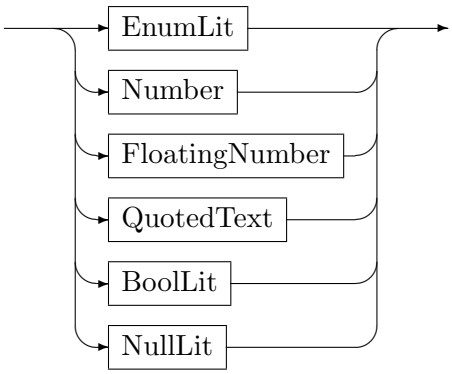
Attributes



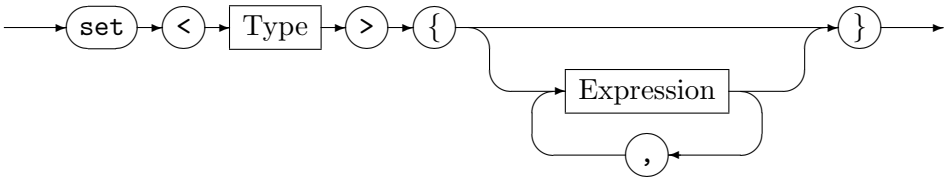
AttributeValue



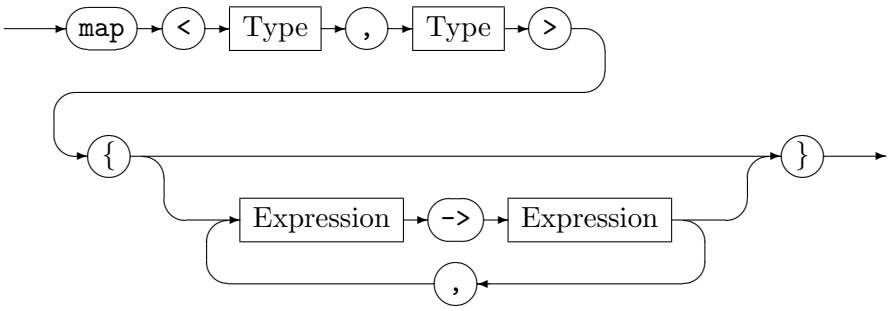
PrimitiveAttributeValue



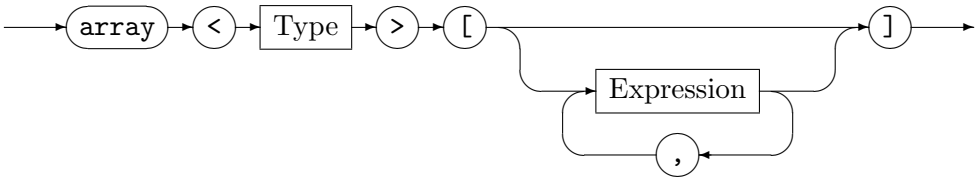
SetConstr



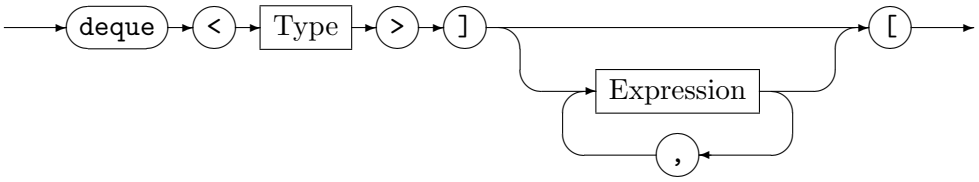
MapConstr



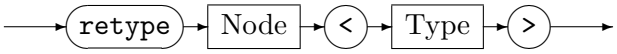
ArrayConstr



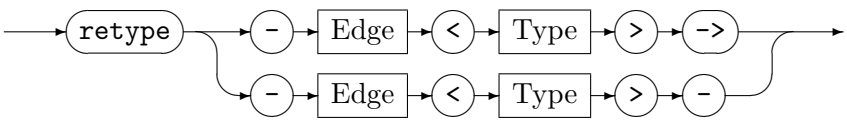
DequeConstr



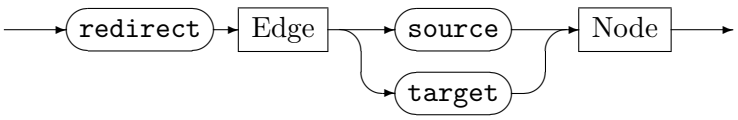
A constructor is used to initialize a new graph element (see `new ...` below). A comma separated list of attribute declarations is supplied to the constructor. Available attribute names are specified by the graph model of the current working graph. All the undeclared attributes will be initialized with default values, depending on their type (`int` ← `0`; `long` ← `0L`; `byte` ← `0Y`; `short` ← `0S`; `boolean` ← `false`; `float` ← `0.0f`; `double` ← `0.0`; `string` ← `""`; `set<T>` ← `set<T>{}`; `map<S,T>` ← `map<S,T>{}`; `array<T>` ← `array<T>[]`; `deque<T>` ← `deque<T>][`; `enum` ← unspecified;).
The `$` is a special attribute name: a unique identifier of the new graph element. This identifier is also called *persistent name* (see Example 83). This name can be specified by a constructor only.



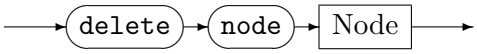
Retypes the node *Node* from its current type to the new type *Type*. Attributes common to initial and final type are kept. Incident edges are kept as well.



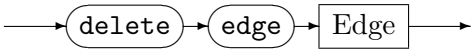
Retypes the edge *Edge* from its current type to the new type *Type*. Attributes common to initial and final type are kept. Incident nodes are kept as well.



Redirects the edge *Edge* from the old source or target node to the new source or target *Node* given.

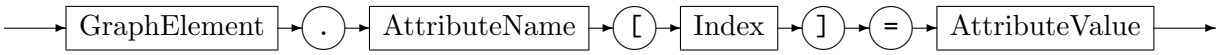


Deletes the node *Node* from the current graph. Incident edges will be deleted as well.

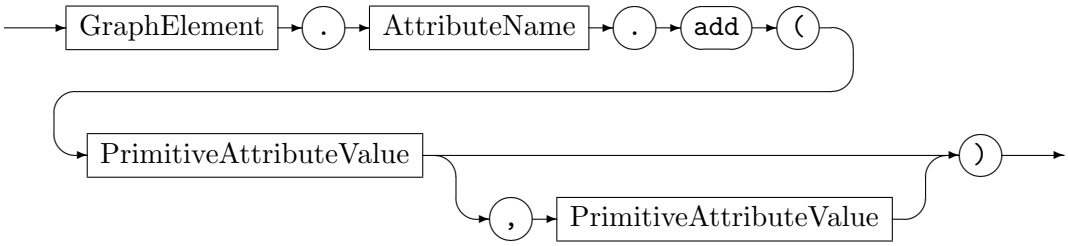


Deletes the edge *Edge* from the current graph.

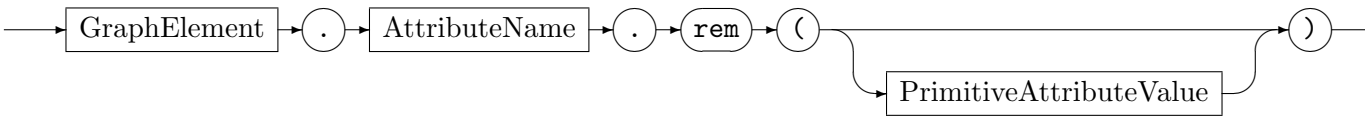
Set the attribute *AttributeName* of the graph element *GraphElement* to the value *AttributeValue* (for the different possible attribute values see above).



Overwrite the value in the array or deque or map attribute *AttributeName* of the graph element *GraphElement* at the integer position or key value *Index* with the value *AttributeValue*.

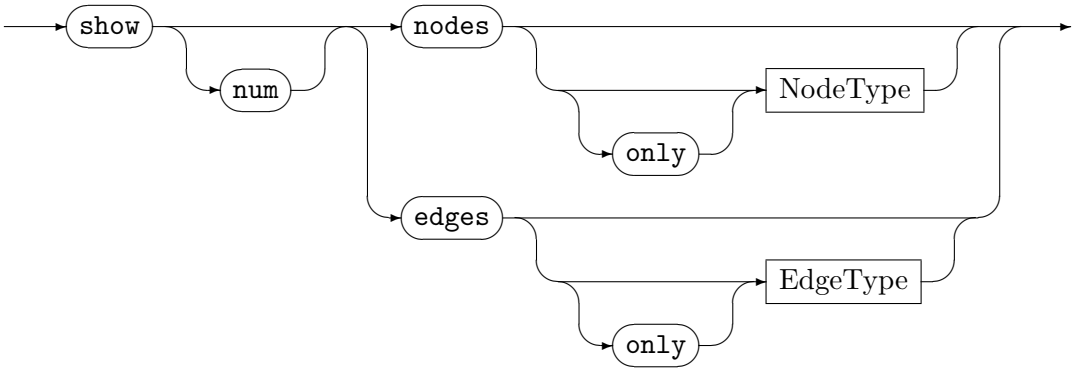


Add the value *PrimitiveAttributeValue* (for the different possible primitive attribute values see above) to the set valued attribute *AttributeName* of the graph element *GraphElement* or add the key-value pair consisting of the two *PrimitiveAttributeValue*s to the map valued attribute *AttributeName* of the graph element *GraphElement*. Or add the value *PrimitiveAttributeValue* to the end of the array/deque valued attribute *AttributeName* of the graph element *GraphElement* in the one parameter case or insert the *PrimitiveAttributeValue* to the of the array/deque valued attribute *AttributeName* of the graph element *GraphElement* at the index given by the second parameter in the two parameter case.

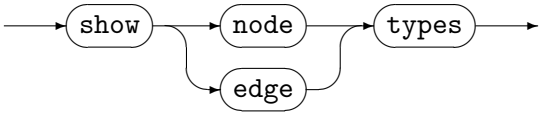


Remove the value *PrimitiveAttributeValue* from the set valued attribute *AttributeName* of the graph element *GraphElement* or remove the key *PrimitiveAttributeValue* from the map valued attribute *AttributeName* of the graph element *GraphElement* or remove the index *PrimitiveAttributeValue* from the array valued attribute *AttributeName* of the graph element *GraphElement* or remove the index *PrimitiveAttributeValue* from the deque valued attribute *AttributeName* of the graph element *GraphElement* or remove the end element from the array valued attribute *AttributeName* of the graph element *GraphElement* in the zero parameter case or remove the first element from the deque valued attribute *AttributeName* of the graph element *GraphElement* in the zero parameter case.

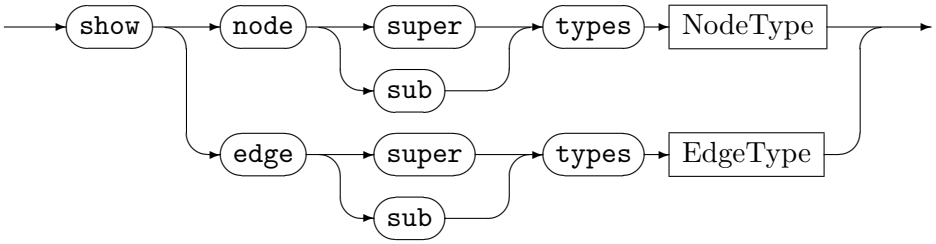
17.2.7 Graph and Model Query Commands



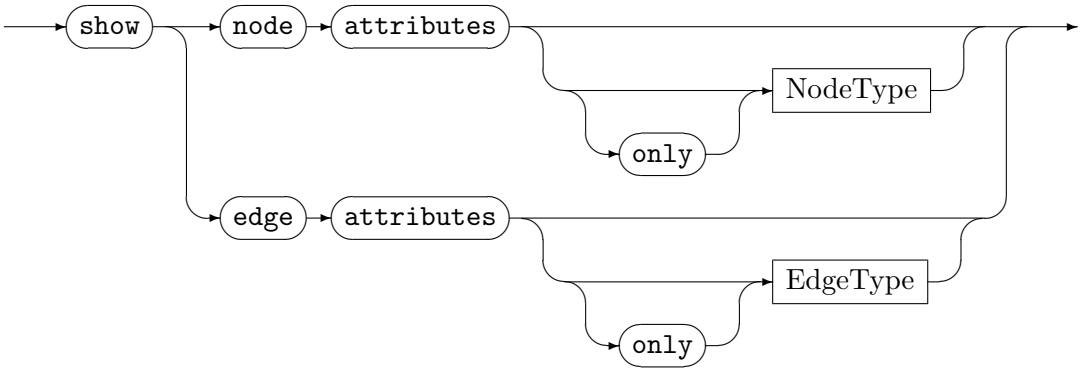
Gets the persistent names and the types of all the nodes/edges of the current graph. If a node type or edge type is supplied, only elements compatible to this type are considered. The `only` keyword excludes subtypes. Nodes/edges without persistent names are shown with a pseudo-name. If the command is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



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



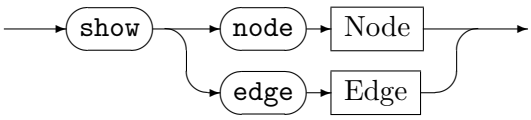
Gets the inherited/descendant types of *NodeType*/*EdgeType*.



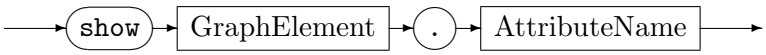
Gets the available node/edge attribute types. If *NodeType*/*EdgeType* is supplied, only attributes defined in *NodeType*/*EdgeType* are diplayed. The `only` keyword excludes inherited attributes.

NOTE (46)

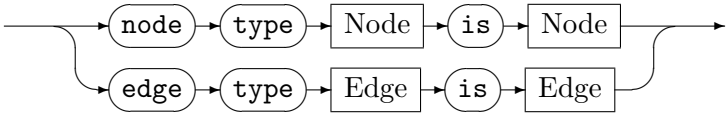
The `show nodes/edges attributes...` command covers types and *inherited* types. This is in contrast to the other `show...` commands where types and *sub*types are specified or the direction in the type hierarchy is specified explicitly, respectively.



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



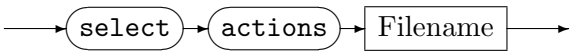
Displays the value of the specified attribute.



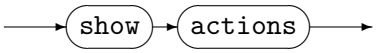
Gets the information whether the first element is type-compatible to the second element.

17.2.8 Action Commands (XGRS)

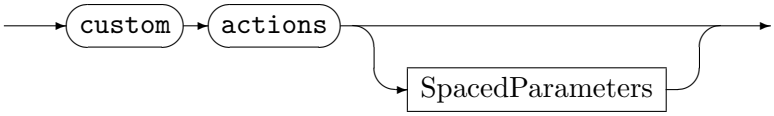
An *action* denotes a graph rewrite rule.



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

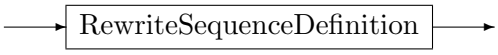


Lists all the rules of the loaded rule set, their parameters, and their return values. Rules can return a set of graph elements.



Executes an action specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSPBackend see Section 17.3.2).

GraphRewriteSequence

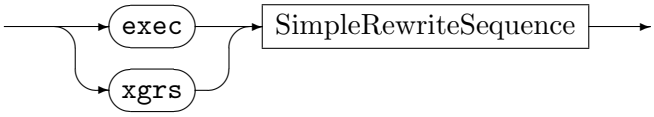


This command allows to define a named sequence at runtime, for *RewriteSequenceDefinition* have a look here 15.1 in the rule application control language chapter. Especially it allows to replace an old sequence definition, but only if the signature is identical. Compiled sequences defined in rule files can't be replaced. The defined sequence can then be used from following graph rewrite sequences (or following sequence definitions) in the shell.

EXAMPLE (87)

```
# a sequence definition (of an interpreted sequence) is only available
# after it was registered the first time
# but it can get overwritten with a sequence of the same signature
# -> (self or mutually) recursive sequences must be constructed with empty body first
def chain(first:A):(last:A){ true }
def chain(first:A):(last:A){ if{(next:A)=chainPiece(first); (last)=chain(next); last=first} }
```

GraphRewriteSequence



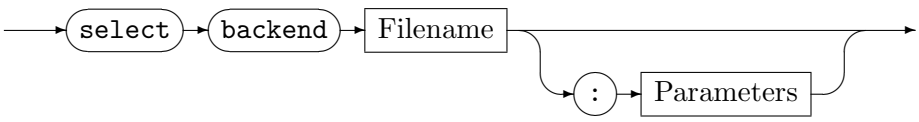
This executes the graph rewrite sequence *SimpleRewriteSequence*. See Chapter 8 for graph rewrite sequences. Additionally to the variable assignment in rule-embedded graph rewrite sequences, you are also able to assign *persistent names* to parameters via `Variable = (Text)`.

Graph elements returned by rules can be assigned to variables using `(Parameters) = Action`. The desired variable identifiers have to be listed in *Parameters*. Graph elements required by rules must be provided using `Action (Parameters)`, where *Parameters* is a list of variable identifiers. For undefined variables see Section 4.2, *Parameters*.

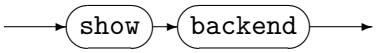
17.3 Backend Commands

GrGen.NET is built to support multiple backends implementing the model and action interfaces of libGr. This is roughly comparable to the different storage engines MySQL offers. Currently only one backend is available, the libGr search plan backend, or short LGSPBackend.

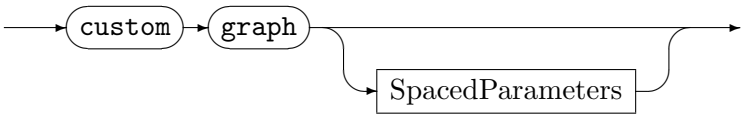
17.3.1 Backend Selection and Custom Commands



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



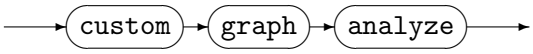
List all the parameters supported by the currently selected backend. The parameters can be provided to the `select backend` command.



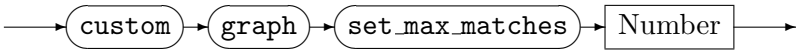
Executes a command specific to the current backend. If *SpacedParameters* is omitted, a list of available commands will be displayed (for the LGSP backend see Sections 17.3.2).

17.3.2 LGSPBackend Custom Commands

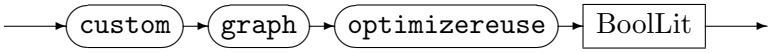
The LGSPBackend supports the following custom commands:



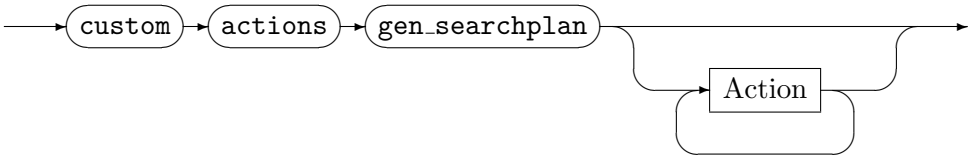
Analyzes the current working graph. The analysis data provides vital information for efficient search plans. Analysis data is available as long as GrShell is running, i.e. when the host graph changes, the analysis data is still available but maybe obsolete.



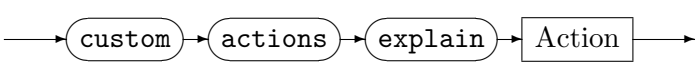
Sets the maximum amount of possible pattern matches to *Number*. This command affects the expression `[Rule]`. If *Number* is less or equal to zero, the constraint is reset.



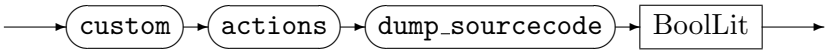
If set to false it prevents deleted elements from getting reused in a rewrite (i.e. it disables a performance optimization). If set to true, new elements may not be discriminable anymore from already deleted elements using object equality, hash maps, etc.



Creates a search plan (and executable code from it) for each rewrite rule *Action* using the data from analyzing the graph (`custom graph analyze`). Otherwise a default search plan is used. If no rewrite rule is specified, all rewrite rules are compiled anew. Analyzing and search plan/code generation themselves take some time, but they can lead to (massively) faster pattern matching, thus overall execution times; the less uniform the type distribution and edge wiring between the nodes, the higher the improvements. During the analysis phase the host graph must be in a shape "similar" to its shape when the main amount of work is done (there may be some trial-and-error steps at different time points needed to get the overall most efficient search plan.) A search plan is available as long as the current rule set remains loaded. Specify multiple rewrite rules instead of using multiple commands for each rule to improve the search plan generation performance.



Shows the search plan currently in use for *Action*, plus the subpatterns called by it. This is an inspection tool comparable to the `explain` command offered by SQL-databases to inspect the search plans of their queries. The explain command allows to evaluate the effects of performance optimization, esp. it allows to change the graph which serves as analysis data source for matcher regeneration, or to annotate the static rules with priorities (cf. 21.7), until a good search plan is built. The search plan is shown as a list of search commands (with commands not doing real matching work shown in parenthesis), which is executed from top to bottom; for more on the search commands have a look at section 22.3.



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

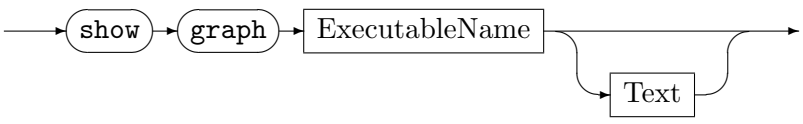
CHAPTER 18

VISUALIZATION AND DEBUGGING

This chapter gives an introduction into the visualization capabilites of yComp and into the graphical debugger of GrGen.NET, which is offered by GrShell in combination with yComp.

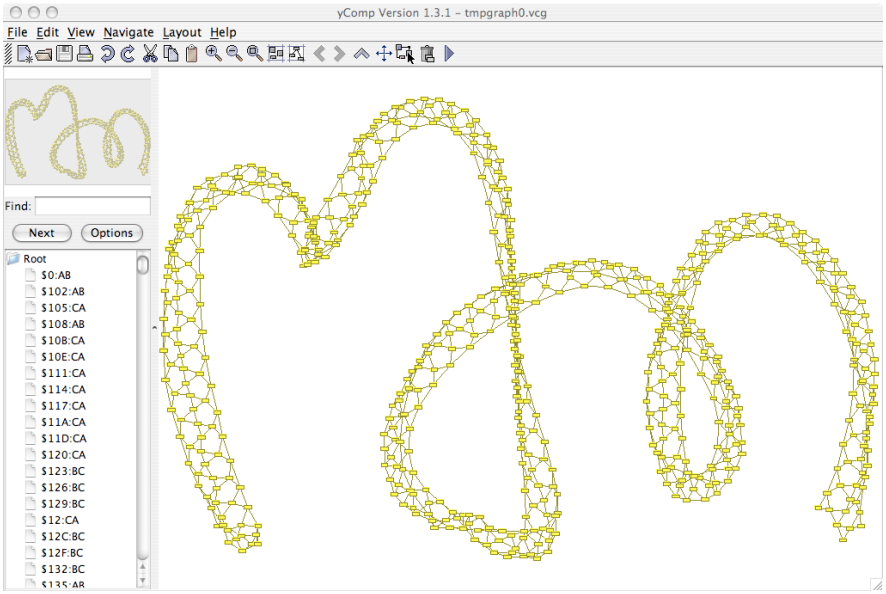
Other commands of use for debugging were already introduced in the shell chapter: `show var <Variable>` to print the content of a variable (but pressing the `v` key in the shell debugger is a lot more convenient) and `show <GraphElement>.<AttributeName>` to print the content of an attribute (but searching with `Ctrl-f` or `/` in yComp for the persistent name or an attribute value, hovering over the then highlighted graph element is more convenient, again); furthermore `record` and `replay` are interesting when you are debugging a transformation where you are choosing from the available matches by hand and want to try other paths later by choosing differently on a previous match: they allow you to save and restore graph states of interest, and to inspect the sequence of changes which lead to them in the `.grs`.

18.1 Graph Visualization Commands (Nested Layout)

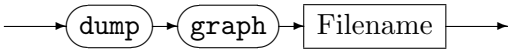


Dumps the current graph in VCG format into a temporary file. The temporary VCG file will be passed to the program *ExecutableName* as first parameter; further parameters, such as program options, can be specified by *Text*. If you use yComp[1] as executable (`show graph ycomp`), this may look like

[1] See Section 1.7.5.

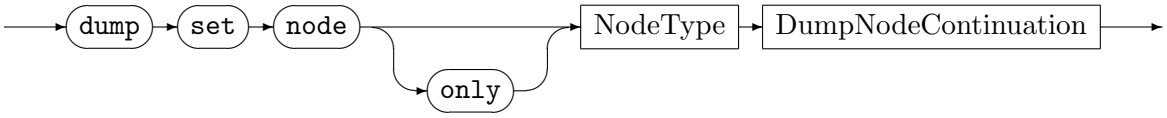


The temporary file will be deleted, when the application *Filename* is terminated if GrShell is still running at this time.

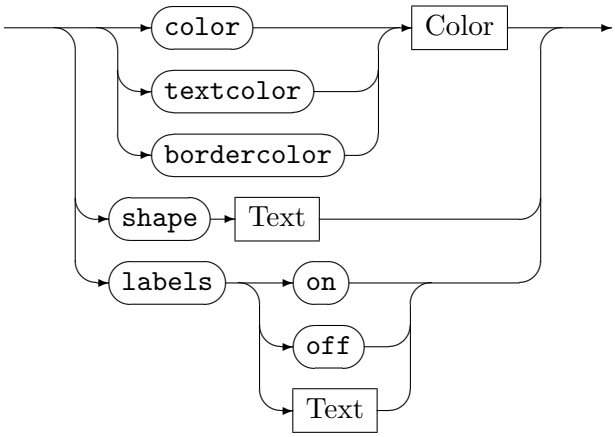


Dumps the current graph in VCG format into the file *Filename*.

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



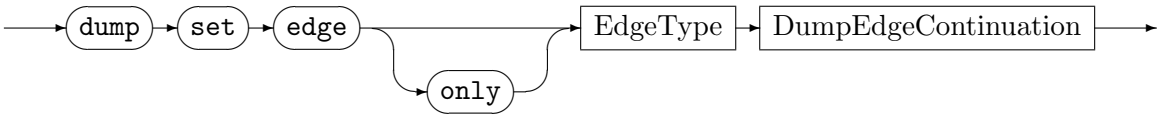
DumpNodeContinuation



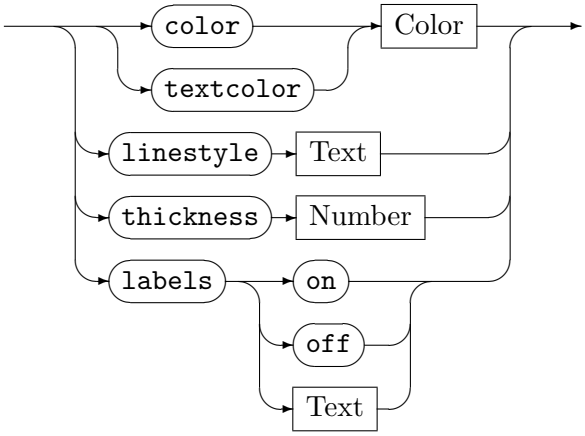
Sets the color, text color, border color, the shape or the label of the nodes of type *NodeType* and all of its subtypes. The keyword `only` excludes the subtypes. The following shapes are supported: `box`, `triangle`, `circle`, `ellipse`, `rhomb`, `hexagon`, `trapeze`, `uptrapeze`, `lparallelogram`, `rparallelogram`. Those are shape names of yComp (for a VCG definition see [San95]). The following colors are supported: `Black`, `Blue`, `Green`, `Cyan`, `Red`, `Purple`, `Brown`, `Grey`, `LightGrey`, `LightBlue`, `LightGreen`, `LightCyan`, `LightRed`, `LightPurple`, `Yellow` (default), `White`, `DarkBlue`, `DarkRed`, `DarkGreen`, `DarkYellow`, `DarkMagenta`, `DarkCyan`,

`Gold`, `Lilac`, `Turquoise`, `Aquamarine`, `Khaki`, `Pink`, `Orange`, `Orchid`, `LightYellow`, `YellowGreen`. These are the same color identifiers as in VCG/yComp files (for a VCG definition see [San95]).

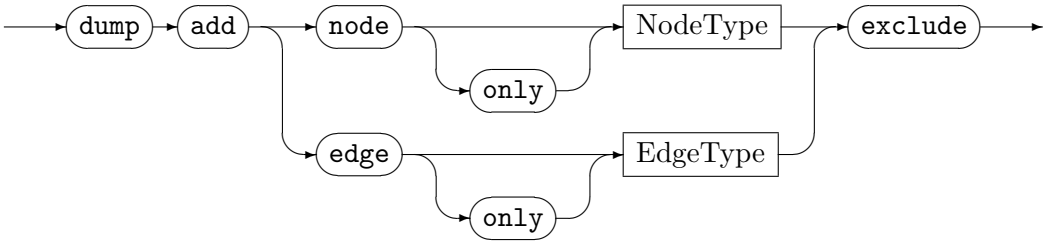
The default labeling is set to `on` with `Name:Type`, it can be overwritten by an specified label string (e.g. the source code line originating the node in a program graph) or switched off.



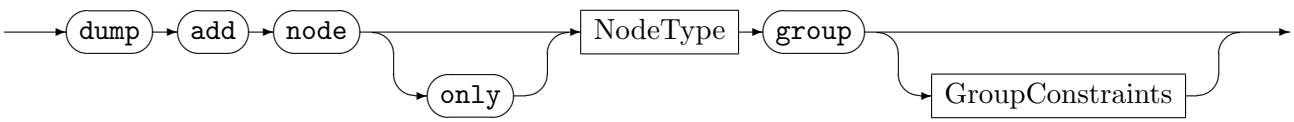
DumpEdgeContinuation



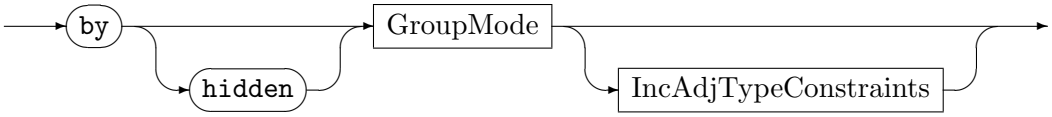
Sets the color, text color, the linestyle, the thickness of the line, or the label of the edges of type *EdgeType* and all of its subtypes. The keyword `only` excludes the subtypes. The available colors are specified above. The default labeling is set to `on` with `Name:Type`, it can be overwritten by an specified label string or switched off. The following linestyles are supported: `continuous` (default), `dotted`, `dashed`. The following thicknesses are supported: `1` (default), `2`, `3`, `4`, `5`.



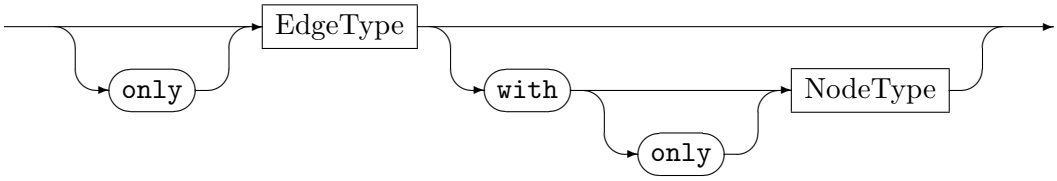
Excludes nodes/edges of type *NodeType*/*EdgeType* and all of its subtypes from output, for a node it also excludes its incident edges. The keyword `only` excludes the subtypes from exlusion, i.e. subtypes of *NodeType*/*EdgeType* are dumped.



GroupConstraints



IncAdjTypeConstraints

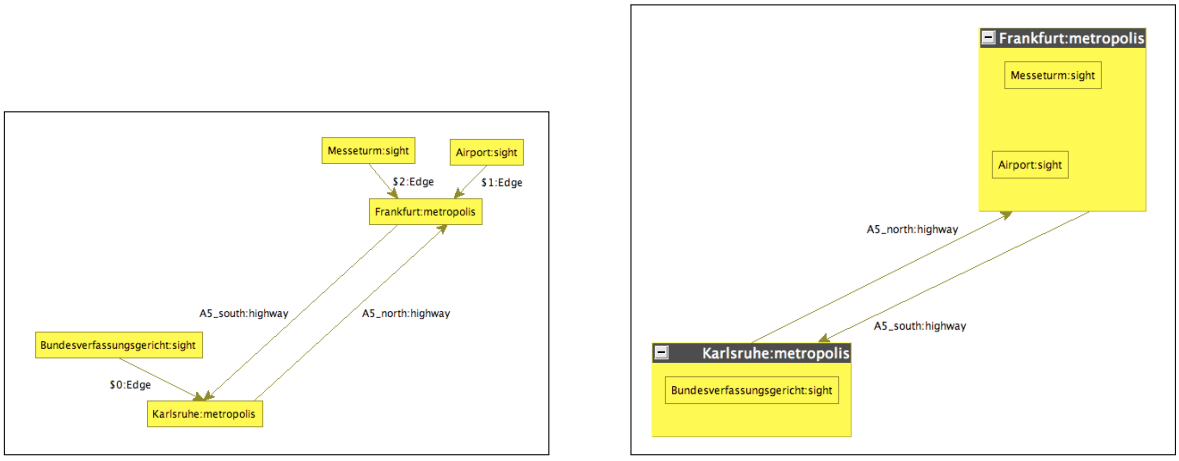


Declares *NodeType* and subtypes of *NodeType* as group node type. All the differently typed nodes that point to a node of type *NodeType* (i.e. there is a directed edge between such nodes) will be grouped and visibly enclosed by the *NodeType*-node (nested graph). `GroupMode` is one of `no`,`incoming`,`outgoing`,`any`; `hidden` causes hiding of the edges by which grouping happens. The `EdgeType` constrains the type of the edges which cause grouping, the `with` clause additionally constrains the type of the adjacent node; `only` excludes subtypes.

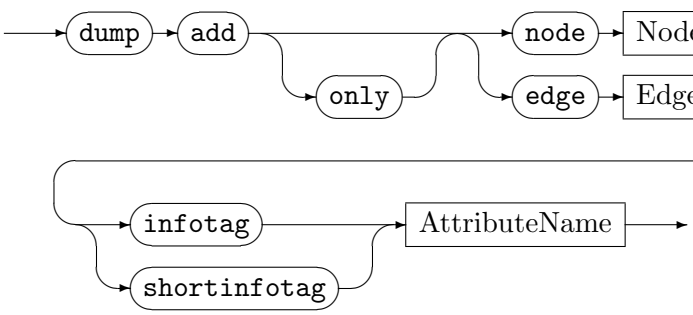
NOTE (47)

Only apply group commands on a graph if they indeed lead to a containment tree of groups. If the group commands would lead to a directed acyclic or even cyclic containment graph, the results are undefined. You may get duplicate edges (and nodes); the implementation is free to choose indeterministically between the possible nestings – it may even grow an arm and stab you in your back. (A conflict resultion heuristic used is to give the earlier executed `add group` command priority. But this mechanism is incomplete – you'd better refine your groups or change the model in that case. Using a model separating edges denoting direct containment from cross-linking edges by type is normally the better design, even disregarding visual node nesting.)

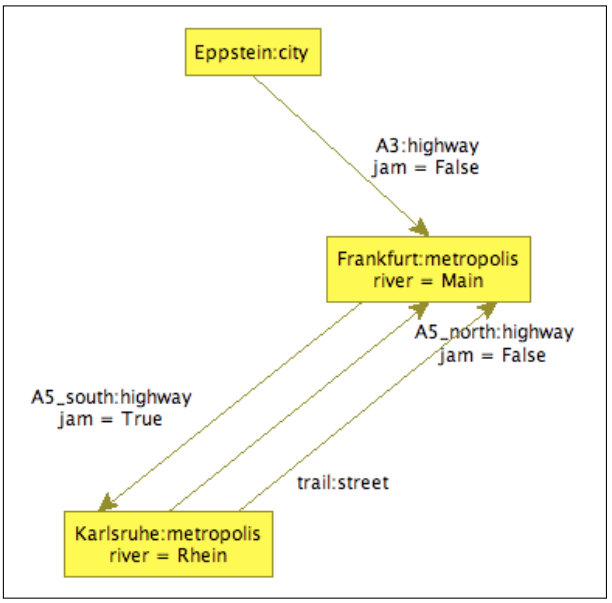
The following example shows *metropolis* ungrouped and grouped:

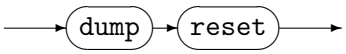


right side: dumped with `dump add node metropolis group`



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





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

NOTE (48)

Small graphs allow for fast visual understanding, but with an increasing number of nodes and edges they quickly loose this property. The group commands are of outstanding importance to keep readability with increasing graph sizes (e.g. for program graphs it allows to lump together expressions of a method inside the method node and attributes of the class inside the class node). Additional helpers in keeping the graph readable are: the capability to exclude elements from dumping (the less hay in the stack the easier to find the needle), the different colors and shapes to quickly find the elements of interest, as well as the labels/infotags/shortinfotags to display the most important information directly. Choose the layout algorithm and the options delivering the best results for your needs, organic and hierarchic or compiler graph (an extension of hierarchic with automatic edge cutting – marking cut edges by fat dots, showing the edge only on mouse over and allowing to jump to the other end on a mouse click) should be tried first.

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

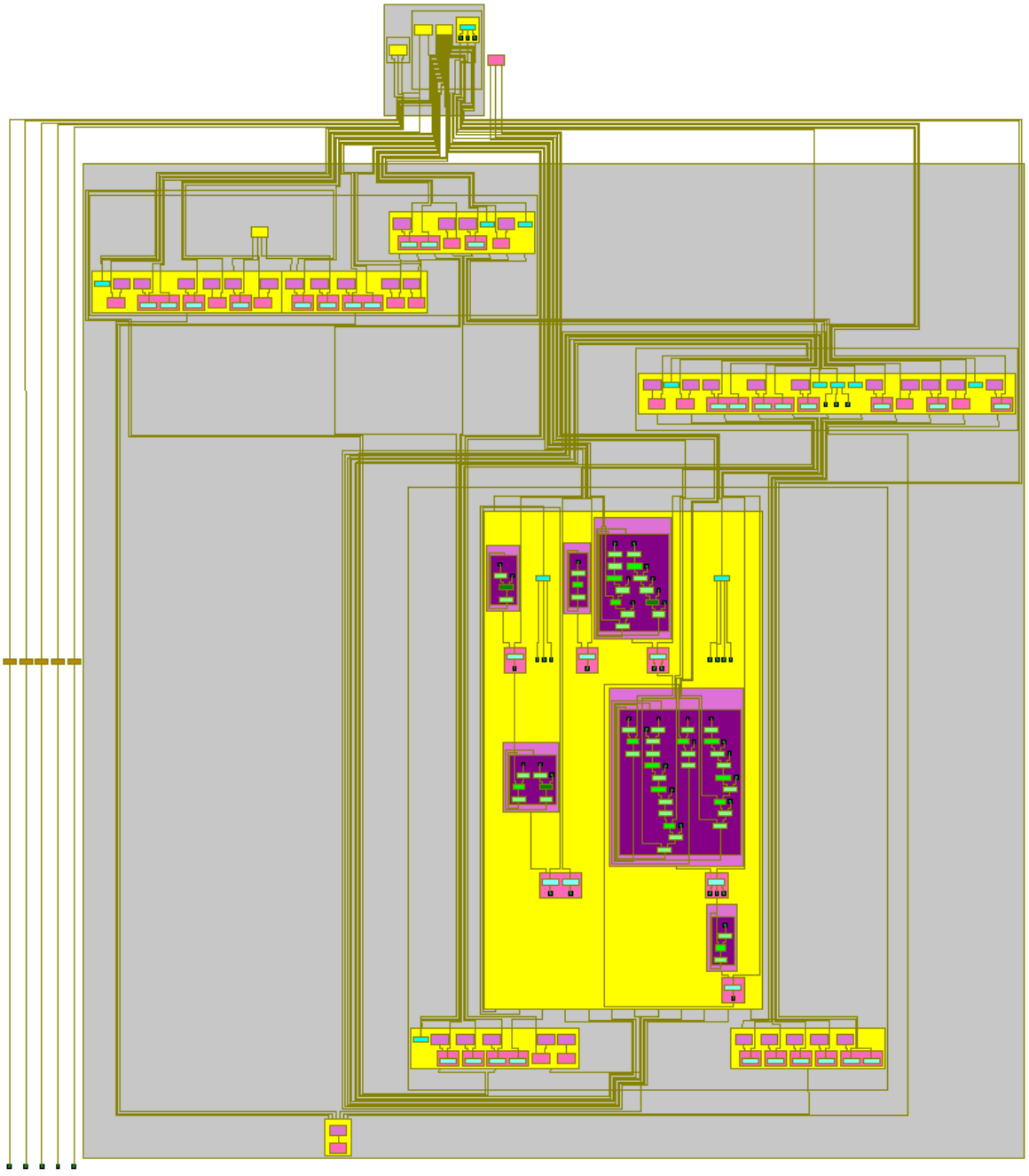


Figure 18.1: Overview of the initial program graph

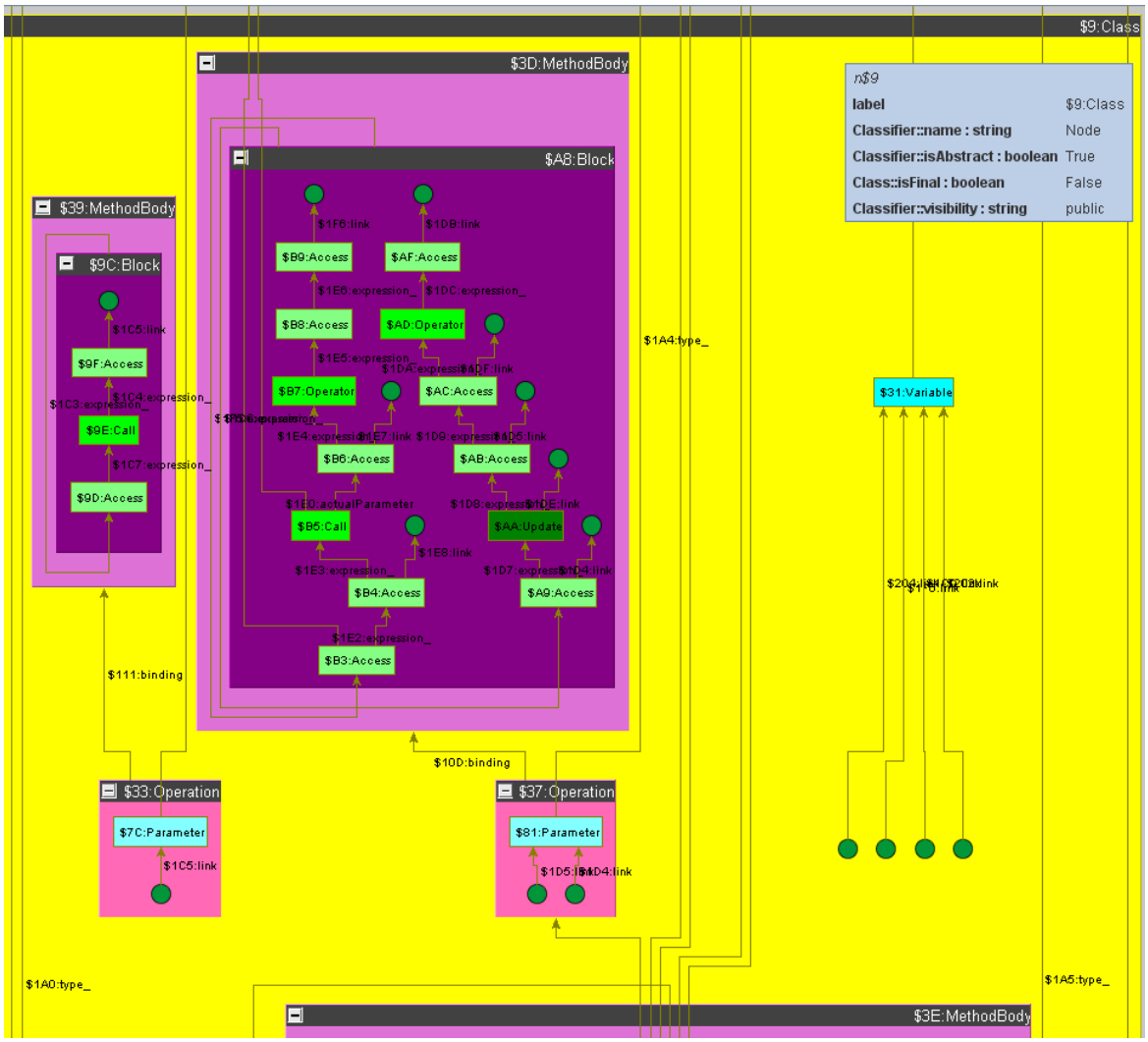


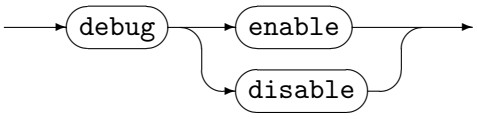
Figure 18.2: Some details of the “Node” class of the initial program graph

18.2 yComp Usage

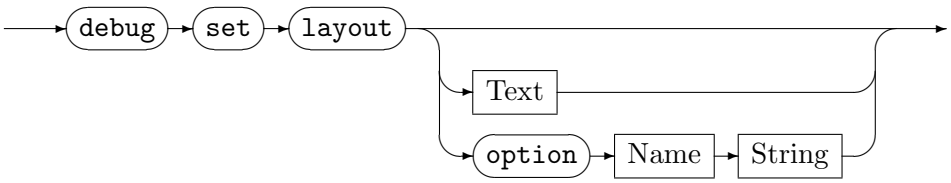
yComp [KBG+07] is the default graph viewer of GrGen, and – when started as a server process – can be controlled by the debugger of GrShell via a TCP/IP connection. Besides the things already mentioned in 1.7.5, we want to give the following hints:

- when started on a dump, you must press the rightmost play button to start layout
- play with the layout options offered in the `Layout` menu until you find a good visualization, configure it then in the GrShell; don't forget to press the play button to apply the changes
- you can pane by pressing and holding the middle mouse button while moving the mouse
- you can zoom with the mouse wheel at the position of the cursor
- hovering over graph elements displays the attributes
- you can select graph elements with the left mouse button and delete them with `del` to gain a better overview
- by activating edit mode with the 3rd rightmost button you can move nodes around, which allows you to fix a bad layout (rather seldom needed)
- the context menu opened by pressing the right mouse button over a graph element allows you to explore the adjacent nodes in non-edit-mode, or delete the element in edit mode
- you can search with `Ctrl-f` or `/` for the persistent name or an attribute value (or by clicking into the left search field), the matching elements get highlighted

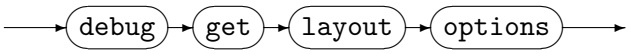
18.3 Debugging Related Commands



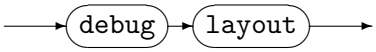
Enables and disables the debug mode. The debug mode shows the current working graph in a yComp window. All changes to the working graph are tracked by yComp immediately.



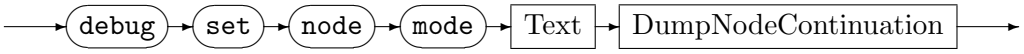
Sets the default graph layout algorithm to *Text*. If *Text* is omitted, a list of the available layout algorithms is displayed. The following layout algorithms are supported: `Random`, `Hierarchic`, `Organic`, `Orthogonal`, `Circular`, `Tree`, `Diagonal`, `Incremental Hierarchic`, `Compilergraph`. For technical graphs `Hierarchic` works normally best; `Compilergraph` is a version of `Hierarchic` cutting some edges, it may be of interest if `Hierarchic` contains too many crossing edges. `Organic` is the other general purpose layout available, the other layouts are rather special, but this should not prevent you from using them if they fit to your task ;). The `option` version allows to specify layout options by name value pairs. The available layout options can be listed by the following command.

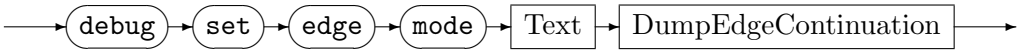


Prints a list of the available layout options of the layout algorithm.



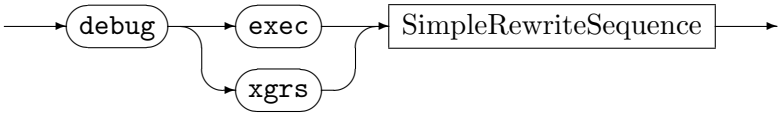
Forces re-layout of the graph shown in yComp (same as pressing the play button within yComp).





Configures the display of the visual debug states for the nodes/edges. The following modes are supported: `matched`, `created`, `deleted`, `retyped`. Change this if you e.g. want the matched elements to be marked more visibly, or added/deleted elements to be colored green/red.

GraphRewriteSequence



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

18.4 Using the Debugger

The debugging process follows of a series of debug situations, which result from a user selection of the underlying execution situations accoring to interest. During each debugging step the debugger – which is a part of the GrShell – prints the debugged sequence with the currently focused/active rule highlighted yellow. What will be shown from executing this rule depends on the commands chosen by the user; and on the fact whether the focused rule matches or not. An active rule which is already known to match is highlighted green. The rules which matched during sequence execution are shown on dark green background, the rules which failed during sequence execution are shown on dark red background; at the begin of a new loop iteration the highlighting state of the contained rules is reset. During execution yComp[2] will display the changes to the graph from every single step. Besides deciding on what is shown from the execution of the current rule, the user determines with the debug commands where to continue the execution (the rule focused next; but again this depends on success/failure of the currently active rule). The debug commands are given in Table 18.1. A run is shown in the following example 88.

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

In addition to the commands for actively stepping or skipping through the sequence execution, there are breakpoints and choicepoints available (toggled with the `b` and `c` commands) which are only processed when they are reached, but on the other hand are also processed if a user command would skip them. The break points halt execution, focus the reached sequence, and cause the debugger to wait for further commands (e.g. `d` to inspect the rule execution en detail versus `s` for just applying it). The choice points halt execution, focus the reached sequence in magenta, and ask for some user input; after the input was received, execution continues according to the command previously issued.

Both break points and choice points are denoted by the `%` modifier. The `%` modifier works as a break point if it is given before: a rule, an all bracketed rule, a variable predicate, or the constants `true`/`false`. The `%` modifier works as a choice point if it is appended to the `$` randomize modifier switching a random decision into a user decision. This holds for the binary operators, the random match selector of all bracketed rules, the random-all-of operators and the one-of-set braces. The idea behind this is: you need some randomization for simulation purposes — then use the randomize modifier `$`. You want to force a certain decision overriding the random decision to try out another execution path while debugging the simulation flow — then modify the randomize modifier with the user (choice) modifier `%`.

The initial breakpoint and choicepoint assignment is given with the `%` characters in the sequences after the `debug exec` commands in the `.grs` file. The breakpoint and choicepoint commands of the debugger allow to toggle them at runtime, overriding the initial assignment (notationally yielding a sequence with added or removed `%` characters). The user input commands `$%(type)` define choice points which can't be toggled off.

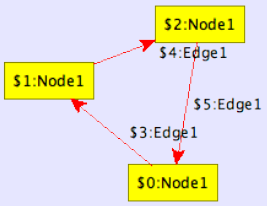
Further commands allow to print the variables at a given situation, the sequence call stack, or a full state dump of the call stack and the variables. Or allow to dump the current graph, or highlight elements in the graph, defined by being contained in a (possibly container valued) variable, or being visited according to a visited flag.

EXAMPLE (88)

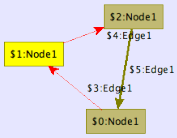
We demonstrate the debug commands with a slightly adjusted script for the Koch snowflake from GrGen.NET's examples (see also Section 19.1). The graph rewriting sequence is

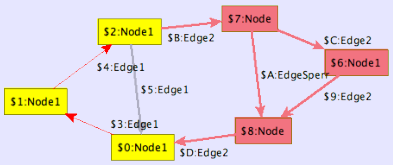
```
1 debug exec (makeFlake1* & (beautify & doNothing)* & makeFlake2* & beautify*)[1]
```

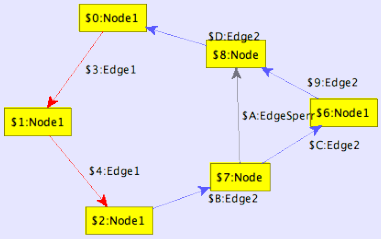
yComp will be opened with an initial graph (resulting from `grs init`):



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
u	beautify
o	makeFlake2
r	—

`s`(tep)	Execute the current rewrite rule (match, and rewrite in case it matched; the resulting graph is shown).
`d`(etailed step)	Execute the current rewrite rule in a three-step procedure: matching - highlighting the found match, rewriting - highlighting the changing elements, and application - doing the rewrite showing the resulting graph. In addition, afterwards the execution of subrules from embedded sequences (`exec`) is shown step by step.
(step) `o`(ut)	Continue execution until the end of the current loop. If the execution is not in a loop at this moment, but in a sequence called, the called sequence will be executed until its end. If neither is the case, the complete sequence will be executed.
(step) `u`(p)	Ascend one level up within the Kantorowitsch tree of the current rewrite sequence (i.e. rule; see Example 88; at the moment the command is pretty useless because only the serialized form is displayed).
`r`(un)	Continue execution (until the end or a breakpoint).
`a`(bort)	Cancel the execution immediately.
`n`(ext)	Go to the next rewrite rule which matches, make it current.
(toggle) `b`(reakpoint)	Toggle a breakpoint at one of the breakpointable locations.
(toggle) `c`(hoicepoint)	Toggle a choicepoint at one of the choicepointable locations.
`v`(ariables)	Prints the global variables and the local variables of the sequence currently executed, which is the topmost sequence of the sequence call stack. Plus the allocated visited flags. To be more precise regarding local variables: all variables which were defined (and have not fallen out of scope again) up to the sequence position focussed.
`t`(race)	Prints the stack trace of the current sequence call stack; the stack trace includes the body of each sequence called at its execution state.
`f`(ull dump)	Prints the stack trace including the local variables of each stack frame plus the global variables.
(dum)`p` (graph)	Dumps the current graph as a `.vcg` file and shows it in yComp. This can be used as a workaround to check the real state in case transaction/backtracking rollback is used on a graph with node nesting, which may lead to a buggy display.
`h`(ighlight)	Highlights the elements in the graph which are marked with the visited flag given, or are contained in the variable given (which might be a simple scalar variable containing a graph element, or a set/map/array valued container of graph elements). Multiple variables of visited flags may be given separated by commas.

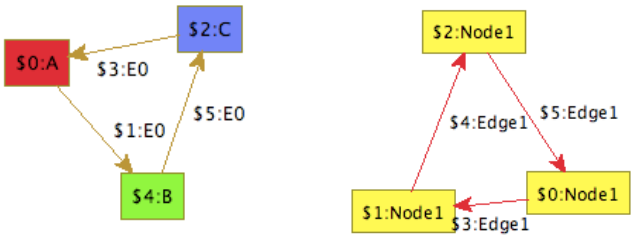
Table 18.1: GrShell debug commands

CHAPTER 19

EXAMPLES

19.1 Fractals

The GrGen.NET package ships with samples for fractal generation. We will construct the Sierpinski triangle and the Koch snowflake. They are created by consecutive rule applications starting with the initial host graphs



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

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

or

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

respectively. If you are on a Unix-like system you have to adjust the path separators of the GrShell scripts. Just edit the first three lines of `/test/Sierpinski.grs` and `/test/Snowflake.grs`. And as we have the file `Sierpinski.grs` already opened, we can increase the number of iterations to get even more beautiful graphs[1]. Just follow the comments. Be careful when increasing the number of iterations of Koch's snowflake—yComp's layout algorithm might need some time and attempts to layout it nicely. We execute the Sierpinski script by

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

or

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

respectively. Because both of the scripts are using the debug mode, we complete execution by typing `r`(un). See Section 17.2.8 for further information. The resulting graphs should look like Figures 19.1 and 19.2.

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

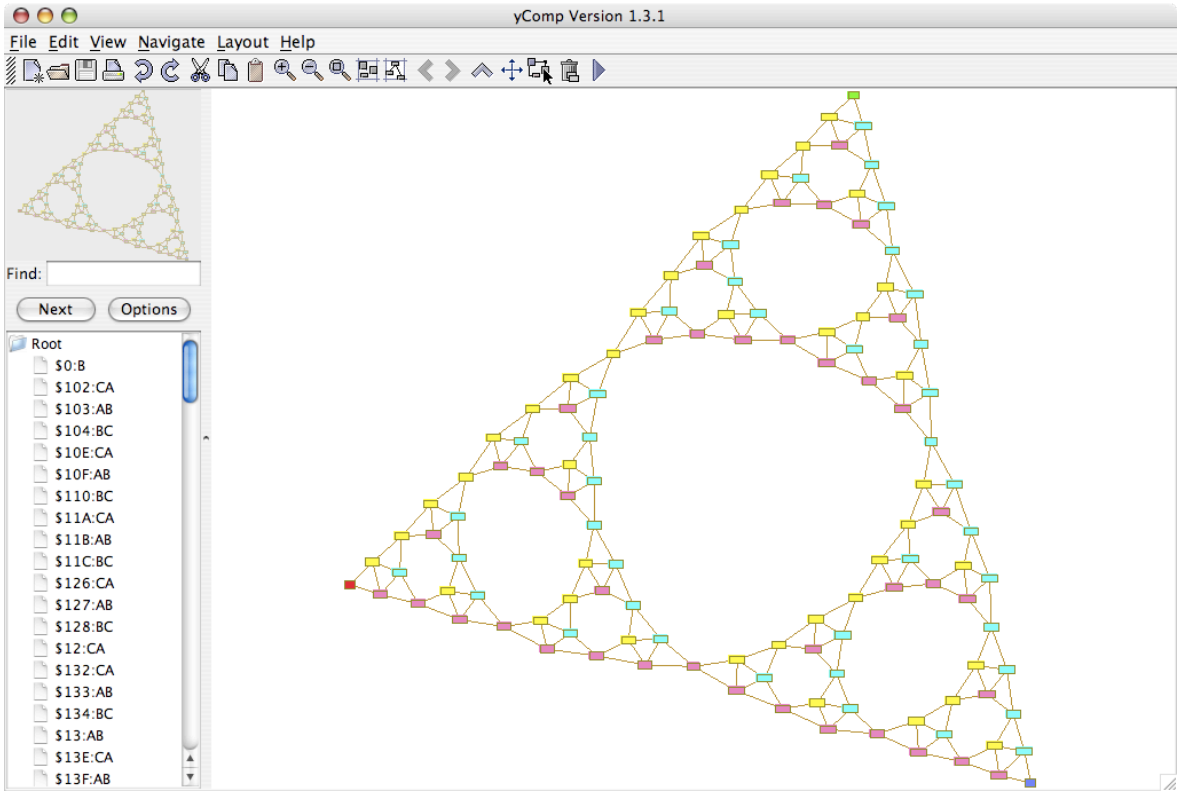


Figure 19.1: Sierpinski triangle

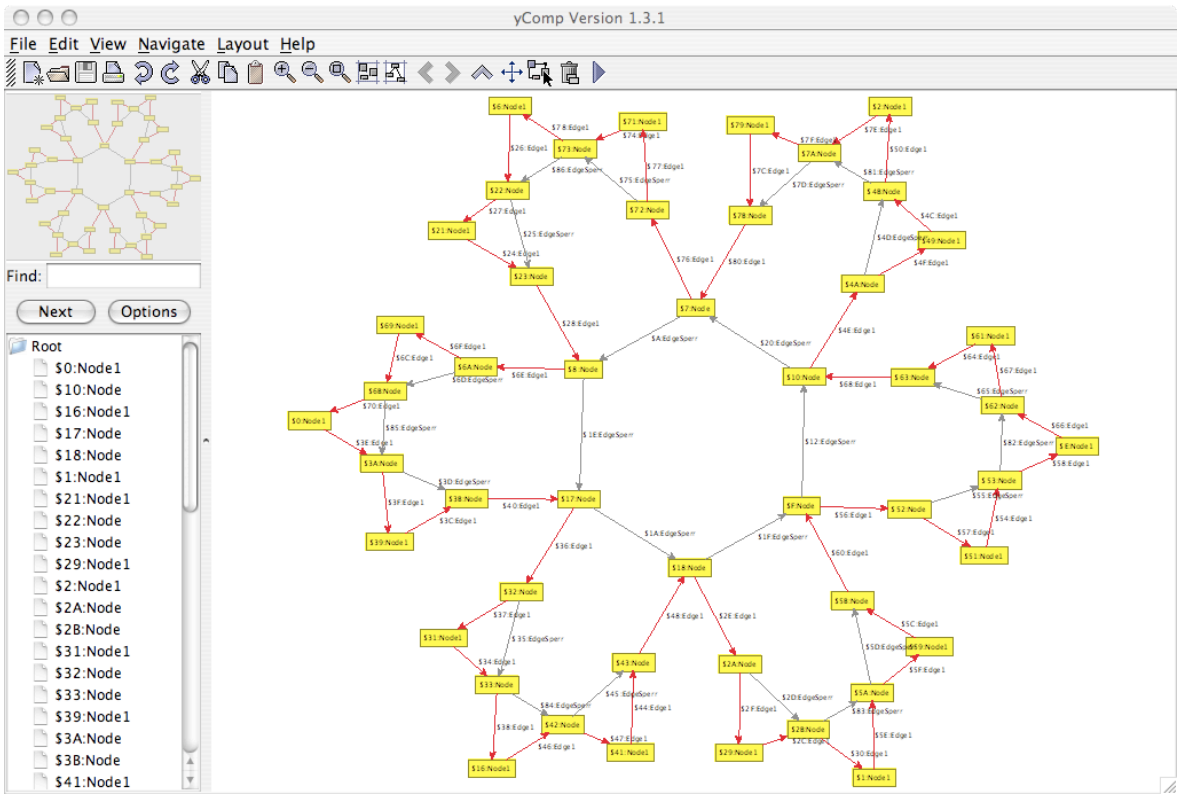


Figure 19.2: Koch snowflake

19.2 Busy Beaver

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

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

19.2.1 Graph Model

The tape will be a chain of `TapePosition` nodes connected by right edges. A cell value is modeled by a reflexive `value` edge, attached to a `TapePosition` node. The leftmost and the rightmost cells (`TapePosition`) do not have an incoming and outgoing edge respectively. Therefore we have the node constraint $[0:1]$.

```
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;
```

Furthermore we need states and transitions. The machine's current configuration is modeled with a `RWHead` edge pointing to a `TapePosition` node. `State` nodes are connected with `WriteValue` nodes via `value` edges, a `moveLeft`/`moveRight`/`dontMove` edge leads from a `WriteValue` node to the next state (cf. the picture on page 211).

```
node class State;

edge 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;
```

19.2.2 Rule Set

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

```
using TuringModel;
```

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

1. Read the value of the current tape cell and select an outgoing edge of the current state.
2. Write a new value into the current cell, according to the sub type of the `WriteValue` node.
3. Move the read-write-head along the tape and select a new state as current state.

As you can see a transition of the Turing machine is split into two graph rewrite steps: Writing the new value onto the tape and performing the state transition. We need eleven rules: Three rules for each step (for "zero", "one", and "empty") and two rules for extending the tape to the left and the right, respectively.

```
rule readZeroRule {
   s:State -h:RWHead-> tp:TapePosition -:zero-> tp;
   s -:zero-> wv:WriteValue;
   modify {
      delete(h);
      wv -:RWHead-> tp;
   }
}
```

We take the current state `s` and the current cell `tp` which is implicitly given by the unique `RWHead` edge and check whether the cell value is zero. Furthermore we check if the state has a transition for zero. The replacement part deletes the `RWHead` edge between `s` and `tp` and adds it between `wv` and `tp`. The remaining rules are analogous:

```
rule readOneRule {
   s:State -h:RWHead-> tp:TapePosition -:one-> tp;
   s -:one-> wv:WriteValue;
   modify {
      delete(h);
      wv -:RWHead-> tp;
   }
}

rule readEmptyRule {
   s:State -h:RWHead-> tp:TapePosition -:empty-> tp;
   s -:empty-> wv:WriteValue;
   modify {
      delete(h);
      wv -:RWHead-> tp;
   }
}

rule writeZeroRule {
   wv:WriteZero -rw:RWHead-> tp:TapePosition -:value-> tp;
   replace {
      wv -rw-> tp -:zero-> tp;
   }
}

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

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

rule moveLeftRule {
   wv:WriteValue -:moveLeft-> s:State;
```

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

Have a look at the negative conditions within the `ensureMove...` rules. They ensure that the current cell is indeed at the end of the tape: An edge to a right/left neighboring cell must not exist. Now don't forget to compile your model and the rule set with `GrGen.exe` (see Section 19.1).

19.2.3 Rule Execution with GrShell

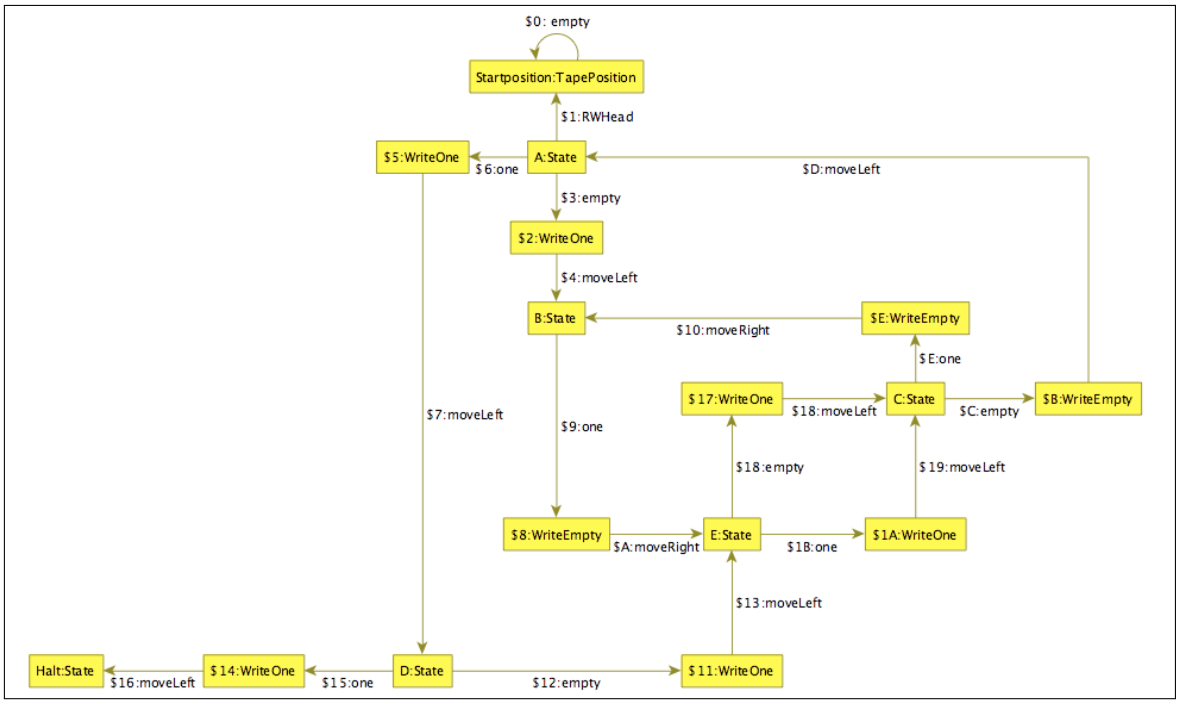
Finally we construct the busy beaver and let it work with GrShell. The following script starts with building the Turing machine that is modeling the six states with their transitions in our Turing machine model:

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

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

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

# 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 and input symbol 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 -:moveRight-> sC

new sE_1: WriteOne
```

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

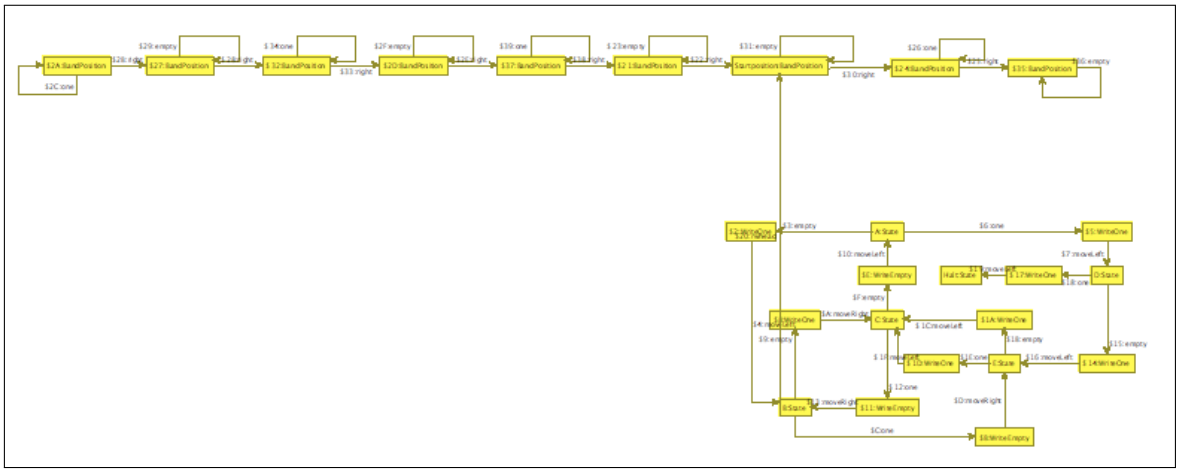
Our busy beaver looks like this:



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

```
64 exec ((readOneRule | readEmptyRule) & (writeOneRule | writeEmptyRule) &
       (ensureMoveLeftValidRule | ensureMoveRightValidRule) & (moveLeftRule |
       moveRightRule))[32]
```

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



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

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

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

Let the beaver run:

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

CHAPTER 20

APPLICATION PROGRAMMING INTERFACE

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

- the interface to the graph and model
- the interface to the rules and matches
- the interface of the graph processing environment
- the porter module for importing and exporting of graphs and miscellaneous stuff
- implementing external class and function declarations
- implementing external match filter and external sequence declarations
- events fired when the graph is changed
- events fired during action execution

From the input file `Foo.grg` `grgen.exe` generates the output files `FooModel.cs` for the model and `FooActions.cs` for the actions,

- defining the exact interface,
- implementing the exact interface with generated code and code from the lgsp backend, i.e. entities from `de.unika.ipd.grGen.lgsp` available from lgspBackend.dll,
- and implementing the generic interface from `de.unika.ipd.grGen.libGr` using the entities mentioned in both points above.

This generative approach bears a great deal of the responsibility for the high execution speed of GrGen.NET, but it comes at the price of flexibility: you can't extend the rule set at runtime with new rules. What you can do at runtime on the other hand is to generate a new rule set file, apply the compiler, and dynamically link the resulting assemblies/dlls.

When working with the API, you could reference the generated binary dlls in your project, in the same way as you have to reference the `libGr.dll` and `lgspBackend.dll`. For easier debugging though, especially when you are integrating the generated code with own extensions (as described in chapter 21), it is recommended to directly include the source code generated (which is thrown away normally after it was compiled into the assemblies lgsp-FooModel.dll and lgsp-FooActions.dll). For this, use the `-keep` option when you call `grgen.exe`, and include the model and the actions file (excluding the intermediate actions file) directly as C# source code files.

The matcher code generated contains the initial, static search plans. When you analyze the graph at runtime and generate new matchers, see section 17.3.2 for more on this, you can request the dumping of the source code of the improved matchers. The custom commands

are available at API level via the `Custom` operation of the `IGraph` interface for the graph commands and the `Custom` operation of the `BaseActions` class for the actions commands, just handing in the same parameters as otherwise specified on the command line. If the intended workflow of "i) loading a typical graph or doing a warm-up run creating a typical graph, ii) analyzing that graph, iii) compiling new matchers which are better suited to the graph" is not easily achievable, or you want to start with the optimized matchers straight from the beginning, you may copy and paste the dumped dynamic matchers of an example run to the existing static code. But this is only a last resort, as the price is that your manual editing is overwritten again with the static search plans at the next time you call `grgen`.

20.1 Interface to the Host Graph

The generated file `FooModel.cs` opens the namespace `de.unika.ipd.grGen.Model_Foo` containing all the generated entities. It contains for every node or edge class `Bar` an interface `IBar`, which offers C# properties giving access to the attributes, and is inheriting in the same way as specified in the model file. This builds the exact interface of the model, it is implemented by a sealed class `Bar` with generated code and with code from the lgsp backend. Furtheron the namespace contains a model class `FooGraphModel` implementing the interface `de.unika.ipd.grGen.libGr.IGraphModel`, which supports iteration over the entities defined in the model using further, generic(i.e. inexact) interfaces from libGr. Finally, the namespace contains a class `FooGraph` which defines an `LGSPGraph` of a model equivalent to `FooGraphModel`; it contains convenience functions to easily create nodes and edges of exact type in the graph. In addition, a class `FooNamedGraph` is available, which defines an `LGSPNamedGraph` of a model equivalent to `FooGraphModel`; the named graph offers persistent names 83 for all its graph elements, otherwise it is identical to an `LGSPGraph`. The naming requires about the same memory as an unnamed graph, but under normal circumstances the named graph is the recommended one to use (and is the one which will be used if employed by the shell).

NOTE (49)

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

20.2 Interface to the Rules

The generated file `FooActions.cs` opens the `namespace de.unika.ipd.grGen.Action_Foo` containing all the generated entities. It contains for every rule or test `bar`

- a class `Rule_bar` inheriting from `de.unika.ipd.grGen.lgsp.LGSPRulePattern`, which contains the exact match interface `IMatch_bar` which defines how a match of the rule looks like, extending the generic rule-unspecific `IMatch` interface. Have a look at section 21.3 for an extended introduction to the matches interfaces. Furtheron there are (but meant only for internal use): a match class `Match_bar` implementing the exact and inexact interface, a description of the pattern to match, and the modify methods doing the rewriting.

- an exact action interface `IAction_bar` which contains the methods:

- `Match`, to match the pattern in the host graph, with in-parameters corresponding to the in-parameters of the rule (name and type), returning matches of the exact type `Rule_bar.IMatch_bar`.

- `Modify`, to modify a given match according to the rewrite specification, with out-parameters corresponding to the out-parameters of the rule.

- `Apply`, to match and modify the found match, with in-parameters corresponding to the in-parameters of the rule, and with ref-parameters corresponding to the out-parameters of the rule.

Furtheron there is (but meant only for internal use) the class `Action_bar` implementing the exact action interface as well as the generic `IAction` interface from libGr; it contains the generated matcher code searching for the patterns.

Moreover the namespace contains an action class `FooActions` implementing the abstract class `de.unika.ipd.grGen.libGr.BaseActions` (in fact `de.unika.ipd.grGen.lgsp.LGSPActions`), which supports iteration over the entities defined in the actions using further, generic(i.e. inexact) interfaces from libGr. Additionally, at runtime it contains the instances of the actions singletons, as member `bar` of the exact type `IAction_bar`.

NOTE (50)

If you want to use the type-safe interface, your entry point is the member `bar` of type `IAction_bar` from `FooActions` (or `Action_bar.Instance`). Actions are used with named parameters of exact types. If you want to use the generic interface, your entry point is the method `GetAction("bar")` of the interface `BaseActions` implemented by `FooActions` returning an `IAction`. Actions are used with `object`-arrays for parameter passing.

NOTE (51)

The old generic interface of string names and entities of node-,edge-, and object-type is implemented with the new interface of exactly typed, named entities. Thus you will receive runtime exceptions when doing operations which are not type-safe with the generic interface, in contrast to GrGen.NET < v2.5. If you need the flexibility of the old input parameters semantics of silently failing rule application on a wrong type, you must declare it explicitly with the syntax `r(x:ExactType<InexactType>)`; then the rule parameter in the exact interface will be of type `InexactType`.

EXAMPLE (89)

Normally you want to use the type-safe interface of the generated code as it is much more convenient. Only if your application must get along with models and actions unknown before it is compiled you have to fall back to the generic interface. An extensive example showing how to cope with the latter is shipped with GrGen.NET in form of the GrShell. Here we'll show a short example on how to use GrGen.NET with the type-safe API; further examples are given in the examples-api folder of the GrGen.NET-distribution.
We'll start with including the namespaces of the libGr and the lgsp backend shipped with GrGen.NET plus the namespaces of our actions and models, generated from `Foo.grg`.

```
using de.unika.ipd.grGen.libGr;
using de.unika.ipd.grGen.lgsp;
using de.unika.ipd.grGen.Action_Foo;
using de.unika.ipd.grGen.Model_Foo;
```

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

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

We could create a named graph instead offering persistent names for its graph elements:

```
FooNamedGraph graph = new FooNamedGraph();
```

EXAMPLE (90)

This is an example doing mostly the same as the previous example 89, in a slightly more complicated way allowing for more control. Here we create the model separate from the graph, then the graph with a model not bound at generation time. We create the actions to apply on the graph, and a single node of type `Bar` in the graph, which we assign again to a variable `b`. Then we get the action from the actions and save it to an action variable `bar`; afterwards we use the action for finding all available matches of `bar` with input `b` – which is different from the first version – and remember the found matches in the matches variable with its exact type. Finally we take the first match from the matches and execute the rewrite with it. We could have inspected the nodes and edges of the match or their attributes before (using element names prefixed with `node_/edge_` or attribute names to get exactly typed entities).

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

We could create a named graph instead offering persistent names for its graph elements:

```
LGSPGraph graph = new LGSPNamedGraph(model);
```

20.3 Interface of the Graph Processing Environment

The interface `IGraphProcessingEnvironment` implemented by the `LGSPGraphProcessingEnvironment` class offers all the additional functionality of GrGen.NET exceeding what is offered by the graph and the actions. It is constructed as `LGSPGraphProcessingEnvironment` given the graph and the actions. It offers execution of the sequences and variable handling, combining actions into transformations (the former regarding control flow, the latter regarding data flow).

EXAMPLE (91)

For all but the simplest transformations you'll end up constructing a graph processing environment from the graph and the actions constructed until now, executing a graph rewrite sequence on the graph processing environment:

```
LGSPGraphProcessingEnvironment procEnv =
    new LGSPGraphProcessingEnvironment(graph, actions);
procEnv.ApplyGraphRewriteSequence("<(::x)=foo && (::y)=bar(::x) | bla(::y)>");
```

In addition to sequences and variables handling, the graph processing environment offers driver or helper objects for transaction management, deferred sequence execution, graph change recording, and emitting. The most important of these is the transaction manager which is utilized when GrGen.NET is used for crawling through a search space or for enumerating a state space, see section 15.2. The operations mentioned there are implemented by calling the function given in example 92.

EXAMPLE (92)

```
LGSPGraphProcessingEnvironment procEnv = ...;
ITransactionManager tm = procEnv.TransactionManager;
public interface ITransactionManager
{
    int StartTransaction();
    void Pause();
    void Resume();
    void Commit(int transactionID);
    void Rollback(int transactionID);
}
```

The `StartTransaction` starts a transaction and returns its id; it may be called multiple times returning different ids for the then nested transactions (i.e. a failing outer one rolls back the changes of an inner transaction which succeeded). Changes to the graph are recorded thereafter into an undo log, unless a `Pause` was called not yet followed by a `Resume`. When the changes of interest were carried out the transaction identified by its id is either `Commit`ed, which causes the changes recorded since the corresponding `StartTransaction` to stay in the graph, or rolled back by calling `Rollback`, in that case all the changes recorded since the corresponding `StartTransaction` are undone.

20.4 Import/Export and Miscellaneous Stuff

GrGen natively supports the following formats:

GRS/GRSI
: Reduced GrShell script files (graph only, model from `.gm`; a very limited version of the normal `.grs`. The recommended standard format.)

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

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

GRG
: Writes a GrGen rule file containing one rule with an empty pattern and a large rewrite part. Export only [1], not for normal use.

While both GRS and GXL importers expect one file (the GXL importer allows to specify a model override, see GrShell import, Note 43), the EMF/ECORE importer expects first one or more `.ecore` files and following optionally a `.xmi` files and/or a `.grg` file (cf. Note 44). To use additional custom graph models you should supply an own `.grg` file which may be based on the automatically generated `.grg` file, if none was supplied (see the Program-Comprehension example in `examples/ProgramComprehension-GraBaTs09`).

To import a graph model and/or a graph instance you can use `Porter.Import()` from the libGr API (the GrShell command `import` is mapped to it) The file format is determined by the file extensions. To export a graph instance you can use `Porter.Export()` from the libGr API

[1] Original German Pisswasser, for export only :)

(the GrShell command `export` is mapped to it). For an example of how to use the importer/-exporter on API level see `examples-api/JavaProgramGraphsExample/JavaProgramGraphsExample.cs`

The GRS(I) importer returns an `INamedGraph`; if you don't need the persistent names, get rid of them by casting to the `LGSPNamedGraph` implementing the interface, (copy-)constructing a `LGSPGraph` from it, and forgetting any references to the named graph. Please be aware that naming is rather expensive: A `LGSPNamedGraph` supplying the name to element and element to name mappings normally uses up about twice the amount of memory of the `LGSPGraph` defining the graph alone (but is worth is most often).

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

- How to use the graph rewrite sequences offered by the libGr on API level is shown in `examples-api/BusyBeaverExample/BusyBeaverExample.cs`.
  But normally you want to use your favourite .NET programming language for control together with the type-safe interface when working on API level.

- How to use the old and new interface for accessing a match on API level is shown in `examples-api/ProgramGraphsExample/ProgramGraphsExample.cs`.

- How to use the visited flags on API level is shown in `examples-api/VisitedExample/VisitedExample.cs`.

- How to analyze the graph and generate (hopefully) better performing matchers based on this information is shown in `examples-api/BusyBeaverExample/BusyBeaverExample.cs`.

- How to compile a `.grg`-specification at runtime and dump a graph for visualization in `.vcg` format on the API level is shown in `examples-api/HelloMutex/HelloMutex.cs`.

- How to access the annotations at API level is shown in `examples-api/MutexDirectExample/MutexDirectExample.cs`.

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

NOTE (52)

While C# allows input arguments values to be of a subtype of the declared interface parameter type (OO), it requires that the argument variables for the `out` parameters are of exactly the type declared (non-OO). Although a variable of a supertype would be fully sufficient – the variable is only assigned. So for `node class Bla extends Bar;` and action `bar(Bar x) : (Bla)` from the rules file rules `Foo.grg` we can't use a desired target variable of type `Bar` as `out`-argument, but are forced to introduce a temporary variable of type `Bla` and assign this variable to the desired target variable after the call.

```
using de.unika.ipd.grGen.libGr;
using de.unika.ipd.grGen.lgsp;
using de.unika.ipd.grGen.Action_Foo;
using de.unika.ipd.grGen.Model_Foo;
FooGraph graph = new FooGraph();
FooActions actions = new FooActions(graph);
Bar b = graph.CreateNodeBar();
IMatchesExact<Rule_bar.IMatch_bar> matches = actions.bar.Match(graph, 1, b);
//actions.bar.Modify(graph, matches.First, out b); // wont work, needed:
Bla bla = null;
actions.bar.Modify(graph, matches.First, out bla);
b = bla;
```

20.5 External Class and Function Implementation

For a model file `Foo` which contains external functions (cf. 21.2) and/or classes (cf. 21.1), GrGen.NET generates a file `FooModelExternalFunctions.cs` located besides the model and rule files, which contains

- within the model namespace public partial classes named as given in the external class declaration, inherting from each other as stated in the external class declarations.
- within the `de.unika.ipd.grGen.expression` namespace a public partial class named `ExternalFunctions` with a body of comments giving the expected function prototypes.

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

- fleshing out the partial classes skeletons with attributes containing data of interest and maybe helper methods
- fleshing out the `ExternalFunctions` partial class skeleton with the functions you declared in the external function declarations, obeying the function signatures as specified; here you can access the now know attributes or methods of the external classes, or do complicated custom computations with the values you receive from a function call.

Don't forget that the source code file `FooModelExternalFunctionsImpl.cs` is an integral part of your GrGen.NET graph transformation project, in contrast to the other C# files generated (and overwritten) for you. In `examples/ExternalAttributeEvaluationExample` and `examples-api/ExternalAttributeEvaluationExample` you find a fabricated example showing how to use the external classes and functions.

When you use third-party assemblies in your source code you must inform GrGen.NET about them so references to them are included into the assembly generated; this can be done

with the -r parameter when calling grgen.exe directly (cf. 1.7.1) or with the new command configurations available in GrShell (cf. 17.2.2). Using the keepdebug configuration of the new command is recommended as it allows for easier debugging.

20.6 External Filter and Sequence Implementation

For an actions file Bar which contains match filter declarations (cf. 21.3) and/or external sequence declarations (21.4), GrGen.NET generates a file BarActionsExternalFunctions.cs located besides the model and rule files, which contains within the action namespace

- a public partial class named MatchFilters with a body of comments giving the expected function prototypes, and for the auto filters even the implementation.

- public partial classes, named Sequence_foo for a sequence foo, with a body containing a comment specifying the expected function prototype of the sequence application function.

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

- fleshing out the MatchFilters partial class skeleton with the match filter functions you declared, obeying the function signatures as specified; you might want to convert the received matches object to an IList in case you want to reorder the list and reinject it into the matches object afterwards.

- fleshing out the partial classes skeletons of the external sequence with the ApplyXGRS_foo methods needed.

Don't forget that the source code file BarActionsExternalFunctionsImpl.cs is an integral part of your GrGen.NET graph transformation project, in contrast to the other C# files generated (and overwritten) for you. In examples/ExternalFiltersAndSequencesExample and examples-api/ExternalFiltersAndSequencesExample you find a fabricated example showing how to use the external classes and functions.

When you use third-party assemblies in your source code you must inform GrGen.NET about them so references to them are included into the assembly generated; this can be done with the -r parameter when calling grgen.exe directly (cf. 1.7.1) or with the new command configurations available in GrShell (cf. 17.2.2). Using the keepdebug configuration of the new command is recommended as it allows for easier debugging.

NOTE (53)

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

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

In C#, this might look like:

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

The external match filters are hooking in between the `Match` and `Modify` functions, they allow you to do this kind of inspection without being forced to resort to fully external control. The most interesting filter can be automatically generated for you, the `auto` filter for filtering symmetric matches of automorphic patterns, see 21.3 for more on this.

20.7 Graph Events

Before or after the host graph is changed, events are fired, notifying listeners about the changes. The GrShell debugger, the transaction handler, and the graph change recorder implement their functionality by listening and reacting to these events. A programmer may add own event handlers to insert custom-made, event-based functionality; or may even implement an event-driven rule execution engine on top of it. The events are fired automatically by the `LGSPGraph` implementing the `IGraph`, or by the rules which get applied. If you operate on API level or with e.g. external sequences, it's your responsibility to fire the events for attribute changes before changing an attribute. Otherwise the changes won't be visible in the debugger, they won't be rolled back at the end of a transactions or during backtracking, and they won't be recorded in case of change recording. If you listen to or fire the rule application events (cf. 20.8), you may be interested in the added names event, too, which tells about the names of the elements which will get added immediately thereafter (this is used in the debugger to display the names of the elements as defined in the rule modify part).

The events available which are fired automatically are:

```
// Fired after a node has been added
event NodeAddedHandler OnNodeAdded;

// Fired after an edge has been added
event EdgeAddedHandler OnEdgeAdded;

// Fired before a node is deleted
event RemovingNodeHandler OnRemovingNode;

// Fired before an edge is deleted
```

```
event RemovingEdgeHandler OnRemovingEdge;

// Fired before all edges of a node are deleted
event RemovingEdgesHandler OnRemovingEdges;

// Fired before the whole graph is cleared
event ClearingGraphHandler OnClearingGraph;

// Fired before the type of a node is changed.
event RetypingNodeHandler OnRetypingNode;

// Fired before the type of an edge is changed.
event RetypingEdgeHandler OnRetypingEdge;

// Fired before an edge is redirected (causing removal then adding again).
event RedirectingEdgeHandler OnRedirectingEdge;
```

The events available which are fired automatically by code generated by GrGen for the actions, but which need to be fired by you in case you change the graph and want e.g. an accurate debugger display.

```
// Fired before an attribute of a node is changed.
event ChangingNodeAttributeHandler OnChangingNodeAttribute;

// Fired before an attribute of an edge is changed.
event ChangingEdgeAttributeHandler OnChangingEdgeAttribute;

// Fires an OnChangingNodeAttribute event.
// To be called before changing an attribute of a node,
// with exact information about the change to occur.
void ChangingNodeAttribute(INode node, AttributeType attrType,
    AttributeChangeType changeType, Object newValue, Object keyValue);

// Fires an OnChangingEdgeAttribute event.
// To be called before changing an attribute of a node,
// with exact information about the change to occur.
void ChangingEdgeAttribute(IEdge edge, AttributeType attrType,
    AttributeChangeType changeType, Object newValue, Object keyValue);

// Fired before each rewrite step (also rewrite steps of subpatterns) to indicate
// the names of the nodes added in this rewrite step in order of addition.
event SettingAddedElementNamesHandler OnSettingAddedNodeNames;

// Fired before each rewrite step (also rewrite steps of subpatterns) to indicate
// the names of the edges added in this rewrite step in order of addition.
event SettingAddedElementNamesHandler OnSettingAddedEdgeNames;
```

20.8 Action Events

When actions are executed, events are fired, notifying listeners about the changes. The GrShell debugger implements its functionality by listening and reacting to these events. A programmer may add own event handlers to insert custom-made, event-based functionality. The events are fired automatically by the code generated by GrGen for the rules or sequences.

If you operate on API level or with e.g. external sequences, it's your responsibility to fire the events in case you want to simulate rules or sequences.

The rule based events declared by the `IActionExecutionEnvironment` are:

```
// Fired after all requested matches of a rule have been matched.
event AfterMatchHandler OnMatched;

// Fired before the rewrite step of a rule, when at least one match has been found.
event BeforeFinishHandler OnFinishing;

// Fired before the next match is rewritten. It is not fired before rewriting the
    first match.
event RewriteNextMatchHandler OnRewritingNextMatch;

// Fired after the rewrite step of a rule.
// Note, that the given matches object may contain invalid entries,
// as parts of the match may have been deleted!
event AfterFinishHandler OnFinished;
```

The sequence based events declared by the `IGraphProcessingEnvironment` extending the `IActionExecutionEnvironment` are:

```
// Fired when a sequence is entered.
event EnterSequenceHandler OnEntereringSequence;

// Fired when a sequence is left.
event ExitSequenceHandler OnExitingSequence;

// Fired when a sequence iteration is ended.
event EndOfIterationHandler OnEndOfIteration;
```

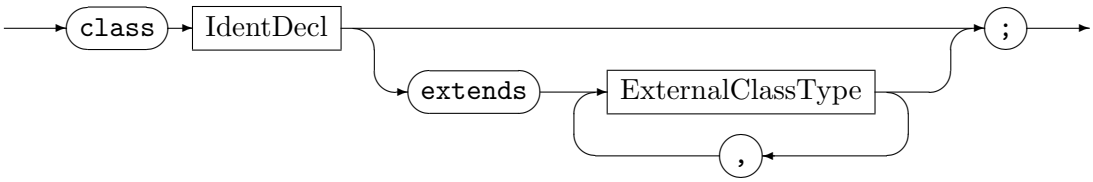
CHAPTER 21

EXTENSIONS

This chapter lists the ways you can customize GrGen.NET without GrGen.NET-language constructs and shows how to interact with the external world outside of GrGen.NET. The primary means available are: external attribute types, external functions, match filters, and external sequences; the secondary helpers available are annotations, command line parameters, and external shell commands.

21.1 External Attribute Types

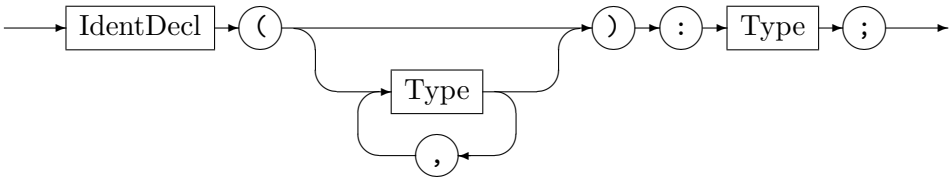
ExternalClassDeclaration



Registers a new attribute type with GrGen.NET. You may declare the base types of the type, but not give attributes. The attribute type must be implemented externally, see 20.5; for GrGen.NET the type is opaque, only external functions can do computations with it. You may extend GrGen.NET with external attribute types if the built-in attribute types (cf. 5.1) are insufficient for your needs.

21.2 External Function Types

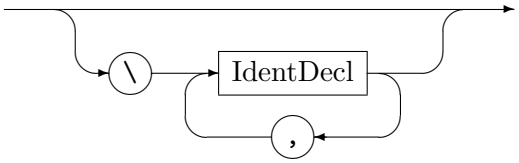
ExternalFunctionDeclaration



Registers an external function with GrGen.NET to be used in attribue computation. An external function declaration specifies the expected input types and the output type. The function must be implemented externally, see 20.5. An external function call (cf. 5.8) may receive and return values of the built-in (attribute) types as well as of the external attribute types; the real arguments on the call sites are type-checked against the declared signature following the subtyping hierarchy of the built-in as well as of the external attribute types. You may extend GrGen.NET with external functions if the built-in attribute computation capabilities (cf. 5.2) are insufficient for your needs.

21.3 External Match Filters

MatchFilter



Registers one or multiple external match filters with GrGen.NET , for the rule the match filters are appended at, to be used from selected applications of that rule (cf. *FilterIdent* in 8.1). The match filter function must be implemented externally, see 20.6. You may extend GrGen.NET with match filters if you need to inspect the matches found for a rule in order to decide which to apply (see note 53) or if you just need a post-match hook which informs you about the found matches.

In addition to the external match filters, you may request from GrGen to generate a match filter for symmetric matches. In order to do so use the special filter name `auto`; this filter name may be used on multiple rules in contrast to the other names which must be unique. The generated code implementing the `auto` filter is to be applied like your own external filters, after a backslash following the rule call. The filter is removing matches due to an automorphic pattern, matches which are covering the same spot in the host graph with a permutation of the nodes, the edges, or subpatterns of the same type at the same level. Other nested pattern constructs which are structurally identical are not recognized to be identical when they appear as commuted subparts; but they are so when they are factored into a subpattern which is instantiated multiple times. It is highly recommended to use this symmetry reduction technique when building state spaces including isomorphic state collapsing, as puring the matches which lead to isomorphic states early saves expensive graph comparisons – and often it gives the semantics you are interested in anyway.

Match object

For a rule or subpattern `r` GrGen generates a match interface `IMatch_r` extending the generic `IMatch` interface from the `libGr`. The basic constituents, i.e. nodes, edges and variables are mapped directly to members of their name and type, containing the graph element matched or the value computed. For alternatives nested inside the pattern a common base interface `IMatch_r_altName` is generated, plus for each alternative case an interface `IMatch_r_altName_altCaseName`. When you walk the matches tree using the type exact interface (which is recommended), you must type switch on the match object in the alternative variable, to determine the case which was finally matched, and cast to its match type. For iterateds nested inside the pattern, in the pattern match object an iterated variable of type `IMatchesExact<IMatch_r_iteratedName>` is created; the IMatchesExact allows you to iterate over the patterns finally found. A subpattern usage is mapped directly to a variable in the match object typed with the match interface of the subpattern. The same holds for independent patterns; negative patterns do not appear in the match objects, for they prevent the matching and thus the building of a match object in the first place.

21.4 External Sequences

ExternalSequenceDeclaration

Registers an external sequence similar to a defined sequence (cf. 15.1) but in contrast to that it must or can be implemented outside in C# code. An external sequence declaration specifies the expected input types and the output types. The sequence must be implemented externally, see 20.6. You may extend GrGen.NET with external sequences if you want to call into external code to interface with libraries from outside the domain of graph rewriting, or if the GrGen.NET-languages are not well suited for parts of the task at hand.

21.5 Shell Commands

CommandLine is execute as-is by the shell of the operating system.

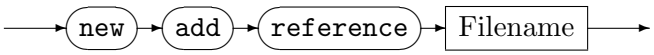
21.6 Shell and Compiler Parameters

When executing the GrGen.NET generator/compiler `GrGen.exe`, the follwing parameters are admissible:

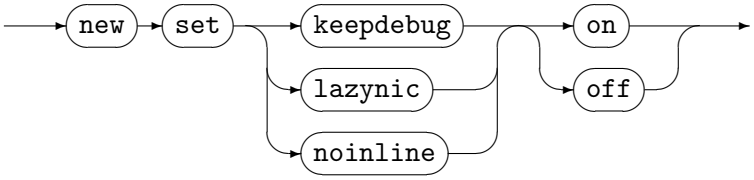
```
[mono] GrGen.exe [-keep [<dest-dir>]] [-debug] [-lazynic] [-noinline] [-r <assembly-path>]
```

The assembly *assembly-path* is linked as reference to the compilation result with the `-r` parameter.

These compiler parameters can be configured in the GrShell to be used, too:



Configures a reference to an external assembly *Filename* to be linked into the generated assemblies, maps to the `-r` option of `grgen.exe` (cf. 1.7.1).

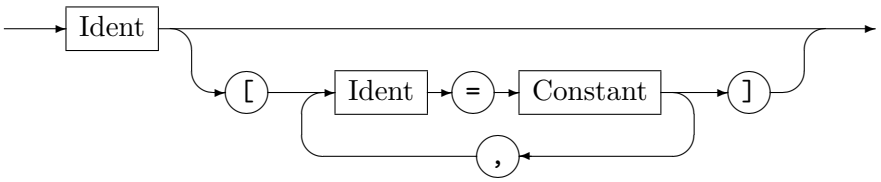


Configures the compilation of the generated assemblies to keep the generated files and to add debug symbols, or configures the generation of the matchers to execute negatives, independents, and conditions only at the end of matching (normally asap), or configures the generation of the matchers to never inline subpatterns. Maps to the `-keep` and the `-debug` options or to the `-lazynic` or to the `-noinline` option of `grgen.exe`.

21.7 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. But you may use the annotations to transmit information from the specification files to API level where they can be enumerated.

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

Table 21.1: Annotations

EXAMPLE (93)

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

CHAPTER 22

UNDERSTANDING AND EXTENDING GRGEN.NET

This chapter describes the inner workings of GrGen.NET to allow external developers

- to understand how GrGen.NET works, maybe just out of curiosity, but especially in order to write more efficient graph rewrite specifications
- extend GrGen.NET with new features, maybe being of general interest, maybe being only special extensions not of interest to other users

It starts with a section on describing how to build GrGen.NET, followed by a section giving an introduction into the generated code, then an introduction into the mechanism of search planning, followed by a section giving some details of the structure of, and the data flow in the GrGen.NET-code generator, ending in a discussion of performance optimizations.

22.1 How to Build

In case you want to build GrGen.NET on your own you should recall the system layout 1.1. The graph rewrite generator consists of a frontend written in Java and a backend written in C#. The frontend was extended and changed since its first version, but not replaced. In contrast to the backend, which has seen multiple engines and versions: a MySQL database based version, a PSQL database based version, a C version executing a search plan with a virtual machine, a C# engine executing code generated from a search plan and finally the current C# engine version 2 capable of matching nested and subpatterns. The frontend is available in the `frontend` subdirectory of the public mercurial repository at `https://bitbucket.org/eja/grgen`. It can be built with the included `Makefile` on Linux or the `Makefile_Cygwin` on Windows with the cygwin environment yielding a `grgen.jar`. Alternatively you may add the `de` subfolder and the jars in the `jars` subfolder to your favourite IDE, but then you must take care of the ANTLR parser generation pre-build-step on your own. The backend is available in the `engine-net-2` subdirectory. It contains a VisualStudio 2008 solution file containing projects for the `libGr`, the `lgspBackend` (libGr-Search-Plan-Backend) and the `GrShell`. Before you can build the solution you must execute `./src/libGr/genparser.bat`, `./src/libGr/GRSImporter/genparser.bat` and `./src/GrShell/genparser.bat` to get the CSharpCC parsers for the rewrite sequences, the GRS importer and the shell generated. Under LINUX you may use `make_linux.sh` to get a complete build. To get the API examples generated you must execute `./genlibs.bat`. The `doc` subdirectory contains the sources of the user manual, for building say `./build_cygwin.sh grgen` on Windows in `cygwin-bash` or `./build grgen` on Linux in `bash`. The `syntaxhighlighting` subdirectory contains syntax highlighting specifications for the GrGen-files for `Notepad++` and `vim`.

You can check the result of your build with the test suite we use to check against regressions. It is split into syntactic tests in `frontend/test` checking that example model and rule files can get compiled by `grgen` (or not compiled, or only compiled emitting warnings) and the resulting code can get compiled by `csc`. The tests get executed by calling `./test.sh` or

`./testpar.sh` from `bash` or `cygwin-bash` (`testpar.sh` executes them in parallel speeding up execution on multi core systems considerably at the price of potential false positive reports); deviations from a gold standard are reported. And semantic tests in `engine-net-2/tests` checking that executing example shell scripts on example models and rules yields the expected results. They get executed by calling `./test.sh`.

22.2 The Generated Code

In this section we'll have a look at what is generated by GrGen.NET of your specifications; firstly the model, secondly the rules, better matching their patterns, ending thirdly with an explanation of the matching of nested and subpatterns.

Internal Graph Representation

The graph structure is maintained in an `LGSPGraph` built of `LGSPNode` and `LGSPEdge` objects, basically without type and attribute information. The types defined in the model are realized by generated node and edge interfaces defining the user visible types and attributes, which are implemented by generated node and edge classes inheriting from and working like `LGSPNode`s and `LGSPEdge`s in the graph, additionally implementing the type and attribute bearing interfaces.

The nodes and edges are contained in a system of ringlists. Top level there are type ringlist, every node or edge is contained in the linked list of its specific type; the dummy head nodes or head edges serving as entry points into these structures are stored in an array in the graph. Every node or edge contains a field `typeNext` giving the next element of the type. These lists allow to quickly look up all elements from the graph bearing a certain type. Figure 22.1 gives an example for a graph with 3 node types, one having no instance nodes, one having only one instance node, and one having 5 instance nodes; the same holds for the edge types.

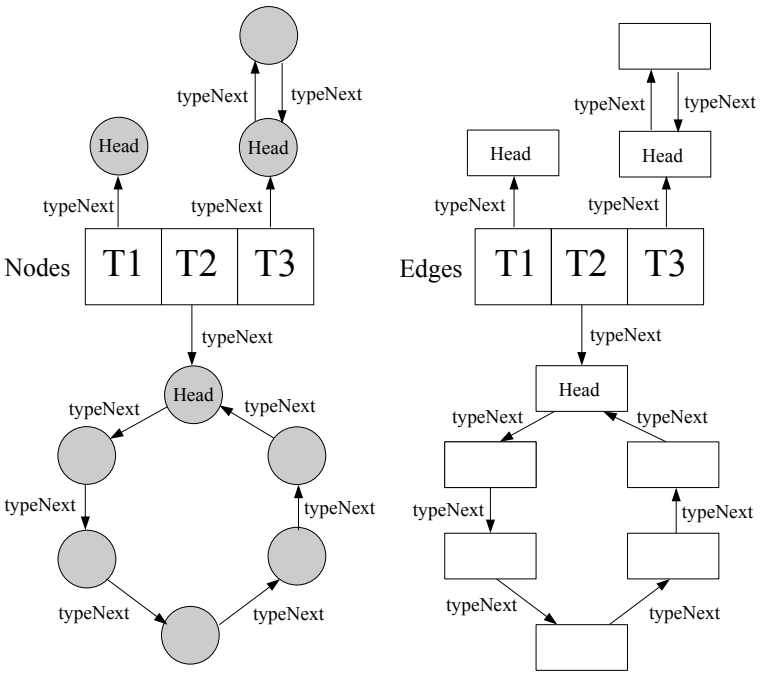


Figure 22.1: Example for type ringlists

The connectedness information is stored in a ringlist containing the incoming edges and a ringlist containing the outgoing edges for every node object; the node object contains a field `inHead` referencing an arbitrary edge object of the incoming edges (or `null` of there

is none) and a field `outHead` referencing an arbitrary edge object of the outgoing edges (or `null` if there is none). These lists allow to quickly retrieve all incoming or all outgoing edges of a node. The edge objects contain fields `source` and `target` referencing the source and the target node. They give instant access to the source and target nodes of an edge. Edges furthermore contain a field `inNext` to give the next edge in the incoming ringlist and a field `outNext` to give the next edge in the outgoing ringlist they are contained in. All the 3 ringlists (type, in, out) are doubly linked (to allow for fast insertion and deletion), so for every `next` field there is also a `prev` field available. The figure 22.3 gives the ringlist implementation of the example graph depicted in figure 22.2.

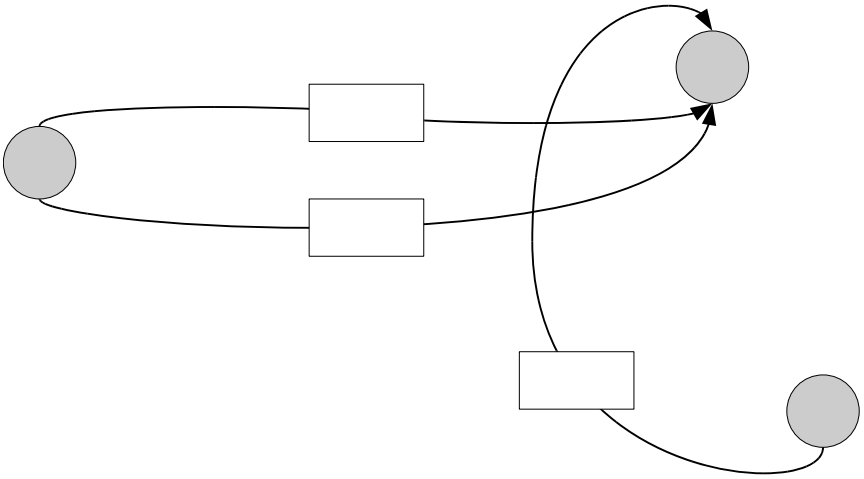


Figure 22.2: Incidence example situation

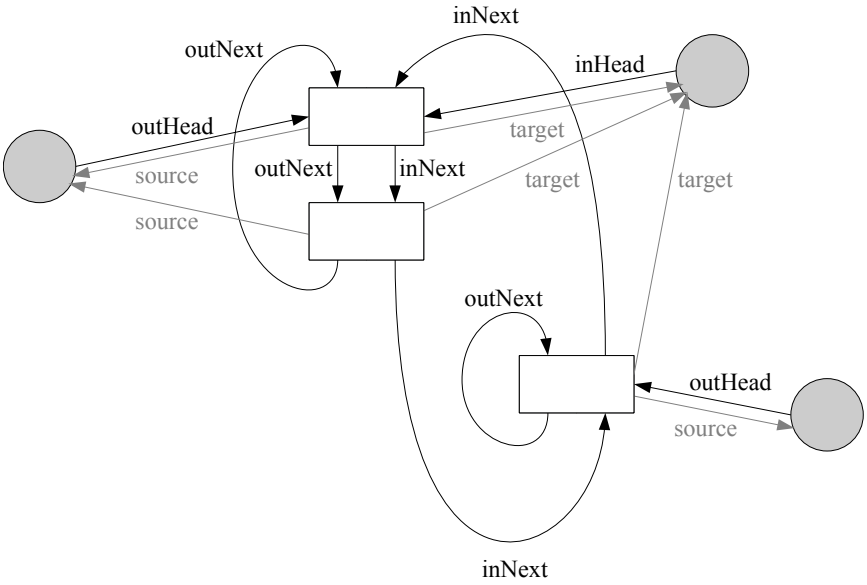


Figure 22.3: Ringlist implementation of incidence example

Pattern Matching and Search Programs

The pattern of a rule (or test) is matched with a backtracking algorithm binding one pattern element after another to a graph element, checking if it fits to the already bound parts. If

it does fit search continues trying to bind the next pattern element (or succeeds building the match object from all the elements bound if the last check succeeds), if if does not fit search continues with the next graph element; if all graph element candidates for this pattern element are exhausted, search backtracks to the previous decision point and continues there with the next element. For every pattern to match a search program implementing this algorithm is generated, basically consisting of nested loops iterating the available graph elements for each pattern element and condition checking code continuing search with the next element to investigate. Figure 22.4 shows a pattern and a search program generated for it.

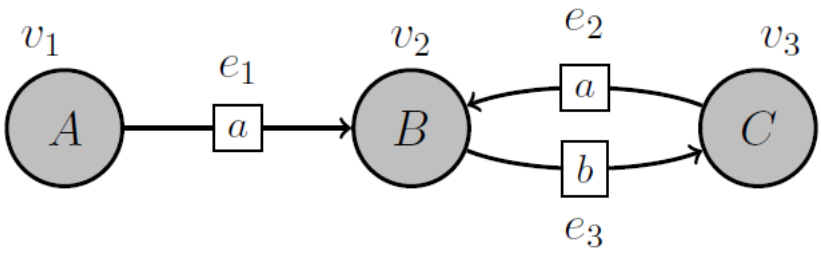


Figure 22.4: Pattern to search

```
1  foreach(v1:A in graph) {
2     foreach(e1 in outgoing(v1)) {
3        if(type(e1)!=a) continue;
4        v2 = e1.tgt;
5        if(type(v2)!=B) continue;
6        foreach(e3 in outgoing(v2)) {
7           if(type(e3)!=b) continue;
8           v3 = e3.tgt;
9           if(type(v3)!=C) continue;
10          foreach(e2 in outgoing(v3)) {
11             if(type(e2)!=a) continue;
12             if(e2.tgt!=v2) continue;
13             // v1,e1,v2,e3,v3,v2 constitute a match
14          }
15       }
16    }
17 }
```

If a non-leaf-type (regarding inheritance hierarchy) is to be matched with a graph lookup (the outermost loop in the example), then an additional loop is used iterating all subtypes of the type of the pattern element. If a pattern is given one or more parameters, search normally follows the graph structure by iterating the incoming or outgoing edges of a node or determining the source or target node of an edge, instead of looking up a type in the graph. If a pattern consists of several unconnected components, several lookups are needed. Undirected or arbitrary directed pattern edges are searched in both, the incoming and the outgoing ringlist; undirected edges are implemented by normal directed edges which can get matched in both directions. Constraints which might cause an candidate to get rejected are the type of the element as shown in the example, structural connections to already bound elements and the isomorphy constraint. Furthermore there are negative checking, independent checking, and attribute checking, which are normally depending on multiple elements. Other matching operations are storage access, storage attribute access and storage mapping.

Compared to pattern matching which may need a long time, pattern rewriting is a very simple task following a simple sequence of node or edge operations, first creating all new nodes then edges, followed by an evaluation of the attributes, then deleting all edges and nodes, finally executing embedded sequences (see table 7.1 for more on this).

A notable performance optimization allowed by the graph model is search state space stepping: after a pattern was matched, the list heads of the type lookup ring lists, the incoming ring lists and the outgoing ring lists are moved to the position of the matched entries with `MoveHeadAfter`, `MoveInHeadAfter` and `MoveOutHeadAfter`. With this optimization the pattern matching during an iteration `r*` will start where the previous iteration step left of, saving the cost of iterating again through all the elements which failed in the previous iteration.

Pushdown Machine for Nested and Subpatterns

Every subpattern (as introduced in chapter 7) is handled with a search program corresponding to its pattern as introduced in the previous section. The interesting part is how the subpatterns used get combined, which happens with a $2 + n$ pushdown machine. It consists of a call stack, containing the subpattern instances found with the bound elements in the local variables of the search program frame, an open tasks stack containing the subpatterns to match (when a subpattern was matched its contained subpatterns are pushed to that open tasks stack, then the top of the stack gets processed), and n result stacks containing the (partial) match object trees; for a normal rule application $n = 1$, for an all-bracketed rule the number of matches is unbound. A simulation of this machine, i.e. the matching process of a pattern using subpatterns is shown on the following pages.

Alternatives are handled like a subpattern with several possible patterns which are tried out, the first one matching is accepted. Iterateds are handled like subpatterns whose tasks are not removed when they get matched, but only if matching failed or the maximum amount of matches was reached. In case of a failure the minimum required amount of matches is inspected, if the amount of found matches is larger or equal then matching partially succeeds and continues with the next open task (or plain succeeds if there are no open tasks left), otherwise matching of the given partial match failes causing matching to backtrack (to the previous decision point of influence).

The advantages of this design linearizing the pattern tree on the call stack are the rather low usage of heap memory, the ability to reuse the match programs for the patterns as introduced in the previous section, and the ability to find all matches (the n in $2+n$ pushdown machine stands for the number of matches found).

Rewriting is carried out by a depth-first traversal of the match object tree, creating new elements before decending, evaluating attributes and deleting old elements before ascending. For each pattern piece match the modification specified in the rewrite part is carried out, i.e. the overall rewrite gets combined from the rewrites of the pattern pieces (cf. table 7.1 for the order of rewriting).

The subpattern usage parameters are computed during matching from matched elements and call expressions (inherited attribution), LHS yielding is carried out after the match was found during match object building within yield statements (synthesized attribution), RHS yielding is carried out during match object tree rewriting within eval statements (left attribution, with a user defined left-relation).

In the following, an example run of the $2 + n$ pushdown machine is given:

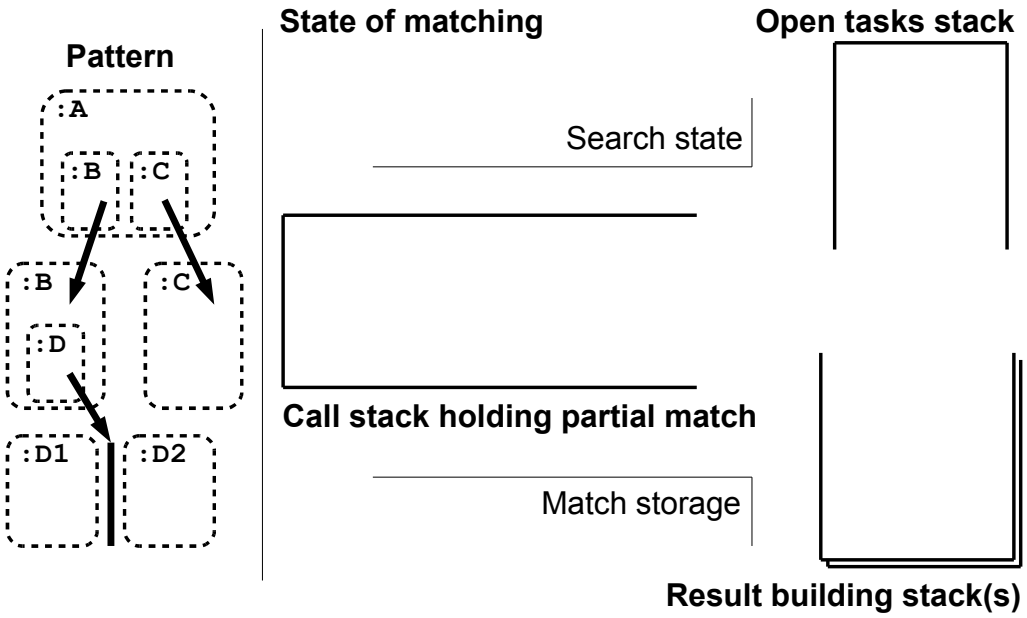


Figure 22.5: 1. Start state

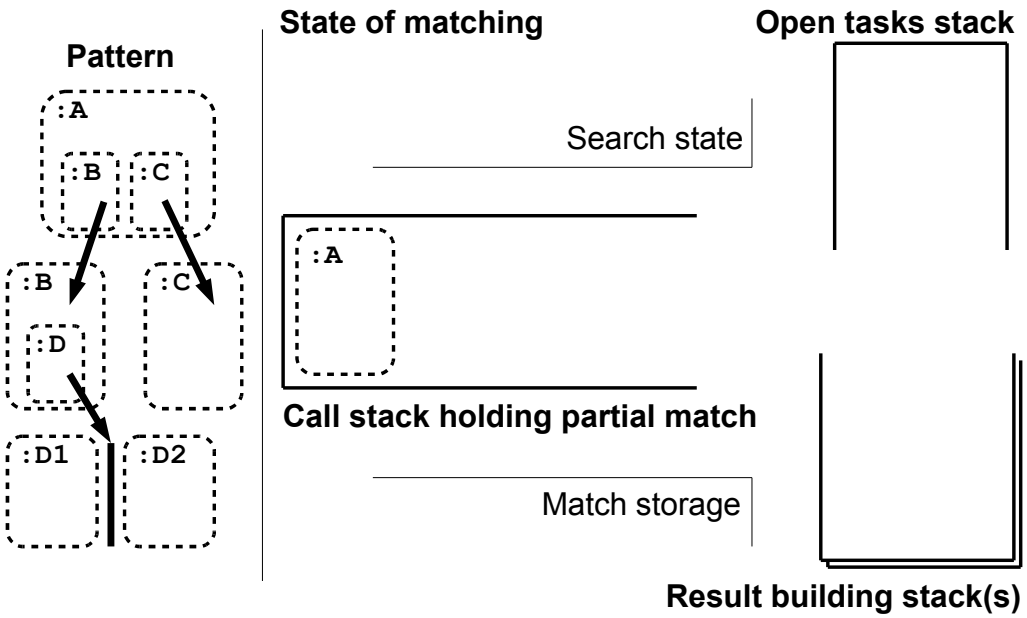


Figure 22.6: 2. The terminal part of pattern A was matched

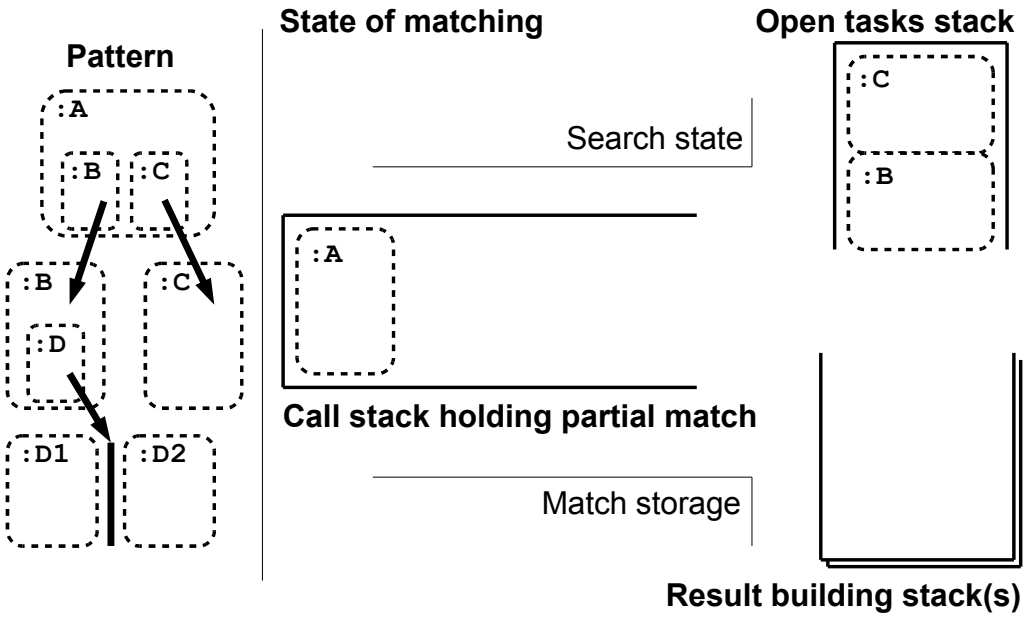


Figure 22.7: 3. The tasks for subpatterns B and C are pushed

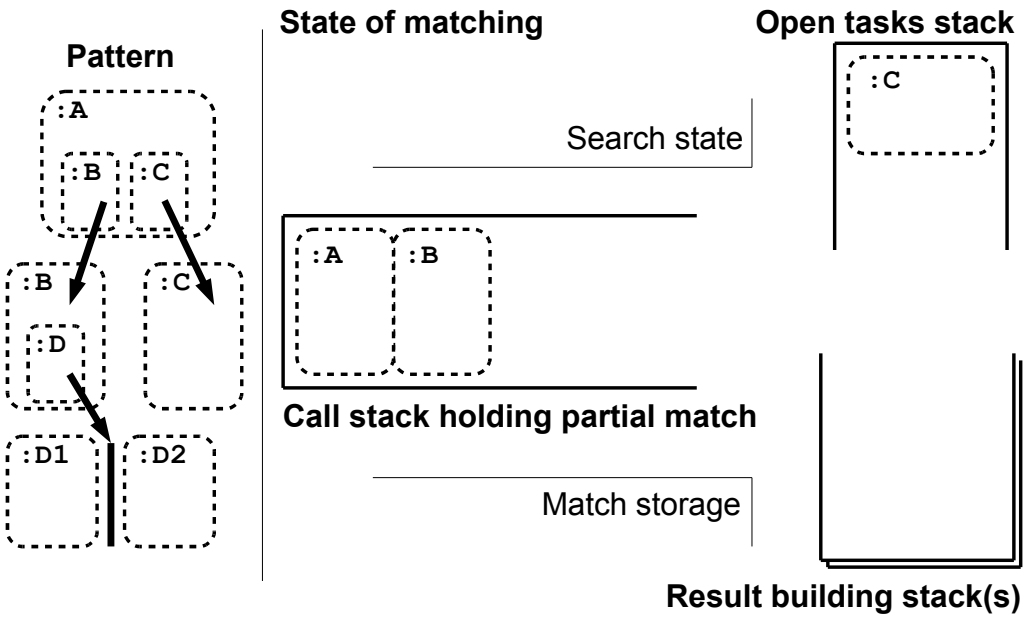


Figure 22.8: 4. The task for B gets executed, the terminal part of B was matched

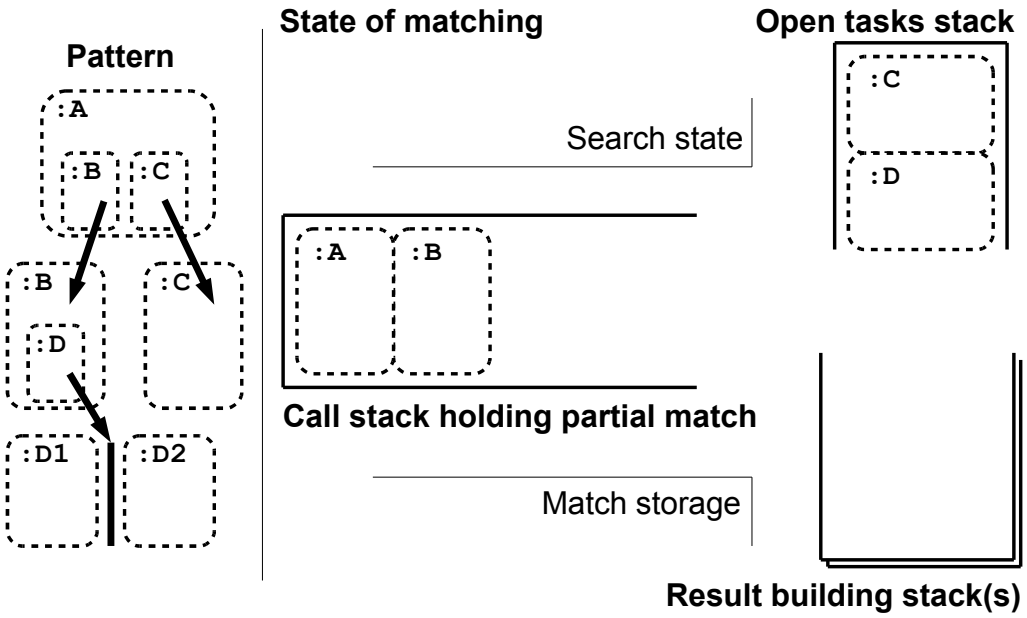


Figure 22.9: 5. The task for alternative D is pushed

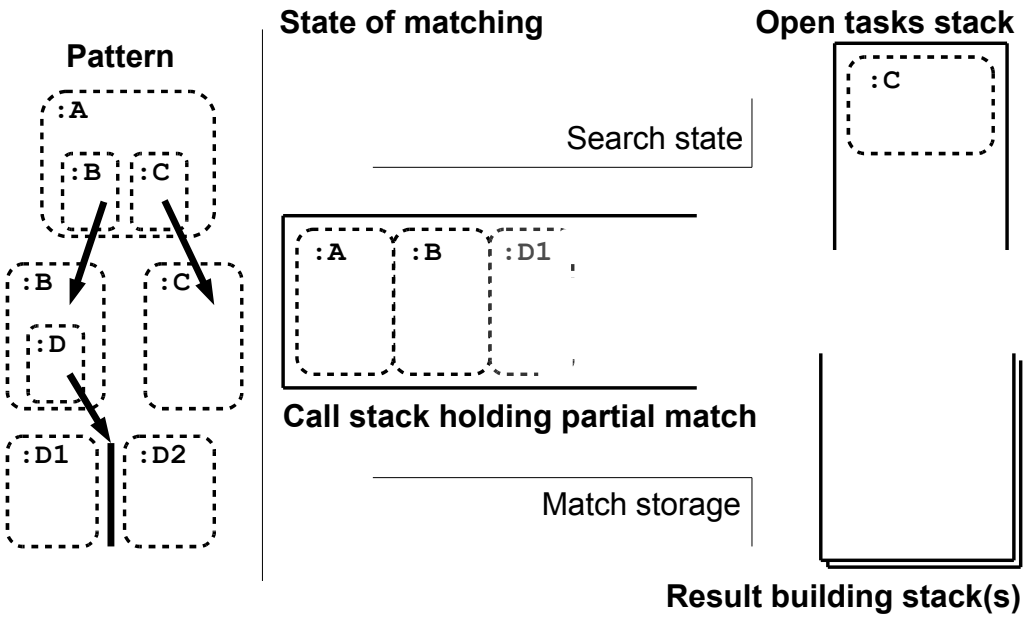


Figure 22.10: 6. The task for D gets executed, D1 is tried, but matching fails

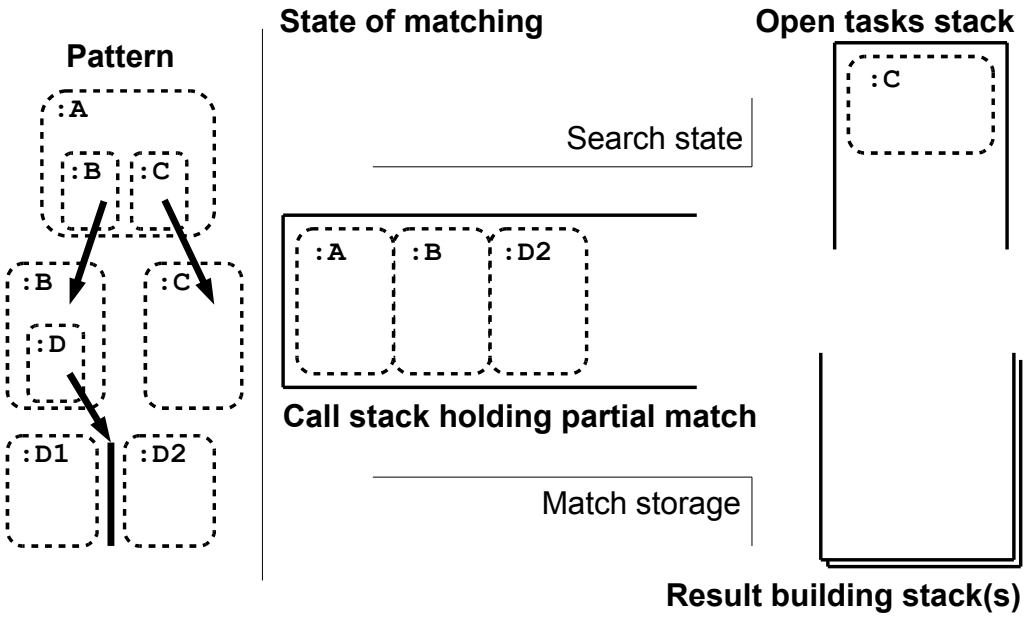


Figure 22.11: 7. The task for D gets executed, D2 was matched

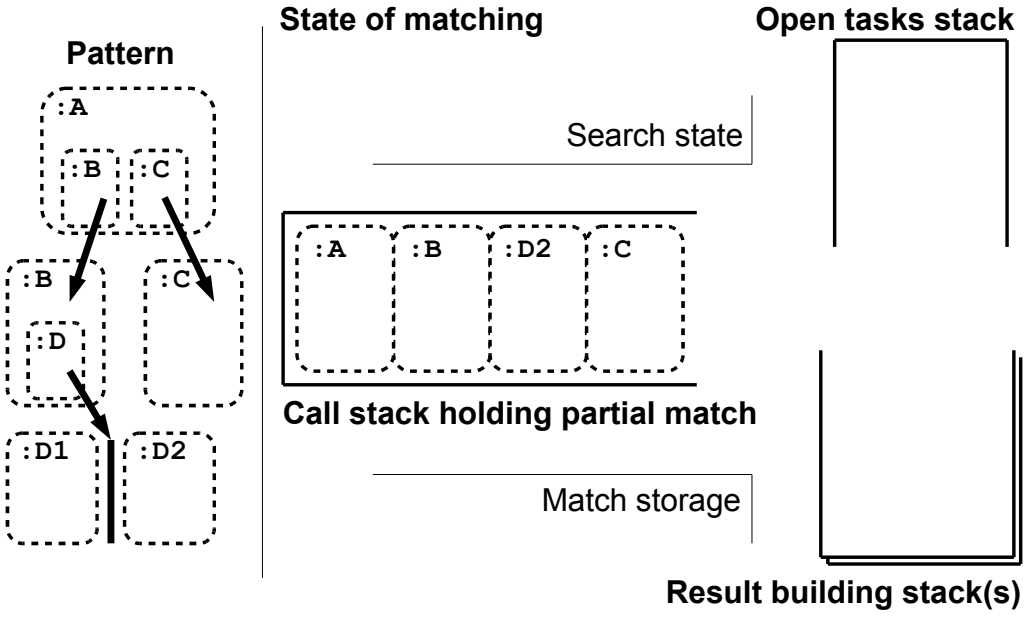


Figure 22.12: 8. The task for C gets executed, C was matched, a match for the overall pattern was found, but is contained only on the call stack

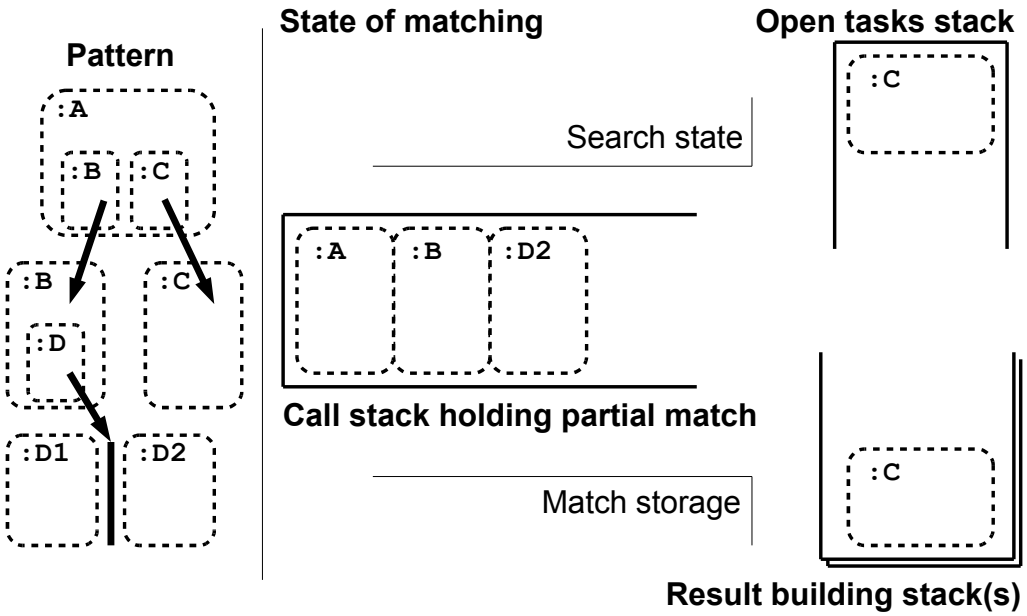


Figure 22.13: 9. The match of C is popped from the call stack and pushed to the result stack

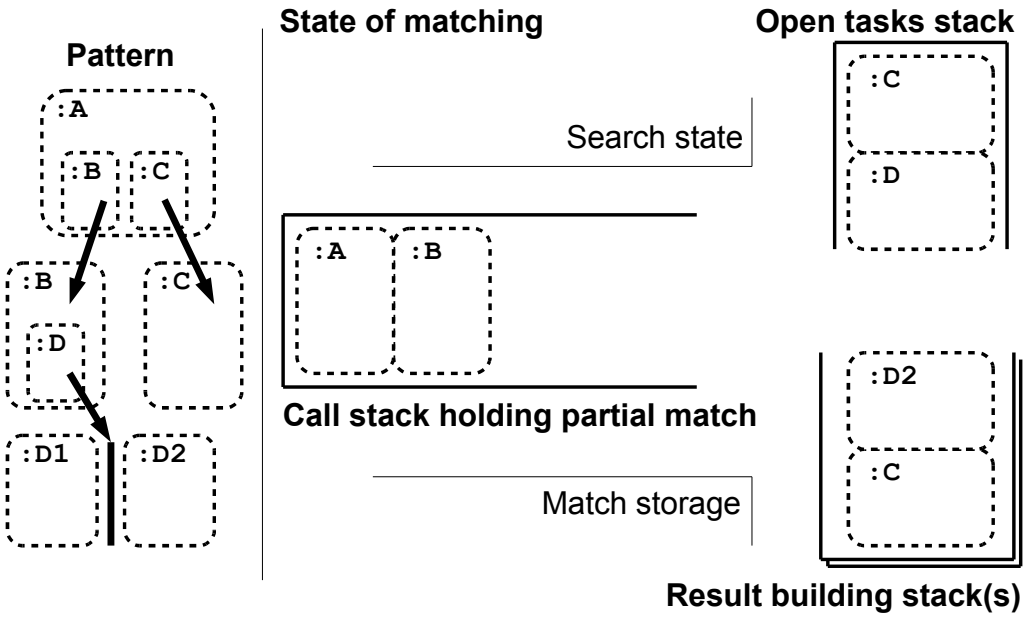


Figure 22.14: 10. The match of D2 is popped from the call stack and pushed to the result stack

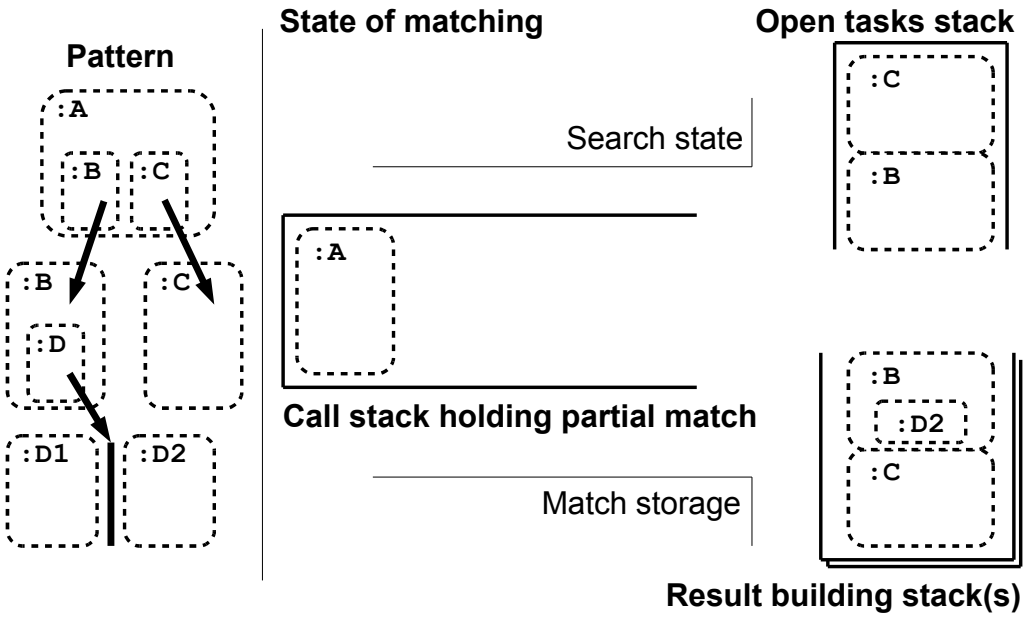


Figure 22.15: 11. The match of B is popped from the call stack, D2 from the result stack is added, the combined match is pushed to the result stack

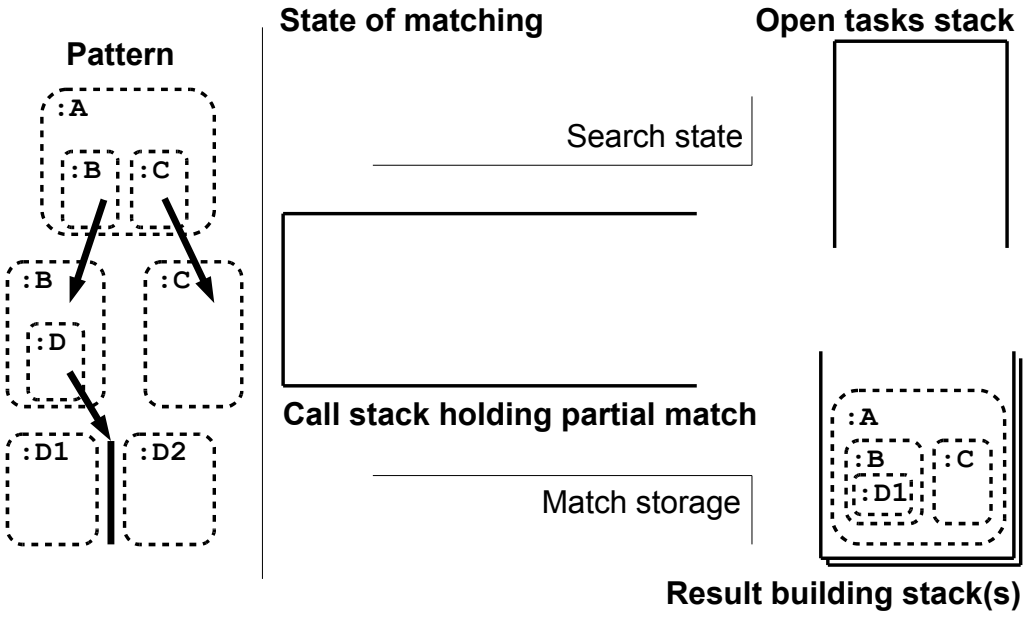


Figure 22.16: 12. The match of A is popped from the call stack, B and C from the result stack are added, now we got the combined match of the overall pattern

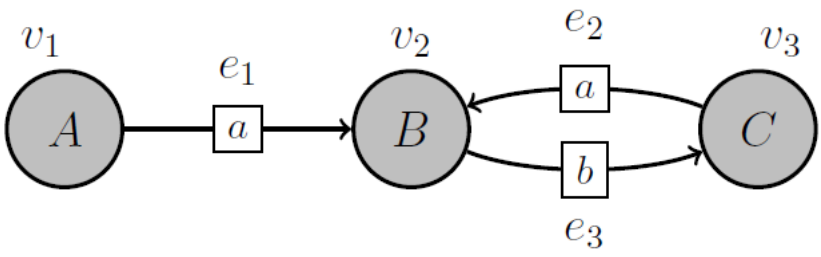


Figure 22.17: Pattern to search

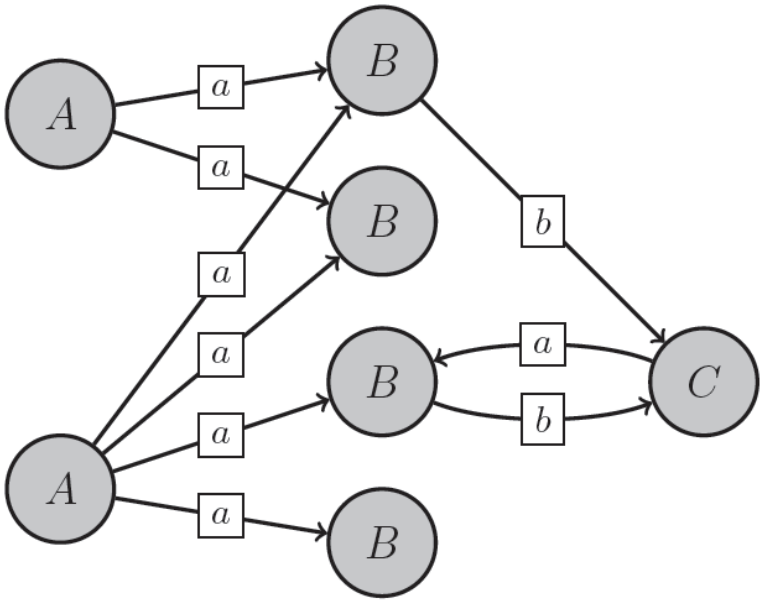


Figure 22.18: Host graph to search in

22.3 Search Planning in Code Generation

In the previous section a search program for the pattern (repeated in figure 22.17) was given. It follows the schedule
`lkp(v1:A); out(v1,e1:a); tgt(e1,v2:B); out(v2,e3:b); tgt(e3,v3:C); out(v3,e2:a)`
A schedule is a more abstract version of a search program, with search operations `lkp` denoting node (or edge) lookup in the graph by iterating the type list, `out` iterating the outgoing edges of the given source node and `in` iterating the incoming edges ot the given target node, `src` fetching the source from the given edge and `tgt` fetching the target from the given edge. For some graphs it might work well, but for the graph given in figure 22.18 is is a bad search plan. Why so can be seen in the search order sketched in figure 22.19. Due to the multiple outgoing edges of `v1` of which only one leads to a match it has to backtrack several times. A better search order would be one matching this edge from `v2` on in reverse direction; due to the graph model containing a list of outgoing and a list of incoming edges, each edge can be traversed in either direction.

For every pattern there are normally multiple search programs available, each able to find all the matches which exist, but with vastly different performance characteristics. In order to improve performance, GrGen.NET tries to prevent following graph structures splitting into breadth as given in this example or lookups on types which are available in high quantities, by first executing a planning phase to choose a good schedule. Due to the planning phase this schedule is chosen:
`lkp(v3:C); out(v3,e2:a); tgt(e2,v2:B); out(v2,e3:b); in(v2,e1:a); src(e1,v1:A)`
corresponding to the search order depicted in figure 22.20 and the search program:

```
1  foreach(v3:C in graph) {
2     foreach(e2 in outgoing(v3)) {
3        if(type(e2)!=a) continue;
4        v2 = e2.tgt;
5        if(type(v2)!=B) continue;
6        foreach(e3 in outgoing(v2)) {
7           if(type(e3)!=b) continue;
8           if(e3.tgt!=v3) continue;
9              foreach(e1 in incoming(v2)) {
10             if(type(e1)!=a) continue;
11             v1 = e1.src;
12             if(type(v1)!=A) continue;
13             buildMatchObjectOfPatternWith(v3,e2,v2,e3,e1,v1);
14          }
15       }
16    }
17 }
```

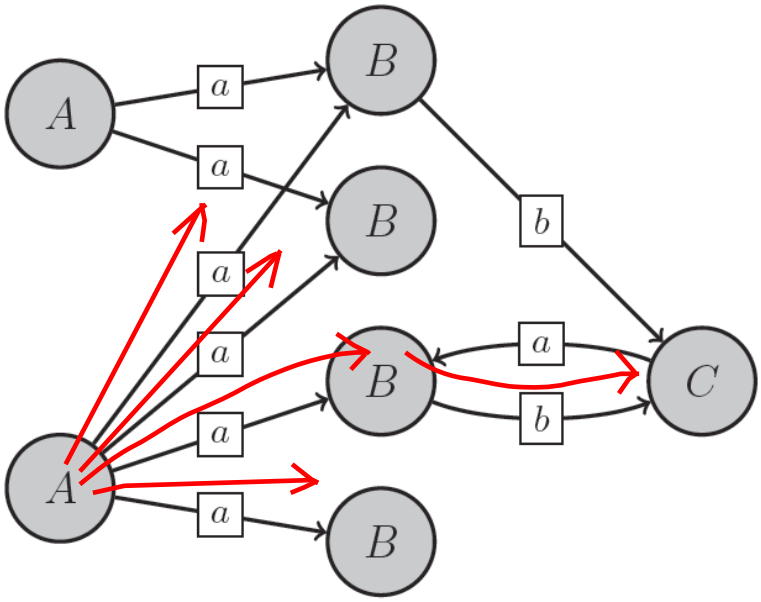


Figure 22.19: Bad search order

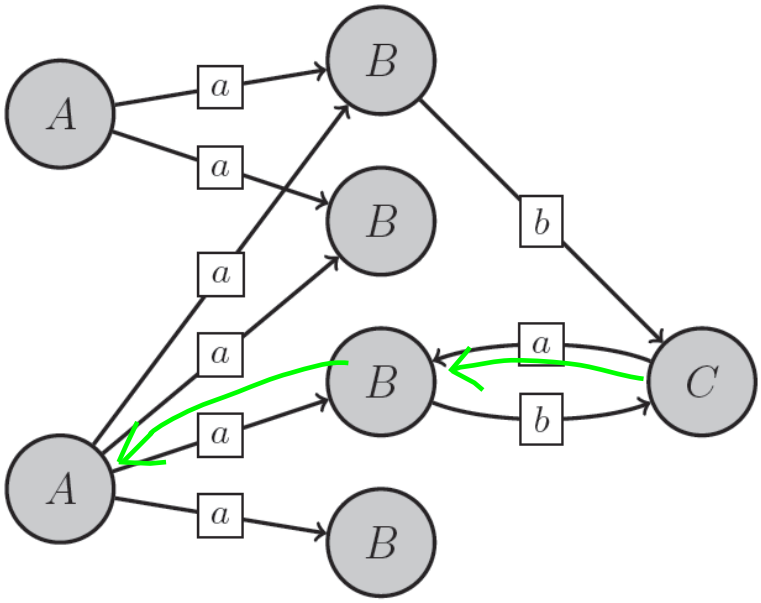


Figure 22.20: Good search order

The mechanism of search planning works by computing from the pattern graph a search plan graph. A search plan graph is an edge-weighted graph, with nodes corresponding to the pattern elements – both nodes and edges – and edges corresponding to operations to match them, with weight attributes giving an estimated cost of the operation. A search plan graph contains an additional root node, with an outgoing edge to each other node defining a `lookup` operation. From plan nodes created for nodes there are `outgoing` and `incoming` operations leaving towards plan nodes created from edges, from plan nodes created for edges there are `source` and `target` operations leaving towards plan nodes created from nodes. The cost is determined by analyzing the amount of splitting between adjacent nodes for every triple of $(nodetype, edgetype, nodetype)$ in the graph – called a V-Structure – and by counting the number of elements of every node type or edge type. In the plan graph a minimum spanning arborescence, i.e. directed tree is computed. The arborescence defining

the matching operations causing least cost is then linearized into a schedule, which is a list of the search operations chosen as already introduced with the good and the bad schedules. The details of search planning and some evaluation how well it works are given in [Bat06b] and in [BKG08].

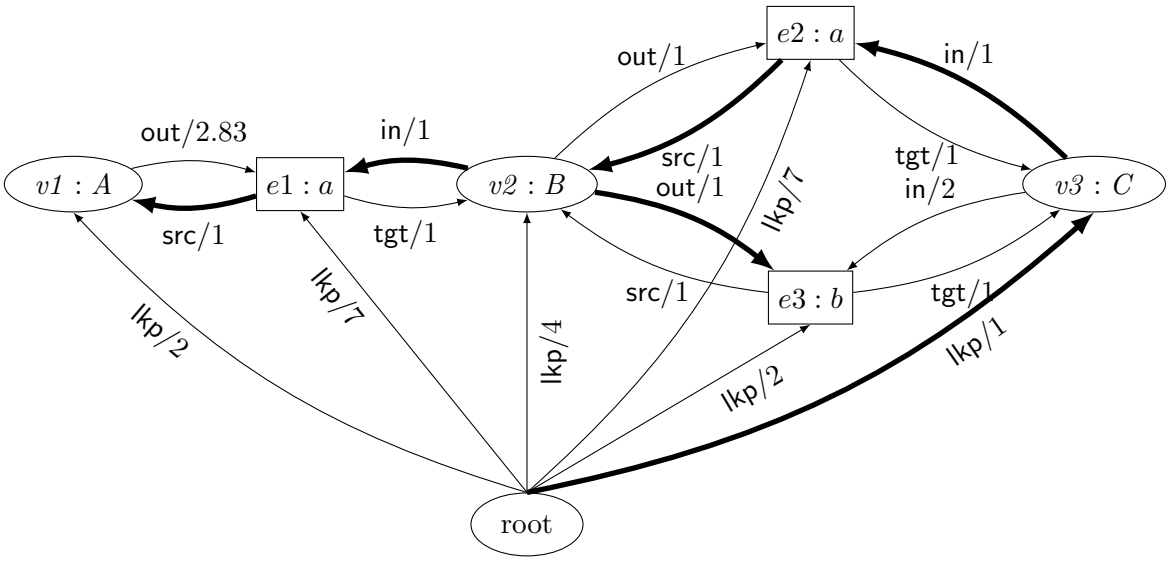


Figure 22.21: The search plan graph for the pattern graph of Figure 22.17 with estimated backtracking costs induced by the host graph of Figure 22.18. The found minimum directed spanning tree is denoted by thick edges.

A search program is a lower abstraction version of a schedule. Structurally it is a tree data structure reflecting the syntax tree of the code to generate (with list entries maybe containing further lists), as sketched in figure 22.22; in contrast to the schedule which is a simple list data structure.

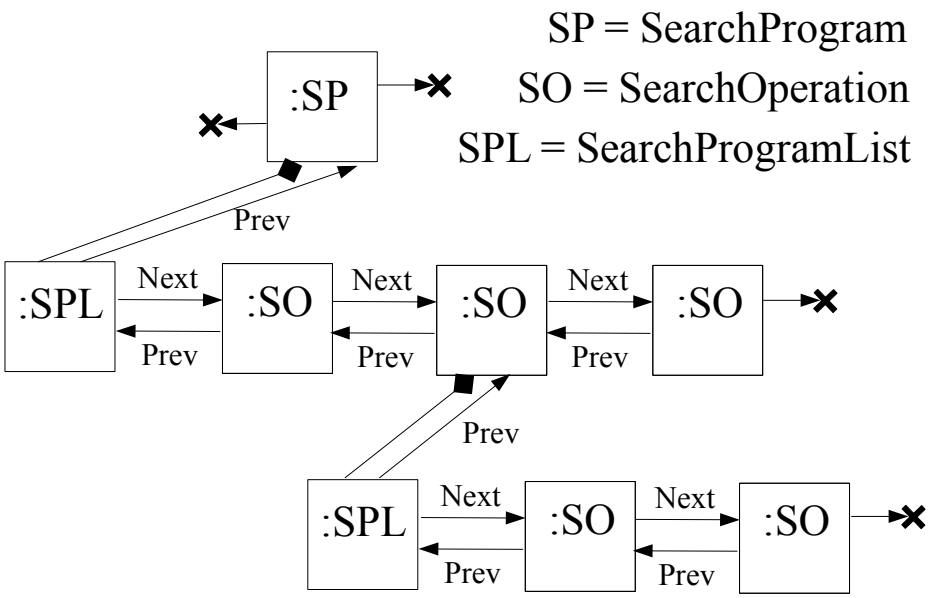


Figure 22.22: A simple Search Program

It contains explicit instructions for isomorphy checking and connectedness checking, which are inserted after the schedule was determined along the exact sequence of instructions. Connectedness checking can be seen in the good search program of the example in the check

that the target of `e3` is indeed `v3`; a target matching operation `tgt(e3,v3:C)` is not used because `v3` was already matched by the lookup operation, and is already contained in the spanning tree. Furthermore it contains exact locations where to continue at, which might be different than the directly preceding search operation (because that one does not define a choice point of influence).

A remark on isomorphy handling: As a consequence of the simple loop based graph element binding to pattern elements, without an explicit check all pattern elements could get matched completely homomorphically to each other (due to the loops starting with the same elements, the homomorphic matches *are* normally the first to be found). To ensure that two pattern elements are not bound to the same graph element, flags contained in the graph elments are set when a graph element is bound to a pattern variable and reset when the binding is given up again (one flag for each negative/independent nesting level, until the implementation defined limit of flags available in the graph elements is reached, then a list of dictionaries is used); the flags are checked then in the following, nested matching operations. An iso-check scheduling pass ensures that checks are only inserted if the elements must be isomorphic and their types do not already ensure that they can't get matched to the same elements.

22.4 The Code Generator

22.4.1 Frontend

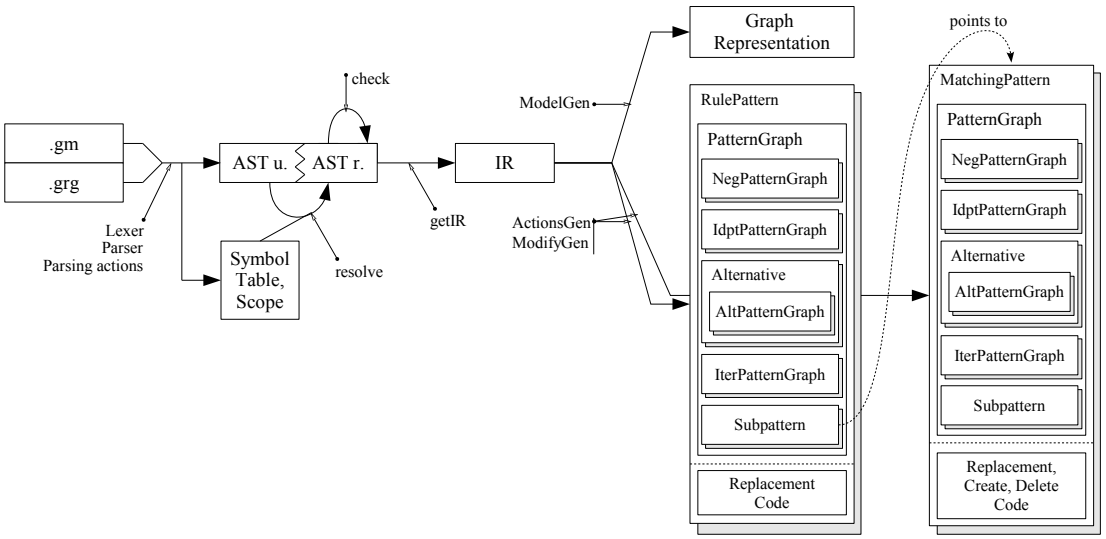


Figure 22.23: Frontend Code Generation

The frontend is spread over the directories `parser`, `ast`, `ir`, `be` and `util`, with their code being used from `Main.java`.

Syntax and Static Semantics

The directory `parser` contains parser helpers like the symbol table and scopes and within the `antlr` subdirectory the ANTLR parser grammar of GrGen.NET in the file `Grgen.g`. The semantic actions of the parser build an abstract syntax tree consisting of instances of the classes given in the directory `ast`, with the base class `BaseNode`. The AST is operated upon in three passes, first resolving by `resolve` and `resolveLocal`, mainly replacing identifier nodes by their declaration nodes during a (largely) preorder walk. Afterwards the AST is checked

by `check` and `checkLocal` during a (largely) postorder walk for type and other semantic constraints. Finally an intermediate representation is built from the abstract syntax tree by the `getIR` and `constructIR` methods.

Intermediate Representation

The IR classes given in the `ir` folder can be seen as more lightweight AST classes; their name is often the same as for their corresponding AST classes, but without the `Node`-suffix which is appended to all AST classes. The most interesting classes are `Rule` used for rules, alternative cases and iterateds, as well as `PatternGraph` used for all pattern graphs including negatives and independents; several data flow analyses are contained in `PatternGraph`, some covering the nesting of patterns, some being even global. A particularily interesting one is `ensureDirectlyNestingPatternContainsAllNonLocalElementsOfNestedPattern` which ensures that, from a pattern which contains a certain entity the first time, up to every pattern that references this entity, all intermediate patterns contain that entity, too.

This is done in a recursive walk over the nested patterns in the IR-structure. Each pattern receives the set of already known elements as parameter, before descending the elements available in the current pattern are added to the set of already known elements, then recursive descent into the directly nested patterns follows handing down the already known elements. On ascending elements are added to the current pattern if they are contained in a directly nested pattern, but not in the current pattern, although they are known in the current pattern.

This is done in the IR to keep the backends simple. The IR classes are the input to the two backends of the frontend, as given in the folders `be/C` and `be/Csharp`.

Backends

The directory `be/C` contains the code generator for the C based backend integrated into the IPD C compiler. (The compiler transforms a C program into a graph and SSA based compiler intermediate representation named FIRM using libFirm (see `libfirm.org`, [TLB99], [Lin02]) and further on to x86 machine code.)

The directory `be/Csharp` contains the code generator for the C# based backend of GrGen.NET. It generates the model source code `FooModel.cs` with the node and edge classes for a rule file named `Foo.grg`, and the intermediate rules source code `FooActions_intermediate` with a description of the patterns to match, a description of the embedded graph rewrite sequences, and the rewriting code. This is done in several recursive passes over the nesting structure of the patterns in the IR.

The backend of the frontend does *not* generate the complete rules source code with the matcher code or the code for the embedded rewrite sequences — this is done by `grgen.exe` which only calls the `grgen.jar` of the frontend. You may call the Java archive on your own to get a visualization of the model and rewrite rules from a `.vcg`-dump of the IR, cf. Note2.

Model Generation

The model generation code in `ModelGen` is rather straight forward:

1. first code for the user defined enums is generated,
2. then the node classes are generated,
3. followed by the node model,
4. then the the edge classes are generated,
5. followed by the edge model,

6. and finally the graph model is generated.

For the nodes as well as for the edges three classes are generated:

1. the first being the interface visible to the user, giving access to the attributes,
2. the second being the implementation implementing the interface and inheriting from `LGSPNode` or `LGSPEdge`,
3. and the third being a type representation class inheriting from `NodeType` or `EdgeType` giving informations about the type and its attributes, used in the node/edge model and thus graph model.

Rule Representation Pass

The action code, better representation generation is implemented in `ActionsGen`, in the code called from `genSubpattern` for the subpatterns and `genAction` for the rules and tests on.

In a prerun from `genRuleOrSubpatternClassEntities` on some needed entities are generated, e.g. type arrays for the allowed types of the pattern elements (the arrays are only filled if the base type was constrained), or indexing enums, which map the pattern element names to their index in the arrays of the matched host graph elements in the match objects of the generic match interface.

In the rule representation pass from `genRuleOrSubpatternInit` on the subpattern- and rule representations of type `MatchingPattern` and `RulePattern` are generated, including the contained `PatternGraph`-objects for the (nested) pattern graph(s), in a mutual recursion game of `genPatternGraph` and `genElementsRequiredByPatternGraph`.

The method `genElementsRequiredByPatternGraph` generates for a pattern graph its contained elements, the pattern nodes, the pattern edges, the used subpatterns as `PatternGraphEmbedding` members, the contained alternatives, and the iterateds; the representations of contained negative subpatterns and alternative cases get generated via `genPatternGraph` before the containing pattern.

A graph element contained in pattern, but defined in a nesting pattern, is saved in the pattern as a reference to the element in the nesting pattern. The `PatternGraph` in which it was used first is remembered in a `PointOfDefinition`-membervariable. The source and target nodes of edges are saved in the `PatternGraph` in hash tables (so that an edge with an undefined target can receive a target in a nested pattern), source and target nodes are determined by ascending along the pattern nesting until a definition is found.

As all graph elements, including the ones of the nested patterns, are created flatly in the `initialize`-method of the `MatchingPattern`, name prefixes are needed to prevent name clashes; this is ensured by the parameter `pathPrefix`. Correct naming of already declared elements is ensured with a `alreadyDefinedEntityToName`-hash-table. The split into constructor and `initialize`-Methode ist needed because a recursive `MatchingPattern` must reference itself at construction.

Furthermore expressions trees for the attribute checks are generated, as well as local and global homomorphy tables for the nodes and the edges, defining which pattern elements may match the same host graph element (the global tables specify the homomorphy in between elements from the `PatternGraph` of an alternative case or an iterated and an enclosing `PatternGraph`).

Rewrite Code Pass

In the rewrite pass the rewrite code gets generated, via `genModify` of `ModifyGen`; the nesting of negative or independent patterns does not need to be taken care of here as they don't possess rewrite parts.

For every pattern piece and its replacement role (dependent rewriting, creation, deletion) are local rewrite methods generated, bringing at runtime the overall rewriting into effect by calling each other. `ModifyGenerationState` holds the state during generation, while the `ModifyGenerationTask`s allow to give a left and right graph independent of the rewriting specified with the rule. The code for subpattern creation is generated by using an empty graph as left graph and the pattern graph as right graph, the code for subpattern deletion is generated by using the pattern graph as left graph and an empty graph as right graph, for a dependent rewrite the graphs from `IR::Rule` are simply referenced, for a test the pattern graph is used as left and right graph. For alternatives and iterateds dispatching methods get generated calling the rewrite methods of the instances of the patterns matched. Rewrite parameters are mapped to local parameters of the rewrite methods.

The core method of rewrite code generation `genModifyRuleOr-Subrule` generates the rewriting of the given pattern and the calls to the rewrite methods of the nested patterns. Because the rewrite methods of the nested patterns are placed flatly in their `MatchingPattern` or `RulePattern`, they receive their nesting path as name prefix, their role is distinguished by one of the name postfixs `Modify`, `Create` or `Delete`; for keeping a pattern unmodified obviously no methods are needed.

For embedded sequences `LGSPEmbeddedSequenceInfo` objects are generated, which do contain the sequence as string plus additional parameter informations, the real sequence code is generated in the backend. For sequences embedded in alternatives, iterateds, or subpatterns additionally closure classes inheriting from `LGSPEmbeddedSequenceClosure` are generated. They store the graph elements the pattern elements were bound to of the pattern containing the exec; the sequence execution function is then called from this closure, which is stored in a queue and executed from the top level rule.

Finally match classes are generated, for every pattern besides negative patterns an interface and a class implementing this interface. The match classes are instantiated after a match of the corresponding pattern was found, giving a highly convenient and type safe interface to the matched entities to the user at API level.

22.4.2 Backend

The real matcher code is generated by the backend given in `engine-net-2`, in the `src/lgspBackend` subdirectory. The processing is done in several passes in `lgspMatcherGenerator.cs`; the base data structure is the `PatternGraph`, resp. the nesting of the `PatternGraph`-objects, contained in the `RulePattern`-objects of the rules/tests or the `MatchingPattern`-objects of the subpatterns.

1. Step

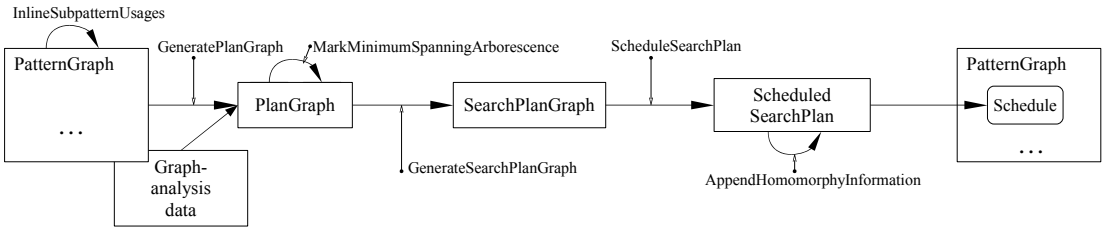


Figure 22.24: Backend Code Generation, step 1

First the subpattern usages are inlined when inlining is assumed to be benefitial; this allows the programmer to extract common patterns into subpatterns increasing readability and modularity without loosing performance (but as of now only one level of inlining is supported, so beware of too much subpattern extraction, especially if the containing pattern would get disconnected by this). After (and partly before) this pattern rewriting the patterns and their relations are analyzed by the `PatternGraphAnalyzer` – the analyze results are used for emitting better code later on (choosing more efficient but more limited implementations for language features which are not sufficient in the general case but are so for the specification at hand). Then a `PlanGraph` is created from the `PatternGraph` and data from analyzing the host graph (for generating the initial matcher some default data given from the frontend is used). A mimimum spanning arborescence is computed defining a hopefully optimal set of operations for matching the pattern (the hopes are founded, see [BKG08]). A `SearchPlanGraph` is built from the arborescence marked in the `PlanGraph` and used thereafter for scheduling the operations into a `ScheduledSearchPlan`, which gets completed by `AppendHomomorphyInformation` with isomorphy checking information. The `ScheduledSearchPlan` is then stored in the `PatternGraph` it was computed from.

2. Step

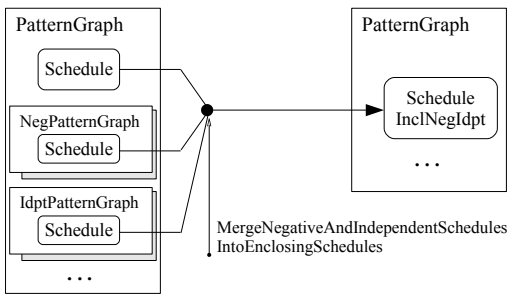


Figure 22.25: Backend Code Generation, step 2

In a second step the `Schedules` of the negative or independent `PatternGraphs` are integrated into the `Schedule` of the enclosing `PatternGraph`, by `MergeNegativeAndIndependentSchedulesIntoEnclosingSchedules` in a recursive run over the nesting structure of the `PatternGraphs` in the `MatchingPatterns` or `RulePatterns`. Due to nested negative or independent graphs this may happen spanning several nesting levels; the result is saved in the `ScheduleIncludingNegativesAndIndependents` field of the non-negative/independent `PatternGraphs`.

3. Step

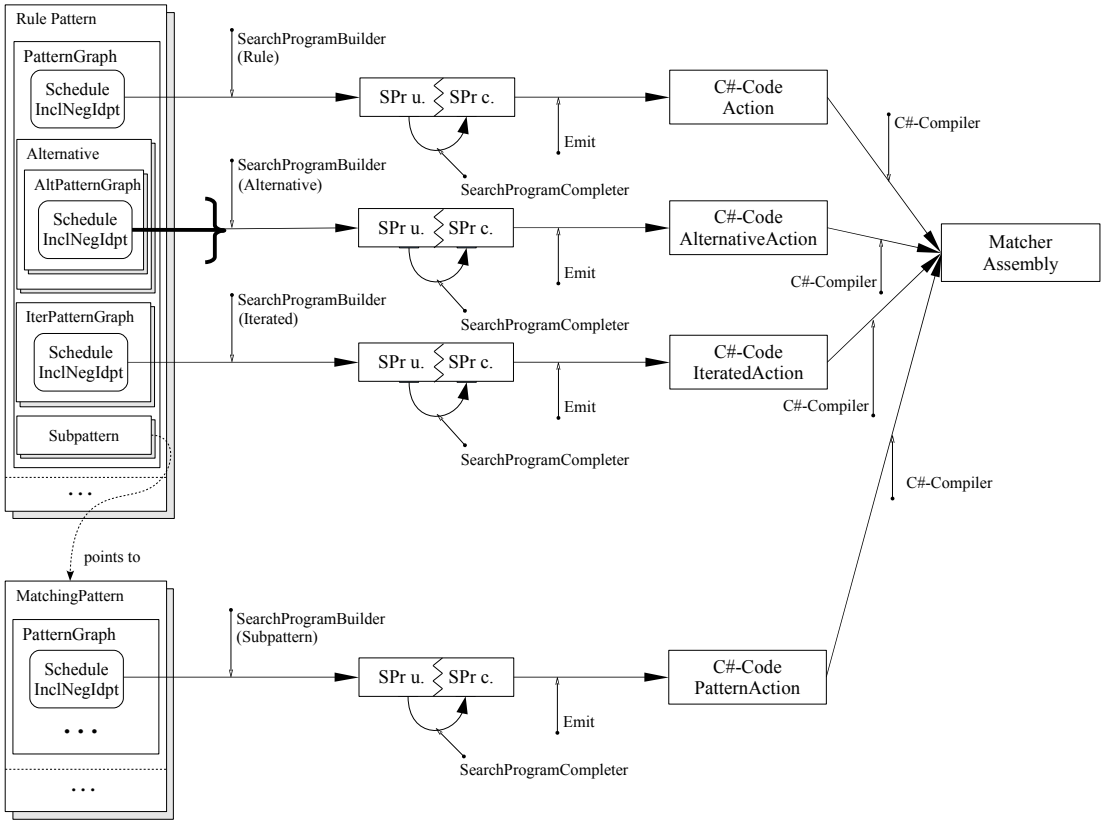


Figure 22.26: Backend Code Generation, step 3

Finally, in a third step code is generated by `GenerateMatcherSourceCode`, again in a recursive run over the nesting structure of the `PatternGraph`s, using the `ScheduleIncludingNegativesAndIndependents` stored in them.

For each `RulePattern` an `Action`-class is generated, and for each `MatchingPattern` a `SubpatternAction`-class is generated. Additionally for each alternative or iterated which is nested in a rule, test, or subpattern, an `AlternativeAction`-class or an `IteratedAction`-class are generated; the alternative matcher will contain code for all alternative cases. The real matching code is then generated into these classes.

The `SearchProgramBuilder` builds a `SearchProgram` tree data structure resembling the syntax tree of the code to generate out of the `ScheduleIncludingNegativesAndIndependents` in the `PatternGraph`s. In a further pass the `SearchProgram` is completed by the `SearchProgramCompleter`, determining the locations to continue at when a check fails, writing undo code for the effects which were applied from that point on to the current one. Finally the C# code gets generated by calling the `Emit` methods of the `SearchProgram`. If you want to extends this code you may be interested in the `Dump` methods which dump the `SearchProgram` in an easier readable form into text files.

The `SubpatternAction`-classes do not only contain the matcher code, but are at the same time the tasks of the $2 + n$ pushdown machine, which are pushed on the open tasks stack; they contain the subpattern parameters as member variables. The same holds for the `AlternativeAction`- and `IteratedAction`-classes, which do not hold parameters but entities from the nesting pattern they reference.

Nested and Subpattern Matching

The higher levels of code generation are in large parts independent from nested and subpattern matching and control of the $2+n$ pushdown machine. Only on the level of search programs it becomes visible, with an `InitializeSubpatternMatching`-search program operation at the begin of a search program and a `FinalizeSubpatternMatching`-search program operation at the end of a search program; but mainly with a call to `buildMatchComplete` when the end of the schedule is reached during search program builing. This corresponds to the innermost location in the search program, the location at which during execution the local pattern was just found; now the control code is inserted by `insertPushSubpatternTasks`, pushing the tasks for the subpatterns used from this pattern, as well as alternatives and iterateds nested in this pattern. To execute the open tasks a `MatchSubpatterns` operation is inserted into the search program. Afterwards `insertPopSubpatternTasks` inserts the operations for cleaning the task stack, `insertCheckForSubpatternsFound` the operations to handle success and failure, and `insertMatchObjectBuilding` the code for maintaining the result stack.

Further Functionality

The compiled graph rewrite sequences are handled by the `lgspSequenceChecker` and `lgspSequenceGenerator` (together with the sequence parser from the libGr). The `src/GrGen` subdirectory contains the `grgen.exe` compiler driver procedure. The `src/libGr` subdirectory contains the libGr, offering the base interfaces a user of GrGen.NET sees on the API level for the model, the actions, the pattern graphs and the host graph. The interfaces get implemented by code from the libGr search plan (lgsp) backend and by the generated code. The libGr further offers a generic, name string and object based interface to access the named entities of the generated code. In addition it offers the rewrite sequence parser which gets generated out of `SequenceParser.csc`, building the rewrite sequence AST from the classes in `Sequence.cs` further utilizing `SymbolTable.cs`. The rewrite sequence classes contain a method `ApplyImpl(IGraph graph)` which executes them. Finally the libGr offers several importers and exporters in the `src/libGr/IO` and `src/libGr/GRSImporter` subfolders.

When GrGen rules are matched in the graph and when the graph is changed by rule rewriting then events are fired. They allow to to display the matches and changes to the user in the debugger, to record changes to a file for later playback, or record changes to a transaction undo log for later rollback, or to execute user code if event handlers are registered to them, allowing users to build an event based graph rewriting mechanism on API level. The graph delegates fired are given in `IGraph.cs`. Graph events recording is implemented in `Recorder.cs` (replaying is normal `.grs` execution); graph transaction handling is implemented in `lgspTransactionManager.cs`, with a list of undo items which know how to undo the effect on the graph which created them, which are purged on commit or executed on rollback. Backtracking is implemented with nested transactions. If you are changing the graph programmatically not using GrGen rules you have to fire the events on your own in case you want to use any of the mechanisms (graphical debugging, record and replay, transactions and backtracking, event based programming) above.

The `src/GrShell` subdirectory contains the GrShell application, which builds upon the generic interface (as it must be capable of coping with arbitrary used defined models and actions at runtime) and the sequence interpretation facilities offered by the libGr. The command line parser of GrShell gets generated out of `GrShell.csc`, the shell implementation is given in `GrShellImpl.cs`. Graphical debugging is offered by the `Debugger.cs` together with the `YCompClient.cs`, which implements the protocol available for controling yComp, communicating with yComp over a tcp connection to localhost.

The `examples` subdirectory of `engine-net-2` contains a bunch of examples for using GrGen.NET with GrShell. The `examples-api` subdirectory contains several examples of how to use GrGen.NET from the API. In case you want to contribute and got further questions

don't hesitate to contact us (via email to `grgen` at the host given by `ipd.info.uni-karlsruhe.de`).

22.5 Performance Optimization

The most important point to understand when optimizing for speed is that the expensive task is the search carried out during pattern matching. The effort for rewriting, the dominant theme in graph rewriting literature, is negligible.

Use Types

A corollary from what you've seen especially in section 22.3 is: *Use types!*
The more fine grain a graph is typed, the better are the statistics regarding the splitting structures (V-Structures) and the number of elements of a certain type, yielding better search plans evading splitting structures and employing least-cost lookups, pruning non-matches earlier in the search. Even if only the initial static schedules are used which do not know about the splitting factors or the type weights, fine grain types cause a quicker search pruning with the type checks. You may improve the static schedules by explicitely giving search priorities. Using fine-grain types is easy in GrGen, for multiple inheritance on node and edge types (cf. 3.2.2) is supported.

NOTE (54)

Search planning is only carried out on request! You must analyze the graph and then re-generate the matchers at runtime, with `custom graph analyze` and `custom actions gen_searchplans` (issued on the command line or to the `Custom` methods of the graph and the actions objects); you may have a look at the effects of replanning with the `custom actions explain <actionname>` command. See subsection 17.3.2 for more on this.

Memorize, Don't Search

Often in a transformation you know the location you want to process from a previous step. In this case, store these locations in variables of node (or edge) type, and hand them in as parameters to the rules and return them out from the rules. In case of a statically not known number of locations as they appear e.g. in wavefront algorithm, you may store the locations in storages (cf. chapter 12), i.e. collection valued variables (which are iterated over in the sequences or passed to the rules as `ref` parameters). Search will then start from these parameters, instead of a looking up a value in the graph by type (unless search planning taking the statistics about the graph into account comes to the conclusion that a lookup on a very seldom type is still the better approach).

A sophisticated way of remembering facts about non-local properties is to computed them with data flow analyses (see section 16.6) and store them as attributes in the graph elements. This allows to replace searching for distant values, or global properties like reachability, by checking a local property, at the price of re-running an analysis every time the graph changes in an important way.

Speaking in database parlance, with storages you can build indices into the graph, indices which are more selective than the automatically built indices by graph element type; but in contrast to the automatically built ones, you must maintain them by hand.

The approach of remembering state instead of searching when needed has a clear caveat: the code becomes less readable and more brittle. As in normal programming, you must balance performance optimizations against maintainability.

Use Compiled Sequences

The compiled sequences given in the rules file are executed a good deal faster than the interpreted sequences given in the shell. So if you need speed, replace interpreted sequences by compiled ones. The price you pay is a loss of debuggability, a compiled sequence can only be executed as one big step.

Beware of Disconnected Patterns

Subpatterns are searched top-down (cf. 22.2), from the input parameters on. If the input arguments are disconnected in the pattern containing the subpattern, the containing pattern enumerates the cross product of the matches of the disconnected parts, which is then later on filtered in the subpattern called for the ones which are connected. This will likely wreak havoc on the search performance. It might be more efficient to just search from a start parameter a connected end location, and yield the found one out (cf. 7.3), or to search from a start parameter all connected end locations, collecting the found ones in a result set; and to check the ones found alongside connectedness in a second step.

The same holds for the nested patterns, just that parameter passing is implicit there, the elements from the containing pattern which are referenced in the nested pattern are passed in automatically as arguments. The holds especially for the `alternative` and `iterated` constructs which are matched with the pushdown machine (cf. 22.2), too, but also for the `negative` and `independent` constructs which are matched with nested local code embedded into the matcher code of their containing pattern.

Just as a side remark: the performance issue is raised by the local disconnected pattern and the caused multiplication of the part matches. This combinatorial explosions appears even within a single local pattern, only that it is more obvious there what is going on compared to the case that you disconnected a pattern later on by factoring out a common part into a subpattern. Luckily in this situation a performance hit is unlikely due to the subpattern inlining supported by GrGen; which in addition to connecting disconnected patterns removes the pushdown machine overhead.

But take into account that the *subpattern inlining* implemented in GrGen is limited to depth one. If a pattern is disconnected over two or more levels of subpattern usage (which might happen statically with one subpattern using another subpattern, and will for sure dynamically on a subpattern recursion path), it will hit performance. You may have a look at the output of the `explain` command (cf. 17.3.2) to see if the subpatterns are disconnected; this is indicated by multiple lookups in the containing pattern, and the fact that the starting points passed as preset parameters to the nested or subpatterns get connected only there with search commands.

Miscellaneous Things

- Visited flags are the most efficient way of marking elements, if a large number of the elements gets marked. Otherwise they are inefficient, cause a lot of elements need to be iterated, just to filter the visited ones out. Storages which give quick access to their contained elements are better then. Or even reflexive marker edges of a special type in the graph.

- Prefer all-bracketing (cf. 8.1) over iteration for rule application. This is only possible if the parts of the matches which are to be modified, e.g. retyped, are disjoint.

BIBLIOGRAPHY

[Ass00] Uwe Assmann. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):583–637, 2000.

[Bat05a] Gernot Veit Batz. Graphersetzung für eine Zwischendarstellung im Übersetzerbau. Master's thesis, Universität Karlsruhe, 2005.

[Bat05b] Veit Batz. Generierung von Graphersetzungen mit programmierbarem Suchalgorithmus. Studienarbeit, 2005.

[Bat06a] Gernot Veit Batz. An Optimization Technique for Subgraph Matching Strategies. Technical Report 2006-7, Universität Karlsruhe, Fakultät für Informatik, April 2006.

[Bat06b] Gernot Veit Batz. An optimization technique for subgraph matching strategies. Internal Report, 2006. "http://www.info.uni-karlsruhe.de/papers/TR_2006_7.pdf".

[BG09] Paul Bédaride and Claire Gardent. Semantic Normalisation: a Framework and an Experiment. In *Eighth International Conference on Computational Semantics*, 2009.

[BJ11a] Sebastian Buchwald and Edgar Jakumeit. A GrGen.NET Solution of the Compiler Case for the Transformation Tool Contest 2011. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/solutions/online/ttc2011_submission_32.pdf, 2011.

[BJ11b] Sebastian Buchwald and Edgar Jakumeit. A GrGen.NET solution of the Reengineering Case for the Transformation Tool Contest 2011. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/solutions/online/ttc2011_submission_31.pdf, 2011.

[BKG08] Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In *Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) Proceedings*, 2008. preliminary version, submitted to AGTIVE 2007.

[Buc08] Sebastian Buchwald. Erweiterung von GrGen.NET um DPO-Semantik und ungerichtete Kanten, 6 2008. Studienarbeit.

[CMR+99] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic concepts and double pushout approach. In *[Roz99]*, volume 1, pages 163–245. 1999.

[Dew84] A. K. Dewdney. A computer trap for the busy beaver, the hardest-working turing machine. *Scientic American*, 251(2):10–12, 16, 17, 8 1984.

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

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

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

[ER97] Engelfriet and Rozenberg. Node Replacement Graph Grammars. Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, 1997.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer. The AGG Approach: Language and Environment. In *[Roz99]*, volume 2, pages 551–603. 1999.

[Fuj07] Fujaba Developer Team. Fujaba-Homepage. http://www.fujaba.de/, 2007.

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2006.

[GDG08] Tom Gelhausen, Bugra Derre, and Rubino Geiß. Customizing GrGen.NET for Model Transformation. In *GraMoT*, pages 17–24, 2008.

[Gei08] Rubino Geiß. *Graphersetzung mit Anwendungen im Übersetzerbau*. PhD thesis, Universität Karlsruhe, Nov 2008.

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

[HE11] Tassilo Horn and Jürgen Ebert. The GReTL Transformation Language. In *ICMT 2011*, volume 6707 of *LNCS*, pages 183–198, 2011.

[HJG08] Berthold Hoffmann, Edgar Jakumeit, and Rubino Geiß. Graph Rewrite Rules with Structural Recursion. 2nd Intl. Workshop on Graph Computational Models (GCM 2008), 2008. "http://www.info.uni-karlsruhe.de/papers/GCM2008.pdf".

[HSESW05] Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. GXL: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 2005.

[Jak08] Edgar Jakumeit. Mit GrGen.NET zu den Sternen – Erweiterung der Regelsprache eines Graphersetzungswerkzeugs um rekursive Regeln mittels Sterngraphgrammatiken und Paargraphgrammatiken. Master's thesis, Universität Karlsruhe, jul 2008.

[JB11] Edgar Jakumeit and Sebastian Buchwald. A GrGen.NET solution of the Hello World Case for the Transformation Tool Contest 2011. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/solutions/online/ttc2011_submission_15.pdf, 2011.

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen.NET. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:263–271, 2010. 10.1007/s10009-010-0148-8.

[KBG+07] Moritz Kroll, Michael Beck, Rubino Geiß, Sebastian Hack, and Philipp Leiß. yComp. http://www.info.uni-karlsruhe.de/software.php/id=6, 2007.

[KG07] Moritz Kroll and Rubino Geiß. Developing Graph Transformations with GrGen.NET. In *Applications of Graph Transformation with Industrial releVancE - AGTIVE 2007*, 2007. preliminary version, submitted to AGTIVE 2007.

[KK07] Ole Kniemeyer and Winfried Kurth. The Modelling Platform GroIMP and the Programming Language XL. Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) Proceedings, 2007.

[Kro] Moritz Kroll. G#: GrGen.NET in C#. Master's thesis, Universität Karlsruhe (TH).

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

[Lin02] Götz Lindenmaier. libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Technical Report 2002-5, Universität Karlsruhe, Fakultät für Informatik, Sep 2002.

[LLMC05] Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. In *Electronic Notes in Theoretical Computer Science*, pages 65–75, 2005.

[LMS99] Litovsky, Métivier, and Sopena. Graph Relabelling Systems and Distributed Algorithms. Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3, 1999.

[MB00] H. Marxen and J. Buntrock. Old list of record TMs. http://www.drb.insel.de/~heiner/BB/index.html, 8 2000.

[Mic07] Microsoft. .NET. http://msdn2.microsoft.com/de-de/netframework/aa497336.aspx, 2007.

[MMJW91] Andrew B. Mickel, James F. Miner, Kathleen Jensen, and Niklaus Wirth. *Pascal user manual and report (4th ed.): ISO Pascal standard*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE '00 Proceedings of the 22nd international conference on Software engineering*, pages 742–745, 2000.

[Plu09] Detlef Plump. The Graph Programming Language GP. In *Proc. Algebraic Informatics (CAI 2009)*, pages 99–122, 2009.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. Applications of Graph Transformation with Industrial Relevance (AGTIVE '03) Proceedings, 2004.

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

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

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

[San95] Georg Sander. VCG Visualization of Compiler Graphs—User Documentation v.1.30. Technical report, Universität des Saarlandes, 1995.

[SGS09] Jochen Schimmel, Tom Gelhausen, and Christoph A. Schaefer. Gene Expression with General Purpose Graph Rewriting Systems. In *Proceedings of the 8th GT-VMT Workshop*, 2009.

[SWZ99] A. Schürr, A. Winter, and A. Zündorf. Progres: Language and environment. In G. Rozenberg, editor, *Handbook on Graph Grammars*, volume Applications, Vol. 2, pages 487–550. World Scientific, 1999.

[Sza05] Adam M. Szalkowski. Negative Anwendungsbedingungen für das suchprogrammbasierte Backend von GrGen, 2005. Studienarbeit, Universität Karlsruhe.

[Tea07] The Mono Team. Mono. http://www.mono-project.com/, 2007.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the intermediate representation firm. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik, Dec 1999.

[VB07] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming*, 68(3):214–234, 2007.

[VHV08] Gergely Varró, Ákos Horváth, and Dániel Varró. Recursive Graph Pattern Matching. In *Applications of Graph Transformations with Industrial Relevance*, pages 456–470. 2008.

[VSV05] G. Varró, A. Schürr, and D. Varró. Benchmarking for Graph Transformation. Technical report, Department of Computer Science and Information Theory, Budapest University of Technology and Economics, March 2005.

[VVF06] Gergely Varró, Dániel Varró, and Katalin Friedl. Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In G. Karsai and G. Taentzer, editors, *GraMot 2005, International Workshop on Graph and Model Transformations*, volume 152 of *ENTCS*, pages 191–205. Elsevier, 2006.

[Wei88] David Weininger. Smiles, a chemical language and information system. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

[WKR02] Winter, Kullbach, and Riediger. An Overview of the GXL Graph Exchange Language. Software Visualization - International Seminar Dagstuhl Castle, 2002.

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

INDEX

Keywords

Non-Terminals

General Index