Chapter 1. EASY Programming with Rascal

Leveraging the Extract-Analyze-SYnthesize Paradigm for Meta-Programming

Paul Klint, Jurgen Vinju, Tijs van der Storm

Fri Apr 24 13:44:21 CEST 2009

Table of Contents

Introduction 2
- EASY Programming 2
- Rascal 4
- Aim and Scope of this Book 4
- Rascal Concepts 4

A Motivating Example 7
- Preparations 7
- Questions 9

The Rascal Language 11
- Types and Values 11
- Declarations 15
- Expressions 20
- Statements 29

Larger Examples (TODO) 33
- Analyzing the Component Structure of an Application 34
- Analyzing the Structure of Java Systems 34
- Finding Uninitialized and Unused Variables in a Program 36
- Using Locations to Represent Program Fragments 37
- McCabe Cyclomatic Complexity 39
- Dataflow Analysis 39
- Program Slicing 46

Built-in Operators and Library Functions 48
- Benchmark 49
- Boolean 50
- Exception 50
- Graph 51
- Integer 51
- IO 52
- Labelled Graph 52
- List 53
- Location 55
- Map 55
- Node 56
- Real 57
- Relation 57
- RSF 59
- Resource (Eclipse only) 59
- Set 59
- String 62

Tuple 63
UnitTest 63
Value 64
ValueIO 64
View (Eclipse only) 64
Void 64
Extracting Facts from Source Code (TODO) 64
Workflow for Fact Extraction 65
Strategies for Fact Extraction 66
Fact Extraction using ASF+SDF 67
Concluding remarks 67
Table of Built-in Operators 67
Table of Built-in Functions 69
Bibliography 71

Warning

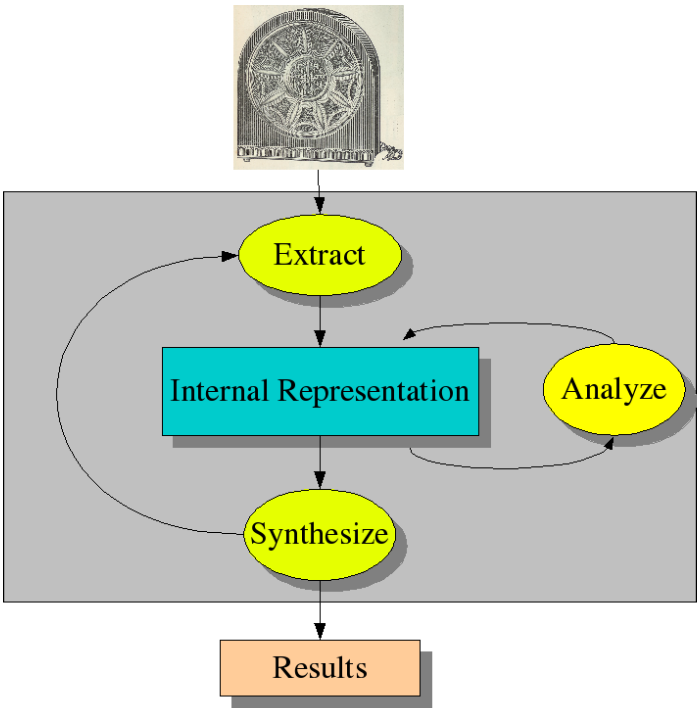
This document is in the process of being written.

Introduction

EASY Programming

Many programming problems follow a fixed pattern. Starting with some *object-of-interest*, first relevant information is extracted from it and stored in an internal representation. This internal representation is then analyzed and used to synthesize results. If the synthesis indicates this, these steps can be repeated over and over again. These steps are shown in Figure 1.1, “EASY: the Extract-Analyze-Synthesize Paradigm” [2] (yes some people do like old radios!).

Figure 1.1. EASY: the Extract-Analyze-Synthesize Paradigm



This is an abstract view on solving programming problems, but is it uncommon? No, so let's illustrate it with a few examples.

Warning

Find better images to illustrate this, e.g. in go a bicyle and a motor, and synthesize a motorbike.

Warning

These examples should come back in the larger examples later on.

Warning

Parsing and constaint solving are not yet well-represented here

Example 1.1. Finding security breaches

Alice is system administrator of a large online marketplace and she is looking for security breaches in her system. The object-of-interest are the system's log files. First relevant entries are extracted. This will include, for instance, messages from the SecureShell demon that reports failed login attempts. From each entry login name and originating IP address are extracted and put in a table (the internal representation in this example). These data are analysed by detecting duplicates and counting frequencies. Finally results are synthesized by listing the most frequently used login names and IP addresses.

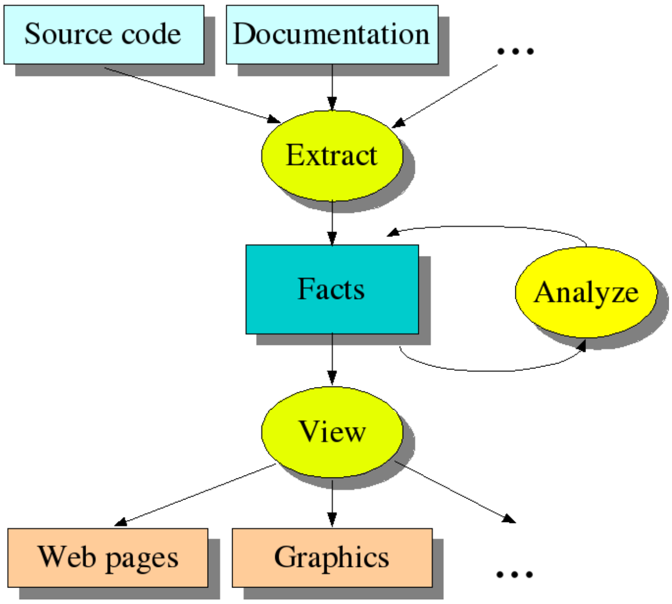
Example 1.2. A Forensic DSL compiler

Bernd is a senior software engineer working at the Berlin headquarters of a forensic investigation lab of the German government. His daily work is to find common patterns in files stored on digital media that have been confiscated during criminal investigations. Text, audio and video files are stored in zillions of different data formats and each data format requires its own analysis technique. For each new investigation ad hoc combinations of tools are used. This makes the process very labour-intensive and error-prone. Bernd convinces his manager that designing a new domain-specific language (DSL) for forensic investigations may relieve the pressure on their lab. After designing the DSL---let's call it DERRICK---he makes an EASY implementation for it. Given a DERRICK program for a specific case under investigation, he first extracts relevant information from it and analyzes it: which media formats are relevant? Which patterns to look for? How should search results be combined? Given this new information, Java code is synthesized that uses the various existing tools and combines their results.

Example 1.3. Renovation Financial Software

Charlotte is software engineer at a large financial institution in Paris and she is looking for options to connect an old and dusty software system to a web interface. She will need to analyze the sources of that system to understand how it can be changed to meet the new requirements. The objects-of-interest are in this case the source files, documentation, test scripts and any other available information. They have to be parsed in some way in order to extract relevant information, say the calls between various parts of the system. The call information can be represented as a binary relation between caller and callee (the internal representation in this example). This relation with 1-step calls is analyzed and further extended with 2-step calls, 3-step calls and so on. In this way call chains of arbitrary length become available. With this new information, we can synthesize results by determining the entry points of the software system, i.e. the points where calls from the outside world enter the system. Having completed this first cycle, Charlotte may be interested in which procedures can be called from the entry points and so on and so forth. Results will be typically represented as pictures that diplay the relationsships that were found. In the case of source code analysis, a variation of our workflow scheme is quite common. It is then called the extract-analyze-view paradigm and is shown in Figure 1.2, “The extract-analyze-view paradigm” [4].

Figure 1.2. The extract-analyze-view paradigm



Example 1.4. Finding Concurrency Errors

Daniel is concurrency researcher at one of the largests hardware manufacturers worldwide. He is working from an office in the Bay Area. Concurrency is the big issue for his company: it is becoming harder and harder to make CPUs faster, therefore more and of them are bundled on a single chip. Programming these multi-core chips is difficult and many programs that worked fine on a single CPU contain hard to detect concurrency errors due to subtle differences in the order of execution that results from executing the code on more than one CPU. Here is where Daniel enters the picture. He is working on tools for finding concurrency errors. First he extracts facts from the code that are relevant for concurrency problems and have to do with calls, threads, shared variables and locks. Next, he analyzes these facts and synthesizes an abstract model that captures the essentials of the concurrency behaviour of the program. Finally he runs a third-party verification tool with this model as input to do the actual verification.

Rascal

With these examples in mind, you have a pretty good picture how EASY applies in different use cases. All these cases involve a form of *meta-programming*: software programs are the objects-of-interest that are being analyzed and transformed. The Rascal language you are about to learn is designed for meta-programming following the EASY paradigm. It can be applied in domains ranging from compiler construction to constraint solving.

Since representation of data is central to the approach, Rascal provides a rich set of built-in data types. To support extraction and analysis, parsing and advanced pattern matching are provided. High-level control structures make analysis and synthesis of complex datastructures simple.

Aim and Scope of this Book

Rascal Concepts

Before explaining the Rascal language in more detail, it is good to have a general understanding of the concepts on which the language is built.

Values

Rascal is a value-oriented language. This means that values are immutable and are always freshly constructed from existing parts and that sharing and aliasing problems are completely avoided. The language does provide assignment to variables either as the result of an explicit assignment statement or as the result of a successful match.

Data structures

Rascal provides a rich set of datatypes. From booleans, infinite precision integers and reals to strings and source code locations. From lists, (optionally labelled) tuples and sets to (optionally labelled) maps and relations. From untyped tree structures to fully typed abstract datatypes. Syntax trees that are the result of parsing source files are represented as ADTs. There is a wealth of built-in operators and library functions available on the standard datatypes. A fragment of the abstract syntax for statements in a programming language would look as follows:

```
data STAT = asgStat(Id name, EXP exp)
          | ifStat(EXP exp, list[STAT] thenpart, list[STAT] elsepart)
          | whileStat(EXP exp, list[STAT] body)
          ;
```

Pattern Matching

Pattern matching is *the* mechanism for case distinction in Rascal. We provide string matching based on regular expressions, list (associative) and set (associative, commutative, identity) matching, matching of abstract datatypes, and matching of concrete syntax patterns. All these forms of matching can be used in a single pattern. Patterns may contain variables that are bound when the match is successful. Anonymous (don't care) positions are indicated by an underscore (_). The following abstract pattern matches the while statement defined above:

```
whileStat(EXP Exp, list[STAT] Stats)
```

Variables in a pattern are either explicitly declared in the pattern itself---as done in this example---or they may be declared in the context in which the pattern occurs. Patterns can also be used in an explicit match operator `:=` and can then be part of larger boolean expressions. Since a pattern match may have more than one solution, local backtracking over the alternatives of a match is provided.

Parsing

All source code analysis projects need to extract information directly from the source code. There are two main approaches to this:

- Use regular expression to extract useful, but somewhat superficial, information. This can be achieved using regular expression patterns.
- Use syntax analysis to extract the complete, nested, structure of the source code in the form of a syntax tree.

In Rascal, we reuse the Syntax Definition Formalism (SDF) and its tooling. SDF modules define grammars and these modules can be imported in a Rascal module. These grammar rules can be applied in writing concrete patterns to match parts of parsed source code. Here is an example of the same pattern we saw above, but now in concrete form:

```
while <EXP Exp> do <list[STAT] Stats> od
```

Generators

Generators enumerate the values in a given (finite) domain, be it the elements in a list, the substrings of a string, or all the nodes in a tree. Each value that is enumerated is first matched against a pattern before it can possibly contribute to the result of the generator. Examples are:

```
int x <- { 1, 3, 5, 7, 11 }
int x <- [ 1 .. 10 ]
asgStat(Id name, _) <- P
```

The first two generate the integer elements of a set of integers, respectively, a range of integers. The third generator traverses the complete program `P` (that is assumed to have a `PROGRAM` as value) and only yields statements that match the assignment pattern. Note the use of an anonymous variable at the `EXP` position in the pattern.

Comprehensions and Control Structures

Rascal generalizes comprehensions in various ways. Comprehensions exist for lists, sets and maps. A comprehension consists of an expression that determines the successive elements to be included in the result and a list of generators and boolean expressions. The generators enumerate values and the boolean expressions filter them. A standard example is

```
{ x * x | int x <- [1 .. 10], x % 3 == 0 }
```

which returns the set `{9, 36, 81}`, i.e., the squares of the integers in the range `[ 1 .. 10 ]` that are divisible by 3. A more intriguing example is

```
{name | asgStat(Id name, _) <- P}
```

which returns a list of all identifiers that occur on the lefthand side of assignment statements in program `P`. Combinations of generators and boolean expressions also drive the control structures. For instance,

```
for(asgStat(Id name, _) <- P, size(Id) > 10){
    println(Id);
}
```

prints all identifiers in assignment statements that consist of more than 10 characters.

Constraint solving

Switching and Visiting

The switch statement as known from C and Java is generalized: the subject value to switch on may be an arbitrary value and the cases are arbitrary patterns. When a match fails, all its side-effects are undone and when it succeeds the statements associated with that case are executed. Visiting the elements of a datastructure is one of the most common operations in our domain and we give it first class support by way of visit expressions that resemble the switch statement. A visit expression consists of an expression that may yield an arbitrarily complex subject value and a number of cases. All the elements of the subject are visited and when one of the cases matches the statements associated with that case are executed. These cases may:

- cause some side effect;
- execute an `insert` statement that replaces the current element;
- execute a `fail` statement that causes the match for the current case to fail (and undoing all side-effects due to the successfull match itself and the execution of the statements sofar).

The value of a visit expression is the original subject value with all replacements made as dictated by matching cases. The traversal order in a visit expressions can be explicitly defined by the programmer.

Functions and Rewrite Rules

Functions are explicitly declared and are fully typed. Here is an example of a function that counts the number of assignment statements in a program:

```
int countAssignments(PROGRAM P){
    int n = 0;
    visit (P){
    case asgStat(Id name, _):
         n += 1;
    }
    return n;
}
```

Rewrite rules are the only implicit control mechanism in the language and are used to maintain invariants during computations.

Typechecking

Rascal has a statically checked type system that prevents type errors and uninitialized variables at runtime. There are no runtime type casts as in Java and there are therefore less opportunities for run-time errors. The language provides higher-order, parametric, polymorphism. A type aliasing mechanism allows documenting specific uses of a type. Builtin operators are heavily overloaded: `1 + [2,3]` , `[1, 2] + 3`, and `[1, 2] + [3]` all give the same result `[1,2,3]`, but `1 + ["1"]` leads to a type error.

Execution

Following the what-you-see-is-what-you-get paradigm, control flow is completely explicit. Boolean expression determine choices that drive the control structures. Rewrite rules form the only exception to the explicit control flow principle. Only local backtracking is provided (no suprise) in the context of boolean expressions and pattern matching; side effects are undone in case of backtracking.

A Motivating Example

Suppose a mystery box ends up on your desk. When you open it, it contains a huge software system with several questions attached to it:

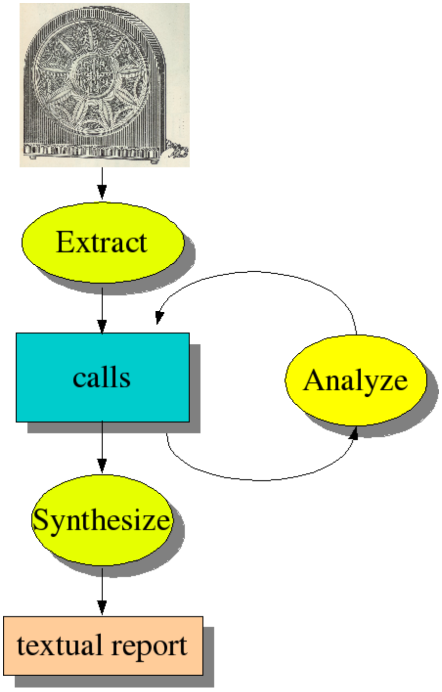
- How many procedure calls occur in this system?
- How many procedures contains it?
- What are the entry points for this system, i.e., procedures that call others but are not called themselves?
- What are the leaves of this application, i.e., procedures that are called but do not make any calls themselves?
- Which procedures call each other indirectly?
- Which procedures are called directly or indirectly from each entry point?
- Which procedures are called from all entry points?

There are now two possibilities. Either you have this superb programming environment or tool suite that can immediately answer all these questions for you or you can use Rascal.

Preparations

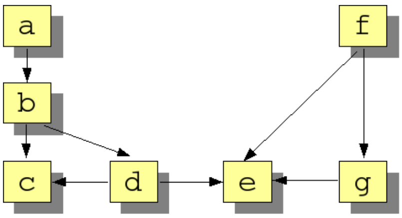
To illustrate this process consider the workflow in Figure 1.3, “Workflow for analyzing mystery box” [8]. First we have to extract the calls from the source code. Rascal is very at this, but to simplify this examples we assume that this call graph has already been extracted. Also keep in mind that a real call graph of a real application will contain thousands and thousands of calls. Drawing it in the way we do later on in Figure 1.4, “Graphical representation of the `calls` relation” [8] makes no sense since we get a uniformly black picture due to all the call dependencies. After the extraction phase, we try to understand the extracted facts by writing queries to explore their properties. For instance, we may want to know *how many calls* there are, or *how many procedures*. We may also want to enrich these facts, for instance, by computing who calls who in more than one step. Finally, we produce a simple textual report giving answers to the questions we are interested in.

Figure 1.3. Workflow for analyzing mystery box



Now consider the call graph shown in Figure 1.4, “Graphical representation of the `calls` relation” [8]. This section is intended to give you a first impression what can be done with Rascal. Please return to this example when you have digested the detailed description of Rascal in the section called “*The Rascal Language*” [11] and the section called “*Built-in Operators and Library Functions*” [48].

Figure 1.4. Graphical representation of the `calls` relation



Rascal supports basic data types like integers and strings which are sufficient to formulate and answer the questions at hand. However, we can gain readability by introducing separately named types for the items we are describing. First, we introduce therefore a new type `proc` (an alias for strings) to denote procedures:

```
rascal> alias proc = str
done
```

Suppose that the following facts have been extracted from the source code and are represented by the relation `Calls`:

```
rascal> rel[proc , proc] Calls =
```

```
  { <"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">, <"d","e">,
    <"f", "e">, <"f", "g">, <"g", "e">
  }
```

This concludes the preparatory steps and now we move on to answer the questions.

Questions

How many procedure calls occur in this system?

To determine the numbers of calls, we simply determine the number of tuples in the `Calls` relation, as follows. First, we need the Relation library (described in the section called “*Relation*”[57]) so we import it:

```
rascal> import Relation;
done.
```

next we describe a new variable and calculate the number of tuples:

```
rascal> int nCalls = size(Calls)
```

The library function `size` determines the number of elements in a set or relation and is explained in the section called “*Relation*” [57]. In this example, `nCalls` will get the value `8`.

How many procedures contains it?

We get the number of procedures by determining which names occur in the tuples in the relation `Calls` and then determining the number of names:

```
rascal> set[proc] procs = carrier(Calls);
{"a", "b", "c", "d", "e", "f", "g"}
> int nprocs = size(procs);
7
```

The built-in function `carrier` determines all the values that occur in the tuples of a relation. In this case, `procs` will get the value `{"a", "b", "c", "d", "e", "f", "g"}` and `nprocs` will thus get value `7`. A more concise way of expressing this would be to combine both steps:

```
rascal> int nprocs = size(carrier(Calls));
7
```

What are the entry points for this system?

The next step in the analysis is to determine which *entry points* this application has, i.e., procedures which call others but are not called themselves. Entry points are useful since they define the external interface of a system and may also be used as guidance to split a system in parts. The `top` of a relation contains those left-hand sides of tuples in a relation that do not occur in any right-hand side. When a relation is viewed as a graph, its top corresponds to the root nodes of that graph. Similarly, the `bottom` of a relation corresponds to the leaf nodes of the graph. See the section called “*Graph*”[51] for more details. Using this knowledge, the entry points can be computed by determining the top of the `Calls` relation:

```
rascal> import Graph;
done.
rascal> set[proc] entryPoints = top(Calls);
{"a", "f"}
```

In this case, `entryPoints` is equal to `{"a", "f"}`. In other words, procedures `"a"` and `"f"` are the entry points of this application.

What are the leaves of this application?

In a similar spirit, we can determine the *leaves* of this application, i.e., procedures that are being called but do not make any calls themselves:

```
rascal> set[proc] bottomCalls = bottom(Calls);
{"c", "e"}
```

In this case, `bottomCalls` is equal to `{"c", "e"}`.

Which procedures call each other indirectly?

We can also determine the *indirect calls* between procedures, by taking the transitive closure of the `Calls` relation:

```
rascal> rel[proc, proc] closureCalls = Calls+;
{<"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">, <"d","e">,
 <"f", "e">, <"f", "g">, <"g", "e">, <"a", "c">, <"a", "d">,
 <"b", "e">, <"a", "e">
}
```

Which procedures are called directly or indirectly from each entry point?

We know now the entry points for this application (`"a"` and `"f"`) and the indirect call relations. Combining this information, we can determine which procedures are called from each entry point. This is done by indexing `closureCalls` with appropriate procedure name. The index operator yields all right-hand sides of tuples that have a given value as left-hand side. This gives the following:

```
rascal> set[proc] calledFromA = closureCalls["a"];
{"b", "c", "d", "e"}
```

and

```
rascal> set[proc] calledFromF = closureCalls["f"];
{"e", "g"}
```

Which procedures are called from all entry points?

Finally, we can determine which procedures are called from both entry points by taking the intersection of the two sets `calledFromA` and `calledFromF:`

```
rascal> set[proc] commonProcs = calledFromA & calledFromF;
{"e"}
```

In other words, the procedures called from both entry points are mostly disjoint except for the common procedure `"e"`.

Wrap-up

These findings can be verified by inspecting a graph view of the calls relation as shown in Figure 1.4, “Graphical representation of the `calls` relation”[8]. Such a visual inspection does *not* scale very well to large graphs and this makes the above form of analysis particularly suited for studying large systems.

Warning

Add a screen dump here of the Eclipse browser.

The Rascal Language

A Rascal program consists of one or more modules. Each module may import other modules and declare data types, variables, functions or rewrite rules. We now describe the basic ingredients of Rascal in more detail:

- Types and values, see the section called "*Types and Values*" [11].
- Declarations, see the section called "*Declarations*" [15].
- Expressions, see the section called "*Expressions*" [20].
- Statements, see the section called "*Statements*" [29].

Types and Values

Elementary Types and Values

Void. Void stands for *nothing* and is represented by the type `void`. It is a type without any values.

Value. Value stands for all possible Rascal values and is represented by the type `value`. This type is a container for all other types and does not have any values itself.

Boolean. The Booleans are represented by the type `bool` and have two values: `true` and `false`.

Integer. The integer values are represented by the type `int` and are written as usual, e.g., `0`, `1`, or `123`. They can be arbitrarily large.

Real. The real values are represented by the type `real` and are written as usual, e.g., `1.5`, or `3.14e-123.` They can have arbitrary size and precision.

String. The string values are represented by the type `str` and consist of character sequences surrounded by double quotes. e.g., `"a"` or `"a\ long\ string"`.

String literals permit *interpolation* of variable values: when `<X>` occurs inside a string literal, the value of the variable *X* is converted to string that replaces <*X*>. As a consequence, the character `<` has to be escaped as`\<` in string literals.

Location. Location values are represented by the type `loc` and serve as text coordinates in a specific source file. It is very handy to associate a source code location which extracted facts. Locations should *always* be generated automatically but for the curious here is an example how they look like: `loc(file:/home/paulk/pico.trm?offset=0&length=1&begin=2,3&end=4,5)`

Source locations have the following syntax:

```
loc(Url?offset=O&length=L&begin=BL,BC&end=EL,EC)
```

where:

- *`Url`* is an arbitrary URL.
- *`O`* and *`L`* are integer expressions giving the offset of this location to the begin of file, respectively , its length.
- *`BL`* and *`BC`* are integers expressions giving the begin line and begin column.
- *`EL`* and *`EC`* are integers expressions giving the end line and end column.

This element of a location value can be accessed and modified using the standard mechanism of field selection and field assignment. The corresponding field names are:

- `url`

- `offset`
- `length`
- `beginLine`, `beginColumn`
- `endLine`, `endColumn`.

List, Set, Map, Tuple, and Relation

List. A list is an ordered sequence of values and has the following properties:

- All elements have the same type.
- The order of the elements matters.
- The list may contain the same element more than once.

Lists are represented by the type `list[`*T*`]`, where *T* is an arbitrary type. Examples are `list[int]`, `list[tuple[int,int]]` and `list[list[str]]`. Lists are denoted by a list of elements, separated by comma's and enclosed in bracket as in $[E_1, E_2, \ldots, E_n]$, where the E_n (1 <= i<= n) are expressions that yield the desired element type. For example,

- `[1, 2, 3]` is of type `list[int]`,
- `{<1,10>, <2,20>, <3,30>}` is of type `set[tuple[int,int]]`,
- `[1, "b", 3]` is of type `list[value]`,
- `[<"a",10>, <"b",20>, <"c",30>]` is of type `list[tuple[str,int]]`, and
- `[["a", "b"], ["c", "d", "e"]]` is of type `list[list[str]]`.

Note

`[1, 2, 3]` and `[3, 2, 1]` are different lists.

Note

`[1, 2, 3]` and `[1, 2, 3, 1]` are also different lists.

When variables of type list occur inside a list, their elements are automatically spliced into the surrounding list. This can be prevented by surrounding them with extra [and] brackets.

Range. For lists of integers, a special shorthand exists to describe ranges of integers:

- `[`*F* `..` *L*`]` ranges from first element *F* to (and including) last element *L* with increments of 1.
- `[`*F*`,`*S*`,..` *E*`]`, ranges from first element *F*, second element S to (and including) last element *L* with increments of S - *F*.

Set. A set is an unordered sequence of values and has the following properties:

- All elements have the same type.
- The order of the elements does not matter.
- A set contains an element only once, no matter how many times it has been added to it.

Sets are represented by the type `set[`*T*`]`, where *T* is an arbitrary type. Examples are `set[int]`, `set[tuple[int,int]]` and `set[set[str]]`. Sets are denoted by a list of elements, separated by comma's and enclosed in braces as in $\{E_1, E_2, \ldots, E_n\}$, where the E_n (1 <= i<= n) are expressions that yield the desired element type. For example,

- `{1, 2, 3}` is of type `set[int]`,
- `{<1,10>, <2,20>, <3,30>}` is of type `set[tuple[int,int]]`,
- `{1, "b", 3}` is of type `set[value]`,
- `{<"a",10>, <"b",20>, <"c",30>}` is of type `set[tuple[str,int]]`, and
- `{{"a", "b"}, {"c", "d", "e"}}` is of type `set[set[str]]`.

> ## Note
>
> `{1, 2, 3}` and `{3, 2, 1}` are identical sets.

> ## Note
>
> `{1, 2, 3]` and `{1, 2, 3, 1}` are also identical sets.

In a similar fashion as with lists, sets variables are automatically spliced into a surrounding set. This can be prevented by surrounding them with extra { and } brackets.

Map. A map is a set of key : value pairs and has the following properties:

- Key and value (may) have different types.
- A key can only occur once.

Maps are represented by the type map[T_1, T_2], where T_1 and T_2 are arbitrary types. Examples are `map[int,int]`, and `map[str,int]`. Sets are denoted by a list of pairs, separated by comma's and enclosed in parentheses as in (K_1: V_1, ..., K_n: V_n}), where the K_n (1 <= i<= n) are expressions that yield the keys of the map and V_n (1 <= i<= n) are expressions that yield the values for each key. Map resemble functions rather than relations in the sense that only a single value can be associated with each key. For example,

- `("pear" : 1, "apple" : 3, "banana" : 0)` is of type `map[str,int]`.

Tuple. A tuple is a sequence of elements with the following properties:

- Each element in a tuple (may) have a different type.
- Each element of a tuple may have a label that can be used to select that element of the tuple.

Tuples are represented by the type tuple[$T_1 L_1$, $T_2 L_2$, ..., T_n L_n], where T_1, T_2, ..., T_n are arbitrary types and L_1, L_2, ..., L_n are optional labels. An example of a tuple type is `tuple[str name, int freq>]`. Examples are:

- `<1, 2>` is of type `tuple[int, int>]`,
- `<1, 2, 3>` is of type `tuple[int, int, int>]`,
- `<"a", 3>` is of type `tuple[str name, int freq].`

Relation. A relation is a set of elements with the following property:

- All elements have the same tuple type.

Relations are thus nothing more than sets of tuples, but since they are used so often we provide a shorthand notation for them. Relations are represented by the type rel[$T_1 L_1$, $T_2 L_2$, ..., T_n L_n], where T_1, T_2, ..., T_n are arbitrary types and L_1, L_2, ..., L_n are optional labels. It is a shorthand for set[tuple[$T_1 L_1$, $T_2 L_2$, ..., T_n L_n]]. Examples are `rel[int,str]` and `rel[int,set[str]]`. An n-ary relations with m tuples is denoted by {<E_{11}, E_{12}, ..., E_{1n}>,<E_{21}, E_{22}, ..., E_{2n}>, ...,<E_{m1}, E_{m2}, ..., E_{mn}>}, where the E_{ij} are expressions that yield the desired element type. For example, `{<1, "a">, <2, "b">, <3,"c">}` is of type `rel[int, str]`. Examples are:

- `{<1,10>, <2,20>, <3,30>}` is of type `rel[int,int]` (yes indeed, you saw this same example before and then we gave `set[tuple[int,int]]` as its type; remember that these types are interchangeable.),
- `{<"a",10>, <"b",20>, <"c",30>}` is of type `rel[str,int]`, and
- `{{"a", 1, "b"}, {"c", 2, "d"}}` is of type `rel[str,int,str]`.

Alias Type

Everything can be expressed using the elementary types and values that are provided by Rascal. However, for the purpose of documentation and readability it is sometimes better to use a descriptive name as type indication, rather than an elementary type. The alias declaration

```
alias Name  = Type;
```

states that *`Name`* can be used everywhere instead of the already defined type *`Type`*. For instance,

```
alias ModuleId = str;
alias Frequency = int;
```

introduces two new type names `ModuleId` and `Frequency`, both an alias for the type `str`. The use of type aliases is a good way to docuement your intentions. Another example is an alias definition for a graph containing integer nodes:

```
alias Graph = rel[int,int];
```

An alias definition may be parameterized. So we can generalize graphs as follows:

```
alias Graph[&Node] = rel[&Node, &Node];
```

Of course, the type variables that are used in the type in the left part should occur as parameters in the right part of the definition and vice versa.

Abstract Data Type

In ordinary programming languages record types or classes exist to introduce a new type name for a collection of related, named, values and to provide access to the elements of such a collection through their name. In Rascal, data declarations provide this facility. The type declaration

```
data Name = Pat1 | Pat1 | ...
```

introduces a new datatype *`Name`* and Pat_1, Pat_2, ... are prefix patterns describing the variants of the datatype. For instance,

```
data Bool = btrue | bfalse | band(Bool L, Bool R)  | bor(Bool L, Bool R);
```

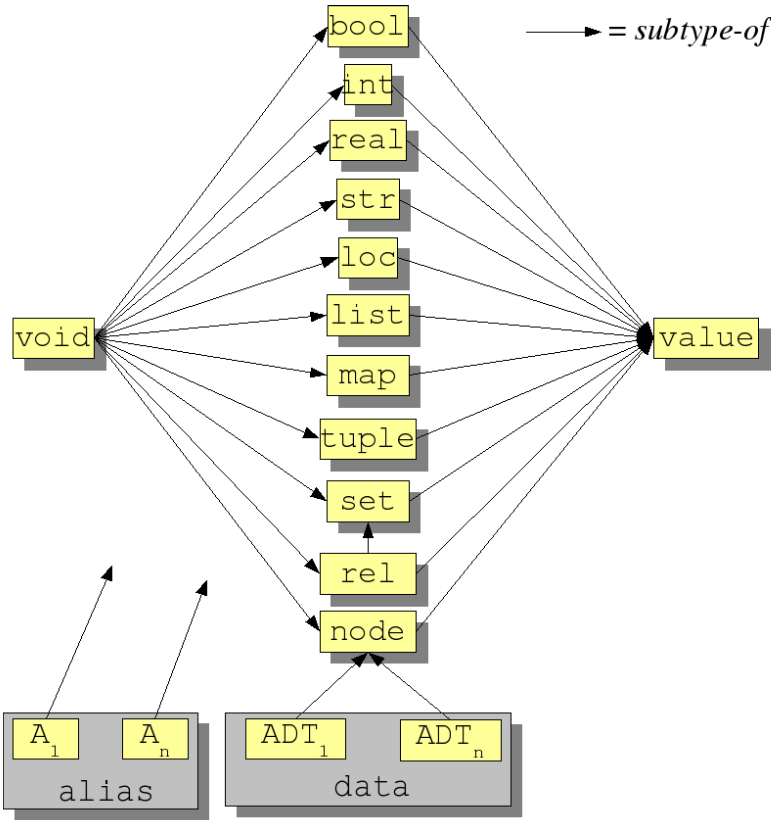
defines the datatype `Bool` that contains various constants and constructor functions.

Parameterized types

Typing

Rascal is based on static typing, this means that as many errors and inconsistencies as possible are spotted before the program is executed. The types introduced earlier are ordered in a so-called *type lattice* shown in the section called “*Typing*”[14]. The arrows describe a *subtype-of* relation between types. The type void is the *smallests* type and is included in all other types and the type value is the *largests* type that includes all other types. We also see that `rel` is a subtype of `set` and that each ADT is a subtype of `node`. Finally, each alias is a subtype of one specific type.

Figure 1.5. The Rascal Type Lattice



Declarations

Rascal Program

A Rascal program consists of a number of modules that may import each other.

Module

A module declaration has the following form:

```
module Name
Imports;
Declaration1;
...
Declarationn;
```

and consists of a `Name` and zero or more imports of other modules (the section called "*Import*" [16]) and declarations for

- Abstract data type, see the section called "*Abstract Data Type*" [14].
- Alias, see the section called "*Alias Type*" [14].
- Variable, see the section called "*Variable*" [16].
- Function, see the section called "*Function*" [17].
- Rewrite rule, see the section called "*Rewrite Rule*" [18].
- Node annotation, see the section called "*Node annotation*" [19].
- Declaration tag, see the section called "*Declaration Tag*" [19].

The module name *Name* will be used when the current module is imported in another module.

The entities that are *visible* inside a module are

- The private or public entities declared in the module itself.
- The public entities declared in any imported module.

The only entities that are visible outside the module, are the public entities declared in the module itself.

Caution

Describe qualified names.

Import

An import has the form:

```
import Name;
```

and has as effect that all public entities declared in module *Name* are made available to the importing module.

Variable

A variable declaration has the form

```
Type Var = Exp
```

where *T*ype is a type, *V*ar is a variable name, and *E*xp is an expression that should have type *Type*. The effect is that the value of expression *Exp* is assigned to *V*ar and can be used later on as *Var*'s value. The folowing rules apply:

- Double declarations in the same scope are not allowed.
- The type of *Exp* should be compatible with *Type*.

As a convenience, also declarations without an initialization expression are permitted and have the form

```
Type Var
```

and only introduce the variable *Var*. When a variable is declared, it has as scope the nearest enclosing block (see the section called “*Block*” [31]).

Rascal provides *convenience typing*, which allows the implicit declaration for variables that are used locally in functions. The following rules apply:

- An implicitly declared variable is declared at the level of the function body.
- An implicitly declared variable gets as type the type of the first value that is assignment to it.
- All uses of an implicitly declared variable must be type compatible.

Examples.

```
rascal>int max = 100;
int: 100
rascal>rel[str,int] day = {<"mon", 1>, <"tue", 2>, <"wed",3>,
                           <"thu", 4>, <"fri", 5>, <"sat",6>, <"sun",7>}
rel[str,int]: {<"mon", 1>, <"tue", 2>, <"wed",3>,
               <"thu", 4>, <"fri", 5>, <"sat",6>, <"sun",7>}
```

Function

A function declaration has the form

```
Type Name(Type1 Var1, ..., Typen Varn) Statement
```

Here *Type* is the result type of the function and this should be equal to the type of the result of Statement (using the return statement, see the section called "*Return*" [31]). Each $Type_i$ Var_i represents a typed formal parameter of the function. The formal parameters may be used in *Statement* and get their value when function *Name* is invoked. Example:

```
rascal> rel[int, int] invert(rel[int,int] R)
        return {<Y, X> | <int X, int Y> <- R }
done.
rascal> invert({<1,10>, <2,20>});
{<10,1>, <20,2>}
```

Variable argument lists. A function may have a variable list of arguments, this has the form:

```
Type Name(ordinary parameters, Type Var...) Statement
```

The last parameter of a function may be followed by `...` and this has as effect that all remaining actual parameters that occur in a call to this function are collected as list value of the last formal parameter. Inside the function body, the type of this parameter will therefore be `list[Type]`.

Exceptions. The exceptions that can be thrown by a function can be (optionally) declared as follows:

```
Type Name(Type1 Var1, ..., Typen Varn) throws Exception1, Exception2, ...
```

See for details.

Caution

Add ref.

Parameterized types in function declaration. The types that occur in function declarations may also contain *type variables* that are written as & followed by an identifier. In this way functions can be defined for arbitrary types. In the following example, we declare an inversion function that is applicable to any binary relation. :

```
rascal> rel[&T2, &T1] invert2(rel[&T1,&T2] R)
        return {<Y, X> | <&T1 X, &T2 Y> <- R };
done.

rascal>invert2({<1,10>, <2,20>});
{<10,1>, <20,2>}

rascal>invert2({<"mon", 1>, <"tue", 2>});
{<1, "mon">, <2, "tue">}
```

Here we declare a function that can be used to swap the elements of pairs of arbitrary types:

```
rascal> <&T2, &T1> swap(&T1 A, &T2 B) return <B, A>;
done.

rascal> swap(<1, 2>);
<2,1>

rascal> swap(<"wed", 3>);
<3, "wed">
```

Rewrite Rule

Functions are the workhorses of Rascal. They can have any value as parameter or result and are explicitly called by the user. Also, functions are declared inside modules and their visibility can be controlled.

Rewrite rules, on the other hand, operate only on nodes and abstract datatypes, they are implicitly applied when a new value (we refer to this as the *subject value*) is constructed and their scope is the whole Rascal program. Rewrite rules are applied to the subject value in a bottom-up fashion. As a result, the subject value may be changed. This process is repeated as long as there are rules that can be applied to the current subject value. When done, the result of rewriting the original subject value is used instead of that original value.

Caution

There is some overlap with the description of switch and visit here.

Rules have the general form:

```
rule Name Rule
```

where `Name` is the name of the rule and `Rule` is the actual rewrite rule itself. A rule can have one of the following forms:

- `Pat => Exp`

  When a subtree of the subject matches `Pat`, it is replaced by the value of `Exp`.

- `Pat => Exp when` Exp_1, Exp_2, ...

  When a subtree of the subject matches `Pat`, and all expressions Exp_i are true, the subtree is replaced by the value of `Exp`.

- `Pat : Statement`

  This is the most general case. When a subtree of the subject matches `Pat`, the `Statement` is executed. The execution of `Statement` should lead to one of the following:

  - Execution of an `insert` statement (see the section called "*Insert*" [31]) that replaces the current subtree.
  - Execution of a `fail` statement (see the section called "*Fail*" [31]): all side effects of `Statement` are undone and the next rule is tried.

Here is an example for user-defined type Booleans:

```
rascal>
data Bool = btrue;
data Bool = bfalse;
data Bool = band(Bool left, Bool right);
data Bool = bor(Bool left, Bool right);

rule a1 band(btrue, Bool B)    => B;
rule a2 band(bfalse, Bool B)   => bfalse;
done.
rascal>band(band(btrue,btrue),band(btrue, bfalse));
bfalse
```

During execution of rules the following applies:

- Rules are applied non-deterministically, and in any order of matching.

- The right hand side of rules can contain fail statements, which cause backtracking over the alternative matches or alternative rules for a certain constructor.
- When the right-hand side is a statement, an `insert` statement determines the value of the actual replacement.

The rule keyword may be immediately followed by a type constraint to limit its applicability.

Caution

Is this implemented?

Node annotation

An annotation may be associated with any node value. Annotations are intended to attach application data to values, like adding position information or control flow information to source code or adding visualization information to a relation. An annotation has a name and the type of its value is explicitly declared. Any value of any named type can be annotated and the type of these annotations can be declared precisely.

For instance, we can add to certain syntactic constructs of programs (e.g., `EXPRESSION`) an annotation with name `posinfo` that contains location information:

```
anno loc EXPRESSION @ posinfo;
```

or location information could be added for all syntax trees:

```
anno loc node @ posinfo;
```

We can add to the graph datatype introduced earlier, the annotation with name `LayoutStrategy` that defines which graph layout algorithm to apply to a particular graph, e.g.,

```
data LayoutStrategy = "dot" | "tree" | "force" |
                      "hierarchy" | "fisheye";

anno LayoutStrategy Graph @ strategy;
```

The folllowing constructs are provided for handling annotations:

- *`Val @ Anno`*: get the value of annotation *`Anno`* of value *`Val`* (may be undefined!).
- *`Val1`*`[@`*`Anno`*` = `*`Val2`*`]`: set the value of annotation *`Anno`* of the value *`Val1`* to *`Val2`*.
- *`Var`*` @ `*`Anno`*` = `*`Val`*: set the value of annotation *`Anno`* of the value of variable *`Var`* to *`Val`*.

Declaration Tag

Warning

Tags are not yet implemented.

Warning

The syntax of tags has to be aligned with the syntax of annotations. This is done in the examples below but not yet in the synyax.

Tags are intended to add metadata to a Rascal program and allow to influence the execution of the Rascal program, for instance, by adding memoization hints or database mappings for relations.

All declarations in a Rascal program may contain (in fixed positions depending on the declaration type) one or more declaration tags (`tag`). A tag is defined by declaring its name, the declaration type to which it can be attached, and the name and type of the annotation. The declaration type `all`, makes the declaration tag applicable for all possible declaration types. All declaration tags have the generic format `@Name{ ... }`, with arbitrary text between the brackets that is further constrained by the declared type. Here is an example of a license tag:

```
tag str license on module;
```

This will allow to write things like:

```
module Booleans
@license{This module is distributed under the GPL}
...
```

Other examples of declaration tags are:

```
tag str todo on all             %% a todo note for all declaration types
tag void deprecated on function %% marks a deprecated function
tag int memo on function        %% bounded memoization of
                                %% function calls
tag str doc on all              %% documentation string
tag str primitive on function   %% a primitive, built-in, function
```

Here is an example of a documentation string as used in the Rascal standard library:

```
public &T max(set[&T] R)
  @doc{Maximum of a set: max}
{
  &T result = arb(R);
  for(&T E : R){
    result = max(result, E);
  }
  return result;
}
```

Expressions

The Expression is the basic unit of evaluation and may consist of:

- An *elementary literal value*, e.g. constants of type `bool`, `int`, `real`, `str` or `loc`. Elementary literals evaluate to themselves.
- A *structured literal value* for `list`, `set`, `map`, `tuple`, or `rel`. The elements are first evaluated before the structured value is built.
- A *variable*, evaluates to its current value.
- A *function call*. First the arguments are evaluated and the corresponding function is called. The value returned by the function is used as value of the function call.
- A *constructor*. First the arguments are evaluated and then a data value is constructed for the corresponding type. This data value is used as value of the constructor.
- An *operator expression*. The operator is applied to the arguments; the evaluation order of the arguments depends on the operator. The result returned by the operator is used as value of the operator expression.

We explain the various operator expressions in more detail. Non-Boolean and Boolean operator expressions behave somewhat differently and therefore we describe separately.

Non-Boolean Operator Expressions

The non-Boolean operators are summarized in Table 1.1, “Non-Boolean Operators”[21]. All operators are higly overloaded and we refer to the section called “*Built-in Operators and Library*

Functions” [48] for a description of their meaning for each specific type. The following rules apply:

- All *name* arguments stand for themselves and are not evaulated.
- For all operators, except, IfThenElse, the argument expressions are first evaluated before the operator is applied.
- The operators in Table 1.1, “Non-Boolean Operators”[21] are listed from high precedence to low precedence. In other words, operators listed higher in the table bind stronger.

Table 1.1. Non-Boolean Operators

Operator	Name	Description
exp_1 [*Name* = Exp_2]	Field assignment	Exp_1should evaluate to a datatype with a field *Name*; assign value Exp_2 to that field
Exp . *Name*	Field selection	*Exp* should evaluate to a datatype with field *Name*; return the value of that field
Exp < *field*, ... >	Field projection	Exp should evaluate to a tuple or relation, and field should be a field name or an integer constant. A new tuple or relation is retruned that only contains the listed fields.
Exp_1 [Exp_2]	Subscription	The value of Exp_2 is used as index in Exp_1's value. On string, list, tuple return the element with given index; for map return the value associated with Exp_2 's value
- *Exp*	Negation	Negative of *Exp*'s integer or real value
Exp +	Transitive closure	Transitive closure on relation
Exp *	Reflexive transitive closure	Reflexive transitive closure on relation
Exp @ *Name*	Attribute selection	value of attribute *Name* of *Exp*'s value
Exp_1 [@ *Name* = Exp_2]	Attribute replacement	Assign value of Exp_2 to attribute *Name* of Exp_1's value
Exp_1 o Exp_2	Composition	Exp_1 and Exp_2 should evaluate to a relation; return their composition
Exp_1 / Exp_2	Division	Divide two integers and reals
Exp_1 % Exp_2	Modulo	Modulo on integer
Exp_1 * Exp_2	Multiplication / Product	Multiply integers or real; product of list, set, relation
Exp_1 & Exp_2	Intersection	Intersection of list, set, map or relation
Exp_1 + Exp_2	Addition / concatenation / union	Add integer and real; concatenate string, list or tuple; union on set, map, or relation
Exp_1 - Exp_2	Subtraction / difference	Subtract integer or real; difference of list, set, map, or relation
Exp_1 join Exp_2	Join	Join on relation

Boolean Operator Expressions

The Boolean operators are summarized in Table 1.2, “Boolean Operators”[23]. All operators are higly overloaded and we refer to the section called “*Built-in Operators and Library*

Functions” [48] for a description of their meaning for each specific type. Most operator are self-explanatory except the Match and NoMatch operators that are also the main reason to treat Boolean operator expressions separately. Although we describe patterns in full detail in the section called “*Patterns*” [23], a preview is usefull here. A pattern can

- match (or not match) any arbitrary value (that we will call the *subject value*);
- during the match variables may be bound to subvalues of the subject value.

The match operator

```
Pat := Exp
```

is evaluated as follows:

- *Exp* is evaluated, the result is a subject value;
- the subject value is matched against the pattern *pat*;
- if the match succeeds, any variables in the pattern are bound to subvalues of the subject value and the match expression yield `true`;
- if the match fails, no variables are bound and the match expression yields `false`.

This looks and *is* nice and dandy, so why all this fuss about Boolean operators? The catch is that--as we will see in detail in the section called “*Patterns*”[23]--a match need not be unique. This means that there may be more than one way of matching the subject value resulting in different variable bindings. A quick example. Consider the following match of a list

```
[1, list[int] L, 2, list[int] M] := [1,2,3,2,4]
```

There are two solutions for this match:

- `L = []` and `M = [2, 3, 2, 4]`; and
- `L = [2,3]` and `M = [4]`.

Depending on the context, only the first solution of a match expression is used, respectively all solutions are used. If a match expression occurs in a larger Boolean expression, a subsequent subexpression may yield fals and -- depending on the actual operator -- evaluation *backtracks* to a previously evaluated match operator to try a next solution. Let's illustrate this by extending the above example:

```
[1, list[int] L, 2, list[int] M] := [1,2,3,2,4] && size(L) > 0
```

where we are looking for a solution in which `L` has a non-empty list as value. Evaluation proceeds as follows:

- The left argument of the && operator is evaluated: the match expression is evaluated resulting in the bindings `L = []` and `M = [2, 3, 2, 4]`;
- The right argument of the && operator is evaluated: `size(L) > 0` yields false;
- Backtrack to the left argument of the && operator to check for more solutions: indeed there are more solutions resulting in the bindings `L = [2,3]` and `M = [4]`;
- Proceed to thr right operator of &&: this time `size(L) > 0` yields `true`;
- The result of evaluating the complete expression is `true`.

This behaviour is applicable in the context of the following Rascal constructs:

- Comprehensions
- Tests in while, do, for, one, all statements.

Caution

Make the above more specific.

Table 1.2. Boolean Operators

Operator	Name	Description
`!` *Exp*	Negation	Negate *Exp*'s boolean value
Exp `?`	IsDefined	true is Exp has a well-defined value
Exp_1 `in` Exp_2	ElementOf	Element of
Exp_1 `notin` Exp_2	NotElementOf	Not element of
Exp_1 <= Exp_2	LessThanOrEqual	Less than or equal on bool, int, real or string; sublist on list; subset on set, map or relation
Exp_1 < Exp_2	LessThan	Less than on bool, int, real or string; strict sublist on list; strict subset on set, map or relation
Exp_1 >= Exp_2	GreaterThanOrEqual	Greater than or equal on bool, int, real or string; superlist on list; superset on set, map or relation
Exp_1 > Exp_2	GreaterThan	Greater than on bool, int, real or string; strict superlist on list; strict superset on set, map or relation
Pat := *Exp*	Match	Value of *Exp* matches with pattern *Pat*
Pat `!:=` *Exp*	NoMatch	Value of *Exp* does not match with pattern *Pat*
Exp_1 == Exp_2	Equal	Equality
Exp_1 `!=` Exp_2	NotEqual	Inequality
Exp_1 ? Exp_2	IfDefinedElse	the value of Exp_1 is it is well-defined, otherwise the value of Exp_2
Exp_1 ? Exp_2 `:` Exp_3	IfThenElse	Conditional expression
Exp_1 ==> Exp_2	Implication	`true`, unless the value of Exp_1 is `true` and that of Exp_2 is `false`
Exp_1 <==> Exp_2	Equivalence	true if Exp_1 and Exp_2 have the same value
Exp_1 && Exp_2	And	`true` if the value of both Exp_1 and Exp_2 is `true`
Exp_1, Exp_2, ..., Exp_n	MultiCondition	`Equivalent to:` Exp_1 && Exp_2 && ... && Exp_n
Exp_1 `||` Exp_2	Or	`true` if the value of either Exp_1 or Exp_2 is `true`
`Pat <- Exp`	Enumerator	`true` for every element in *Exp*'s value that matches *pat*

Patterns

Patterns come in three flavours:

- *Regulars patterns* to do string matching with regular expressions, see the section called "*Regular Patterns*" [24].
- *Abstract patterns* to matching on arbitrary values, see the section called "*Abstract Patterns*" [24].
- *Concrete patterns* to match syntax trees that are the result of parsing, see the section called "*Concrete patterns*" [26].

Regular Patterns

Regular expression patterns are ordinary regular expressions that are used to match a string value and to decompose it in parts and also to compose new strings. Regular expression patterns bind variables of type `str` when the match succeeds, otherwise they do not bind anything. Their syntax and semantics parallels abstract and concrete syntax patterns as much as possible. This means that they can occur in cases of `visit` and `switch` statements, on the left-hand side of the match operator (`:= or !:=`) and as declarator in generators.

We use a regular expression language that slightly extends the Java Regex language with the following exceptions:

- Regular expression are delimited by / and / optionally followed by a modifier (see below).
- We allow named groups, syntax `<Name:Regex>`, which introduce a variable of type `str` named *`Name`*. Currently, these names have to be unique in the pattern.
- Java regular expressions might have optional groups, which may introduce null bindings. Since unitialized variables are not allowed in Rascal, we limit the kinds of expressions one can write here by not allowing nesting of named groups.
- Named groups have to be outermost, such that they can only bind in one way.
- Unlike Perl, Java uses the notation `(?Option)` inside the regular expression to set options like multi-line matching `(?m)`, case-insensitive matching `(?i)` etc. We let these options follow the regular expression.
- We allow name use in a regular expression: `<Name>` which inserts the string value of *`Name`* in the pattern.

Here are some examples of regular patterns.

```
/\brascal\b/i
```

does a case-insensitive match (`i`) of the word rascal between word boundaries (`\b`). And

```
/^.*?<word:\w+><rest:.*$>/m
```

does a multi-line match (`m`), matches the first consecutive word characters (`\w`) and assigns them to the variable `word`. The remainder of the string is assigned to the variable `rest`.

Abstract Patterns

An abstract pattern is recursively defined and may contain the following elements:

- *Literal* of one of the basic types `bool`, `int`, `real`, `str`, or `loc`. A literal pattern matches with a value that is identical to the literal.
- A *variable declaration pattern*

  ```
  Type Var
  ```

  A variable declaration introduces a new variable that matches any value of the given type. That value is assigned to *`Var`* when the whole match succeeds.

- A *variable pattern*

  ```
  Var
  ```

  A variable pattern can act in two roles:

  - If `Var` has already a defined value then it matches with that value.
  - If Var has not been defined before (or it has been declared but not initalized) then it matches any value. That value is assigned to `Var`.

  ### Warning

  Explain scope.

- A *list pattern*

  ```
  [ Pat1, Pat2, ..., Patn ]
  ```

  A list pattern matches a list value, provided that Pat_1, Pat_2, ..., Pat_n match the elements of that list in order. Two special cases exist when one of the patterns Pat_i is

  - a variable declaration pattern with a list type that is identical to the type of the list that is being matched.
  - a variable pattern, where the variable has been declared, but not initalized, outside the pattern with a list type that is identical to the type of the list that is being matched.

  In both cases list matching is applied and the variable can match an arbitrary number of elements of the subject list.

- A *set pattern*

  ```
  { Pat1, Pat2, ..., Patn }
  ```

  A set pattern matches a set value, provided that Pat_1, Pat_2, ..., Pat_n match the elements of that set in any order. Completely analogous to list patterns, there are two special cases when one of the patterns Pat_i is

  - a variable declaration pattern with a set type that is identical to the type of the set that is being matched.
  - a variable pattern, where the variable has been declared, but not initalized, outside the pattern with a set type that is identical to the type of the set that is being matched.

  In both cases set matching is applied and the variable can match an arbitrary number (in arbitrary order!) of elements of the subject set.

- A *tuple pattern*

  ```
  < Pat1, Pat2, ..., Patn >
  ```

  A tuple pattern matches a tuple value, provided that Pat_1, Pat_2, ..., Pat_n match the elements of that tuple in order.

- A *node pattern*

  ```
  Name ( Pat1, Pat2, ..., Patn )
  ```

  A node pattern matches a node value or an abstract datatype value, provided that `Name` matches with the constructor symbol of that value and Pat_1, Pat_2, ..., Pat_n match the children of that value in order.

- A *labelled pattern*

  ```
  Var : Pat
  ```

  A labelled pattern matches the same values as *`Pat`*, but has as side-effect that the matched value is assigned to *`Var`*.

- A *typed, labelled, pattern*

  ```
  Type Var : Pat
  ```

  A typed, labelled, pattern matches when the subject value has type *`Type`* and *`Pat`* matches. The matched value is assigned to *`Var`*.

  ### Note

  Map patterns are currently not supported.

Concrete patterns

Note

Concrete patterns are not yet implemented.

A *concrete pattern* is a (possibly quoted) concrete syntax fragment that may contain variables. We want to cover the whole spectrum from maximally quoted patterns that can unambiguously describe **any** syntax fragment to minimally quoted patterns as we are used to in ASF+SDF. A concrete pattern may have the following forms:

- A *typed variable pattern*

  ```
  <Type Var>
  ```

- A *variable pattern*

  ```
  <Var>
  ```

- *Quoted pattern*

  ```
  [| Token_1 Token_2 ... Token_n |]
  ```

  Inside a quoted pattern arbitrary lexical tokens may occur, but the characters `<`, `>` and `|` have to be escaped as `\<`, `\>`, `\|`. Quoted patterns may contain variable declaration patterbs and variable patterns.

- A *typed quoted pattern*

  ```
  Symbol [| Token_1 Token_2 ... Token_n |]
  ```

  is a quoted pattern that is preceeded by an SDF symbol to define its desired syntactic type.

- An *unquoted pattern*

  ```
  Token_1 Token_2 ... Token_n
  ```

  is a quoted pattern without the surrounding quotes.

- Inside syntax patterns, layout is ignored.

Examples:

- Quoted syntax pattern with two pattern variable declarations:

```
[| while <EXP Exp> do <{STATEMENT ";"}* Stats> od |]
```

- Quoted syntax pattern with two pattern variable uses:

```
[| while <Exp> do <Stats> od |]
```

- Identical to the previous example, but with a declaration of the desired syntactic type:

```
STATEMENT [| while <Exp> do <Stats> od |]
```

- Unquoted syntax pattern with two pattern variable declarations:

```
while <EXP Exp> do <{STATEMENT ";"}* Stats> od
```

- Unquoted syntax pattern with two pattern variable uses:

```
while <Exp> do <Stats> od
```

Obviously, with less quoting and type information, the probability of ambiguities increases. Our assumption is that a type checker can resolve most of them.

Comprehensions

We will use the familiar notation for *list comprehension*

$$[Exp \mid Gen_1, ..., Gen_n]$$

to denote the construction of a list consisting of the successive values of the *contributing expression* *Exp*. The values and the resulting list are determined by *Exp* and the *generators* Gen_1 ,..., Gen_n. *E*xp is computed for all possible combinations of values produced by the generators. Each generator may introduce new variables that can be used in subsequent generators as well as in the expressions *Exp*. A generator can use the variables introduced by preceeding generators. Generators may enumerate all the values in a set or relation, or they may perform an arbitrary test.

In addition to list comprehensions, Rascal also supports *set comprehension*

$$\{Exp \mid Gen_1, ..., Gen_n\}$$

that also serve as relation comprehension in the case that *Exp* is of a tuple type.

Finally, *map comprehension*s are written as:

$$(Exp_1 : Exp_2 \mid Gen_1, ..., Gen_n)$$

Since the entries in a map require both a key and a value for each entry, two expressions are needed in this case.

Enumerator

An enumerator generates all the values in a given list, set, map, tuple, relation or abstract datatype. It has the following form:

```
Pat <- Exp
```

where *Pat* is a pattern and *Exp* is an expression. An enumerator is evaluated as follows:

- Expression *Exp* is evaluated and may have an arbitrary value *V*.
- The elements of *V* are enumerated one by one.
- Each element value is matched against the pattern *Pat*. There are two cases:

- The match succeeds, any variables in *Pat* are bound, and the next generator in the comprehension is evaluated. The variables that are introduced by an enumartor are only available to generators that appear later (i.e., to the right) in the comprehension. When this enumerator is the last generator in the comprehension its contributing expression is evaluated.
- The match fails, no variables are bound. If V has more elements, a next element is tried. Otherwise, a previous generator (i.e., to the left) is tried. If this enumerator is the first generator in the comprehension, the evaluation of the comprehension is complete.

This are examples of enumerators:

- int N <- {1, 2, 3, 4, 5},
- str K <- KEYWORDS, where KEYWORDS should evaluate to a value of set[str].
- <str K, int N> <- {<"a",10>, <"b",20>, <"c",30>}.
- <str K, int N> <- FREQUENCIES, where FREQUENCIES should evaluate to a value of type rel[str,int].
- <str K, 10> <- FREQUENCIES, will only generate pairs with 10 as second element.

Note

Type information will be used to check the plausibity of an enumerator and guard you against mistakes. An impossible enumerator like int N <- {"apples", "oranges"} will be flagged as an error since the pattern can never match.

Note

An enumerator may be preceeded by a *strategy indication*:

- top-down
- bottom-up (this is the default)

These take only effect for enumerators that produce the elements of an abstract data type and determine the order in which the elements are enumerated.

Test

A test is a boolean-valued expression. If the evaluation yields true this indicates that the current combination of generated values up to this test is still as desired and execution continues with subsequent generators. If the evaluation yields false this indicates that the current combination of values is undesired, and that another combination should be tried by going back to the previous generator.

Examples:

- N >= 3 tests whether N has a value greater than or equal 3.
- S == "coffee" tests whether S is equal to the string "coffee".

In both examples, the variable (N, respectively, S) should have been introduced by a generator that occurs earlier in the comprehension.

Examples of Comprehensions

- {X | int X : {1, 2, 3, 4, 5}, X >= 3} yields the set {3,4,5}.
- {<X, Y> | int X : {1, 2, 3}, int Y : {2, 3, 4}, X >= Y} yields the relation {<2, 2>, <3, 2>, <3, 3>}.

- {<Y, X> | <int X, int Y> : {<1,10>, <2,20>}} yields the inverse of the given relation: {<10,1>, <20,2>}.
- {X, X * X | X : {1, 2, 3, 4, 5}, X >= 3} yields the set {3,4,5,9,16,25}.

Statements

Assignment

Switch statement

A switch statement is similar to a switch statement in C or Java and has the form:

```
switch ( Exp ) {
case Rule1;
case Rule2;
...
default: ...
}
```

The value of the expression *Exp* is the subject term that will be matched by the successive cases in the switch statement. This corresponds to the matching provided by the left-hand sides of a set of rewrite rules. However, the switch statement provides **only** matching at the top level of the subject term and does not traverse it. The type of the pattern in each case must be identical to the type of the subject term (or be a subtype of it). If no case matches, the switch acts as a dummy statement. There is no fall through from one case to the next.

Each case contains a *Rule* that can have one of the following forms:

- *Pat => Exp*

  When the subject matches *Pat*,the value of Exp is returned from the enclosing function.

  ### Warning

  Check this; how is this implemented?

- *Pat* : *Statement*

  This is the most general case. When the subject matches *Pat*, the *Statement* is executed. The execution of *Statement* should lead to one of the following:

  - Execution of a return statement that returns a value from the enclosing function.
  - Execution of a fail statement: all side effects of *Statement* are undone and the next case is tried.
  - None of the above: execution continues with the statement following the switch.

Visit expression

Caution

Technically, visit is an expression and not a statement; Move it from here?

Visiting the nodes in a tree is a very common task in the EASY domain. In many cases (but certainly not all) the tree is a syntax tree of some source code file and the nodes correspond to expressions or statements. Computing metrics or refactoring are examples of tasks that require a tree visit. In object-oriented programming, the *visitor pattern* is in common use for this. There are three frequently occurring scenarios:

- Accumulator: traverse the tree and collect information.
- Transformer: traverse the tree and transform it into another tree.
- Accumulating Transformer: traverse the tree, collect information and also transform the tree.

The `visit` expression in Rascal can accommodate all these (and more) use cases and has the form:

```
Strategy visit ( Exp ) {
case Rule1;
case Rule2;
...
default: ...
}
```

Given a subject term (the current value of *Exp*) and a list of cases (resembling cases in a case statement or rewrite rules) it traverses the term. Depending on the precise rules it may perform replacement (mimicking a transformer), update local variables (mimicking an accumulator) or a combination of these two (accumulating transformer). If **any** of the cases contains an `insert` statement, the value of the visit expression is a new value that is obtained by successive insertions in the subject term by executing one or more cases. Otherwise, the original value of the subject term is returned.

The visit expression is optionally preceeded by one of the following strategy indications that determine the traversal order of the subject:

- top-down: visit the subject from root to leaves.
- `top-down-break`: visit the subject from root to leaves, but stop at the current path when a case matches.
- `bottom-up`: visit the subject from leaves to root (this is the default).
- `bottom-up-break`: visit the subject from leaves to root, but stop at the current path when a case matches.
- `innermost`: repeat a bottom-up traversal as long as the traversal changes the resulting value (compute a fixed-point).
- `outermost`: repeat a top-down traversal as long as the traversal changes the resulting value (compute a fixed-point).

The execution of the cases is similar to the cases in a switch statement with the following exceptions:

- Rules of the form *`Pat`* => *`Exp`* insert their result in the subject (instead of returning a value).
- For rules of the form *`Pat : Statement`*, executing *`Statement`* should lead to one of the following:
  - Execution of an `insert` statement of the form `insert` *`Expression`*. The value of *`Expression`* replaces the subtree of the subject that is currently being visited. Note that a copy of the subject is created at the start of the visit statement and all insertions are made in this copy. As a consequence, insertions cannot influence matches later on.

    Note

    An `insert` statement may only occur inside a `visit` expression.

  - Execution of a `fail` statement: all side effects of *`Statement`* are undone, no insertion is made, and the next case is tried.
  - Execution of a `return` statement that returns a value from the enclosing function.

Each case keyword may be followed by a type constraint of the form `[Type]` that limits the type of the values to which the case applies.

The precise behaviour of the visit statement depends on the type of the subject:

- For type `node or ADT`, all nodes of the tree are visited (in the order determined by the strategy). concrete patterns and abstract patterns directly match tree nodes. Regular expression patterns match only values of type string.
- For structured types (list, set, map, tuple, rel), the elements of the structured type are visited and matched against the cases. When inserts are made, a new structured value is created.

Caution

Have strategies any effect for non-tree subjects?

Try Catch

Throw

Return

Insert

Fail

Block

A block consists of a sequence of statements separated by semi-colons:

```
{ Statement1; ... Statementn }
```

Since a block is itself a statement, it may be used in all places where a statement is required. A block also introduces a new scope and variables that are declarated in the block are local to that block.

If

The if-statement is completely standard and comes in an if-then and an if-then-else variant:

```
if ( Bool ) Statement
```

```
if ( Bool ) Statement1 else Statement2
```

In both cases the test `Bool` is evaluated and its outcome determines the statement to be executed. Recall from the section called "*Boolean Operator Expressions*" [21] that boolean expression maybe multi-valued. In this case only the first true value (if any) is used.

While

The while-statement is completely standard:

```
while ( Bool ) Statement
```

The test `Bool` is evaluated repeatedly and `Statement` is executed when the test is true. Execution ends the first time that the test yields false. The test `Bool` is executed anew in each repetition and only the first true value (if any) is used.

Do

The do-while-statement is completely standard:

```
do Statement while ( Bool )
```

Statement is executed repeatedly, as long as the test Bool yields true. The test `Bool` is executed anew in each repetition and only the first true value (if any) is used.

For

The for-statement is more exciting:

```
for ( Exp1 , Exp2 , ... , Expn ) Statement
```

It generates executes Statement for all possible combinations of values of the expressions Exp_i. Note that is one of the expressions is a boolean expression, we do try all its possible values.

One

```
one ( Exp1 , Exp2 , ... , Expn )
```

The one expression yields true when one combination of values of Exp_i is true.

Caution

The status of one is under discussion.

Caution

Technically, one is an expression.

All

```
all ( Exp1 , Exp2 , ... , Expn )
```

The all expression yields true when all combinations of values of Exp_i are true.

Caution

The status of all is under discussion.

Caution

Technically, one is an expression.

Solve

Rascal provides a solve statement for performing arbitrary fixed-point computations. This means, repeating a certain computation as long as it causes changes. This can, for instance, be used for the solution of sets of simultaneous linear equations but has much wider applicability. The format is:

```
with {
  Type1 Var1 = Exp1;
  Type2 Var2 = Exp2;
  ...
} solve
  Statement
```

The solve statement consists of an initialization section and a statement. Optionally, an expression directly following the `solve` keyword, gives an upperbound on the number of iterations.

In the `initial` section, the variables Var_i are declared and initialized. In the `solve` section, the statement can use and modify these variables. The statement is executed, assigning new values to the variables Var_i, and this is repeated as long as the value of any of the variables was changed compared to the previous repetition.

Let's consider transitve closure as an example (transitive closure is already available as built-in operator, we use it here just as a simple illustration). Transitive closure of a relation is usually defined as:

```
R+ = R + (R o R) + (R o R o R) + ...
```

This can be expressed as follows:

```
rascal> rel[int,int] R = {<1,2>, <2,3>, <3,4>};
{<1,2>, <2,3>, <3,4>}
rascal>
  with
     rel[int,int] T = R;
  solve
     T = T + (T o R1);

{<1,2>, <1,3>,<1,4>,<2,3>,<2,4>,<3,4>}
```

Assertions

An assert statement may occur everywhere where a declaration is allowed. It has two forms:

```
assert Exp1
```

and

```
assert Exp1 : Exp2
```

where Exp_1 is a boolean-value expression and Exp_2 is a string-valued expression that serves as a identifying message for this assertion. During execution, a list of true and false assertions is maintained. When the script is executed as a *test suite* a summary of this information is shown to the user. When the script is executed in the standard fashion, the assert statement has no affect.

Note

Update the above description.

Example:

```
rascal> assert {1, 2, 3, 1} == {3, 2, 1, 1} : "Equality on Sets";
```

Larger Examples (TODO)

Now we will have a closer look at some larger applications of Rascal. We start by analyzing the global structure of a software system. You may now want to reread the example of call graph analysis given earlier in the section called “*A Motivating Example*” [7] as a reminder. The component structure of an application is analyzed in the section called “*Analyzing the Component Structure of an Application*” [34] and Java systems are analyzed in the section called “*Analyzing the Structure of Java Systems*” [34]. Next we move on to the detection of initialized variables in the section called “*Finding Uninitialized and Unused Variables in a Program*” [36] and we explain how source code locations can be included in a such an analysis (the section called “*Using Locations to Represent Program Fragments*” [37]). As an example of computing code metrics, we describe the calculation of McCabe's cyclomatic complexity in the section called “*McCabe Cyclomatic Complexity*”. Several examples of dataflow analysis follow in the section called “*Dataflow Analysis*” [39]. A description of program slicing concludes the chapter (the section called “*Program Slicing*” [46]).

Analyzing the Component Structure of an Application

A frequently occurring problem is that we know the call relation of a system but that we want to understand it at the component level rather than at the procedure level. If it is known to which component each procedure belongs, it is possible to *lift* the call relation to the component level as proposed in [Kri99]. First, introduce new types to denote procedure calls as well as components of a system:

```
type proc = str
type comp = str
```

Given a calls relation `Calls2`, the next step is to define the components of the system and to define a `PartOf` relation between procedures and components.

```
rel[proc,proc] Calls = {<"main", "a">, <"main", "b">, <"a", "b">,
                         <"a", "c">, <"a", "d">, <"b", "d">
                       }

set[comp] Components = {"Appl", "DB", "Lib"}

rel[proc, comp] PartOf = {<"main", "Appl">, <"a", "Appl">,
                          <"b", "DB">, <"c", "Lib">, <"d", "Lib">
                         }
```

Actual lifting, amounts to translating each call between procedures by a call between components. This is achieved by the following function `lift`:

```
rel[comp,comp] lift(rel[proc,proc] aCalls, rel[proc,comp] aPartOf)=
   { <C1, C2> | <proc P1, proc P2> : aCalls,
                <comp C1, comp C2> : aPartOf[P1] x aPartOf[P2]
   }
```

In our example, the lifted call relation between components is obtained by

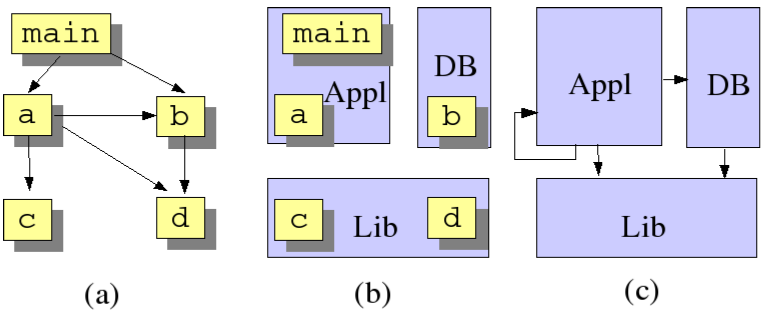
```
rel[comp,comp] ComponentCalls = lift(Calls2, PartOf)
```

and has as value:

```
{<"DB", "Lib">, <"Appl", "Lib">, <"Appl", "DB">, <"Appl", "Appl">}
```

The relevant relations for this example are shown in Figure 1.6, “(a) `Calls2`; (b) `PartOf`; (c) `ComponentCalls`.” [34].

Figure 1.6. (a) `Calls2`; (b) `PartOf`; (c) `ComponentCalls`.



Analyzing the Structure of Java Systems

Now we consider the analysis of Java systems (inspired by [BNL03]. Suppose that the type `class` is defined as follows

```
type class = str
```

and that the following relations are available about a Java application:

- `rel[class,class] CALL`: If <C_1, C_2> is an element of `CALL`, then some method of C_2 is called from C_1.
- `rel[class,class] INHERITANCE`: If <C_1, C_2> is an element of `INHERITANCE`, then class C_1 either extends class C_2 or C_1 implements interface C_2.
- `rel[class,class] CONTAINMENT`: If <C_1, C_2> is an element of `CONTAINMENT`, then one of the attributes of class C_1 is of type C_2.

To make this more explicit, consider the class `LocatorHandle` from the JHotDraw application (version 5.2) as shown here:

```
package CH.ifa.draw.standard;

import java.awt.Point;
import CH.ifa.draw.framework.*;
/**
 * A LocatorHandle implements a Handle by delegating the
 * location requests to a Locator object.
 */
public class LocatorHandle extends AbstractHandle {
    private Locator       fLocator;
    /**
     * Initializes the LocatorHandle with the given Locator.
     */
    public LocatorHandle(Figure owner, Locator l) {
        super(owner);
        fLocator = l;
    }
    /**
     * Locates the handle on the figure by forwarding the request
     * to its figure.
     */
    public Point locate() {
        return fLocator.locate(owner());
    }
}
```

It leads to the addition to the above relations of the following tuples:

- To `CALL` the pairs `<"LocatorHandle", "AbstractHandle">` and `<"LocatorHandle", "Locator">` will be added.
- To `INHERITANCE` the pair `<"LocatorHandle", "AbstractHandle">` will be added.
- To `CONTAINMENT` the pair `<"LocatorHandle", "Locator">` will be added.

Cyclic structures in object-oriented systems makes understanding hard. Therefore it is interesting to spot classes that occur as part of a cyclic dependency. Here we determine cyclic uses of classes that include calls, inheritance and containment. This is achieved as follows:

```
rel[class,class] USE = CALL union CONTAINMENT union INHERITANCE
set[str] ClassesInCycle =
   {C1 | <class C1, class C2> : USE+, C1 == C2}
```

First, we define the `USE` relation as the union of the three available relations `CALL`, `CONTAINMENT` and `INHERITANCE`. Next, we consider all pairs <C_1, C_2> in the transitive closure of the `USE` relation such that C_1 and C_2 are equal. Those are precisely the cases of a class with a cyclic dependency on itself. Probably, we do not only want to know which classes occur in a cyclic dependency, but we also want to know which classes are involved in such a cycle. In other words, we want to associate with each class a set of classes that are responsible for the cyclic dependency. This can be done as follows.

```
rel[class,class] USE = CALL union CONTAINMENT union INHERITANCE
set[class] CLASSES = carrier(USE)
rel[class,class] USETRANS = USE+
rel[class,set[class]] ClassCycles =
   {<C, USETRANS[C]> | class C : CLASSES, <C, C> in USETRANS }
```

First, we introduce two new shorthands: `CLASSES` and `USETRANS`. Next, we consider all classes `C` with a cyclic dependency and add the pair `<C, USETRANS[C]>` to the relation `ClassCycles`. Note that `USETRANS[C]` is the right image of the relation `USETRANS` for element `C`, i.e., all classes that can be called transitively from class `C`.

Finding Uninitialized and Unused Variables in a Program

Consider the following program in the toy language Pico: (This is an extended version of the example presented earlier in [Kli03].)

```
[ 1] begin declare x : natural, y : natural,
[ 2]               z : natural, p : natural;
[ 3]  x := 3;
[ 4]  p := 4;
[ 5]  if q then
[ 6]        z := y + x
[ 7]  else
[ 8]        x := 4
[ 9]  fi;
[10]  y := z
[11] end
```

Inspection of this program learns that some of the variables are being used before they have been initialized. The variables in question are `q` (line 5), `y` (line 6), and `z` (line 10). It is also clear that variable `p` is initialized (line 4), but is never used. How can we automate these kinds of analysis? Recall from the section called “*EASY Programming*”[2] that we follow Extract-Analyze-SYnthesize paradigm to approach such a problem. The first step is to determine which elementary facts we need about the program. For this and many other kinds of program analysis, we need at least the following:

- The *control flow graph* of the program. We represent it by a relation `PRED` (for predecessor) which relates each statement with each predecessors.
- The *definitions* of each variable, i.e., the program statements where a value is assigned to the variable. It is represented by the relation `DEFS`.
- The *uses* of each variable, i.e., the program statements where the value of the variable is used. It is represented by the relation `USES`.

In this example, we will use line numbers to identify the statements in the program. (In the section called “*Using Locations to Represent Program Fragments*”[37], we will use locations to represent statements.) Assuming that there is a tool to extract the above information from a program text, we get the following for the above example:

```
type expr = int
type varname = str
expr ROOT = 1
rel[expr,expr] PRED = { <1,3>, <3,4>, <4,5>, <5,6>, <5,8>,
```

```
                        <6,10>, <8,10>
                      }
rel[expr,varname] DEFS = { <3,"x">, <4,"p">, <6,"z">,
                           <8,"x">, <10,"y">
                         }
rel[expr,varname] USES = { <5,"q">, <6,"y">, <6,"x">, <10,"z"> }
```

This concludes the extraction phase. Next, we have to enrich these basic facts to obtain the initialized variables in the program. So, when is a variable *V* in some statement *S* initialized? If we execute the program (starting in ROOT), there may be several possible execution path that can reach statement *S*. All is well if *all* these execution path contain a definition of *V*. However, if one or more of these path do *not* contain a definition of *V*, then *V* may be uninitialized in statement *S*. This can be formalized as follows:

```
rel[expr,varname] UNINIT =
   { <E, V> | <expr E, varname V>: USES,
              E in reachX({ROOT}, DEFS[-,V], PRED)
   }
```

We analyze this definition in detail:

- `<expr E, varname V> : USES` enumerates all tuples in the USES relation. In other words, we consider the use of each variable in turn.
- `E in reachX({ROOT}, DEFS[-,V], PRED)` is a test that determines whether statement *S* is reachable from the ROOT without encountering a definition of variable *V*.
  - {ROOT} represents the initial set of nodes from which all path should start.
  - DEFS[-,V] yields the set of all statements in which a definition of variable V occurs. These nodes form the exclusion set for reachX: no path will be extended beyond an element in this set.
  - PRED is the relation for which the reachability has to be determined.
  - The result of `reachX({ROOT}, DEFS[-,V], PRED)` is a set that contains all nodes that are reachable from the ROOT (as well as all intermediate nodes on each path).
  - Finally, `E in reachX({ROOT}, DEFS[-,V], PRED)` tests whether expression E can be reached from the ROOT.
- The net effect is that UNINIT will only contain pairs that satisfy the test just described.

When we execute the resulting Rascal code (i.e., the declarations of ROOT, PRED, DEFS, USES and UNINIT), we get as value for UNINIT:

```
{<5, "q">, <6, "y">, <10, "z">}
```

and this is in concordance with the informal analysis given at the beginning of this example.

As a bonus, we can also determine the *unused* variables in a program, i.e., variables that are defined but are used nowhere. This is done as follows:

```
set[var] UNUSED = range(DEFS) \ range(USES)
```

Taking the range of the relations DEFS and USES yields the variables that are defined, respectively, used in the program. The difference of these two sets yields the unused variables, in this case {"p"}.

Using Locations to Represent Program Fragments

Warning

Fix the following

\begin{figure}[tb] \begin{center} \epsfig{figure=figs/meta-pico.eps,width=6cm} \hspace*{0.5cm} \epsfig{figure=figs/pico-example.eps,width=6cm} \end{center} \hrulefill \caption{\label{FIG:meta-pico}Checking undefined variables in Pico programs using the ASF+SDF Meta-Environment. On the left, main window of Meta-Environment with error messages related to Pico program shown on the right.{\bf THIS FIGURE IS OUTDATED}} \end{figure}

One aspect of the example we have just seen is artificial: where do these line numbers come from that we used to indicate expressions in the program? One solution is to let the extraction phase generate *locations* to precisely indicate relevant places in the program text. Recall from the section called "*Elementary Types and Values*" [11], that a location consists of a file name, a begin line, a begin position, an end line, and an end position. Also recall that locations can be compared: a location A_1 is smaller than another location A_2, if A_1 is textually contained in A_2. By including locations in the final answer of a relational expression, external tools will be able to highlight places of interest in the source text.

The first step, is to define the type `expr` as aliases for `loc` (instead of `int`):

```
type expr = loc
type varname = str
```

Of course, the actual relations are now represented differently. For instance, the `USES` relation may now look like

```
{ <area-in-file("/home/paulk/example.pico",
                area(5,5,5,6,106,1)), "q">,
  <area-in-file("/home/paulk/example.pico",
                area(6,13,6,14,127,1)), "y">,
  <area-in-file("/home/paulk/example.pico",
                area(6,17,6,18,131,1)), "x">,
  <area-in-file("/home/paulk/example.pico",
                area(10,7,10,8,168,1)), "z">
}
```

The definition of `UNINIT` can be nearly reused as is. The only thing that remains to be changed is to map the (expression, variable-name) tuples to (variable-name, variable-occurrence) tuples, for the benefit of the precise highlighting of the relevant variables. We define a new type `var` to represent variable occurrences and an auxiliary set that `VARNAMES` that contains all variable names:

```
type var = loc
set[varname] VARNAMES = range(DEFS) union range(USES)
```

Remains the new definition of `UNINIT`:

```
rel[var, varname] UNINIT =
   { <V, VN>| var-name VN : VARNAMES,
              var V : USES[-,VN],
              expr E : reachX({ROOT}, DEFS[-,VN], PRED),
              V <= E
   }
```

This definition can be understood as follows:

- `var-name VN : VARNAMES` generates all variable names.
- `var V : USES[-,VN]` generates all variable uses `V` for variables with name `VN`.
- As before, `expr E : reachX({ROOT}, DEFS[-,VN], PRED)` generates all expressions `E` that can be reached from the start of the program without encountering a definition for variables named `VN`.
- `V <= E` tests whether variable use `V` is enclosed in that expression (using a comparison on locations). If so, we have found an uninitialized occurrence of the variable named `VN`.

Warning

Fix reference

In Figure~\ref{FIG:meta-pico} it is shown how checking of Pico programs in the ASF+SDF Meta-Environment looks like.

McCabe Cyclomatic Complexity

The *cyclomatic complexity* of a program is defined as *e* - *n* + 2, where *e* and *n* are the number of edges and nodes in the control flow graph, respectively. It was proposed by McCabe [McC76] as a measure of program complexity. Experiments have shown that programs with a higher cyclomatic complexity are more difficult to understand and test and have more errors. It is generally accepted that a program, module or procedure with a cyclomatic complexity larger than 15 is *too complex*. Essentially, cyclomatic complexity measures the number of decision points in a program and can be computed by counting all if statement, case branches in switch statements and the number of conditional loops. Given a control flow in the form of a predecessor relation `rel[stat,stat] PRED` between statements, the cyclomatic complexity can be computed in an Rascal as follows:

```
int cyclomatic-complexity(rel[stat,stat] PRED) =
    #PRED - #carrier(PRED) + 2
```

The number of edges *e* is equal to the number of tuples in `PRED`. The number of nodes *n* is equal to the number of elements in the carrier of PRED, i.e., all elements that occur in a tuple in `PRED`.

Dataflow Analysis

Dataflow analysis is a program analysis technique that forms the basis for many compiler optimizations. It is described in any text book on compiler construction, e.g. [ASU86]. The goal of dataflow analysis is to determine the effect of statements on their surroundings. Typical examples are:

- Dominators (the section called "*Dominators*"[39]): which nodes in the flow dominate the execution of other nodes?
- Reaching definitions (the section called "*Reaching Definitions*"[41]): which definitions of variables are still valid at each statement?
- Live variables (the section called "*Live Variables*"[45]): of which variables will the values be used by successors of a statement?
- Available expressions: an expression is available if it is computed along each path from the start of the program to the current statement.

Dominators

A node *d* of a flow graph *dominates* a node *n*, if every path from the initial node of the flow graph to *n* goes through *d* [ASU86] (Section 10.4). Dominators play a role in the analysis of conditional statements and loops. The function `dominators` that computes the dominators for a given flow graph `PRED` and an entry node `ROOT` is defined as follows:

```
rel[stat,stat] dominators(rel[stat,stat] PRED, int ROOT) =
   DOMINATES
where
    set[int] VERTICES = carrier(PRED)

    rel[int,set[int]] DOMINATES =
    { <V,  VERTICES \ {V, ROOT} \ reachX({ROOT}, {V}, PRED)> |
      int V : VERTICES }
endwhere
```

First, the auxiliary set `VERTICES` (all the statements) is computed. The relation `DOMINATES` consists of all pairs `<S, {S1,...,Sn}>` such that

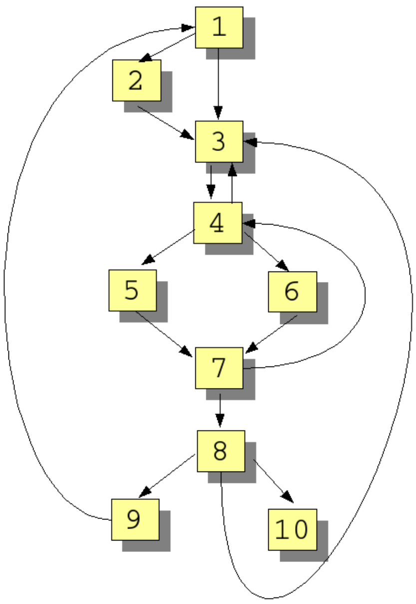
- S_i is not an initial node or equal to S.
- S_i cannot be reached from the initial node without going through S.

Consider the flow graph

```
rel[int,int] PRED = {
<1,2>, <1,3>,
<2,3>,
<3,4>,
<4,3>,<4,5>, <4,6>,
<5,7>,
<6,7>,
<7,4>,<7,8>,
<8,9>,<8,10>,<8,3>,
<9,1>,
<10,7>
}
```

It is illustrated inFigure 1.7, “Flow graph” [40]

Figure 1.7. Flow graph



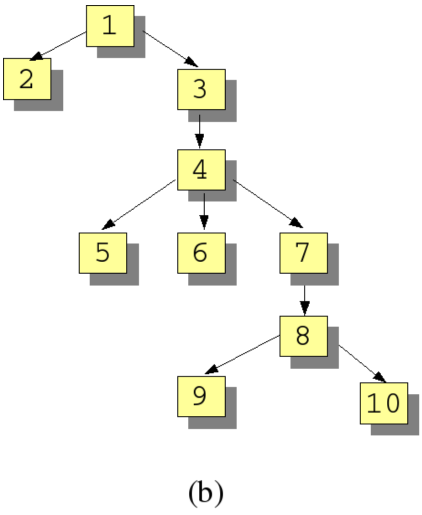
(a)

The result of applying `dominators` to it is as follows:

```
{<1, {2, 3, 4, 5, 6, 7, 8, 9, 10}>,
<2, {}>,
<3, {4, 5, 6, 7, 8, 9, 10}>,
<4, {5, 6, 7, 8, 9, 10}>,
<5, {}>,
<6, {}>,
<7, {8, 9, 10}>,
<8, {9, 10}>,
<9, {}>,
<10, {}>}
```

The resulting *dominator tree* is shown in Figure 1.8, “Dominator tree”[41]. The dominator tree has the initial node as root and each node *d* in the tree only dominates its descendants in the tree.

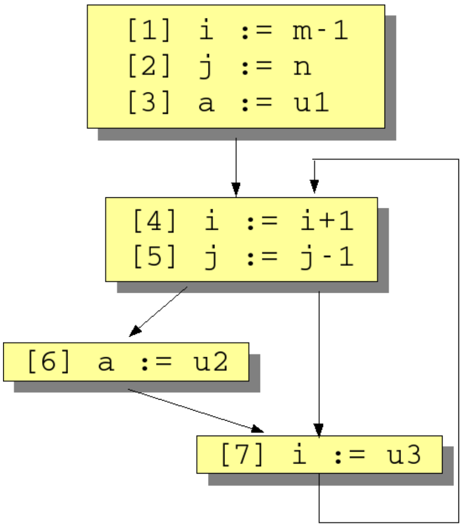
Figure 1.8. Dominator tree



Reaching Definitions

We illustrate the calculation of reaching definitions using the example in Figure 1.9, “Flow graph for various dataflow problems” [41] which was inspired by [ASU86] (Example 10.15).

Figure 1.9. Flow graph for various dataflow problems



As before, we assume the following basic relations PRED, DEFS and USES about the program:

```
type stat = int
type var = str
rel[stat,stat] PRED = { <1,2>, <2,3>, <3,4>, <4,5>, <5,6>,
                        <5,7>, <6,7>, <7,4>
                      }
rel[stat, var] DEFS = { <1, "i">, <2, "j">, <3, "a">, <4, "i">,
                       <5, "j">, <6, "a">, <7, "i">
                      }
rel[stat, var] USES = { <1, "m">, <2, "n">, <3, "u1">, <4, "i">,
                        <5, "j">, <6, "u2">, <7, "u3">
```

```
                          }
```

For convenience, we introduce a notion `def` that describes that a certain statement defines some variable and we revamp the basic relations into a more convenient format using this new type:

```
type def  = <stat theStat, var theVar>

rel[stat, def] DEF = {<S, <S, V>> | <stat S, var V> : DEFS}
rel[stat, def] USE = {<S, <S, V>> | <stat S, var V> : USES}
```

The new `DEF` relation gets as value:

```
{ <1, <1, "i">>, <2, <2, "j">>, <3, <3, "a">>, <4, <4, "i">>,
  <5, <5, "j">>, <6, <6, "a">>, <7, <7, "i">>
}
```

and `USE` gets as value:

```
{ <1, <1, "m">>, <2, <2, "n">>, <3, <3, "u1">>, <4, <4, "i">>,
  <5, <5, "j">>, <6, <6, "u2">>, <7, <7, "u3">>
}
```

Now we are ready to define an important new relation `KILL`. `KILL` defines which variable definitions are undone (killed) at each statement and is defined as follows:

```
rel[stat, def] KILL =
   {<S1, <S2, V>> | <stat S1, var V> : DEFS,
                    <stat S2, V> : DEFS,
                    S1 != S2
   }
```

In this definition, all variable definitions are compared with each other, and for each variable definition all *other* definitions of the same variable are placed in its kill set. In the example, `KILL` gets the value

```
{ <1, <4, "i">>, <1, <7, "i">>, <2, <5, "j">>, <3, <6, "a">>,
  <4, <1, "i">>, <4, <7, "i">>, <5, <2, "j">>, <6, <3, "a">>,
  <7, <1, "i">>, <7, <4, "i">>
}
```

and, for instance, the definition of variable `i` in statement `1` kills the definitions of `i` in statements `4` and `7`. Next, we introduce the collection of statements

```
set[stat] STATEMENTS = carrier(PRED)
```

which gets as value `{1, 2, 3, 4, 5, 6, 7}` and two convenience functions to obtain the predecessor, respectively, the successor of a statement:

```
set[stat] predecessor(stat S) = PRED[-,S]
set[stat] successor(stat S) = PRED[S,-]
```

After these preparations, we are ready to formulate the reaching definitions problem in terms of two relations `IN` and `OUT`. `IN` captures all the variable definitions that are valid at the entry of each statement and `OUT` captures the definitions that are still valid after execution of each statement. Intuitively, for each statement `S`, `IN[S]` is equal to the union of the `OUT` of all the predecessors of `S`. `OUT[S]`, on the other hand, is equal to the definitions generated by `S` to which we add `IN[S]` minus the definitions that are killed in `S`. Mathematically, the following set of equations captures this idea for each statement:

Warning

Fix expression

[IN[S] = \bigcup_{P \in predecessor of S} OUT[P] \]

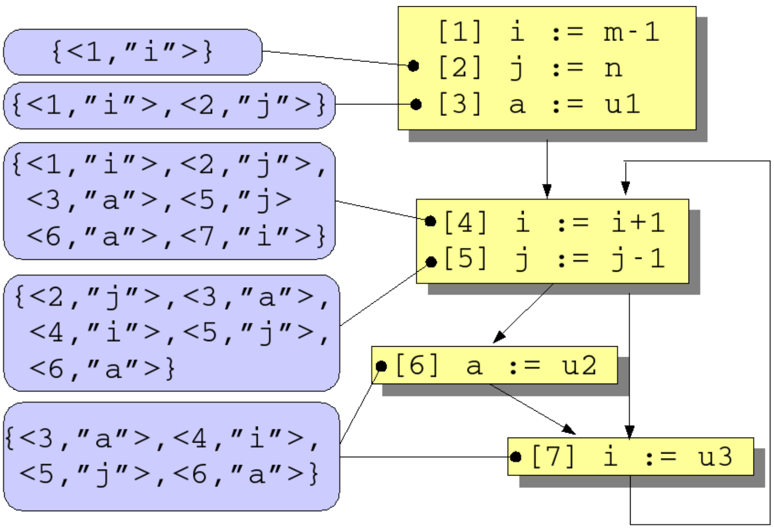
\[OUT[S] = DEF[S] \cup (IN[S] - KILL[S]) \]

This idea can be expressed in Rascal quite literally:

```
equations
  initial
        rel[stat,def] IN init {}
        rel[stat,def] OUT init DEF
  satisfy
        IN  = {<S, D> | stat S : STATEMENTS,
                        stat P : predecessor(S),
                        def D : OUT[P]}
        OUT = {<S, D> | stat S : STATEMENTS,
                        def D : DEF[S] union (IN[S] \ KILL[S])}
end equations
```

First, the relations `IN` and `OUT` are declared and initialized. Next, two equations are given that very much resemble the ones given above.

Figure 1.10. Reaching definitions for flow graph in Figure 1.9, “Flow graph for various dataflow problems” [41]



For our running example (Figure 1.10, “Reaching definitions for flow graph in Figure 1.9, “Flow graph for various dataflow problems”’[43]) the results are as follows (see Figure 1.10, “Reaching definitions for flow graph in Figure 1.9, “Flow graph for various dataflow problems”’[43]). Relation `IN` has as value:

```
{ <2, <1, "i">>, <3, <2, "j">>, <3, <1, "i">>, <4, <3, "a">>,
  <4, <2, "j">>, <4, <1, "i">>, <4, <7, "i">>, <4, <5, "j">>,
  <4, <6, "a">>, <5, <4, "i">>, <5, <3, "a">>, <5, <2, "j">>,
  <5, <5, "j">>, <5, <6, "a">>, <6, <5, "j">>, <6, <4, "i">>,
  <6, <3, "a">>, <6, <6, "a">>, <7, <5, "j">>, <7, <4, "i">>,
  <7, <3, "a">>, <7, <6, "a">>
}
```

If we consider statement `3`, then the definitions of variables `i` and `j` from the preceding two statements are still valid. A more interesting case are the definitions that can reach statement `4`:

- The definitions of variables `a`, `j` and `i` from, respectively, statements `3`, `2` and `1`.

- The definition of variable `i` from statement `7` (via the backward control flow path from `7` to `4`).
- The definition of variable `j` from statement `5` (via the path `5`, `7`, `4`).
- The definition of variable `a` from statement `6` (via the path `6`, `7`, `4`).

Relation `OUT` has as value:

```
{ <1, <1, "i">>, <2, <2, "j">>, <2, <1, "i">>, <3, <3, "a">>,
  <3, <2, "j">>, <3, <1, "i">>, <4, <4, "i">>, <4, <3, "a">>,
  <4, <2, "j">>, <4, <5, "j">>, <4, <6, "a">>, <5, <5, "j">>,
  <5, <4, "i">>, <5, <3, "a">>, <5, <6, "a">>, <6, <6, "a">>,
  <6, <5, "j">>, <6, <4, "i">>, <7, <7, "i">>, <7, <5, "j">>,
  <7, <3, "a">>, <7, <6, "a">>
}
```

Observe, again for statement `4`, that all definitions of variable `i` are missing in `OUT[4]` since they are killed by the definition of `i` in statement `4` itself. Definitions for `a` and `j` are, however, contained in `OUT[4]`. The result of reaching definitions computation is illustrated in Figure 1.10, “Reaching definitions for flow graph in Figure 1.9, “Flow graph for various dataflow problems”” [43]. The above definitions are used to formulate the function `reaching-definitions`. It assumes appropriate definitions for the types `stat` and `var`. It also assumes more general versions of `predecessor` and `successor`. We will use it later on in the section called “*Program Slicing*” [46] when defining program slicing. Here is the definition of `reaching-definitions`:

```
type def  = <stat theStat, var theVar>
type use  = <stat theStat, var theVar>

set[stat] predecessor(rel[stat,stat] P, stat S) = P[-,S]

set[stat] successor(rel[stat,stat] P, stat S) = P[S,-]

rel[stat, def] reaching-definitions(rel[stat,var] DEFS,
                                    rel[stat,stat] PRED) =
   IN
where
    set[stat] STATEMENT = carrier(PRED)

    rel[stat,def] DEF  = {<S,<S,V>> | <stat S, var V> : DEFS}

    rel[stat,def] KILL =
        {<S1, <S2, V>> | <stat S1, var V> : DEFS,
                         <stat S2, V> : DEFS,
                         S1 != S2
        }

    equations
       initial
           rel[stat,def] IN init {}
           rel[stat,def] OUT init DEF
       satisfy
           IN  =  {<S, D> | int S : STATEMENT,
                            stat P : predecessor(PRED,S),
                            def D : OUT[P]}
           OUT = {<S, D> |  int S : STATEMENT,
                            def D : DEF[S] union (IN[S] \ KILL[S])}
    end equations
end where
```

Live Variables

The live variables of a statement are those variables whose value will be used by the current statement or some successor of it. The mathematical formulation of this problem is as follows:

Warning

Fix expression

\[IN[S] =USE[S] \cup (OUT[S] - DEF[S]) \]

\[OUT[S] = \bigcup_{S' \in successor of S} IN[S'] \]

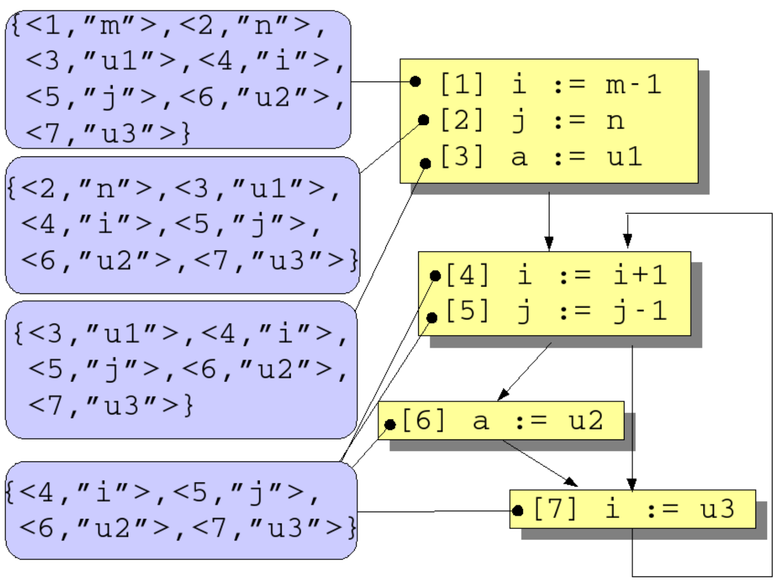
The first equation says that a variable is live coming into a statement if either it is used before redefinition in that statement or it is live coming out of the statement and is not redefined in it. The second equation says that a variable is live coming out of a statement if and only if it is live coming into one of its successors.

This can be expressed in Rascal as follows:

```
equations
  initial
      rel[stat,def] LIN init {}
      rel[stat,def] LOUT init DEF
  satisfy
      LIN = { < S, D> | stat S : STATEMENTS,
                        def D : USE[S] union (LOUT[S] \ (DEF[S]))
            }
      LOUT= { < S, D> | stat S : STATEMENTS,
                        stat Succ : successor(S),
                        def D : LIN[Succ]
            }
end equations
```

The results of live variable analysis for our running example are illustrated in Figure 1.11, “Live variables for flow graph in Figure 1.9, “Flow graph for various dataflow problems”” [45].

Figure 1.11. Live variables for flow graph in Figure 1.9, “Flow graph for various dataflow problems” [41]



Program Slicing

Program slicing is a technique proposed by Weiser [Wei84] for automatically decomposing programs in parts by analyzing their data flow and control flow. Typically, a given statement in a program is selected as the *slicing criterion* and the original program is reduced to an independent subprogram, called a *slice*, that is guaranteed to represent faithfully the behavior of the original program at the slicing criterion. An example will illustrate this:

```
[ 1] read(n)          [ 1] read(n)          [ 1] read(n)
[ 2] i := 1           [ 2] i := 1           [ 2] i := 1
[ 3] sum := 0         [ 3] sum := 0
[ 4] product := 1                           [ 4] product := 1
[ 5] while i<= n do   [ 5] while i<= n do   [ 5] while i<= n do
     begin                 begin                 begin
[ 6]   sum := sum + i [ 6]   sum := sum + i
[ 7]   product :=                           [ 7]   product :=
         product * i                                 product * i
[ 8]   i := i + 1     [ 8]  i := i + 1      [ 8]   i := i + 1
     end                   end                   end
[ 9] write(sum)       [ 9] write(sum)
[10] write(product)                         [10] write(product)

(a) Sample program    (b) Slice for         (c) Slice for
                         statement [9]         statement [10]
```

The initial program is given as (a). The slice with statement [9] as slicing criterion is shown in (b): statements `[4]` and `[7]` are irrelevant for computing statement `[9]` and do not occur in the slice. Similarly, (c) shows the slice with statement `[10]` as slicing criterion. This particular form of slicing is called *backward slicing*. Slicing can be used for debugging and program understanding, optimization and more. An overview of slicing techniques and applications can be found in [Tip95]. Here we will explore a relational formulation of slicing adapted from a proposal in [JR94]J. The basic ingredients of the approach are as follows:

- We assume the relations `PRED`, `DEFS` and `USES` as before.
- We assume an additional set `CONTROL-STATEMENT` that defines which statements are control statements.
- To tie together dataflow and control flow, three auxiliary variables are introduced:
  - The variable `TEST` represents the outcome of a specific test of a conditional statement. The conditional statement defines `TEST` and all statements that are control dependent on this conditional statement will use `TEST`.
  - The variable `EXEC` represents the potential execution dependence of a statement on some conditional statement. The dependent statement defines `EXEC` and an explicit (control) dependence is made between `EXEC` and the corresponding `TEST`.
  - The variable `CONST` represents an arbitrary constant.

The calculation of a (backward) slice now proceeds in six steps:

- Compute the relation `rel[use,def] use-def` that relates all uses to their corresponding definitions. The function `reaching-definitions` as shown earlier in the section called "*Reaching Definitions*" [41]does most of the work.
- Compute the relation `rel[def,use] def-use-per-stat` that relates the *internal* definitions and uses of a statement.

- Compute the relation `rel[def,use] control-dependence` that links all `EXEC`s to the corresponding `TEST`s.
- Compute the relation `rel[use,def] use-control-def` combines use/def dependencies with control dependencies.
- After these preparations, compute the relation `rel[use,use] USE-USE` that contains dependencies of uses on uses.
- The backward slice for a given slicing criterion (a use) is now simply the projection of `USE-USE` for the slicing criterion.

This informal description of backward slicing can now be expressed in Rascal:

```
set[use] BackwardSlice(
    set[stat] CONTROL-STATEMENT,
    rel[stat,stat] PRED,
    rel[stat,var] USES,
    rel[stat,var] DEFS,
    use Criterion)
= USE-USE[Criterion]

where
    rel[stat, def] REACH = reaching-definitions(DEFS, PRED)

    rel[use,def] use-def =
       { <<S1,V>, <S2,V>> | <stat S1, var V> : USES,
                            <stat S2, V> : REACH[S1]
       }

    rel[def,use] def-use-per-stat  =
       {<<S,V1>, <S,V2>> | <stat S, var V1> : DEFS,
                           <S, var V2> : USES
       }
             union
       {<<S,V>, <S,"EXEC">> | <stat S, var V> : DEFS}
             union
       {<<S,"TEST">,<S,V>> | stat S : CONTROL-STATEMENT,
                             <S, var V> : domainR(USES, {S})
       }

    rel[stat, stat] CONTROL-DOMINATOR =
       domainR(dominators(PRED), CONTROL-STATEMENT)

    rel[def,use] control-dependence  =
       { <<S2, "EXEC">,<S1,"TEST">> |
         <stat S1, stat S2> : CONTROL-DOMINATOR
       }

    rel[use,def] use-control-def = use-def union control-dependence

    rel[use,use] USE-USE = (use-control-def o def-use-per-stat)*

endwhere
```

Let's apply this to the example from the start of this section and assume the following:

```
rel[stat,stat] PRED = { <1,2>, <2,3>, <3,4>, <4,5>, <5,6>, <5,9>,
                        <6,7>, <7,8>,<8,5>, <8,9>, <9,10>
```

```
                      }

rel[stat,var] DEFS  = { <1, "n">, <2, "i">, <3, "sum">,
                        <4,"product">, <6, "sum">, <7, "product">,
                        <8, "i">
                      }

rel[stat,var] USES  = { <5, "i">, <5, "n">, <6, "sum">, <6,"i">,
                        <7, "product">, <7, "i">, <8, "i">,
                        <9, "sum">, <10, "product">
                      }

set[int] CONTROL-STATEMENT = { 5 }
```

The result of the slice

```
BackwardSlice(CONTROL-STATEMENT, PRED, USES, DEFS, <9, "sum">)
```

will then be

```
 { <1, "EXEC">, <2, "EXEC">,  <3, "EXEC">, <5, "i">, <5, "n">,
   <6, "sum">, <6, "i">, <6, "EXEC">, <8, "i">, <8, "EXEC">,
   <9, "sum"> }
```

Take the domain of this result and we get exactly the statements in (b) of the example.

Built-in Operators and Library Functions

The built-in operators and library functions can be subdivided in the following categories:

- Benchmark: measuring functions, see the section called “*Benchmark*” [49].
- Boolean: operators and functions on Boolean values, see the section called “*Boolean*” [50].
- Exception: data definition of all soft exceptions that can be caught by Rascal programs, see the section called “*Exception*” [50].
- Graph: graphs are a special kind of binary relation, see the section called “*Graph*” [51].
- Integer: operators and functions on integers, see the section called “*Integer*” [51].
- IO: simple print functions, see the section called “*IO*” [52].
- Labelled Graph: labelled graphs with addition edge information, see the section called “*Labelled Graph*” [52].
- List: operators and functions on lists, see the section called “*List*” [53].
- Location: operators and functions on source locations, see the section called “*Location*” [55].
- Map: operators and functions on maps, see the section called “*Map*” [55].
- Node: operators and functions on nodes, see the section called “*Map*” [55].
- Real: operators and functions on reals, see the section called “*Real*” [57].
- Relation: operators and funcions on relations, see the section called “*Relation*” [57].
- Resource: functions to retrieve resources from an Eclipse workspace, see the section called “*Resource (Eclipse only)*” [59].

- RSF: function for reading files in Rigi Standard Format, see the section called “*RSF*” [59].
- Set: operators and functions on sets, see the section called “*Set*” [59].
- String: operators and functions on strings, see the section called “*String*” [62].
- Tuple: operators and functions on tuples, see the section called “*Tuple*” [63].
- UnitTest: functions for unit testing, see the section called “*UnitTest*” [63].
- ValueIO: functions for reading and writing Rascal values, both in textual and in binary form, see the section called “*ValueIO*” [64].
- View: functions for graphical display of values in Eclipse, see the section called “*View (Eclipse only)*” [64].
- Void: the type void, see

All operators are directly available for each program, but library functions have to be imported in each module that uses them.

We use some notational conventions to describe the argument of operators, as shown in Table 1.3, “Notational conventions” [49]. When an operator has more than one argument of the same type, they are distinguished by subscripts.

Table 1.3. Notational conventions

Argument	**Describes expression of type**
`Bool`	bool
`Int`	int
`Real`	real
`Str`	str
`Loc`	loc
`Node`	node
`List`	Any list type
`Set`	Any set type
`Map`	Any map type
`Tuple`	Any tuple type
`Rel`	Any rel type
`Value`	value
`Elm`	Compatible with element type of list, set, map, relation

Benchmark

Table 1.4. Benchmark Functions

Function	**Description**
`real currentTimeMillis()`	current time in milliseconds since January 1, 1970 GMT.
p.m. benchmark	measure and report the execution time of name:void-closure pairs.

Boolean

Table 1.5. Boolean Operators

Operator	Description
Bool$_1$ == *Bool*$_2$	`true` if both arguments are identical
Bool$_1$!= *Bool*$_2$	`true` if both arguments are not identical
Bool$_1$ <= *Bool*$_2$	`true` if both arguments are identical or *Bool*$_1$ is `false` and *Bool*$_2$ is `true`
Bool$_1$ < *Bool*$_2$	`true` if *Bool*$_1$ is `false` and *Bool*$_2$ is `true`
Bool$_1$ >= *Bool*$_2$	`true` if both arguments are identical or *Bool*$_1$ is `true` and *Bool*$_2$ is `false`
Bool$_1$ > *Bool*$_2$	`true` if *Bool*$_1$ is `true` and *Bool*$_2$ is `false`
Bool$_1$ && *Bool*$_2$	yields `true` if both arguments have the value true and `false` otherwise
Bool$_1$ || *Bool*$_2$	yields `true` if either argument has the value true and `false` otherwise
Bool$_1$ ==> *Bool*$_2$	yields `false` if *Bool*$_1$ has the value `true` and *Bool*$_2$ has value `false`, and `true` otherwise
! *Bool*	yields true if Bool is false and true otherwise
Bool$_1$? *Bool*$_2$: *Bool*$_3$	if *Bool*$_1$ is true then *Bool*$_2$ else *Bool*$_3$

Table 1.6. Boolean Functions

Function	Description
`bool arbBool()`	arbitrary boolean value
`bool fromInt(int i)`	convert an integer to a bool
`bool fromString(str s)`	convert the strings "true" or "false" to a bool
`int toInt(bool b)`	convert a boolean value to integer
`real toReal(bool b)`	convert a boolean value to a real value
`str toString(bool b)`	convert a boolean value to a string

Exception

The following "soft" exceptions are defined:

```
data RuntimeException =
      EmptyList
    | EmptyMap
    | EmptySet
    | IndexOutOfBounds(int index)
    | AssertionFailed
    | AssertionFailed(str label)
    | NoSuchElement(value v)
    | IllegalArgument(value v)
    | IllegalArgument
    | IO(str message)
    | FileNotFound(str filename)
    | LocationNotFound(loc location)
    | PermissionDenied
    | PermissionDenied(str message)
```

```
    | ModuleNotFound(str name)
    | NoSuchKey(value key)
    | NoSuchAnnotation(str label)
    | Java(str message)
    ;
```

Graph

The graph datatype is a special form of binary relation defined as follows:

```
alias graph[&T] = rel[&T from, &T to];
```

Table 1.7. Graph Functions

Function	Description
`set[&T] bottom(graph[&T] G)`	bottom nodes of a graph
`set[&T] top(graph[&T] G)`	top nodes of a graph
`set[&T] reach(graph[&T] G, set[&T] Start)`	Reachability from set of start nodes.
`set[&T] reachR(graph[&T] G, set[&T] Start, set[&T] Restr)`	Reachability from set of start nodes with restriction to certain nodes.
`set[&T] reachX(graph[&T] G, set[&T] Start, set[&T] Excl)`	Reachability from set of start nodes with exclusion of certain nodes
`list[&T] shortestPathPair(graph[&T] G, &T From, &T To)`	Shortest path between pair of nodes

The following examples illustrate these functions:

```
rascal> top({<1,2>, <1,3>, <2,4>, <3,4>});
{1}
rascal> bottom({<1,2>, <1,3>, <2,4>, <3,4>});
{4}
rascal> reachR({1}, {1, 2, 3}, {<1,2>, <1,3>, <2,4>, <3,4>});
{2, 3}
rascal> reachX({1}, {2}, {<1,2>, <1,3>, <2,4>, <3,4>});
{3, 4}
```

Integer

Rascal integers are unbounded in size.

Table 1.8. Integer Operators

Operator	Description
Int_1 == Int_2	`true` if both arguments are numerically equal and `false` otherwise
Int_1 != Int_2	`true` if both arguments are numerically unequal and `false` otherwise
Int_1 <= Int_2	`true` if Int_1 is numerically less than or equal to Int_2 and `false` otherwise
Int_1 < Int_2	`true` if Int_1 is numerically less than Int_2 and `false` otherwise
Int_1 >= Int_2	`true` if Int_1 is numerically greater than or equal than Int_2 and `false` otherwise
Int_1 > Int_2	`true` if Int_1 is numerically greater than Int_2 and `false` otherwise
Int_1 + Int_2	sum of Int_1 and Int_2
Int_1 - Int_2	difference of Int_1 and Int_2
Int_1 * Int_2	Int_1 multiplied by Int_2
Int_1 / Int_2	Int_1 divided by Int_2
Int_1 % Int_2	remainder of dividing Int_1 by Int_2
- *Int*	negate sign of *Int*
Bool ? Int_1 : Int_2	if *Bool* is true then Int_1 else Int_2

Table 1.9. Integer Functions

Function	Description
`int abs(int N)`	absolute value of integer N
`int arbInt()`	arbitrary integer value
`int arbInt(int limit)`	arbitrary integer value in the interval [0, limit)
`int max(int n, int m)`	largest of two integers
`int min(int n, int m)`	smallest of two integers
`real toReal(int n)`	convert an integer value to a real value
`str toString(int n)`	convert an integer value to a string

IO

Table 1.10. IO Functions

Function	Description
`void java println(value V...)`	print a list of values on the output stream
`list[str] readFile(str filename) throws NoSuchFileError(str msg), IOError(str msg)`	read a named file as list of strings

Labelled Graph

The labelled graph datatype is a special form of binary relation with labelled edges and is defined as follows:

```
alias lgraph[&T,&L] = rel[&T from, &L label, &T to];
```

Table 1.11. Labelled Graph Functions

Function	Description
`set[&T] bottom(lgraph[&T] G)`	bottom nodes of a labelled graph
`set[&T] top(lgraph[&T] G)`	top nodes of a labelled graph
`set[&T] reach(lgraph[&T] G, set[&T] Start)`	Reachability from set of start nodes.
`set[&T] reachR(lgraph[&T] G, set[&T] Start, set[&T] Restr)`	Reachability from set of start nodes with restriction to certain nodes.
`set[&T] reachX(lgraph[&T] G, set[&T] Start, set[&T] Excl)`	Reachability from set of start nodes with exclusion of certain nodes
`list[&T] shortestPathPair(lgraph[&T] G, &T From, &T To)`	Shortest path between pair of nodes

Warning

shortestPath not yet implemented for lgraph.

List

Table 1.12. List Operators

Operator	Description
$List_1$ == $List_2$	`true` if both arguments have the same elements in the same order
$List_1$!= $List_2$	`true` if both arguments have different elements
$List_1$ <= $List_2$	`true` if both sists are equal or $List_1$ is a sublist of $List_2$
$List_1$ < $List_2$	`true` if $List_1$ is a sublist of $List_2$
$List_1$ >= $List_2$	`true` if both lists are equal or $List_2$ is a sublist of $List_1$
$List_1$ > $List_2$	`true` if $List_2$ is a sublist of $List_1$
$List_1$ + $List_2$	concatenation of $List_1$ and $List_2$
$List_1$ - $List_2$	list consisting of all elements in $List_1$ that do not occur in $List_2$
$List_1$ * $List_2$	$List_1$ multiplied by $List_2$
Elm `in` *List*	`true` if *Elm* occurs as element in *List*
Elm `notin` *List*	`true` if *Elm* does not occur as element in *List*
Bool ? $List_1$: $List_2$	if *bool* is true then $List_1$ else $List_2$
List [*int*]	element at position *int* in *List*

Table 1.13. List Functions

Function	Description
`&T average(list[&T] lst, &T zero)`	average of elements of a list
`list[&T] delete(list[&T] lst, int n)`	delete nth element from list
`set[int] domain(list[&T] lst)`	a set of all legal index values for a list
`&T java head(list[&T] lst) throws EmptyListError`	get the first element of a list
`list[&T] head(list[&T] lst, int n) throws IndexOutOfBoundsError`	get the first n elements of a list
`&T getOneFrom(list[&T] lst)`	get an arbitrary element from a list
`list[&T] insertAt(list[&T] lst, int n, &T elm) throws IndexOutOfBoundsError`	add an element at a specific position in a list
`bool isEmpty(list[&T] lst)`	is list empty?
`list[&T] mapper(list[&T] lst, &T (&T) fn)`	apply a function to each element of a list
`&T max(list[&T] lst)`	largest element of a list
`&T min(list[&T] lst)`	smallest element of a list
`&T multiply(list[&T] lst, &T unity)`	multiply the elements of a list
`list[list[&T]] permutations(list[&T] lst)`	all permutations of a list
`&T reducer(list[&T] lst, &T (&T, &T) fn, &T unit)`	apply function F to successive elements of a list
`list[&T] reverse(list[&T] lst)`	elements of a list in reverse order
`int size(list[&T] lst)`	number of elements in a list
`list[&T] slice(list[&T] lst, int start, int len)`	sublist from start of length len
`list[&T] sort(list[&T] lst)`	sort the elements of a list
`&T sum(list[&T] lst, &T zero)`	add elements of a List
`list[&T] tail(list[&T] lst)`	all but the first element of a list
`list[&T] tail(list[&T] lst, int len) throws IndexOutOfBoundsError`	last n elements of a list
`tuple[&T, list[&T]] takeOneFrom(list[&T] lst)`	remove an arbitrary element from a list, returns the element and the modified list
`map[&A,&B] toMap(list[tuple[&A, &B]] lst)`	convert a list of tuples to a map
`set[&T] toSet(list[&T] lst)`	convert a list to a set
`str toString(list[&T] lst)`	convert a list to a string

Location

Table 1.14. Operations on Locations

Operator	Description
`Loc`$_1$ == `Loc`$_2$	`true` if both arguments are identical and `false` otherwise
`Loc`$_1$!= `Loc`$_2$	`true` if both arguments are not identical and `false` otherwise
`Loc`$_1$ <= `Loc`$_2$	`true` if `Loc`$_1$ is textually contained in or equal to `Loc`$_2$ and `false` otherwise
`Loc`$_1$ < `Loc`$_2$	`true` if `Loc`$_1$ is strictly textually contained in `Loc`$_2$ and `false` otherwise
`Loc`$_1$ >= `Loc`$_2$	`true` if `Loc`$_1$ is textually encloses or is equal to `Loc`$_2$ and `false` otherwise
`Loc`$_1$ >= `Loc`$_2$	`true` if `Loc`$_1$ is textually encloses `Loc`$_2$ and `false` otherwise
`Loc . Field`	retrieve one of the fields of location value

The field names for locations are:

- `url`
- `offset`
- `length`
- `beginLine`, `beginColumn`
- `endLine`, `endColumn`.

Map

Table 1.15. Map Operators

Operator	Description
`Map`$_1$ == `Map`$_2$	`true` if both arguments consist of the same pairs
`Map`$_1$!= `Map`$_2$	`true` if both arguments have different pairs
`Map`$_1$ <= `Map`$_2$	`true` if all pairs in `Map`$_1$ occur in `Map`$_2$ or `Map`$_1$ and `Map`$_2$ are equal
`Map`$_1$ < `Map`$_2$	`true` if all pairs in `Map`$_1$ occur in `Map`$_2$ but `Map`$_1$ and `Map`$_2$ are not equal
`Map`$_1$ >= `Map`$_2$	`true` if all pairs in `Map`$_2$ occur in `Map`$_1$ or `Map`$_1$ and `Map`$_2$ are equal
`Map`$_1$ > `Map`$_2$	`true` if all pairs in `Map`$_2$ occur in `Map`$_1$but `Map`$_1$ and `Map`$_2$ are not equal
`Map`$_1$ + `Map`$_2$	union of `Map`$_1$ and `Map`$_2$
`Map`$_1$ - `Map`$_2$	difference of `Map`$_1$ and `Map`$_2$
`Key in Map`	true if `Key` occurs in a key:value pair in `Map`
`Key`$_1$ `notin` `Map`$_2$	true if `Key` does not occur in a key:value pair in map
`Bool ?` `Map`$_1$: `Map`$_2$	if `Bool` is true then `Map`$_1$ else `Map`$_2$
`Map [ Key ]`	the value associated with `Key` in `Map` if that exists, undefined otherwise

Table 1.16. Map Functions

Function	Description
`set[&K] domain(map[&K, &V] M)`	the domain (keys) of a map
`&K getOneFrom(map[&K, &V] M)`	arbitrary key of a map
`map[&V, &K] invert(map[&K, &V] M)`	map with key and value inverted
`bool isEmpty(map[&K, &V] M)`	is map empty?
`map[&K, &V] mapper(map[&K, &V] M, &K (&K) F, &V (&V) G)`	apply two functions to each key/value pair in a map.
`set[&V] range(map[&K, &V] M)`	the range (values) of a map
`int size(map[&K, &V] M)`	number of elements in a map.
`list[tuple[&K, &V]] toList(map[&K, &V] M)`	convert a map to a list
`rel[&K, &V] toRel(map[&K, &V] M)`	convert a map to a relation
`str toString(map[&K, &V] M)`	convert a map to a string.

Node

Table 1.17. Node Operators

Operator	Description
Node$_1$ == *Node*$_2$	`true` if both arguments are identical
Node$_1$!= *Node*$_2$	`true` if both arguments are not identical
Node$_1$ <= *Node*$_2$	
Node$_1$ < *Node*$_2$	
Node$_1$ >= *Node*$_2$	
Node$_1$ > *Node*$_2$	
Bool ? *Node*$_1$: *Node*$_2$	if *Bool* is true then *Node*$_1$ else *Node*$_2$
Node [*Int*]	child of *Node* at position *Int*

Table 1.18. Node Functions

Function	Description
`int arity(node T)`	number of children of a node
`list[value] getChildren(node T)`	get the children of a node
`str getName(node T)`	get the function name of a node
`node makeNode(str N, value V...)`	create a node given its function name and arguments

Real

Table 1.19. Real Operators

Operator	Description
$Real_1 == Real_2$	`true` if both arguments are numerically equal and `false` otherwise
$Real_1\ != Real_2$	`true` if both arguments are numerically unequal and `false` otherwise
$Real_1 <= Real_2$	`true` if $Real_1$ is numerically less than or equal to $Real_2$ and `false` otherwise
$Real_1 < Real_2$	`true` if $Real_1$ is numerically less than $Real_2$ and `false` otherwise
$Real_1 >= Real_2$	`true` if $Real_1$ is numerically greater than or equal than $Real_2$ and `false` otherwise
$Real_1 > Real_2$	`true` if $Real_1$ is numerically greater than $Real_2$ and `false` otherwise
$Real_1 + Real_2$	sum of $Real_1$ and $Real_2$
$Real_1 - Real_2$	difference of $Real_1$ and $Real_2$
$Real_1 * Real_2$	$Real_1$ multiplied by $Real_2$
$Real_1 / Real_2$	$Real_1$ divided by $Real_2$
$-Real$	negate sign of $Real$
$Real_1 \% Real_2$	remainder of dividing $Real_1$ by $Real_2$
$Bool\ ?\ Real_1 : Real_2$	if $Bool$ is true then $Real_1$ else $Real_2$

Table 1.20. Real Functions

Function	Description
`real arbReal()`	an arbitrary real value in the interval [0.0,1.0).
`real max(real n, real m)`	largest of two reals
`int toInteger(real d)`	convert a real to integer.
`str toString(real d)`	convert a real to a string.

Relation

Relation are sets of tuples, therefore all set operators (see, Table 1.25, “Set Operators”[60]) apply to relations as well

Table 1.21. Operations on Relations

Operator	Description
$Rel_1 \circ Rel_2$	yields the relation resulting from the composition of the two arguments
$Set_1 \times Set_2$	yields the relation resulting from the Cartesian product of the two arguments
$Rel +$	yields the relation resulting from the transitive closure of Rel
$Rel *$	yields the relation resulting from the reflexive transitive closure of Rel
$Rel\ [\ elem\ ]$	yields the right image of Rel
$Rel\ [\ set\ ]$	yields the right image of Rel
$Rel < Index_1, Index_2, \ldots >$	

Examples:

```
{<1,10>, <2,20>, <3,15>} o {<10,100>, <20,200>} yields {<1,100>, <2,200>}.


{1, 2, 3} x {9} yields {<1, 9>, <2, 9>, <3, 9>}.



Rel has value {<1,10>, <2,20>, <1,11>, <3,30>, <2,21>} in the following example

Rel[1] yields {10, 11}.

Rel[{1}] yields {10, 11}.

Rel[{1, 2}] yields {10, 11, 20, 21}.
```

Table 1.22. Relation Functions

Function	Description
`set[&T] carrier (rel[&T,&T] R)`	all elements in any tuple in a relation
`rel[&T,&T] carrierR (rel[&T,&T] R, set[&T] S)`	relation restricted to tuples with elements in a set S
`rel[&T,&T] carrierX (rel[&T,&T] R, set[&T] S)`	relation excluded tuples with some element in S
`rel[&T0, &T1] complement(rel[&T0, &T1] R)`	complement of relation
`set[&T0] domain (rel[&T0,&T1] R)`	first element of each tuple in binary relation
`rel[&T0,&T1] domainR (rel[&T0,&T1] R, set[&T0] S)`	restriction of a relation to tuples with first element in S}
`rel[&T0,&T1] domainX (rel[&T0,&T1] R, set[&T0] S)`	relation excluded tuples with first element in S
ident?	
`rel[&T1, &T0] invert (rel[&T0, &T1] R)`	inverse the tuples in a relation
`rel[&T2, &T1, &T0] invert (rel[&T0, &T1, &T2] R)`	all but the first element of each tuples in binary relation
`rel[&T0,&T1] rangeR (rel[&T0,&T1] R, set[&T2] S)`	restriction of a binary relation to tuples with second element in set S

Examples:

```
id({1,2,3}) yields {<1,1>, <2,2>, <3,3>}.

id({"mon", "tue", "wed"}) yields {<"mon","mon">, <"tue","tue">, <"wed","wed">}.
inv({<1,10>, <2,20>}) yields {<10,1>,<20,2>}.

compl({<1,10>} yields {<1, 1>, <10, 1>, <10, 10>}.

domain({<1,10>, <2,20>}) yields {1, 2}.

domain({<"mon", 1>, <"tue", 2>}) yields {"mon", "tue"}.
```

```
range({<1,10>, <2,20>}) yields {10, 20}.

range({<"mon", 1>, <"tue", 2>}) yields {1, 2}.

carrier({<1,10>, <2,20>}) yields {1, 10, 2, 20}

domainR({<1,10>, <2,20>, <3,30>}, {3, 1});
{<1,10>, <3,30>}.


rangeR({<1,10>, <2,20>, <3,30>}, {30, 10}) yields {<1,10>, <3,30>}.

carrierR({<1,10>, <2,20>, <3,30>}, {10, 1, 20}) yields {<1,10>}.

domainX({<1,10>, <2,20>, <3,30>}, {3, 1}) yields {<2, 20>}.
rangeX({<1,10>, <2,20>, <3,30>}, {30, 10}) yields {<2, 20>}.

carrierX({<1,10>, <2,20>, <3,30>}, {10, 1, 20}) yields {<3,30>}.
```

RSF

Table 1.23. RSF Functions

Function	Description
`map[str, rel[str,str]] readRSF(str nameRSFFile)`	read a file in Rigi Standard Format (RSF).

Resource (Eclipse only)

```
data Resource = root(set[Resource] projects)
              | project(str name, set[Resource] contents)
              | folder(str name, set[Resource] contents)
              | file(str name, str extension);
```

Table 1.24. Resource Functions

Function	Description
`Resource java root()`	The root of the Eclipse workspace
`set[str] java projects()`	The projects in the Eclipse workspace
`set[str] java references(str project)`	The project references of a given project
`loc java location(str project)`	Source location of given project
`set[loc] java files(str project)`	The files contained in a project

Set

Table 1.25. Set Operators

Operator	Description
Set_1 == Set_2	`true` if both arguments are equal sets and `false` otherwise
Set_1 != Set_2	`true` if both arguments are unequal sets and `false` otherwise
Set_1 <= Set_2	`true` if Set_1 is a subset of Set_2 and `false` otherwise
Set_1 < Set_2	`true` if Set_1 is a strict subset of Set_2 and `false` otherwise
Set_1 >= Set_2	`true` if Set_1 is a superset of Set_2 and `false` otherwise
Set_1 > Set_2	`true` if Set_1 is a strict superset of Set_2 and `false` otherwise
Set_1 + Set_2	set resulting from the union of the two arguments
Set_1 - Set_2	the set resulting from the difference of the two arguments
Set_1 * Set_2	set resulting from the product of the two arguments
Set_1 & Set_2	set resulting from the intersection of the two arguments
Elm `in` *Set*	`true` if *Elm* occurs as element in *Set* and `false` otherwise
Elm `notin` *Set*	`false` if *Elm* occurs as element in *Set* and `false` otherwise
Set_1 `join` Set_2	
Bool ? Set_1 : Set_2	

Examples:

```
rascal> {1, 2, 3} + {4, 5, 6};
{1, 2, 3, 4, 5, 6}

rascal> {1, 2, 3} + {1, 2, 3};
{1, 2, 3}

rascal> {1, 2, 3, 4} - {1, 2, 3};
{4}

rascal> {1, 2, 3} - {4, 5, 6};
{1, 2, 3}

rascal> {1, 2, 3} & {4, 5, 6};
{ }

rascal> {1, 2, 3} & {1, 2, 3};
{1, 2, 3}

rascal> 3 in {1, 2, 3};
true

rascal> 4 in {1, 2, 3};
false

rascal> 3 notin {1, 2, 3};
false

racsal> 4 notin {1, 2, 3};
true

rascal> <2,20> in {<1,10>, <2,20>, <3,30>};
true
```

```
rascal> <4,40> notin {<1,10>, <2,20>, <3,30>};
true
```

Table 1.26. Set Functions

Function	Description
`&T average(set[&T] st, &T zero)`	average of the elements of a set
`&T getOneFrom(set[&T] st)`	pick a random element from a set
`bool isEmpty(set[&T] st)`	Is set empty?
`set[&T] mapper(set[&T] st, &T (&T,&T) fn)`	apply a function to each element of a setacter
`&T max(set[&T] st)`	largest element of a set
`&T min(set[&T] st)`	smallest element of a set
`&T multiply(set[&T] st, &T unity)`	multiply the elements of a set
`set[set[&T]] power(set[&T] st)`	all subsets of a set
`set[set[&T]] power1(set[&T] st)`	all subsets (excluding empty set) of a set
`&T reducer(set[&T] st, &T (&T,&T) fn, &T unit)`	apply function F to successive elements of a set
`int size(set[&T] st)`	number of elements in a set
`&T sum(set[&T] st, &T zero)`	add the elements of a set
`tuple[&T, set[&T]] takeOneFrom(set[&T] st)`	remove an arbitrary element from a set, returns the element and the modified set
`list[&T] toList(set[&T] st)`	convert a set to a list
`map[&A,&B] toMap(rel[&A, &B] st)`	convert a set of tuples to a map
`str toString(set[&T] st)`	convert a set to a string

Examples:

```
rascal> power({1, 2, 3, 4});
{ {}, {1}, {2}, {3}, {4},{1,2}, {1,3}, {1,4}, {2,3}, {2,4},
  {3,4}, {1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}, {1,2,3,4}
}

rascal> power1({1, 2, 3, 4});
{ {1}, {2}, {3}, {4},{1,2}, {1,3}, {1,4}, {2,3}, {2,4},
  {3,4}, {1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}, {1,2,3,4}
}

rascal> size({1,2,3});
3

rascal> size({<1,10>, <2,20>, <3,30>});
3
```

String

Table 1.27. Operations on Strings

Operator	Description
`Str`$_1$ == `Str`$_2$	yields `true` if both arguments are equal and `false` otherwise
`Str`$_1$!= `Str`$_2$	yields `true` if both arguments are unequal and `false` otherwise
`Str`$_1$ <= `Str`$_2$	yields `true` if `Str`$_1$ is lexicographically less than or equal to `Str`$_2$ and `false` otherwise
`Str`$_1$ < `Str`$_2$	yields `true` if `Str`$_1$ is lexicographically less than `Str`$_2$ and `false` otherwise
`Str`$_1$ >= `Str`$_2$	yields `true` if `Str`$_1$ is lexicographically greater than or equal to `Str`$_2$ and `false` otherwise
`Str`$_1$ > `Str`$_2$	yields `true` if `Str`$_1$ is lexicographically greater than `Str`$_2$ and `false` otherwise
`Str`$_1$ + `Str`$_2$	concatenates `Str`$_1$ and `Str`$_2$
`Bool` ? `Str`$_1$: `Str`$_2$	if `Bool` is true then `Str`$_1$ else `Str`$_2$

Table 1.28. String Functions

Function	Description
`int charAt(str s, int i) throws out_of_range(str msg)`	character at position i in string s.
`bool endsWith(str s, str suffix)`	true if string s ends with given string suffix.
`str center(str s, int n)`	center s in string of length n using spaces
`str center(str s, int n, str pad)`	center s in string of length n using a pad character
`bool isEmpty(str s)`	is string empty?
`str left(str s, int n)`	left align s in string of length n using spaces
`str left(str s, int n, str pad)`	left align s in string of length n using pad character
`str right(str s, int n)`	right align s in string of length n using spaces
`str reverse(str s)`	string with all characters in reverse order.
`int size(str s)`	the length of string s.
`bool startsWith(str s, str prefix)`	true if string s starts with the string prefix.
`str toLowerCase(str s)`	convert all characters in string s to lowercase.
`str toUpperCase(str s)`	convert all characters in string s to uppercase.

Tuple

Table 1.29. Tuple Operators

Operator	Description
$Tuple_1$ == $Tuple_2$	`true` if both arguments are identical
$Tuple_1$!= $Tuple_2$	`true` if both arguments are not identical
$Tuple_1$ <= $Tuple_2$	`true` if both arguments are identical or if the leftmost element in $Tuple_1$ that differs from the corresponding in $Tuple_2$ is smaller than that element in $Tuple_2$
$Tuple_1$ < $Tuple_2$	`true` if both arguments are not identical and the leftmost element in $Tuple_1$ that differs from the corresponding in $Tuple_2$ is smaller than that element in $Tuple_2$
$Tuple_1$ >= $Tuple_2$	`true` if both arguments are identical or if the leftmost element in $Tuple_1$ that differs from the corresponding in $Tuple_2$ is greater than that element in $Tuple_2$
$Tuple_1$ > $Tuple_2$	`true` if both arguments are not identical and the leftmost element in $Tuple_1$ that differs from the corresponding in $Tuple_2$ is greater than that element in $Tuple_2$
$Tuple_1$ + $Tuple_2$	concatenates $Tuple_1$ and $Tuple_2$
Bool ? $Tuple_1$: $Tuple_2$	if *Bool* is true then $Tuple_1$ else $Tuple_2$
Tuple . *Name*	select field *Name* from *Tuple*
Tuple [*Int*]	select field at position *int* from *Tuple*

UnitTest

We provided a very rudimentary library for unit testing that will certainly evolve over time:

Table 1.30. UnitTest Functions

Function	Description
`void assertTrue(bool outcome)`	check that outcome is true
`void assertEqual(value V1, value V2)`	check that two values are equal
`bool report()`	print unit test summary
`bool report(str msg)`	print unit test summary, including msg

Value

Table 1.31. Value Operators

Operator	Description
Value$_1$ == *Value*$_2$	`true` if both arguments are identical
Value$_1$!= *Value*$_2$	`true` if both arguments are not identical
Value$_1$ <= *Value*$_2$	
Value$_1$ < *Value*$_2$	
Value$_1$ >= *Value*$_2$	
Value$_1$ > *Value*$_2$	
Bool ? *Value*$_1$: *Value*$_2$	if *Bool* is true then *Value*$_1$ else *Value*$_2$

ValueIO

Table 1.32. ValueIO Functions

Function	Description
`value readValueFromBinaryFile(str namePBFFile)`	read a value from a binary file in PBF format
`value readValueFromTextFile(str namePBFFile)`	read a value from a text file
`void writeValueToBinaryFile(str namePBFFile, value val)`	write a value to a binary file in PBF format
`void writeValueToTextFile(str namePBFFile, value val)`	write a value to a binary file in PBF format

View (Eclipse only)

Table 1.33. View Functions

Function	Description
`void show(value v)`	Show value v in a graphical viewer
`void browse(value v)`	Show value v a graphical browser

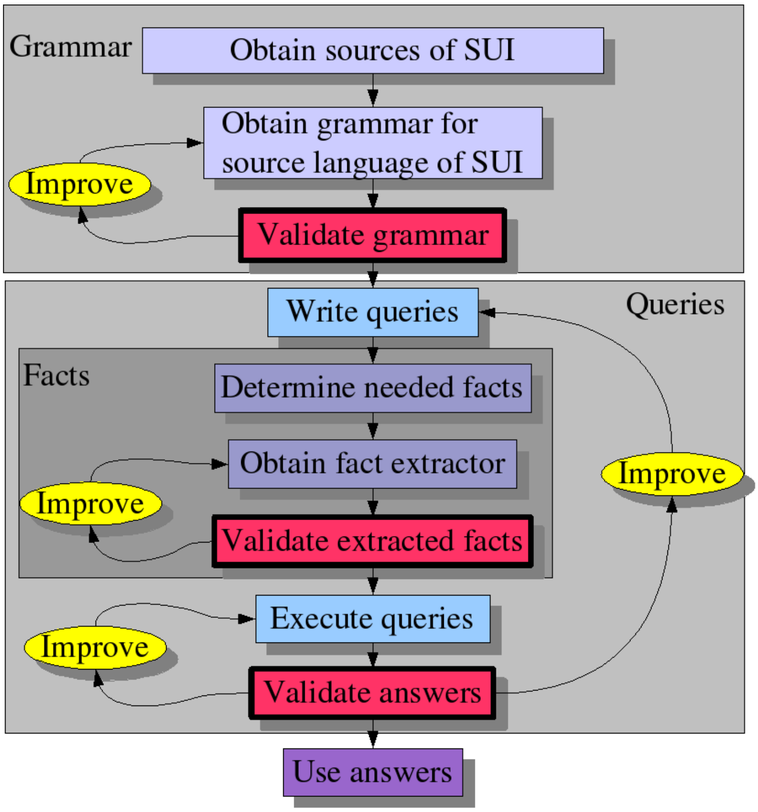
Void

There are no operators or functions defined on the type `void`.

Extracting Facts from Source Code (TODO)

In this tutorial we have, so far, concentrated on querying and enriching facts that have been extracted from source code. As we have seen from the examples, once these facts are available, a concise Rascal program suffices to do the required processing. But how is fact extraction achieved and how difficult is it? To answer these questions we first describe the workflow of the fact extraction process (the section called "*Workflow for Fact Extraction*" [65]) and give strategies for fact extraction (the section called "*Strategies for Fact Extraction*" [66]).

Figure 1.12. Workflow for fact extraction



Workflow for Fact Extraction

Figure 1.12, "Workflow for fact extraction"[65] shows a typical workflow for fact extraction for a *System Under Investigation* (SUI). It assumes that the SUI uses only *one* programming language and that you need only one grammar. In realistic cases, however, several such grammars may be needed. The workflow consists of three main phases:

- Grammar: Obtain and improve the grammar for the source language of the SUI.
- Facts: Obtain and improve facts extracted from the SUI.
- Queries: Write and improve queries that give the desired answers.

Of course, it may happen that you have a lucky day and that extracted facts are readily available or that you can reuse a good quality fact extractor that you can apply to the SUI. On ordinary days you have the above workflow as fall-back. It may come as a surprise that there is such a strong emphasis on validation in this workflow. The reason is that the SUI is usually a huge system that defeats manual inspection. Therefore we must be very careful that we validate the outcome of each phase.

Grammar. In many cases there is no canned grammar available that can be used to parse the programming language dialect used in the SUI. Usually an existing grammar can be adjusted to that dialect, but then it is then mandatory to validate that the adjusted grammar can be used to parse the sources of the SUI.

Facts. It may happen that the facts extracted from the source code are *wrong*. Typical error classes are:

- Extracted facts are *wrong*: the extracted facts incorrectly state that procedure `P` calls procedure `Q` but this is contradicted by a source code inspection.

- Extracted facts are *incomplete*: the inheritance between certain classes in Java code is missing.

The strategy to validate extracted facts differ per case but here are three strategies:

- Postprocess the extracted facts (using Rascal, of course) to obtain trivial facts about the source code such as total lines of source code and number of procedures, classes, interfaces and the like. Next validate these trivial facts with tools like `wc` (word and line count), `grep` (regular expression matching) and others.

- Do a manual fact extraction on a small subset of the code and compare this with the automatically extracted facts.

- Use another tool on the same source and compare results whenever possible. A typical example is a comparison of a call relation extracted with different tools.

Queries. For the validation of the answers to the queries essentially the same approach can be used as for validating the facts. Manual checking of answers on random samples of the SUI may be mandatory. It also happens frequently that answers inspire new queries that lead to new answers, and so on.

Strategies for Fact Extraction

The following global scenario's are available when writing a fact extractor:

- *Dump-and-Merge*: Parse each source file, extract the relevant facts, and return the resulting (partial) Rstore. In a separate phase, merge all the partial Rstores into a complete Rstore for the whole SUI. The tool {\tt merge-rstores} is available for this.

- *Extract-and-Update*: Parse each source file, extract the relevant facts, and add these directly to the partial Rstore that has been constructed for previous source files.

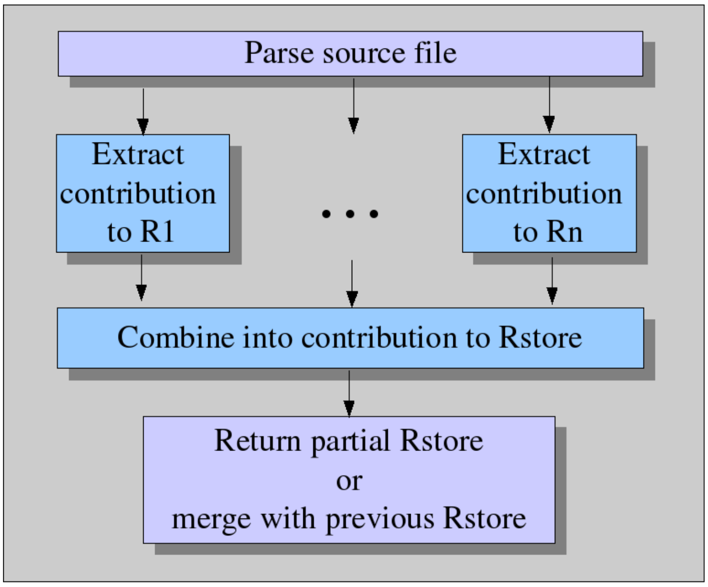
The experience is that the *Extract-and-Update* is more efficient. A second consideration is the scenario used for the fact extraction per file. Here there are again two possibilities:

- *All-in-One*: Write one function that extracts all facts in one traversal of the source file. Typically, this function has an Rstore as argument and returns an Rstore as well. During the visit of specific language constructs additions are made to named sets or relations in the Rstore.

- *Separation-of-Concerns*: Write a separate function for each fact you want to extract. Typically, each function takes a set or relation as argument and returns an updated version of it. At the top level all these functions are called and their results are put into an Rstore. This strategy is illustrated in Figure 1.13, “Separation-of-Concerns strategy for fact extraction” [67].

The experience here is that everybody starts with the *All-in-One* strategy but that the complexities of the interactions between the various fact extraction concerns soon start to hurt. The advice is therefore to use the *Separation-of-Concerns* strategy even if it may be seem to be less efficient since it requires a traversal of the source program for each extracted set or relation.

Figure 1.13. Separation-of-Concerns strategy for fact extraction



Fact Extraction using ASF+SDF

Although facts can be extracted in many ways, ASF+SDF is the tool of preference to do this. Examples are given In XXX.

Warning

Fix reference

Concluding remarks

It is not unusual that the effort that is needed to write a fact extractor is much larger than the few lines of Rascal that are sufficient for the further processing of these facts. What can we learn from this observation? First, that even in simple cases fact extraction is more complicated than the processing of these facts. This may be due to the following:

- The facts we are interested in may be scattered over different language constructs. This implies that the fact extractor has to cover all these cases.
- The extracted facts are completely optimized for relational processing but places a burden on the fact extractor to perform this optimization.

Second, that several research questions remain unanswered:

- Is it possible to solve (parts of) the fact extraction in a language-parametric way. In other words, is it possible to define generic extraction methods that apply to multiple languages?
- Is a further integration of fact extraction with relational processing desirable? Is it, for instance, useful to bring some of the syntactic program domains like expressions and statements to the relational domain?

Table of Built-in Operators

Table 1.34. All Operators

Operator	Description	See
Exp_1 [*Name* = Exp_2]	Field assignment	
Exp . *Name*	Field selection	
Exp < *field*, ... >	Field projection	
Exp_1 [Exp_2]	Index	
Exp ?	Isdefined: true is Exp has a well-defined value	
! *Exp*	Negation	
- *Exp*	Negation	
Exp +	Transitive closure	
Exp *	Reflexive transitive	
Exp @ Name	Attribute value	
Exp_1 [@ Name = Exp_2]	Assign attribute value	
Exp_1 o Exp_2	Composition	
Exp_1 / Exp_2	Division	
Exp_1 % Exp_2	Modulo	
Exp_1 * Exp_2	Multiplication/product	
Exp_1 & Exp_2	Intersection	
Exp_1 + Exp_2	Addition/concatenation/ union	
Exp_1 - Exp_2	Subtraction/difference	
Exp_1 join Exp_2	Join	
Exp_1 in Exp_2	Element of	
Exp_1 notin Exp_2	Not element of	
Exp_1 <= Exp_2	Less than/sublist /subset	
Exp_1 < Exp_2	Less than/strict sublist/ strict subset	
Exp_1 >= Exp_2	Greater than/superlist/ superset	
Exp_1 > Exp_2	Greater than/strict superlist/ strict superset	
Pat := *Exp*	Match	
Pat !:= *Exp*	No Match	
Exp_1 == Exp_2	Equality	
Exp_1 != Exp_2	Inequality	
Exp_1 ? Exp_2	Ifdefined Otherwise	
Exp_1 ? Exp_2 : Exp_3	Conditional Expression	
Exp_1 ==> Exp_2	Implication	
Exp_1 <==> Exp_2	Equivalence	
Exp_1 && Exp_2	Boolean and	
Exp_1 || Exp_2	Boolean or	

Table of Built-in Functions

Table 1.35. All Functions (part I)

Operator	Module	See
`abs`	Integer	
`arbBool`	Boolean	
`arbInt`	Integer	
`arbReal`	Real	
`arity`	Node	
`assertEqual`	UnitTest	
`assertTrue`	UnitTest	
`average`	List, Set	
`bottom`	Graph, LabelledGraph	
`browse`	View	
`carrier`	Relation	
`carrierR`	Relation	
`carrierX`	Relation	
`center`	String	
`charAt` ? remove	String	
`complement`	Relation	
`currentTimeMillis`	Benchmark	
`delete`	List	
`domain`	List, Map, Relation	
`domainR`	Relation	
`domainX`	Relation	
`endsWith`	String	
`files`	Resource	
`fromInt`	Boolean	
`fromString`	Boolean	

Table 1.36. All Functions (part II)

Operator	Module	See
`getChildren`	Node	
`getName`	Node	
`getOneFrom`	List, Map, Set	
`head`	List	
`insertAt`	List	
`invert`	Map, Relation	
`isEmpty`	List, Map, Set, String	
`left`	String	
location ?name	Resource	
`makeNode`	Node	
`mapper?`	Map, Set	
`max`	Integer, List, Real, Set	
`min`	Integer, List, Real?, Set	
`multiply`	List, Set	
`permutations`	List	
`power`	Set	
`power1`	Set	
`println`	IO	
`projects`	Resource	
`range`	Map	
`reach`	Graph, LabelledGraph	
`reachR`	Graph, LabelledGraph	
`reachX`	Graph, LabelledGraph	
`readFile`	IO	
`readRSF`	RSF	

Table 1.37. All Functions (part III)

Operator	Module	See
`readValueFromBinaryFile`	ValueIO	
`readValueFromTextFile`	ValueIO	
`reducer`	List, Set	
`references`	Resource	
`report`	UnitTest	
`reverse`	List, String	
`right`	String	
`root`	Resource	
`shortestPathPair`	Graph, LabelledGraph	
`show`	View	
`size`	List?, Map, Set, String	
`slice`	List	
`sort`	List	
`startsWith`	String	
`sum`	List, Set	
`tail`	List	
`takeOneFrom`	List, Set	
`toInt`	Boolean	
`toInteger!`	Real	
`toList`	Map, Set	
`toLowerCase`	String	
`toMap`	List, Set	
`top`	Graph, LabelledGraph	
`toReal`	Boolean, Integer	
`toRel`	Map	
`toSet`	List	
`toString`	Boolean, Integer, List, Map, Real, Set	
`toUpperCase`	String	
`writeValueToBinaryFile`	ValueIO	
`writeValueToTextFile`	ValueIO	

Bibliography

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley. 1986.

[BNL03] D. Beyer, A Noack, and C. Lewerentz. *Simple and efficient relational querying of software structures*. Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE 2003). . 2003. To appear.

[KN96] E. Koutsofios and S.C. North. *Drawing graphs with dot*. Technical report. AT&T Bell Laboratories. Murray Hill, NJ. 1996. See also www.graphviz.org.

[FKO98] L.M.G. Feijs, R. Krikhaar, and R.C. Ommering. *A relational approach to support software architecture analysis*. 371--400. *Software Practice and Experience*. 28. 4. april 1998.

[Hol96] R.C. Holt. *Binary relational algebra applied to software architecture*. CSRI345. University of Toronto. march 1996.

[JR94] D.J. Jackson and E.J. Rollins. *A new model of program dependences for reverse engineering*. 2--10. Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering. . ACM SIGSOFT Software Engineering Notes.
<seriesvolnum>19</seriesvolnum>
1994.

[Kli03] P. Klint. *How understanding and restructuring differ from compiling---a rewriting perspective*. 2--12. Proceedings of the 11th International Workshop on Program Comprehension (IWPC03). . 2003. IEEE Computer Society.

[Kri99] R.L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis. University of Amsterdam. 1999.

[McC76] T.J. McCabe. *A complexity measure*. 308--320. *IEEE Transactions on Software Engineering*. SE-12. 3. 1976.

[MK88] H. Müller and K. Klashinsky. *Rigi -- a system for programming-in-the-large.*. 80--86,. Proceedings of the 10th International Conference on Software Engineering (ICSE 10),. . April 1988.

[Tip95] F. Tip. *A survey of program slicing techniques*. 121--189. *Journal of Programming Languages*. 3. 3. 1995.

[Wei84] M. Weiser. *Program slicing*. 352--357. *IEEE Transactions on Software Engineering*. SE-10. 4. July 1984.