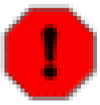
Chapter 1. EASY Meta-Programming with Rascal

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

Paul Klint, Jurgen Vinju, Tijs van der Storm

Fri Jun 26 23:31:43 CEST 2009

Table of Contents



Rascal is a work in progress both regarding implementation and documentation. The current version of this document is a preview version only. Comments labelled "Warning" (like this one) or other *remarks* (like this one) are temporary notes that will disappear in the final version.

A New Language for Meta-Programming

Meta-programs are programs that analyze, transform or generate other programs. Ordinary programs work on data; meta-programs work on programs. The range of programs to which meta-programming can be applied is large: from programs in standard languages like C and Java to domain-specific languages for describing high-level system models or applications in specialized areas like gaming or finance. In some cases, even test results or performance data are used as input for meta-programs.

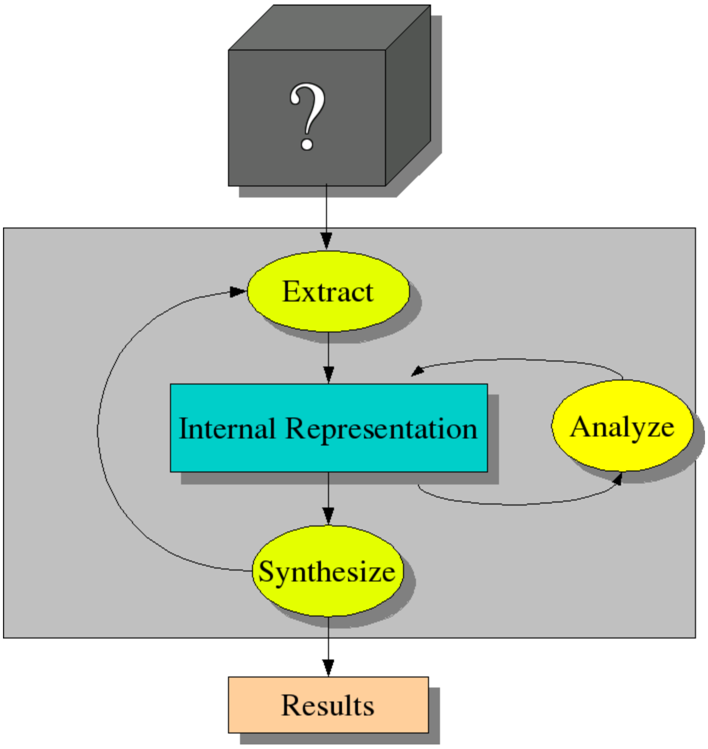
Rascal is a new language for *meta-programming*, this is the activity of writing meta-programs.

EASY Programming

Many meta-programming problems follow a fixed pattern. Starting with some input system (a black box that we usually call *system-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” (page 3).

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



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

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 objects-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.

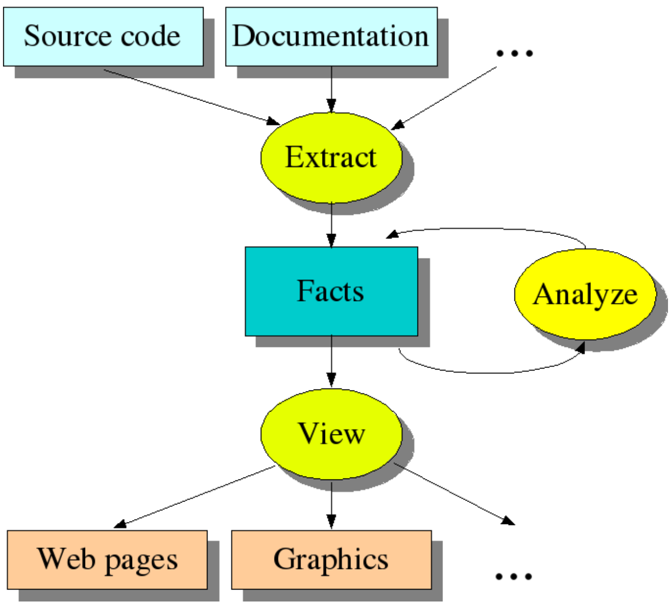
Example 1.2. A Forensic DSL compiler

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. Renovating 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 display the relationships 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” (page 4).

Figure 1.2. The extract-analyze-view paradigm



Example 1.4. Finding Concurrency Errors

Daniel is concurrency researcher at one of the largest 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 more 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.

Example 1.5. Model driven engineering

Elisabeth is a software architect at a large airplane manufacturer and her concern is reliability and dependability of airplane control software. She and her team have designed a UML model of the control software and have extended it with annotations that describe the reliability of individual components. She will use this annotated model in two ways: (a) to extract relevant information from it to synthesize input for a statistical tool that will compute overall system reliability from the reliability of individual components; (b) to generate executable code that takes the reliability issues into account.

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 (in a wide sense) 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 and implementing domain-specific languages to constraint solving and software renovation.

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.

Benefits of Rascal

Before you spend your time on studying the Rascal language it may help to first hear our elevator pitch about the main benefits offered by the language:

- **Sophisticated built-in data types** provide standard solutions for many meta-programming problems.
- **Safety** is achieved by finding most errors even before the program is executed and by making common errors like missing initializations or wrong pointers impossible. *At the time of writing, this checking is done during execution.*
- **Pattern matching** is used to analyze even the most complex datastructures.
- **Syntax definitions** make it possible to define new and existing languages and to write tools for them.
- **Visiting** makes it easy to traverse datastructures and to extract information from them or to synthesize results.
- **Functions as values** permit programming styles with high re-use.
- **Generic types** allow writing functions that are applicable for many different types.
- **Local type inference** makes local variable declarations redundant.
- **Familiar syntax** in a *what-you-see is-what-you-get style* is used even for sophisticated concepts and this makes the language easy to learn and easy to use.
- **Eclipse integration** makes Rascal programming a breeze. All familiar tools are at your fingertips.

Interested? Read on!

Aim and Scope of this Book

Aim. The aim of this book is to give an easy to understand but comprehensive overview of the Rascal language and to offer problem solving strategies to handle realistic problems that require meta-programming. Problems may range from security analysis and model extraction to software renovation, domain-specific languages and code generation.

Audience. The book is intended for students, practitioners and researchers who want to solve meta-programming problems.

Background. Readers should have some background in computer science, software engineering or programming languages. Familiarity with several main stream programming languages and experience with larger software projects will make it easier to appreciate the relevance of the meta-programming domain that Rascal is addressing. Some familiarity with concepts like sets, relations and pattern matching is assumed.

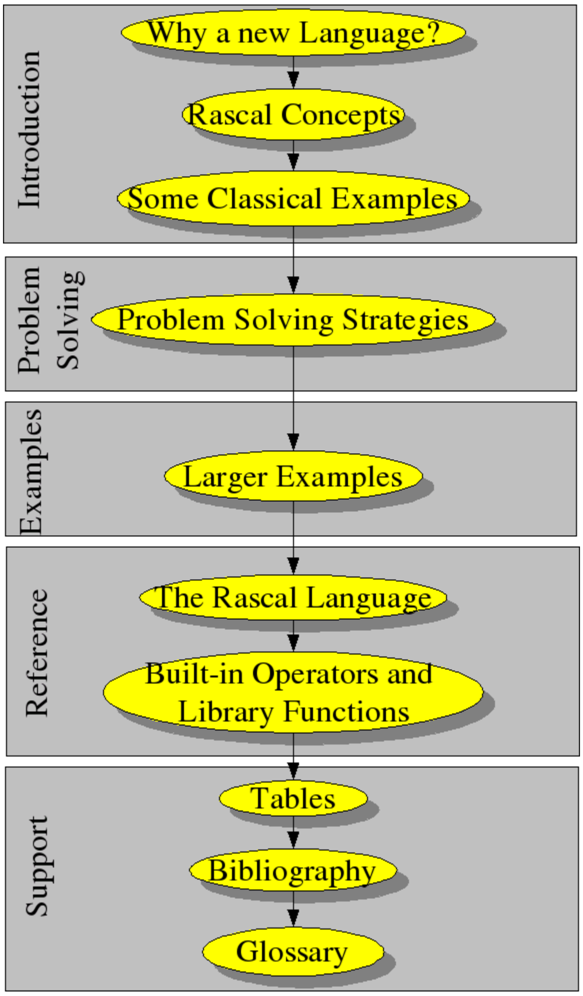
Scope. The scope of the book is limited to the Rascal language and its applications but does not address implementation aspects of the language.

Installing and Running Rascal

See http://www.meta-environment.org/Meta-Environment/Rascal for information.

Reading Guide

Figure 1.3. Structure of this Book



The structure of this book is shown in Figure 1.3, "Structure of this Book" (page 6). It consists of five parts:

- **Introduction**: gives a high-level overview of Rascal and consists of the section called “A New Language for Meta-Programming” (page 2) and the section called “Rascal Concepts” (page 8) . It also presents some simple examples in the section called “Some Classical Examples” (page 14).
- **Problem Solving**: describes the major problem solving strategies in Rascal's application domain, see the section called “Problem Solving Strategies” (page 18).
- **Examples**: gives a collection of larger examples, see the section called “Larger Examples” (page 25).
- **Reference**: gives a detailed description of the Rascal language, see the section called “The Rascal Language” (page 44) and all built-in operators and library functions, see the section called “Built-in Operators and Library Functions” (page 76).
- **Support**: gives tables with operators, see Table 1.38, “All Operators” (page 97) and library functions, see Table 1.39, “All Functions” (page 99) a bibliography, see the section called “Bibliography” (page 102) and a glossary, see Glossary of Terminology (page 104) that explains many concepts that are used in this book and tries to make the book self-contained.

Typographic Conventions

Rascal code fragments are always shown as a listing like this:

```
.. here is some Rascal code ...
```

Interactive sessions are show as a screen like this (*make a more clear typographic distinction between listings and screens*):

```
rascal> Command;
Type: Value
```

where:

- `rascal>` is the prompt of the Rascal system.
- **`Command`** is an arbitrary Rascal statement or declaration typed in by the user.
- `Type: Value` is the type of the answer followed by the value of the answer as computed by Rascal. In some cases, the response will simply be `ok` when there is no other meaningful answer to give.

For typographic reasons the output is abbreviated or slightly edited in some examples.

Rascal Concepts

Before explaining the Rascal language in more detail, we detail our elevator pitch a bit and give you a general understanding of the concepts on which the language is based.

Values

Values are the basic building blocks of a language and the type of values determines how they may be used.

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 datastructures. Syntax trees that are the result of parsing source files are represented as datatypes. There is a wealth of built-in operators and library functions available on the standard datatypes. The basic Rascal datatypes are illustrated in Table 1.1, “Basic Rascal Types” (page 8).

These builtin datatypes are closely related to ech other:

- In a **list** all elements have the same static type and the order of elements matters. A list may contain the same value more than once.
- In a **set** all elements have the same static type and the order of elements does not matter. A set contains an element only once. In other words, duplicate elements are eliminated and no matter how many times an element is added to a set, it will occur in it only once.
- In a **tuple** alle elements (may) have a different static type. Each element of a tuple may have a label that can be used to select that element of the tuple.
- A **relation** is a set of tuples which all have the same static tuple type.
- A **map** is a binary relation consisting of (key, value) pairs. Key and value (may) have different static type and a key can only be associated with a value once

User-defined datatypes allow the introduction of problem-specific types. 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)
          ;
```

Table 1.1. Basic Rascal Types

Type	**Examples**
`bool`	`true`, `false`
`int`	`1`, `0`, `-1`, `123456789`

Type	**Examples**
`real`	`1.0`, `1.0232e20`, `-25.5`
`str`	`"abc"`, `"first\nnext"`, `"result: <X>"`
`loc`	`!file:///etc/passwd`
`tuple[`T_1`,...,`T_n`]`	`<1,2>`, `<"john", 43, true>`
`list[`T`]`	`[]`, `[1]`, `[1,2,3]`, `[true, 2, "abc"]`
`set[`T`]`	`{}`, `{1,2,3,5,7}`, `{"john", 4.0}`
`rel[`T_1`,...,`T_n`]`	`{<1,2>,<2,3>,<1,3>}`, `{<1,10,100>, <2,20,200>}`
`map[`T`, `U`]`	`()`, `(1:true, 2:false)`, `("a":1, "b":2)`
node	`f`, `add(x,y)`, `g("abc", [2,3,4])`

Pattern Matching

Pattern matching determines whether a given pattern matches a given value. The outcome can be false (no match) or true (a match).

Pattern matching is *the* mechanism for case distinction and search in Rascal and a very rich pattern language is provided that includes string matching based on regular expressions, list (associative) and set (associative, commutative, identity) matching, matching of abstract patterns, matching of concrete syntax patterns, and matching of descendant (nested) patterns. All these forms of matching can be used in a single pattern and can be nested. Patterns may contain variables that are bound when the match is successful. Anonymous (don't care) positions are indicated by an underscore (_). Here is a regular expression that matches a line of text, finds the first alphanumeric word in it, and extracts the word itself as well as the before and after it:

```
/^<before:\W*><word:\w+><after:.*$>/
```

Regular expressions follow the Java regular expression syntax with one exception: instead of using `$1`, `$2` as names for parts of the subject string that have been matched by a part of the regular expression we use the notation:

```
<Name:RegularExpression>
```

If *`RegularExpression`* matches, the matched substring is assigned to string variable *`Name`*.

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 the example---or they may be declared in the context in which the pattern occurs. So-called multi-variables in list and set patterns are declared by a `*` suffix. The above pattern can then be written as

```
whileStat(EXP Exp, Stats*)
```

or, if you are not interested in the actual value of the statements as

```
whileStat(EXP Exp, _*)
```

When there is a grammar for this example language (in the form of an imported SDF definition), we can also write concrete patterns as we will see below.

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. Patterns can also be used in control structures, enumerators and visits.

Enumerators

Enumerators 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 enumerator. Examples are:

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

The first two produce the integer elements of a set of integers, respectively, a range of integers. The third enumerator 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

A comprehension is a notation inspired by mathematical set-builder notation that helps to write succinct definitions of lists and sets.

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 enumerators and tests (boolean expressions). The enumerators produce values and the tests 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 left hand side of assignment statements in program `P`.

Control structures

Control structures like `if` and `while` statement are driven by Boolean expressions, for instance

```
if(N <= 0)
     return 1;
  else
     return N * fac(N - 1);
```

Actually, combinations of generators and boolean expressions can be used to 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.

Switching

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. Here is an example where we take a program P and distinguish two cases for while and if statement:

```
switch (P){
case whileStat(EXP Exp, Stats*):
     println("A while statement");
case ifStat(Exp, Stats1*, Stat2*):
     println("An if statement");
}
```

Visiting

Visiting the elements of a datastructure is one of the most common operations in our domain and the visitor design pattern is a solution known to every software engineer. Given a tree-like datastructure we want to perform an operation on some (or all) nodes of the tree. The purpose of the visitor design pattern is to decouple the logistics of visiting each node from the actual operation. In Rascal the logistics of visiting is completely automated.

Visiting is achieved 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, i.e., assign a value to local or global variables;
- 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 successful match itself and the execution of the statements so far).

Side-effects including assignments to local and global variables in the Rascal program are undone on failure. External side-effects like I/O and side-effects in user-defined Java code are not undone.

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. An example of visiting is given in the section called “Colored Trees” (page 15).

Functions

Functions allow the definition of frequently used operations. They have a name and formal parameters. They 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;
}
```

Syntax Definition and Parsing

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

- Use regular expressions 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> do <Stats> od
```

Importing an SDF module has the following effects:

- All non-terminals (*sorts* in SDF jargon) that are used in the imported grammar are implicitly declared as Rascal types. This makes it possible to handle parse trees and parse tree fragments as fully typed values and assign them to variables, store them in larger datastructures or pass them as arguments to functions.
- For all *start symbols* of the grammar *parse functions* are implicitly declared that can parse source files according to a specific start symbol.
- Concrete syntax patterns for that specific grammar can be used.
- Concrete syntax constructors can be used that allow the construction of new parse trees.

The following example parses a Java compilation unit from a text file and counts the number of method declarations:

```
module Count
import languages::syntax::Java;

public int countMethods(str file){
  int n = 0;
  for(MethodDeclaration md <- parseCompilationUnit(file))
      n += 1;
  return n;
}
```

The function `parseCompilationUnit` is implicitly defined as a result of importing the Java grammar and `MethodDeclaration` is a non-terminal from the Java grammar that becomes available as type. This example ignores many potential error issues but does illustrate some of Rascal's syntax and parsing features.

Rewrite Rules

A *rewrite rule* is a recipe how to simplify values. Remember: $(a + b)^2 = a^2 + 2ab + b^2$? A rewrite rule has a pattern as left-hand side (here: $(a + b)^2$) and a replacement as right-hand side (here: $a^2 + 2ab + b^2$). Given a value and a set of rewrite rules the patterns are tried on every subpart of the value and replacements are made on the way. This is repeated as long as some pattern matches.

Rewrite rules are the only implicit control mechanism in the language and are used to maintain invariants during computations. For example, in a package for symbolic differentiation it is desirable to keep expressions in simplified form in order to avoid intermediate results like `sum(product(1, x), product(0, y))` that can be simplified to `x`. The following rules achieve this:

```
rule simplify1 product(1, Expression e) => e;
rule simplify2 product(Expression e, 1) => e;
rule simplify3 product(0, Expression e) => 0;
rule simplify4 product(Expression e, 0) => 0;
rule simplify5 sum(0, Expression e)     => e;
rule simplify6 sum(Expression e, 0)     => e;
```

Whenever a new expression is constructed during symbolic differentiation, these rules are applied to that expression and all its subexpressions and when a pattern at the left-hand side of a rule applies the matching subexpression is replaced by the right-hand side of the rule. This is repeated as long as any rule can be applied.

Since rewrite rules are activated automatically, one may always assume that expressions are in simplified form.

Rewrite rules are *computationally complete*, in other words any computable function can be defined using rewrite rules, including functions that do not terminate. This is a point of attention when using rewrite rules.

Constraint solving

Many problems can be solved by forms of *constraint solving*. This is a declarative way of programming: specify the constraints that a problem solution should satisfy and how potential solutions can be generated. The actual solution (if any) is found by enumerating solutions and testing their compliance with the constraints.

Rascal provides a `solve` statement that helps writing constraint solvers. A typical example is dataflow analysis where the propagation of values through a program can be described by a set of equations. Their solution can be found with the `solve` statement. See the section called “Dataflow Analysis” (page 33) for examples.

Other features

All language features (including the ones just mentioned) are described in more detail later on in this book. Some features we have not yet mentioned are:

- Rascal programs consist of modules that are organized in packages.
- Modules can import other modules.
- The visibility of entities declared in modules can be controlled by a public/private mechanism.
- Datastructures may have annotations that can be explicitly used and modified.

Typechecking and Execution

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. Built-in operators are heavily overloaded. For instance, the operator + is used for addition on integers and reals but also for list concatenation, set union etc.

The flow of Rascal program execution is completely explicit. Boolean expressions 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 surprise) in the context of boolean expressions and pattern matching; side effects are undone in case of backtracking.

Some Classical Examples

The following simple examples will help you to grasp the main features of Rascal quickly. You can also look ahead and consult the section called "The Rascal Language" (page 44)for details of the language or the section called "Built-in Operators and Library Functions"(page 76)for specific operators or functions.

Hello

The ubiquitous hello world program looks in Rascal as follows:

```
rascal> import IO;
ok

rascal> println("Hello, this is my first Rascal program");
Hello, this is my first Rascal program
ok
```

First, the library module IO (see the section called "IO" (page 80**)** is imported since hello world requires printing. Next, we call `println` and proudly observe our first Rascal output!

A slightly more audacious approach is to wrap the print statement in a function and call it:

```
rascal> void hello() {
   println("Hello, this is my first Rascal program");
}
Rascal.Function ...

rascal> hello();
Hello, this is my first Rascal program
ok
```

The summit of hello-engineering can be reached by placing all the above in a separate module:

```
module demo::Hello

public void hello() {
   println("Hello, this is my first Rascal program");
}
```

Note that we added a public modifier to the definition of hello, since we want it to be visible outside the Hello module. Using this Hello module is now simple:

```
rascal> import demo::Hello;
ok

rascal> hello();
Hello, this is my first Rascal program
ok
```

All examples in this book can be found in the `demo` directory of the Rascal distribution. That is why we prefix all names of examples modules with `demo::`.

Factorial

Here is another classical example, computing the factorial function:

```
module demo::Factorial

public int fac(int N)
{
  if(N <= 0)
     return 1;
  else
     return N * fac(N - 1);
}
```

It uses a conditional statement to distinguish cases and here is how to use it:

```
rascal> import demo::Factorial;
ok

rascal> fac(47);
int: 2586232415111681806429643551536119799691976323891200
00000000
```

Indeed, Rascal has infinite length integers.

Colored Trees

Suppose we have binary trees---trees with exactly two children--that have integers as their leaves. Also suppose that our trees can have red and black nodes. Such trees can be defined as follows:

```
module demo::ColoredTrees

data ColoredTree =
            leaf(int N)
          | red(ColoredTree left, ColoredTree right)
          | black(ColoredTree left, ColoredTree right);
```

We can use them as follows:

```
rascal> import demo::ColoredTrees;
ok

rascal> rb = red(black(leaf(1), red(leaf(2),leaf(3))),
                 black(leaf(3), leaf(4)));
ColoredTree: red(black(leaf(1),red(leaf(2),leaf(3))),
                 black(leaf(3),leaf(4)))
```

We define two operations on ColoredTrees, one to count the red nodes, and one to add the values contained in all leaves:

```
// continuing module demo::ColoredTrees

public int cntRed(ColoredTree t){
   int c = 0;
   visit(t) {
     case red(_,_): c = c + 1;❶
   };
   return c;
}

public int addLeaves(ColoredTree t){
   int c = 0;
   visit(t) {
```

```
    case leaf(int N): c = c + N;❷
  };
  return c;
}
```

❶ Visit all the nodes of the tree and increment the counter `c` for each red node.

❷ Visit all nodes of the tree and add the integers in the leaf nodes.

This can be used as follows:

```
rascal> cntRed(rb);
int: 2
rascal> addLeaves(rb);
int: 13
```

A final touch to this example is to introduce green nodes and to replace all red nodes by green ones:

```
// continuing module demo::ColoredTrees

data ColoredTree = green(ColoredTree left,
                         ColoredTree right);❶

public ColoredTree makeGreen(ColoredTree t){
   return visit(t) {
     case red(l, r) => green(l, r)      ❷
   };
}
```

❶ Extend the ColoredTree datatype with a new green constructor.

❷ Visit all nodes in the tree and replace red nodes by green ones. Note that the variables `l` and `r` are introduced here without a declaration.

This is used as follows:

```
rascal> makeGreen(rb);
ColoredTree: green(black(leaf(1),green(leaf(2),leaf(3))),
                   black(leaf(3),leaf(4)))
```

Word Replacement

Suppose you are in the publishing business and are responsible for the systematic layout of publications. Authors do not systematically capitalize words in titles---"Word replacement" instead of Word Replacement"--- and you want to correct this. Here is one way to solve this problem:

```
module demo::WordReplacement
import String;

public str capitalize(str word)
{
   if(/^<letter:[a-z]><rest:.*$>/ := word) ❶
      return toUpperCase(letter) + rest;❷
   else
     return word;❸
}
```

❶ The function capitalize takes a string as input and capitalizes its first character if that is a letter. This is done using a regular expression match that anchors the match at the beginning (`^`), expects a single letter and assigns it to the variable letter (`letter:[a-z]`) followed by an arbitrary sequence of letters until the end of the string that is assigned to the variable rest (`<rest:.*$>`).

❷ If the regular expression matches we return a new string with the first letter capitalized.
❸ Otherwise we return the word unmodified.

The next challenge is how to capitalize all the words in a string. Here are two solutions:

```
// continuing module demo::WordReplacement

public str capAll1(str S)
{
 result = "";
 while (/^<before:\W*><word:\w+><after:.*$>/ := S) { ❶
    result += before + capitalize(word);
    S = after;
  }
  return result;
}

public str capAll2(str S)
{
   return visit(S){❷
     case /<word:\w+>/i ❸ => capitalize(word)❹
   };
}
```

❶ In the first solution `capAll1` we just loop over all the words in the string and capitalize each word. The variable `result` is used to collect the successive capitalized words.
❷ In the second solution we use a visit expression to visit all the substrings of S. Each matching case advances the substring by the length of the pattern it matches and replaces that pattern by another string. If no case matches the next substring is tried.
❸ The single case matches a word (note that `\w` matches a word character).
❹ When the case matches a word, it is replaced by a capitalized version.

We can apply this all as follows:

```
rascal> import demo::WordReplacement;
ok

rascal> capitalize("rascal");
str: "rascal"

rascal> capAll1("rascal is great");
str: "Rascal Is Great"
```

Problem Solving Strategies

Before we study more complicated examples, it is useful to discuss some general problem solving strategies that are relevant in Rascal's application domain.

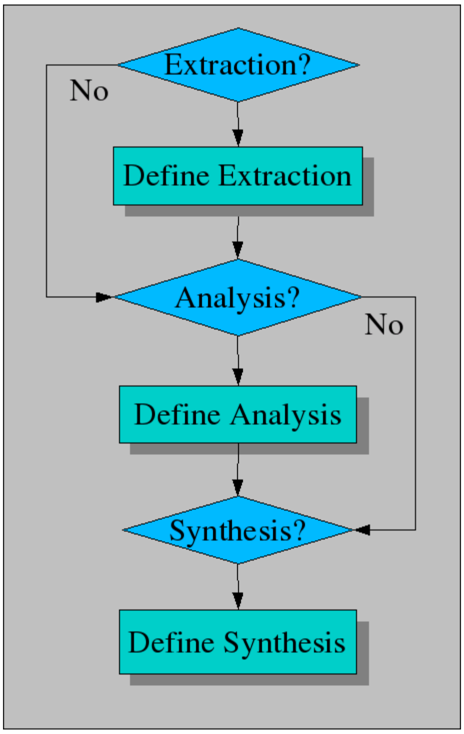
To appreciate these general strategies, it is good to keep some specific problem areas in mind:

- **Documentation generation**: extract facts from source code and use them to generate textual documentation. A typical example is generating web-based documentation for legacy languages like Cobol and PL/I.
- **Metrics calculation**: extract facts from source code (and possibly other sources like test runs) and use them to calculate code metrics. Examples are cohesion and coupling of modules and test coverage.
- **Model extraction**: extract facts from source code and use them to build an abstract model of the source code. An example is extracting lock and unlock calls from source code and to build an automaton that guarantees that lock/unlock occurs in pairs along every control flow path.
- **Model-based code generation**: given a high-level model of a software system, described in UML or some other modelling language, transform this model into executable code. UML-to-Java code generation falls in this category.
- **Source-to-source transformation**: large-scale, fully automated, source code transformation with certain objectives like removing deprecated language features, upgrading to newer APIs and the like.
- **Interactive refactoring**: given known "code smells" a user can interactively indicate how these smells should be removed. The refactoring features in Eclipse and Visual Studio are examples.

With these examples in mind, we can study the overall problem solving workflow as shown in Figure 1.4, “General 3-Phased Problem Solving Workflow” (page 19). It consists of three optional phases:

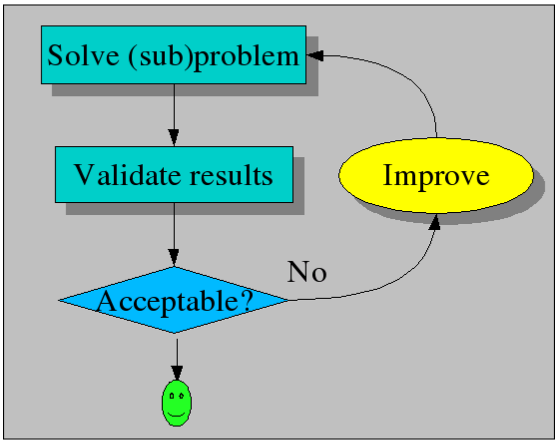
- Is **extraction needed** to solve the problem, then define the extraction phase, see the section called “Defining Extraction” (page 20).
- Is **analysis needed**, then define the analysis phase, see the section called “Defining Analysis” (page 22).
- Is **synthesis needed**, then define the synthesis phase, see the section called “Defining Synthesis” (page 23).

Figure 1.4. General 3-Phased Problem Solving Workflow



Each phase is subject to a validation and improvement workflow as shown in Figure 1.5, “Validation and Improvement Workflow” (page 19). Each individual phase as well as the combination of phases may introduce errors and has thus to be carefully validated. In combination with the detailed strategies for each phase, this forms a complete approach for problem solving and validation using Rascal.

Figure 1.5. Validation and Improvement Workflow

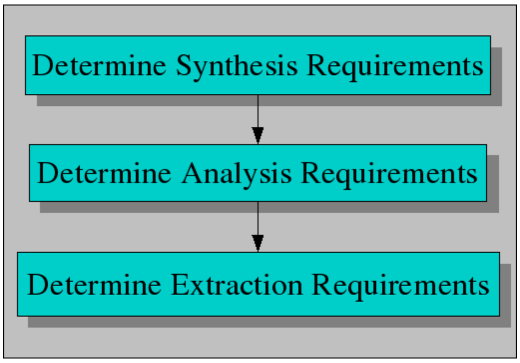


A major question in every problem solving situation is how to determine the requirements for each phase of the solution. For instance, how do we know what to extract from the source code if we do not know what the desired end results of the project are? The standard solution is to use a workflow for requirements gathering that is the inverse of the phases needed to solve the complete problem. This is shown in Figure 1.6, “Requirements Workflow” (page 20) and amounts to the phases:

- **Requirements of the synthesis phase**. This amounts to making an inventory of the desired results of the whole project and may include generated source code, abstract models, or visualizations.

- **Requirements of the analysis phase**. Once these results of the synthesis phase are known, it is possible to list the analysis results that are needed to synthesize desired results. Possible results of the analysis phase include type information, structural information of the original source.
- **Requirements of the extraction phase**. As a last step, one can make an inventory of the facts that have to be extracted to form the starting point for the analysis phase. Typical facts include method calls, inheritance relations, control flow graphs, usage patterns of specific library functions or language constructs.

Figure 1.6. Requirements Workflow



You will have no problem in identifying requirements for each phase when you apply them to a specific example from the list given earlier.

When these requirements have been established, it becomes much easier to actually carry out the project using the three phases of Figure 1.4, “General 3-Phased Problem Solving Workflow” (page 19).

Defining Extraction

How can we extract facts from the *System under Investigation* (SUI) that we are interested in? The extraction workflow is shown in Figure 1.7, “Extraction Workflow” (page 22) and consists of the following steps:

- First and foremost we have to determine which facts we need. This sounds trivial, but it is not. The problem is that we have to anticipate which facts will be needed in the next---not yet defined---analysis phase. A common approach is to use look-ahead and to sketch the queries that are likely to be used in the analysis phase and to determine which facts are needed for them. Start with extracting these facts and refine the extraction phase when the analysis phase is completely defined.
- If relevant facts are already available (and they are reliable!) then we are done. This may happen when you are working on a system that has already been analyzed by others.
- Otherwise you need the source code of the SUI. This requires:
  - Checking that all sources are available (and can be compiled by the host system on which they are usually compiled and executed). Due to missing or unreliable configuration management on the original system this may be a labour-intensive step that requires many iterations.
  - Determining in which languages the sources are written. In larger systems it is common that three or more different languages are being used.
- If there are reliable third-party extraction tools available for this language mix, then we only have to apply them and we are done. Here again, validation is needed that the extracted facts are as expected.
- The extraction may require syntax analysis. This is the case when more structural properties of the source code are needed such as the flow-of-control, nesting of declarations, and the like. There two approaches here:

  - Use a third-party parser, convert the source code to parse trees and do the further processing of these parse trees in Rascal. The advantage is that the parser can be re-used, the disadvantage is that data conversion is needed to adapt the generated parse tree to Rascal. Validate that the parser indeed accepts the language the SUI is written in, since you will not be the first who has been bitten by the language dialect monster when it turns out that the SUI uses a local variant that slightly deviates from a mainstream language.

  - Use an existing SDF definition of the source language or write your own definition. In both cases you can profit from Rascal's seamless integration with SDF. Be aware, however, that writing a grammar for a non-trivial language is a major undertaking and may require weeks to month of work. Whatever approach you choose, validate that the result.

- The extraction phase may only require lexical analysis. This happens when more superficial, textual, facts have to be extracted like procedure calls, counts of certain statements and the like. Use Rascal's full regular expression facilities to do the lexical analysis.

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. This may happen when the fact extractor uses a conservative approximation when precise information is not statically available. In the language C, when procedure P performs an indirect call via a pointer variable, the approximation may be that P calls all procedures in the procedures.

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

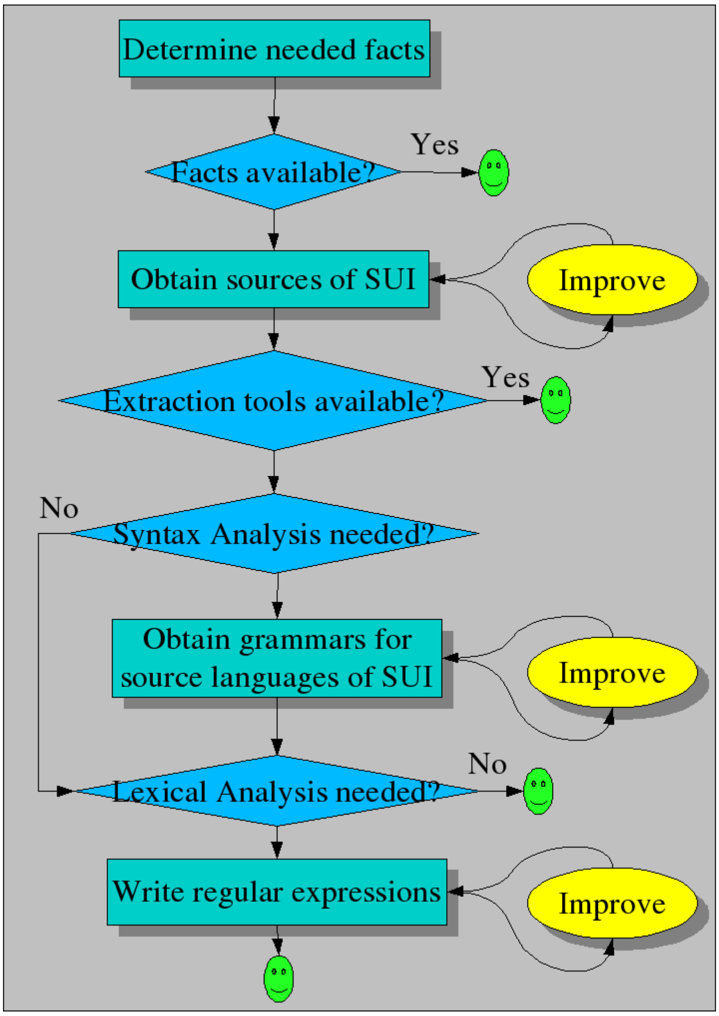
The strategy to validate extracted facts differ per case but here are three strategies:

- Post process 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.

Figure 1.7. Extraction Workflow



The Rascal features that are most frequently used for extraction are:

- Regular expression patterns to extract textual facts from source code.
- Syntax definitions and concrete patterns to match syntactic structures in source code.
- Pattern matching (used in many Rascal statements).
- Visits to traverse syntax trees and to locally extract information.
- The repertoire of built-in datatypes (like lists, maps, sets and relations) to represent the extracted facts.

Defining Analysis

The analysis workflow is shown in Figure 1.8, “Analysis Workflow” (page 23)and consists of two steps:

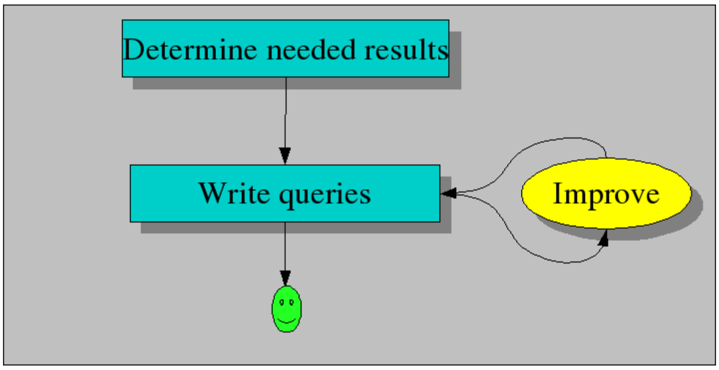
- Determine the results that are needed for the synthesis phase.
- Write the Rascal code to perform the analysis. This may amount to:
  - Reordering extracted facts to make them more suitable for the synthesis phase.

    - Enriching extracted facts. Examples are computing transitive closures of extracted facts (e.g., A may call B in one or more calls), or performing data reduction by abstracting aways details (i.e., reducing a program to a finite automaton).

    - Combining enriched, extracted, facts to create new facts.

As before, validate, validate and validate the results of analysis. 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.

Figure 1.8. Analysis Workflow



The Rascal features that are frequently used for analysis are:

- List, set and map comprehensions.
- The built-in operators and library functions, in particular for lists, maps, sets and relations.
- Pattern matching (used in many Rascal statements).
- Visits and switches to further process extracted facts.
- The solve statement for constraint solving.
- Rewrite rules to simplify results and to enforce constraints.

Defining Synthesis

Results are synthesized as shown in Figure 1.9, “Synthesis Workflow” (page 24) This consists of the following steps:

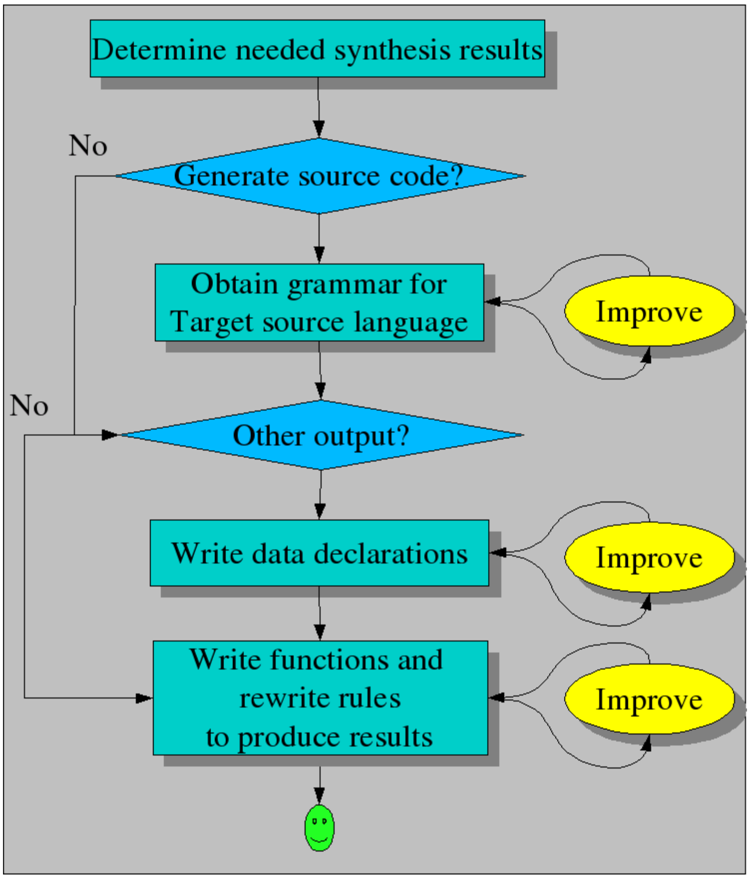
- Determine the results of the synthesis phase. Wide range of results is possible including:
    - Generated source code.
    - Generated abstract representations, like finite automata or other formals models that capture properties of the SUI.
    - Generated data for visualizations that will be used by visualization tools.
- If source code is to be generated, there are various options.
    - Print strings with embedded variables.
    - Convert abstract syntax trees to strings (perhaps using forms of pretty printing).

  - Use a grammar of the target source language, also for code generation. Note that this approach guarantees the generation of syntactically correct source code as opposed to code generation using print statements

- If other output is needed (e.g., an automaton or other formal structure) write data declarations to represent that output.

- Finally, write functions and rewrite rules that generate the desired results.

Figure 1.9. Synthesis Workflow



The Rascal features that are frequently used for synthesis are:

- Syntax definitions or data declarations to define output formats.

- Pattern matching (used in many Rascal statements).

- Visits of datastructures and on-the-fly code generation.

- Rewrite rules.

Larger Examples

Now we will have a closer look at some larger applications of Rascal. We start with a call graph analysis in the section called "Call Graph Analysis" (page 25) and then continue with the analysis of the component structure of an application in the section called "Analyzing the Component Structure of an Application" (page 28) and of Java systems in the section called "Analyzing the Structure of Java Systems" (page 30). Next we move on to the detection of uninitialized variables in the section called "Finding Uninitialized and Unused Variables in a Program" (page 31). 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" (page 33). A description of program slicing concludes the chapter, see the section called "Program Slicing" (page 40).

The examples in this section are biased towards pure analysis. We intend to add more extraction and synthesis examples.

Call Graph Analysis

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:

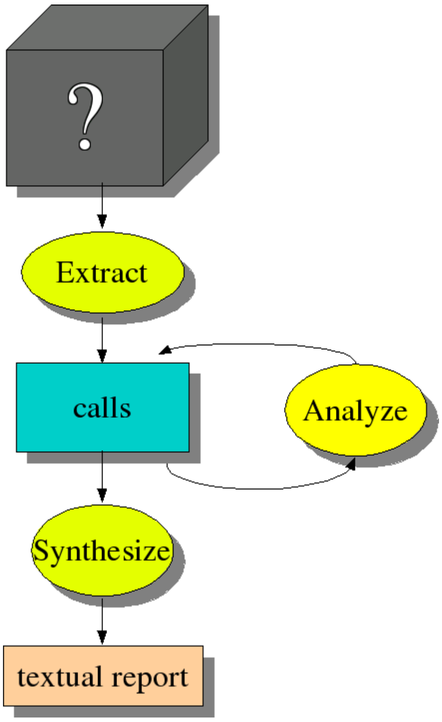
- How many procedure calls occur in this system?
- How many procedures does it contains?
- What are the entry points for this system, i.e., procedures that call others but are not called themselves?
- What are the leaves of this application, i.e., procedures that are called but do not make any calls 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

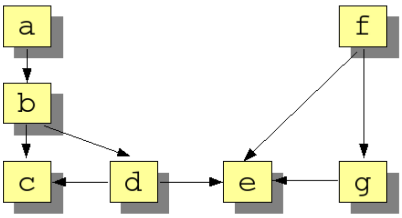
To illustrate this process consider the workflow in Figure 1.10, "Workflow for analyzing mystery box" (page 26). First we have to extract the calls from the source code. Rascal is very good at this, but to simplify this example we assume that this call graph has already been extracted. Also keep in mind 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 1.11, "Graphical representation of the `calls` relation" (page 26) 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 1.10. Workflow for analyzing mystery box



Now consider the call graph shown in Figure 1.11, "Graphical representation of the `calls` relation" (page 26) 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" (page 44)and the section called "Built-in Operators and Library Functions" (page 76).

Figure 1.11. 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;
ok
```

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

Here we should illustrate how to do this.

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

```
   };
rel[proc,proc]: { <"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” (page 88) so we import it:

```
rascal> import Relation;
ok
```

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

```
rascal> nCalls = size(Calls);
int: 8
```

The library function `size` determines the number of elements in a set or relation and is explained in the section called “Relation” (page 88). In this example, `nCalls` will get the value `8`.

How many procedures are contained in 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> procs = carrier(Calls);
set[proc]: {"a", "b", "c", "d", "e", "f", "g"}

rascal> nprocs = size(procs);
int: 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> nprocs = size(carrier(Calls));
int: 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” (page 78) for more details. Using this knowledge, the entry points can be computed by determining the top of the `Calls` relation:

```
rascal> import Graph;
ok

rascal> entryPoints = top(Calls);
set[proc]: {"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> bottomCalls = bottom(Calls);
set[proc]: {"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. Observe that the transitive closure will contain both the direct and the indirect calls.

```
rascal> closureCalls = Calls+;
rel[proc, proc]: {<"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 now know 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> calledFromA = closureCalls["a"];
set[proc]: {"b", "c", "d", "e"}
```

and

```
rascal> calledFromF = closureCalls["f"];
set[proc]: {"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> commonProcs = calledFromA & calledFromF;
set[proc]: {"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 1.11, “Graphical representation of the `calls` relation” (page 26) 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.

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]. Actual lifting, amounts to translating each call between procedures by a call between components. This described in the following module:

```
module demo::Lift

alias proc = str;
alias comp = str;

public rel[comp,comp] lift(rel[proc,proc] aCalls,
                           rel[proc,comp] aPartOf){
   return { <C1, C2> | <proc P1, proc P2> <- aCalls,
                       <comp C1, comp C2> <- aPartOf[P1] *
                                             aPartOf[P2]
          };
}
```

Let's now apply this. First import the above module, and define a call relation and a partof relation:

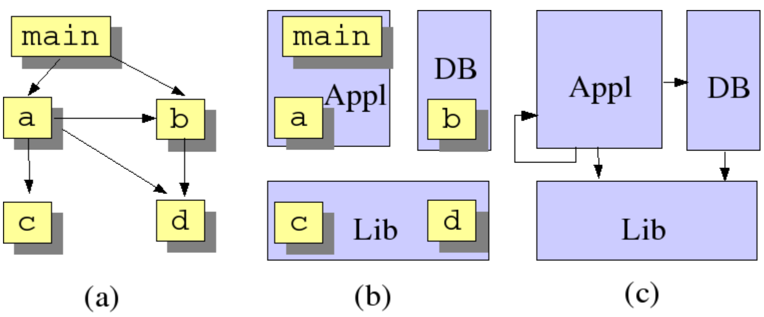
```
rascal> import demo::Lift;
ok

rascal> Calls = {<"main", "a">, <"main", "b">, <"a", "b">,
                 <"a", "c">, <"a", "d">, <"b", "d">
                };
rel[str,str] : {<"main", "a">, <"main", "b">, <"a", "b">,
                <"a", "c">, <"a", "d">, <"b", "d">
               }

rascal> Components = {"Appl", "DB", "Lib"};
set[str] : {"Appl", "DB", "Lib"}

rascal> PartOf = {<"main", "Appl">, <"a", "Appl">,
                  <"b", "DB">, <"c", "Lib">,
                  <"d", "Lib">};
rel[str,str] : {<"main", "Appl">, <"a", "Appl">,
                <"b", "DB">, <"c", "Lib">,
                <"d", "Lib">}
```

The lifted call relation between components is now obtained by:

```
rascal> ComponentCalls = lift(Calls, PartOf);
rel[str,str] : {<"DB", "Lib">, <"Appl", "Lib">,
                <"Appl", "DB">, <"Appl", "Appl">}
```

The relevant relations for this example are shown in Figure 1.12, "(a) `Calls`; (b) `PartOf`; (c) `ComponentCalls`." (page 29).

Figure 1.12. (a) `Calls`; (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

```
alias 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 fields 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 + CONTAINMENT + 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 + CONTAINMENT + 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” (page 2) that we follow the 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 graph `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. Assuming that there is a tool to extract the above information from a program text, we get the following for the above example:

```
module demo::Uninit
import Relation;
import Graph;

alias expr = int;
alias varname = str;

public expr ROOT = 1;

public graph[expr] PRED = { <1,3>, <3,4>, <4,5>, <5,6>,
                            <5,8>, <6,10>, <8,10> };

public rel[varname,expr] DEFS = { <"x", 3>, <"p", 4>,
                                  <"z", 6>, <"x", 8>,
                                  <"y", 10> };

public rel[varname, expr] USES = { <"q", 5>, <"y", 6>,
                                   <"x", 6>, <"z", 10> };
```

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 paths 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:

```
// module demo::Unit continued
public rel[varname,expr] UNINIT =
   { <V,E> | <varname V, expr E> <- USES,
             E in reachX(PRED, {ROOT}, DEFS[V])
   };
```

We analyze this definition in detail:

- `<varname V, expr E> : USES` enumerates all tuples in the `USES` relation. In other words, we consider the use of each variable in turn.
- `E in reachX(PRED, {ROOT}, DEFS[V])` is a test that determines whether expression *E* 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(PRED, {ROOT}, DEFS[V])` 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(PRED, {ROOT}, DEFS[V])` 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`:

```
rascal> import demo::Uninit;
ok

rascal> UNINIT;
rel[varname,expr]: {<"q", 5>, <"y", 6>, <"z", 10>}
```

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:

```
// module demo::Unit continued

public set[varname] UNUSED = domain(DEFS) - domain(USES);
```

Taking the domain 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"}`.

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 graph `graph[&T] PRED` between elements of arbitrary type `&T`, the cyclomatic complexity can be computed in Rascal as follows:

```
module demo::McCabe
import Graph;

public int cyclomaticComplexity(graph[&T] PRED){
    return size(PRED) - size(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” (page 34): which nodes in the flow dominate the execution of other nodes?
- Reaching definitions (the section called “Reaching Definitions” (page 36): which definitions of variables are still valid at each statement?
- Live variables (the section called “Live Variables” (page 39): 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:

```
module demo::Dominators
import Set;
import Relation;
import Graph;

public rel[&T, set[&T]] dominators(rel[&T,&T] PRED,
                                   &T ROOT)
{
  set[&T] VERTICES = carrier(PRED);
  return  { <V,  (VERTICES - {V, ROOT}) -
                 reachX(PRED,{ROOT},{V})>
             |  &T V <- VERTICES
          };
}
```

First, the auxiliary set `VERTICES` (all the statements) is computed. The relation `DOMINATES` consists of all pairs $<S,\ \{S_1, ..., S_n\}>$ such that

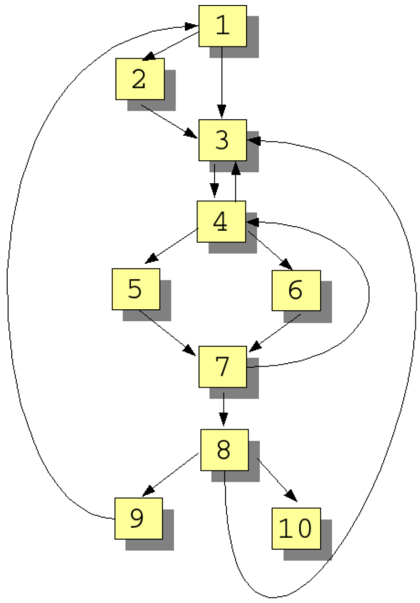
- S_i is not an initial node or equal to S.
- S_i cannot be reached from the initial node without going through S.

First import the above module and consider the sample flow graph PRED:

```
rascal> import demo::Dominators;
ok

rascal> 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>
};

rel[int,int]: { <1,2>, <1,3>, ...
```

It is illustrated inFigure 1.13, “Flow graph” (page 35)

Figure 1.13. Flow graph



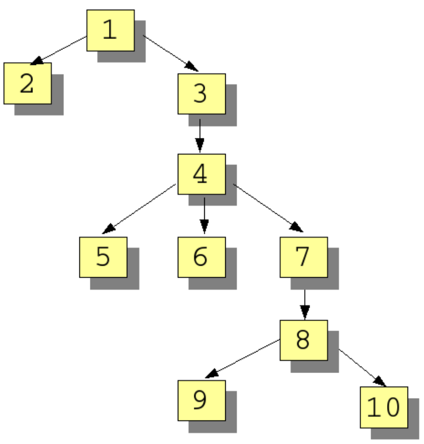
(a)

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

```
rascal> dominators(PRED);
rel[int,int]: {<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 1.14, “Dominator tree” (page 35) The dominator tree has the initial node as root and each node *d* in the tree only dominates its descendants in the tree.

Figure 1.14. Dominator tree

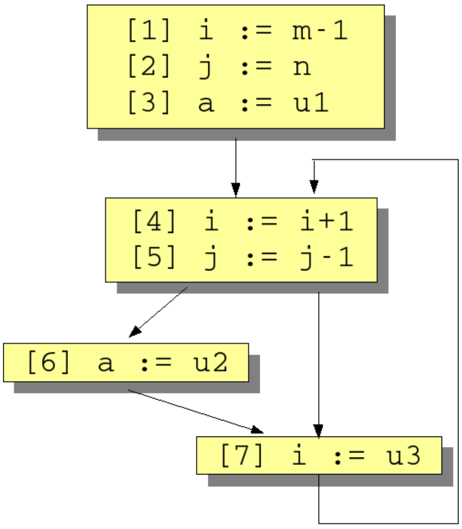


(b)

Reaching Definitions

We illustrate the calculation of reaching definitions using the example in Figure 1.15, "Flow graph for various dataflow problems" (page 36) which was inspired by [ASU86] (Example 10.15).

Figure 1.15. Flow graph for various dataflow problems



We assume the following basic definitions to represent information about the program:

```
module demo::ReachingDefs

import Relation;
import Graph;
import IO;

public alias stat = int;
public alias var = str;
public alias def  = tuple[stat, var];
public alias use = tuple[stat,var];

public rel[stat,def] definition(rel[stat,var] DEFS){
  return {<S,<S,V>> | <stat S, var V> <- DEFS};
}

public rel[stat,def] use(rel[stat, var] USES){
  return {<S, <S, V>> | <stat S, var V> <- USES};
}
```

Let's use the following values to represent our example:

```
rascal> rel[stat,stat] PRED = { <1,2>, <2,3>, <3,4>,
                                <4,5>, <5,6>, <5,7>,
                                <6,7>, <7,4> };
rel[stat,stat]: { <1,2>, <2,3>, ...

rascal> rel[stat, var] DEFS = { <1, "i">, <2, "j">,
                                <3, "a">, <4, "i">,
                                <5, "j">, <6, "a">,
                                <7, "i"> };
rel[stat,var]: { <1, "i">, <2, "j">, ...
```

```
rascal> rel[stat,var] USES = { <1, "m">, <2, "n">,
                               <3, "u1">, <4, "i">,
                               <5, "j">, <6, "u2">,
                               <7, "u3"> };
rel[stat,var]: { <1, "m">, <2, "n">,...
```

For convenience, we have introduced above 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 and the auxiliary functions `definition` and `use`:

```
rascal> definition(DEFS);
rel[stat,def]: { <1, <1, "i">>, <2, <2, "j">>,
                 <3, <3, "a">>, <4, <4, "i">>,
                 <5, <5, "j">>, <6, <6, "a">>,
                 <7, <7, "i">> }

rascal> use(USES);
rel[stat,def]: { <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 by the following function `kill`:

```
// continuing module demo::ReachingDefs

public rel[stat,def] kill(rel[stat,var] DEFS) {
  return {<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

```
rascal> kill(DEFS);
rel[stat,def]:
{ <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`.

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:

$IN[S] = UNION_{\text{P in predecessors of S}}\ OUT[P]$

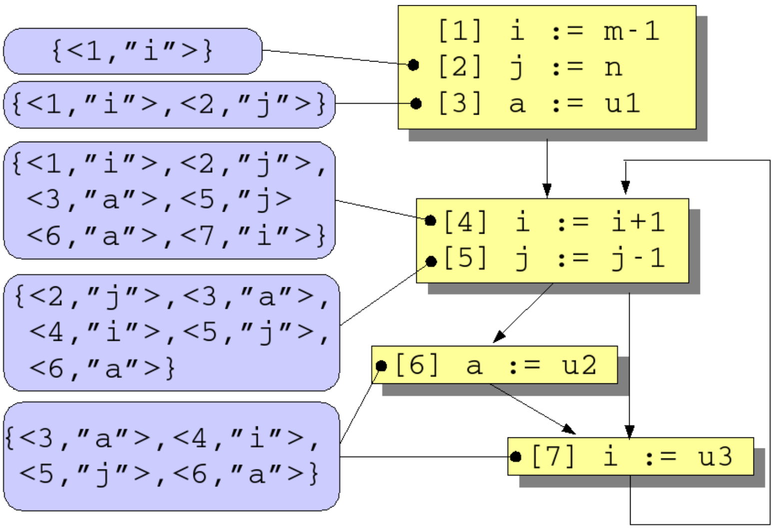
$OUT[S] = DEF[S] + (IN[S] - KILL[S])$

This idea can be expressed in Rascal quite literally:

```
public rel[stat, def] reachingDefinitions(
                            rel[stat,var] DEFS,
                            rel[stat,stat] PRED){
  set[stat] STATEMENT = carrier(PRED);
  rel[stat,def] DEF  = definition(DEFS);
  rel[stat,def] KILL = kill(DEFS);

  // The set of mutually recursive dataflow equations
  // that has to be solved:

  with
    rel[stat,def] IN = {};
    rel[stat,def] OUT = DEF;
  solve {
    IN  = {<S, D> | int S <- STATEMENT,
                    stat P <- predecessors(PRED,S),
                    def D <- OUT[P]};
    OUT = {<S, D> | int S <- STATEMENT,
                    def D <- DEF[S] + (IN[S] - KILL[S])};
  };
  return IN;
}
```

First, the relations `IN` and `OUT` are declared and initialized. Next, two equations are given that resemble the mathematical equations given above. Note the use of the library function `predecessors` to obtain the predecessors of a statement for a given control flow graph.

Figure 1.16. Reaching definitions for flow graph in Figure 1.15, “Flow graph for various dataflow problems” (page 36)



For our running example (Figure 1.16, “Reaching definitions for flow graph in Figure 1.15, “Flow graph for various dataflow problems”” (page 38) the results are as follows (see Figure 1.16, “Reaching definitions for flow graph in Figure 1.15, “Flow graph for various dataflow problems”” (page 38)). 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 1.16, “Reaching definitions for flow graph in Figure 1.15, “Flow graph for various dataflow problems”(page 38)” We will use the function `reachingDefinitions` later on in the section called “Program Slicing” (page 40) when defining program slicing.

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:

$IN[S] = USE[S] + (OUT[S] - DEF[S])$

$OUT[S] = UNION_{S' \text{ in successors of } S}\ IN[S']$

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.

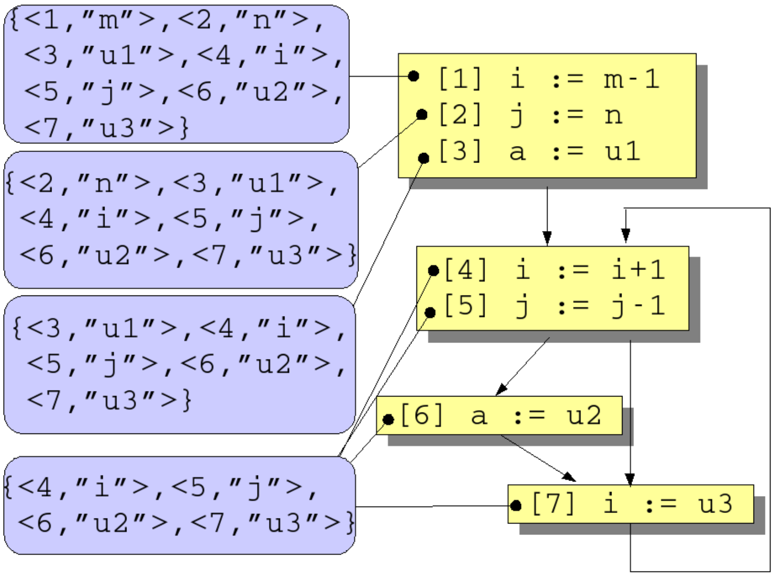
This can be expressed in Rascal as follows:

```
public rel[stat,def] liveVariables(rel[stat,var] DEFS,
                                   rel[stat, var] USES,
                                   rel[stat,stat] PRED){
  set[stat] STATEMENT = carrier(PRED);
  rel[stat,def] DEF  = definition(DEFS);
  rel[stat,def] USE = use(USES);
  with
    rel[stat,def] LIN = {};
    rel[stat,def] LOUT = DEF;
  solve {
```

```
    LIN  = { <S, D> | stat S <- STATEMENT,
                      def D <- USE[S] +
                               (LOUT[S] - (DEF[S]))};
    LOUT = { <S, D> | stat S <- STATEMENT,
                      stat Succ <- successors(PRED,S),
                      def D <- LIN[Succ] };
  }
  return LIN;
}
```

The results of live variable analysis for our running example are illustrated in Figure 1.17, "Live variables for flow graph in Figure 1.15, "Flow graph for various dataflow problems"" (page 40).

Figure 1.17. Live variables for flow graph in Figure 1.15, "Flow graph for various dataflow problems" (page 36)



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 (we use line numbers for later reference):

```
[ 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     [5] while i<= n   [ 5] while i<= n
     do                  do                 do
     begin               begin              begin
[ 6]  sum :=         [6]  sum :=
          sum + i             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]. 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" (page 36)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:

```
module demo::Slicing

import Set;
import Relation;
import demo::ReachingDefs;
import demo::Dominators;
import UnitTest;

set[use] BackwardSlice(set[stat] CONTROLSTATEMENT,
```

```
                       rel[stat,stat] PRED,
                       rel[stat,var] USES,
                       rel[stat,var] DEFS,
                       use Criterion) {

  rel[stat, def] REACH = reachingDefinitions(DEFS, PRED);

  // Compute the relation between each use and
  // corresponding definitions: use_def

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

  // Internal dependencies per statement

  rel[def,use] def_use_per_stat  =
       {<<S,V1>, <S,V2>> | <stat S, var V1> <- DEFS,
                           <S, var V2> <- USES}
       +
       {<<S,V>, <S,"EXEC">> | <stat S, var V> <- DEFS}
       +
       {<<S,"TEST">,<S,V>> | stat S <- CONTROLSTATEMENT,
                             <S, var V> <-
                                      domainR(USES, {S})};

  // Control dependence: control-dependence

  rel[stat, set[stat]] CONTROLDOMINATOR =
  domainR(dominators(PRED, 1), CONTROLSTATEMENT);

  rel[def,use] control_dependence  =
  { <<S2, "EXEC">,<S1,"TEST">>
    | <stat S1, stat S2> <- CONTROLDOMINATOR};

  // Control and data dependence: use-control-def

  rel[use,def] use_control_def =
               use_def + control_dependence;
  rel[use,use] USE_USE =
               (use_control_def o def_use_per_stat)*;

  return USE_USE[Criterion];
}
```

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

```
rascal> import demo::Slicing;
ok

rascal> 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,stat]: {<1,2>, ...

rascal> rel[stat,var] DEFS  = { <1, "n">, <2, "i">,
                                <3, "sum">,
```

```
                                <4,"product">,
                                <6, "sum">,
                                <7, "product">,
                                <8, "i"> };
rel[stat,var]: {<1, "n">, ...

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

rascal> set[int] CONTROL-STATEMENT = { 5 };
set[int]: {5}

rascal> BackwardSlice(CONTROL-STATEMENT,
                      PRED, USES, DEFS, <9, "sum">);
set[use]: { <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.

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” (page 44).
- Declarations, see the section called “Declarations” (page 50).
- Expressions, see the section called “Expressions” (page 55).
- Statements, see the section called “Statements” (page 69).

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` which has 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\nlong\tstring"`.

String literals permit *interpolation* of variable values: when `<X>` occurs inside a string literal, the value of the variable `X` is converted to a 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.

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.

Locations should *always* be generated automatically but for the curious here is an example:

```
loc(file:///home/paulk/pico.trm?offset=0&length=1&begin=2,3&end=4,5)
```

The elements 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 static 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, ..., 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]]`.

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

`[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.

```
rascal> L = [1, 2, 3];
list[int]: [1,2,3]

rascal> [10, L, 20];
list[int]: [10, 1, 2, 3, 20]

rascal> [10, [L], 20];
list[value]: [10, [1,2,3], 20]
```

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

- `[`*F*` .. `*L*`]` ranges from first element *F* up to (and including) last element *L* with increments of 1.
- `[`*F*`,`*S*`,.. `*E*`]`, ranges from first element *F*, second element S up 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 static type.
- The order of the elements does not matter.
- A set contains an element only once. In other words, duplicate elements are eliminated and no matter how many times an element is added to a set, it will occur in it only once.

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, ..., 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]]`.

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

`{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.

```
rascal> S = {1, 2, 3};
set[int]: {1,2,3]}

rascal> {10, S, 20};
set[int]: {10, 1, 2, 3, 20]}

rascal> {10, {L}, 20};
lsetist[value]: {10, {1,2,3}, 20}
```

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

- Key and value may have different static 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]`. Maps 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. Maps 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 $\texttt{tuple[}T_1 L_1\texttt{, } T_2 L_2\texttt{, ..., } T_n L_n\texttt{]}$, 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]`.

The elements of a tuple can also be labelled and can then be accessed using the field selection operator (`.`). Fields can be changed (yielding a new tuple value) by a combination of field selection and assignment. For instance,

```
rascal> tuple[str first, str last, int age] P = <"Jo", "Jones", 35>;
tuple[str first, str last, int age] P = <"Jo", "Jones", 35>;

rascal> P.first;
str: "Jo"

rascal> P.first = "Bo";
tuple[str first,str last,int age]: <"Bo","Jones",35>
```

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

- All elements have the same static 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 $\texttt{rel[}T_1 L_1\texttt{, } T_2 L_2\texttt{, ..., } T_n L_n\texttt{]}$, 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 $\texttt{set[tuple[}T_1 L_1\texttt{ , } T_2 L_2\texttt{, ..., } T_n L_n\texttt{]]}$. Examples are `rel[int,str]` and `rel[int,set[str]]`. An n-ary relations with m tuples is denoted by $\{<E_{11}, E_{12}, \ldots, E_{1n}>, <E_{21}, E_{22}, \ldots, E_{2n}>, \ldots, <E_{m1}, E_{m2}, \ldots, E_{mn}>\}$, 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`*. Both types are thus structurally equivalent. 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 document your intentions. Another example is an alias definition for a graph containing integer nodes:

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

Note that Rascal Standard Library provides a graph data type that is defined as follows:

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

In other words the standard graph datatype can be parameterized with any element type. See the section called “Graph” (page 78) for details.

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 = T | F | conj(Bool L, Bool R)  | disj(Bool L, Bool R);
```

defines the datatype `Bool` that contains various constants (`T` and `F`) and constructor functions `conj` and `disj`.

Type Parameters and Parameterized types

In addition to the types that we have already discussed, a type may also be a *type parameter* of the form

```
&Name
```

A type parameter may occur at every syntactic position where a type is required and turns an ordinary type into a parameterized type. Parameterized types are used to define polymorphic functions and data types, i.e., functions and data types that are applicable for more than one type. Type parameters are bound to an actual type when the function or data type is applied and further uses of the type parameter are consistently replaced by the actual type.

The following syntactic positions are *binding occurrences* for type parameters:

- Type parameters in the type declaration of a function are bound to the types of the actual parameters in the call of that function. Type parameters that occur in the body of the function are replaced by the corresponding actual type.
- The left-hand side of an alias. The type parameters are bound when the alias is used and occurrences of type parameters in the right hand side are replaced by corresponding actual types.
- The alternatives of a data type. Binding and replacement is identical to that of function declarations.

All other occurrences of type parameters are *using occurrences*. The following rules apply:

- When the same type parameter is used at different binding occurrences it should be bound to the same actual type.
- For every using occurrence of a type parameter there should be a binding occurrence of a type parameter with the same name.

We refer to the section called “Function Declaration” (page 51)for a full description of function declaration, but here

The following function `swap` returns a tuple in which its arguments are swapped and can be applied to arbitrary values in a type safe manner:

```
rascal> tuple[&B, &A] swap(&A a, &B b) { return <b, a>; }
ok

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

```
tuple[int,int]: <2,1>

rascal> swap("abc", 3);
tuple[int,str]: <3, "abc">
```

Observe that the type parameters that occur in the return type should occur in the formal parameter types of the function.

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

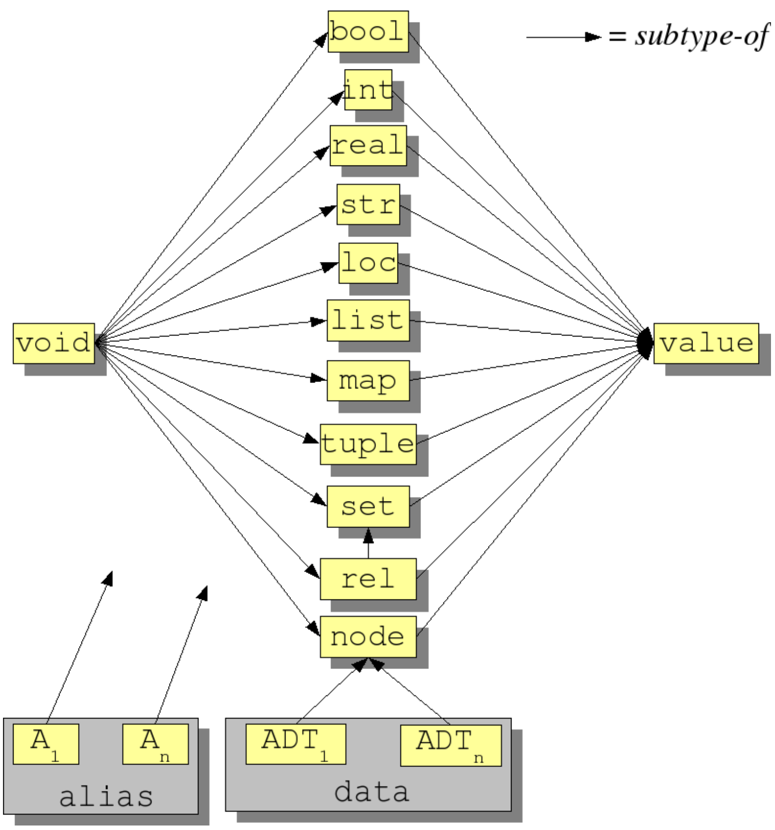
```
rascal> alias Graph[&Node] = rel[&Node, &Node];
ok

rascal> Graph[int] GI = {<1,2>, <3,4>, <4,1>};
Graph[int] :  {<1,2>, <3,4>, <4,1>}

rascal> Graph[str] GS = {<"a", "b">, <"c","d">, <"d", "a">};
Graph[str] : {<"a", "b">, <"c","d">, <"d", "a">}
```

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

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” (page 49). The arrows describe a *subtype-of* relation between types. The type void is the *smallest* type and is included in all other types and the type value is the *largest* 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 structurally equivalent to one or more specific other types.

Figure 1.18. 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;
Declaration_1;
...
Declaration_n;
```

and consists of a *Name* and zero or more imports of other modules (the section called “Import” (page 51)) and declarations for

- Data type, see the section called “Data Type” (page 48).
- Alias, see the section called “Alias Type” (page 47).
- Variable, see the section called “Variable Declaration” (page 52).
- Function, see the section called “Function Declaration” (page 51).
- Rewrite rule, see the section called “Rewrite Rule Declaration” (page 54).
- Node annotation, see the section called “Node Annotation Declaration” (page 54).
- Declaration tag, see the section called “Declaration Tag” (page 53).

The module name *Name* will be used when the current module is imported in another module. A module is usually a qualified name of the form:

```
Name_1::Name_2:: ... ::Name_n
```

which corresponds to a path relative to the root of the current workspace.

Explain this better.

The constituents of a module are also shown in Figure 1.19, “Constituents of a Module” (page 51) The more sophisticated features are shown in a separate color.

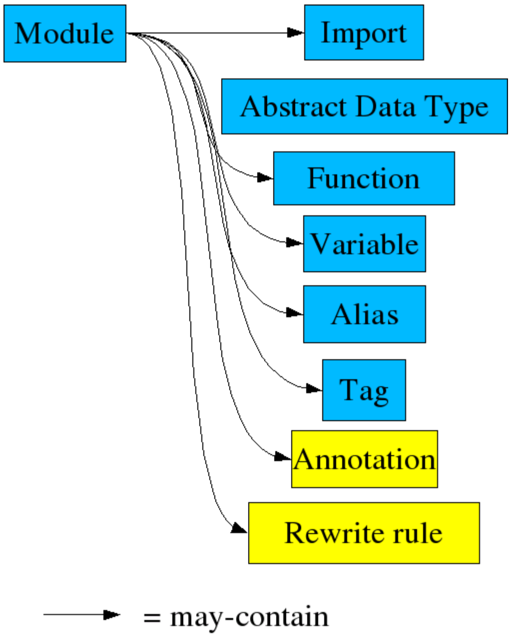
The entities that are *visible* inside a module are

- 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. The visible entities of an imported may be explicitly qualified with their module name:

```
Module::Name
```

Figure 1.19. Constituents of a Module



Import

An import has the form:

```
import QualifiedName;
```

and has as effect that all public entities declared in module `QualifiedName` are made available to the importing module. Circular imports are allowed.

Function Declaration

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 Statement” (page 72). 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 }
        }
ok

rascal> invert({<1,10>, <2,20>});
rel[int,int] : {<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 the section called “Try Catch Statement”(page 73)and the section called “Throw Statement” (page 73) for details.

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 };
        }
ok

rascal> invert2({<1,10>, <2,20>});
rel[int,int] : {<10,1>, <20,2>}

rascal> invert2({<"mon", 1>, <"tue", 2>});
rel[int,str] : {<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>;;}
ok

rascal> swap(<1, 2>);
tuple[int,int] : <2,1>

rascal> swap(<"wed", 3>);
tuple[int,str] : <3, "wed">
```

Variable Declaration

A variable declaration has the form

```
Type Var = Exp
```

where *Type* is a type, *Var* is a variable name, and *Exp* is an expression that should have type *Type*. The effect is that the value of expression *Exp* is assigned to *Var* and can be used later on as *Var*'s value. The following 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 inside functions (but not at the module level) 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 Statement” (page 74)).

Rascal provides *local type inference*, 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.
- If a variable is implicitly declared in different execution path of a function, all these implicit declarations should result in the same type.
- All uses of an implicitly declared variable must be compatible with its implicit type.

Examples.

```
rascal> int max = 100;
int: 100

rascal> min = 0;
int : 0

rascal> 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>}
```

Declaration Tag

Tags are not yet implemented.

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 syntax.

Tags are intended to add meta data 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               %% todo note for all 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;
}
```

Node Annotation Declaration

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 following constructs are provided for handling annotations:

- *`Val @ Anno`*: is an expression that retrieves the value of annotation *`Anno`* of value *`Val`* (may be undefined!).
- *`Val1`*`[@`*`Anno`* `=` *`Val2`*`]`: is an expression that set the value of annotation *`Anno`* of the value *`Val1`* to *`Val2`* and return *XXX* as result.
- *`Var @ Anno = Val`*: is an assignment statement that sets the value of annotation *`Anno`* of the value of variable *`Var`* to *`Val`*.

Rewrite Rule Declaration

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 defined datatypes, they are implicitly applied when a new value (we refer to this as the *subject value*) is constructed. The scope of rewrite rules 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. Technically, this is called *innermost rewriting*. When done, the result of rewriting the original subject value is used instead of that original value.

Rules have the general form:

```
rule Name PatternWithAction
```

where *`Name`* is the name of the rule and *`PatternWithAction`* is the body of the rule consisting of a pattern and an associated action (see the section called “PatternWithAction” (page 65)for a detailed description).

Here is an example for a 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;
ok

rascal> band(band(btrue,btrue),band(btrue, bfalse));
Bool: 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.

Expressions

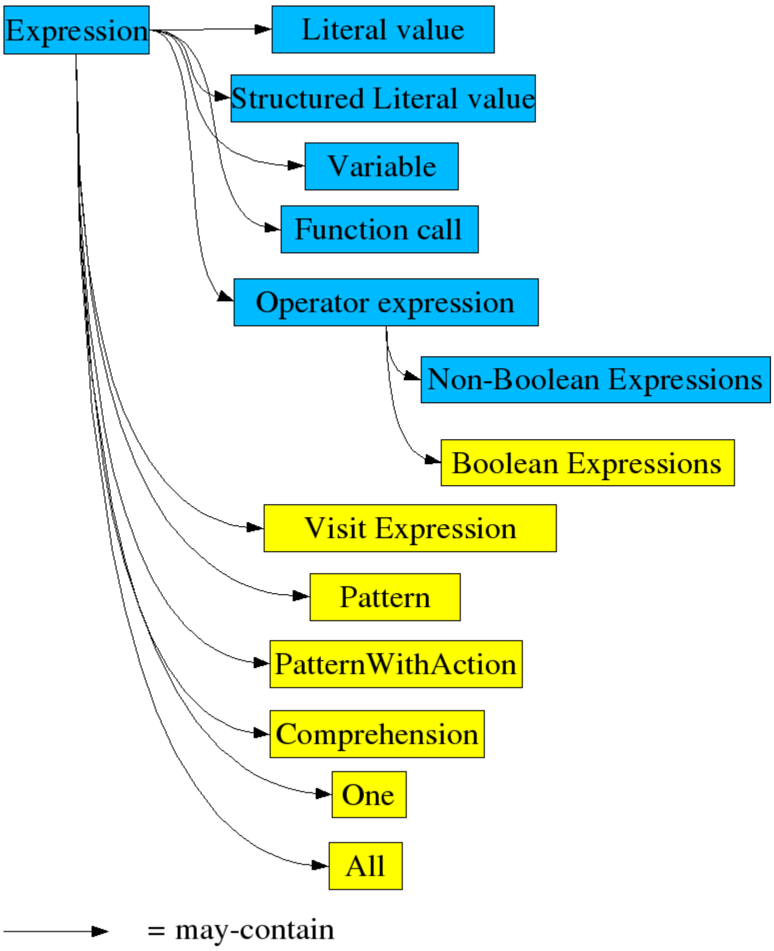
The Expression is the basic unit of evaluation and may consist of (see Figure 1.20, “Expression Forms” (page 56)):

- 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.
- A visit expression, see the section called “Visit expression” (page 67).
- A pattern, see the section called “Patterns” (page 61).
- A pattern with associated action, see the section called “PatternWithAction” (page 65).
- A comprehension, see the section called “Comprehension Expression” (page 65).
- A one expression, see the section called “One Expression” (page 69).

- An all expression, see the section called “All Expression” (page 69).

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

Figure 1.20. Expression Forms



Non-Boolean Operator Expressions

The non-Boolean operators are summarized in Table 1.2, “Non-Boolean Operators” (page 56) All operators are highly overloaded and we refer to the section called “Built-in Operators and Library Functions” (page 76) for a description of their meaning for each specific type. The following rules apply:

- All *name* arguments stand for themselves and are not evaluated.
- For all operators, except, IfThenElse, the argument expressions are first evaluated before the operator is applied.
- The operators in Table 1.2, “Non-Boolean Operators” (page 56)are listed from high precedence to low precedence. In other words, operators listed higher in the table bind stronger.

Table 1.2. Non-Boolean Operators

Operator	Name	Description
`Exp . Name`	Field selection	`Exp` should evaluate to a tuple or datatype with field `Name`; return the value of that field

Operator	Name	Description
Exp_1 [*Name* = Exp_2]	Field assignment	Exp_1 should evaluate to a tuple or datatype with a field *Name*; assign value Exp_2 to 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 returned that only contains the listed fields.
Exp_1 [Exp_2 , Exp_3, ...]	Subscription	The value of Exp_2 , Exp_3, ... are used as index in Exp_1's value. On list, tuple return the element with given (single!) index value; for map return the value associated with Exp_2 's value. On relations more than one index is allowed. All tuples are returned that have the values of Exp_2, Exp_3, ... as first elements. These values are removed from each tuple.
- *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*	Annotation selection	Value of annotation *Name* of *Exp*'s value
Exp_1 [@ *Name* = Exp_2]	Annotation replacement	Assign value of Exp_2 to annotation *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

Most of these operators are described in detail in the section called “Built-in Operators and Library Functions” (page 76) for specific types. Here we describe the remaining generic operators.

Field Selection and Field Assignment. Field selection and field assignment apply to all values that have named components like tuples and relations with named elements, data types, and locations. Field selection returns the value of the named component and Field assignment returns a new value in which the named component has been replaced by a new value. We illustrate this for tuples:

```
rascal> tuple[int key, str val] T = <1, "abc">;
tuple[int, str] : <1, "abc">

rascal> T.val;
str : "abc"
```

```
rascal> T[val = "def"];
tuple[int, str] : <1, "def">

rascal> T;
tuple[int, str] : <1, "abc">
```

Observe that field assignment creates a *new* value with an updated field. The old value remains unchanged as can be seen from the unchanged value of T in the above example. Inthe section called “Assignment Statement” (page 70) we explain how to change a field of variable value.

Field projection. Field projection applies to tuples and relations and may contain element names or integer constants that refer to elements in the order in which they occur in the original value (counting from 0). A field projection returns a new value that consists of the selected elements. Suppose we have a relation with traffic information that records the name of the day, the day number, and the length of the traffic jams at that day.

```
rascal> rel[str day, int daynum, int length] Traffic =
        {<"mon", 1, 100>, <"tue", 2, 150>, <"wed", 3, 125>,
         <"thur", 4, 110>, <"fri", 5, 90>};
rel[str, int, int]: {<"thur",4,110>,<"tue",2,150>,<"wed",3,125>,
         <"fri",5,90>,<"mon",1,100>}

rascal> Traffic<length,daynum>;
rel[int, int]: {<110,4>,<150,2>,<90,5>,<100,1>,<125,3>}

rascal> Traffic<2,day>;
rel[int, str]: {<125,"wed">,<110,"thur">,<100,"mon">,<90,"fri">,
                <150,"tue">}
```

Field projection thus selects parts from a larger value that has a fixed number of parts. The selection is based on position and not on value and can be used to completely reorder or remove the parts of a larger value.

Subscription. Subscription selects values with a given computed index from a larger value that has a variable number of elements. For lists and tuples a single integer expression is allowed as index and the returned value is the element with that index (counting from 0). For maps, the index type should correspond to the key type of the map and the value associated with the index is returned. For relations, more than one index expression is allowed and as value a new, reduced, relation is returned with all elements that contained the index values at the corresponding tuples positions (but these values are removed, hence a reduced relation). Some examples illustrate this:

```
rascal> L = [10, 20, 30];
list[int] : [10,20,30]

rascal> L[1];
int : 20

rascal> T = <"mon", 1>;
tuple[str,int] : <"mon", 1>;

rascal> T[0];
str : "mon"

rascal> colors = ("hearts":"red", "clover":"black",
                  "trumps":"black", "clubs":"red");
map[str,str] : ("hearts":"red", "clover":"black",
                "trumps":"black", "clubs":"red")

rascal> colors["trumps"];
```

```
str: "black"

rascal> colors[0];
Static Error: -:1,0: Expected str, but got int

rascal> colors["square"];
Uncaught Rascal Exception: -:1,0: NoSuchKey("square")

rascal> rel[str country, int year, int amount] GDP =
           {<"US", 2008, 14264600>, <"EU", 2008, 18394115>,
            <"Japan", 2008, 4923761>, <"US", 2007, 13811200>,
            <"EU", 2007, 13811200>, <"Japan", 2007, 4376705>};
rel[str,int,int] :
           {<"US", 2008, 14264600>, <"EU", 2008, 18394115>,
            <"Japan", 2008, 4923761>, <"US", 2007, 13811200>,
            <"EU", 2007, 13811200>, <"Japan", 2007, 4376705>}

rascal> GDP["Japan"];
rel[int,int] : {<2008, 4923761>, <2007, 4376705>}

rascal> GDP["Japan", 2008];
set[int] : {4923761}
```

Other Non-Boolean Operator Expressions. The other non-Boolean operator expressions are explained in more detail for each datatype, see the section called “Built-in Operators and Library Functions” (page 76).

Boolean Operator Expressions

The Boolean operators are summarized in Table 1.3, “Boolean Operators” (page 60). All operators are highly overloaded and we refer to the section called “Built-in Operators and Library Functions” (page 76) 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” (page 61), a preview is useful 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.

Match Operator. 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 yields `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” (page 61)--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]
```

By definition `list[int] L` and `list[int] M` match list elements that are part of the enclosing list in which they occur. If they should match a nested list each should be enclosed in list brackets.

There are two solutions for the above 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 false 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:

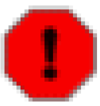
```
[1, list[int] L, 2, list[int] M] := [1,2,3,2,4] && size(L) > 0
```

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 the 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, see the section called “Comprehension Expression” (page 65).
- Tests in for, one, all statements, see the section called “Statements” (page 69).



Make the above more specific.

Table 1.3. 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

Operator	Name	Description
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 if that is defined (e.g., excluding uninitialized variables and undefined map elements) 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 (set/list element, subtree) that matches `Pat`

Patterns

Patterns come in three flavours:

- *Regular expression patterns* to do string matching with regular expressions, see the section called "Regular Expression Patterns" (page 61).
- *Abstract patterns* to matching on arbitrary values, see the section called "Abstract Patterns" (page 62).
- *Concrete syntax patterns* to match syntax trees that are the result of parsing, see the section called "Concrete Syntax Patterns" (page 64).

Regular Expression 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 uninitialized 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 *multi-variable pattern*

  ```
  Var*
  ```

  A multi-variable is an abbreviation for a variable declaration pattern. It can occur in a list pattern or set pattern and can match zero or more list or set elements.
- 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 initialized) then it matches any value. That value is assigned to *`Var`*. *Explain scope.*
- A *list pattern*

  $[\ Pat_1,\ Pat_2,\ \ldots,\ Pat_n\ ]$

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 initialized, 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 initialized, 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 a 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 *descendant pattern*

```
/ Pat
```

performs a deep match of the pattern *Pat*. In other words, it matches when any element of the subject at any depth matches *Pat* and is used to match, for instance, tree nodes at an arbitrary distance from the root.

- 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*.

- A *type constrained pattern*

```
[Type] Pat
```

matches provided that the subject has type *Type* and *Pat* matches.

Map patterns are currently not supported.

Concrete Syntax Patterns

Concrete patterns are currently being 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 patterns and variable patterns.

- A *typed quoted pattern*

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

is a quoted pattern that is preceded 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.

PatternWithAction

Patterns can be used in various contexts, but a common context is a *PatternWithAction*, which in its turn, may be used in a visit expression (see the section called “Visit expression” (page 67)), a switch statement (see the section called “Switch Statement” (page 70)), or a rewrite rule (see the section called “Rewrite Rule Declaration” (page 54)).

A *PatternWithAction* can have one of the following forms:

- `Pat => Exp`

  When the subject matches *Pat*, the expression *Exp* is evaluated. The use of the resulting value depends on the context and is described in the relevant section.

- `Pat : Statement`

  This is the most general case. When the subject matches *Pat*, the *Statement* is executed. The execution of *Statement* should, depending on the context, 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 PatternWithAction as a whole fails.
  - None of the above: execution continues with the statement following the switch.

Comprehension Expression

We will use the familiar notation for *list comprehension*

$$[Exp_1, \ldots, Exp_n \mid Gen_1, \ldots, Gen_m]$$

to denote the construction of a list consisting of the successive values of the *contributing expressions* $Exp_1, \ldots, Exp_n$. The values and the resulting list are determined by $Exp_1, \ldots, Exp_n$ and the *generators* Gen_1 ,..., Gen_m. Exp_1, ..., Exp_n are 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. A generator can use the variables introduced by preceding 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*

$\{Exp_1,\ ...,\ Exp_n \mid Gen_1,\ ...,\ Gen_m\}$

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

Finally, *`map comprehension`*s are written as:

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

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 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 enumerator 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.

Type information will be used to check the plausibility 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.

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

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

These take only effect for enumerators that produce the elements of a 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

Here are some examples of comprehensions:

```
rascal> {X | int X <- {1, 2, 3, 4, 5}, X >= 3};
set[int] : {3,4,5}

rascal> {<X, Y> | int X <- {1, 2, 3}, int Y : {2, 3, 4}, X >= Y};
rel[int,int] : {<2, 2>, <3, 2>, <3, 3>}

rascal> {<Y, X> | <int X, int Y> <- {<1,10>, <2,20>}};
rel[int,int] : {<10,1>, <20,2>}

rascal> {X, X * X | int X <- {1, 2, 3, 4, 5}, X >= 3};
set[int] : {3,4,5,9,16,25}
```

Visit expression

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 PatternWithAction1;
case PatternWithAction2;
...
default: ...
}
```

Given a subject term (the current value of *`Exp`*) and a list of cases (consisting of PatternWithActions, see the section called “PatternWithAction” (page 65**)** it traverses the term. Depending on the precise actions 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 actions 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 preceded 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 has the following effect:

- PatternWithActions of the form *`Pat`* => *`Exp`* insert their result in the subject.
- For PatternWithActions of the form *`Pat`* `:` *`Statement`*, executing *`Statement`* should lead to one of the following:
  - Execution of an `insert` statement of the form

    ```
    insert Expr
    ```

    The value of *`Exp`* 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.

    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.

The precise behaviour of the visit expression 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.

Have strategies any effect for non-tree subjects?

One Expression

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

The one expression yields true when one combination of values of Exp_i is true. *The status of one is under discussion.*

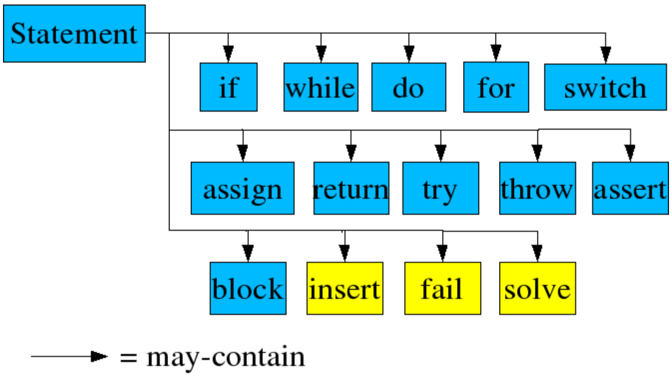
All Expression

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

The all expression yields true when all combinations of values of Exp_i are true. *The status of one is under discussion.*

Statements

Figure 1.21. Various Statement Forms



The various statements forms are summarized in Figure 1.21, "Various Statement Forms" (page 69). The more advanced statements are show in a different color.

If Statement

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"(page 59)that boolean expression maybe multi-valued. In this case only the first true value (if any) is used.

While Statement

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 Statement

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 Statement

The for-statement is more exciting:

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

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

Switch Statement

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

```
switch ( Exp ) {
case PatternWithAction1;
case PatternWithAction2;
...
default: ...
}
```

The value of the expression `Exp` is the subject term that will be matched by the successive PatternWithActions (see the section called “PatternWithAction” (page 65)) in the switch statement. 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.

Assignment Statement

The purpose of an assignment is to assign a new value to a simple variable or to an element of a more complex data structure. The most general form of an assignment statement is

```
Assignable AssignmentOp Exp
```

where `AssignmentOp` may be `=`, `+=`, `-=`, `*=`, `/=`, or `?=`. Here = is the ordinary assignment operators and the other forms can be derived from it according to Table 1.4, “Assignment Operators” (page 70).

Table 1.4. Assignment Operators

Assignment Operator	Equivalent to
`Assignable += Exp`	`Assignable = Assignable + Exp`
`Assignable -= Exp`	`Assignable = Assignable - Exp`
`Assignable *= Exp`	`Assignable = Assignable * Exp`
`Assignable /= Exp`	`Assignable = Assignable / Exp`
`Assignable ?= Exp`	`Assignable = Assignable ? Exp`

An assignable is either a single variable, (the *base variable*), optionally followed by subscriptions or field selections. The assignment statement always results in assigning a *completely new value* to the base variable. We distinguish the following forms of assignment:

- `Var = Exp`

The expression *Exp* is evaluated and its value is assigned to the base variable *Var*.

- *Assignable [Exp_1] = Exp_2*

  First the value V of *Assignable* is determined. Next the value of Exp_1 is used as index in *V* and the value of Exp_2 replaces the original value at that index position. The result is a new value *V'* that is assigned to the *Assignable*.

- *Assignable . Name = Exp*

  The value V of *Assignable* is determined and should be of a type that has a field Name. The value of that field is replaced in *V* by the value of *Exp* resulting in a new value V' that is assigned to Assignable.

- *< $Assignable_1$, $Assignable_2$, ..., $Assignable_n$ > = Exp*

  First the value *Exp* is determined and should be a tuple of the form <V_1, V_2, ..., V_n>. Next the assignments $Assignable_i$ = V_i are performed for 1 <= i <= n.

- *Assignable ? Exp_1 = Exp_2*

  First the value of Exp_2 is determined and if that is defined it is assigned to Assignable. Otherwise, the value of Exp_1 is assigned to *Assignable*.

- *Assignable @ Name = Exp*

  The value V of *Assignable* is determined and should be of a type that has an annotation *Name*. The value of that annotation is replaced in *V* by the value of *Exp* resulting in a new value V' that is assigned to *Assignable*.

- *Name ($Assignable_1$, $Assignable_2$, ...) = Exp*

  First the value *Exp* is determined and should be a data value of the form *Name*(V_1, V_2, ...,V_n). Next the assignments $Assignable_i$ = V_i are performed for 1 <= i <= n.

  Constructor assignable is not yet implemented.

Here are some examples:

```
rascal> N = 3;
int : 3

rascal> N;
int : 3

rascal> L = [10,20,30];
list[int] : [10,20,30]

rascal> P = L;
list[int] : [10,20,30]

rascal> L[1] = 200;
list[int] : [10,200,30]

rascal> P;  // Value of P is unchanged!
list[int] : [10,20,30]
```

```
rascal> M = ("abc": 1, "def" : 2);
map[str,int] : ("abc": 1, "def" : 2)

rascal> M["def"] = 3;
map[str,int] : ("abc": 1, "def" : 3)

rascal> T = <1, "abc", true>;
tuple[int,str,bool] : <1, "abc", true>

rascal> T[1] = "def";
tuple[int,str,bool] : <1, "def", true>

rascal> data FREQ = wf(str word, int freq);
ok

rascal> W = wf("rascal", 1000);
FREQ : wf("rascal", 1000)

rascal> W.freq = 100000;
FREQ : wf("rascal",100000)

rascal> <A, B, C> = <"abc", 2.5, [1,2,3]>;
tuple[str,real,list[int]] : <"abc", 2.5, [1,2,3]>

rascal> A;
str : "abc"

rascal> B;
real : 2.5

rascal> C;
list[int] : [1,2,3]

rascal> ** good V ? E1 = E2 example here **


rascal> anno str FREQ@color;
ok

rascal> W @ color = "red";
FREQ: wf("rascal",100000)[@color="red"]

rascal> wf(S, I) = W;
ok

rascal> S;
str : "rascal"

rascal> I;
int : 100000
```

Return Statement

A return statement has either the form

```
return;
```

or

```
return Exp;
```

both end the execution of the current function. The first form applies to functions with `void` as return type. The second form applies to non-void functions and returns the value of *`Exp`* as result of the function. The following rules apply:

- The static type of *`Exp`* should be compatible with the declared return type of the function in which the return statement occurs.
- In each function with a return type that is not `void`, every possible execution path through the body of the function end in a return statement.
- In each function with a return type that is `void`, a return statement is implicitly assumed at the end of each execution path through the function body.

Try Catch Statement

A try catch statement has the form

```
try
   Statement1;
catch PatternWithAction1;
catch PatternWithAction2;
...
catch: Statement2;
finally: Statement3;
```

and has as purpose to catch any exceptions that are raised during the execution of *`Statement`*$_1$. These exceptions may caused by:

- The execution of an explicit throw statement, see the section called “Throw Statement” (page 73).
- The Rascal system that discover an abnormal condition, e.g., an out of bounds error when accessing a list element.

Note that all elements of the try catch statement are optional but that at least one has to be present. Their meaning is as follows:

- If a pattern of some *`PatternWithAction`*$_i$ matches, the corresponding action is executed.
- Otherwise, *`Statement`*$_2$ is executed (when present).
- Before leaving the try catch statement *`Statement`*$_3$ is always executed (when present).

Throw Statement

A throw statement has the form

```
throw Exp;
```

and causes the immediate abortion of the execution of the current function with *`Exp`*'s value as exception value. The exception can be caught by a try catch statement (the section called “Try Catch Statement” (page 73**)** in the current function or in one of its callers. If the exception is not caught, the execution of the Rascal program is terminated. The following rules apply:

- The static type of *`Exp`* should be `RuntimeException`, see the section called “Exception” (page 78).
- The Rascal program may contain data declarations that extend the type RuntimeException.

Assert Statement

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

$$\texttt{assert}\ Exp_1$$

and

$$\texttt{assert}\ Exp_1\ \texttt{:}\ Exp_2$$

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.

Update the above description.

Example:

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

Insert Statement

An insert statement has the form

$$\texttt{insert}\ Exp\texttt{;}$$

and replaces the value of the *current subject* (see below) by the value of *Exp*. An insert statement may only occur in the action part of a PatternWithAction (see the section called “PatternWithAction” (page 65)), more precisely in

- A case in a visit expression, see the section called “Visit expression” (page 67) The current subject is the value matched by the pattern of this case.
- An action of a rewrite rule, see the section called “Rewrite Rule Declaration” (page 54) The current subject is the value matched by the pattern of the rewrite rule.

The following rule applies:

- The static type of Exp and of the current subject should be comparable.

Fail Statement

A fail statement has the form

```
fail;
```

and may only occur in the action of PatternWithAction (see the section called “PatternWithAction” (page 65**)**. The fail statement forces the failure of that action. Any bindings caused by the pattern or side-effects caused by the action are undone.

Block Statement

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

$$\texttt{\{}\ Statement_1\texttt{;}\ \texttt{...}\ Statement_n\ \texttt{\}}$$

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 declared in the block are local to that block.

Solve Statement

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

rel[int,int] : {<1,2>, <1,3>,<1,4>,<2,3>,<2,4>,<3,4>}
```

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” (page 77).
- Boolean: operators and functions on Boolean values, see the section called “Boolean” (page 77).
- Exception: data definition of all soft exceptions that can be caught by Rascal programs, see the section called “Exception” (page 78).
- Graph: graphs are a special kind of binary relation, see the section called “Graph” (page 78).
- Integer: operators and functions on integers, see the section called “Integer” (page 79).
- IO: simple print functions, see the section called “IO” (page 80).
- JDT: Java fact extraction functions, see the section called “JDT (Eclipse only)” (page 80).
- Labelled Graph: labelled graphs with addition edge information, see the section called “Labelled Graph” (page 83).
- List: operators and functions on lists, see the section called “List” (page 84).
- Location: operators and functions on source locations, see the section called “Location” (page 85).
- Map: operators and functions on maps, see the section called “Map” (page 86).
- Node: operators and functions on nodes, see the section called “Node” (page 87).
- Real: operators and functions on reals, see the section called “Real” (page 87).
- Relation: operators and functions on relations, see the section called “Relation” (page 88).
- Resource: functions to retrieve resources from an Eclipse workspace, see the section called “Resource (Eclipse only)” (page 90).
- RSF: function for reading files in Rigi Standard Format, see the section called “RSF” (page 90).
- Set: operators and functions on sets, see the section called “Set” (page 91).
- String: operators and functions on strings, see the section called “String” (page 93).
- Tuple: operators and functions on tuples, see the section called “Tuple” (page 94).
- UnitTest: functions for unit testing, see the section called “UnitTest” (page 95).
- ValueIO: functions for reading and writing Rascal values, both in textual and in binary form, see the section called “ValueIO” (page 95).
- View: functions for graphical display of values in Eclipse, see the section called “View (Eclipse only)” (page 95).
- Void: the type void, see the section called “Void” (page 96).

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 1.5, “Notational conventions” (page 77) When an operator has more than one argument of the same type, they are distinguished by subscripts.

Table 1.5. 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 1.6. 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 1.7. Boolean Operators

Operator	Description
$Bool_1$ == $Bool_2$	Yields `true` if both arguments are identical
$Bool_1$!= $Bool_2$	Yields `true` if both arguments are not identical
$Bool_1$ <= $Bool_2$	Yields `true` if both arguments are identical or $Bool_1$ is `false` and $Bool_2$ is `true`
$Bool_1$ < $Bool_2$	Yields `true` if $Bool_1$ is `false` and $Bool_2$ is `true`
$Bool_1$ >= $Bool_2$	Yields `true` if both arguments are identical or $Bool_1$ is `true` and $Bool_2$ is `false`
$Bool_1$ > $Bool_2$	Yields `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

Operator	Description
! *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 1.8. 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)
    | ParseError(loc location)
    ;
```

Graph

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

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

Table 1.9. Graph Functions

Function	Description
`set[&T] bottom(graph[&T] G)`	Leaf nodes of a graph
set[&T] predecessors(graph[&T], &T From)	Direct predecessors of node From
set[&T] predecessors(graph[&T], set[&T] FromSet)	Direct predecessors of all nodes in FromSet

Function	**Description**
`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
set[&T] successors(graph[&T], &T From)	Direct successors of node From
set[&T] successors(graph[&T], set[&T] FromSet)	Direct successors of all nodes in FromSet
`set[&T] top(graph[&T] G)`	Root nodes of a graph

The following examples illustrate these functions:

```
rascal> top({<1,2>, <1,3>, <2,4>, <3,4>});
set[int] : {1}

rascal> bottom({<1,2>, <1,3>, <2,4>, <3,4>});
set[int] : {4}

rascal> reachR({<1,2>, <1,3>, <2,4>, <3,4>}, {1}, {1, 2, 3});
set[int] : {2, 3}

rascal> reachX({<1,2>, <1,3>, <2,4>, <3,4>}, {1}, {2});
set[int,int] : {3, 4}
```

Integer

Rascal integers are unbounded in size.

Table 1.10. Integer Operators

Operator	**Description**
Int_1 == Int_2	Yields `true` if both arguments are numerically equal and `false` otherwise
Int_1 != Int_2	Yields `true` if both arguments are numerically unequal and `false` otherwise
Int_1 <= Int_2	Yields `true` if Int_1 is numerically less than or equal to Int_2 and `false` otherwise
Int_1 < Int_2	Yields `true` if Int_1 is numerically less than Int_2 and `false` otherwise
Int_1 >= Int_2	Yields `true` if Int_1 is numerically greater than or equal than Int_2 and `false` otherwise
Int_1 > Int_2	Yields `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

Operator	Description
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 1.11. 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 1.12. IO Functions

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

JDT (Eclipse only)

Detailed information can be extracted from Java projects in the current Eclipse workspace. This proceeds in two steps:

- First all facts are extracted from given projects or files. The result is a `FactMap` that contains all data. A FactMap is a datatype only used by the JDT functions and is not further described here. The project/file extraction functions are shown in Table 1.13, “JDT Project/File Extraction Functions” (page 81).
- Next, specific facts about the Java source code can be retrieved from the `FactMap`. These functions are shown in Table 1.14, “JDT Fact Retrieval Functions” (page 82) They return values of type `BindingRel` or `TypeInfoRel` that are explained below.

The Java constructs that may occur in extracted facts are represented by the datatype Entity that is defined in the library module `Java` (not further described here). Entity is defined as follows:

```
data Entity = entity(list[Id] id);

data Id = package(str name)
        | class(str name)
        | class(str name, list[Entity] params)
        | interface(str name)
        | interface(str name, list[Entity] params)
        | anonymousClass(int nr)
        | enum(str name)
```

```
        | method(str name, list[Entity] params, Entity returnType)
        | constructor(list[Entity] params)
        | initializer
        | initializer(int nr)

        | field(str name)
        | parameter(str name)
        | variable(str name)
        | enumConstant(str name)

        | primitive(PrimitiveType type)
        | array(Entity elementType)

        | typeParameter(str name)
        | wildcard
        | wildcard(Bound bound)
        ;


data PrimitiveType =
          byte
        | short
        | \int
        | long
        | float
        | double
        | char
        | boolean
        | \void
        | null
        ;

data Bound =
          extends(Entity type)
        | super(Entity type)
        ;
```

Observe how the Rascal keywords `int` and `void` are used constructo symbols in the above data definition by escaping them as `\int` and `\void`.

The extraction functions use the following additional types:

```
public alias JDTlocation =
             tuple[str fileName, int offset, int length];
public alias BindingRel = rel[JDTlocation, Entity];
public alias TypeInfoRel = rel[Entity, Entity];
```

`JDTLocation` describes a text area in a given file. `BindingRel` associates JDTLocations with Entities. `TypeInfoRel` associates Entities with each other.

Table 1.13. JDT Project/File Extraction Functions

Function	Description
`FactMap java extractFacts(str project, str file)`	Import JDT facts from file (path relative to project root)
`FactMap extractFacts(str project, loc file)`	Import JDT facts from file (absolute file system path)

Function	Description
`FactMap extractFacts(set[str] projects)`	Extract facts from projects
`FactMap extractFacts(str projectName)`	Extract facts from a single project
`FactMap extractFactsTransitive(set[str] projects)`	Extracts facts from projects and all projects they depends on (transitively)
`tuple[rel[&T1, &T2] found, rel[JDTlocation, &T2] notfound] matchLocations(rel[&T1, loc] RSClocs, rel[JDTlocation, &T2] JDTlocs)`	Compose two relations by matching JDT locations with Rascal locations and return a tuple with the composition result and the locations that could not be matched. The source code locations of the JDT AST nodes might not always be the same as the ones in your own parse tree. This function picks the 'best fitting' user provided locations for JDT locations that don't have a direct match. It works correctly for the Java 1.4 grammar in the sdf-library, but other grammars might need a specific implementation.
`FactMap unionFacts(FactMap m1, FactMap m2)`	Union fact maps, union values for facts that appear in both maps (if possible)

Table 1.14. JDT Fact Retrieval Functions

Function	Description
`BindingRel getConstructorBindings(FactMap fm)`	Get all fully-qualified constructor name that occur in declarations or calls
`TypeInfoRel getDeclaredFields(FactMap fm)`	Get all declared fields in each class
`TypeInfoRel getDeclaredMethods(FactMap fm)`	Get all declared methods in each class
`TypeInfoRel getDeclaredTypes(FactMap fm)`	Get the relation between each class and its inner classes
`TypeInfoRel getExtends(FactMap fm)`	Get the subclass (extends) relation
`BindingRel getFieldBindings(FactMap fm)`	Get all fully-qualified field names
`TypeInfoRel getImplements(FactMap fm)`	Get the interface implementation (implements) relation
`BindingRel getMethodBindings(FactMap fm)`	Get all fully-qualified method names that occur in declarations or calls
`BindingRel getTypeBindings(FactMap fm)`	Get the fully-qualified types of all expressions and identifiers
`BindingRel getVariableBindings (FactMap fm)`	Get all fully-qualified variable names of all local variables and method parameters

This is the implementation of `extractFacts(set[str] projects)`, showing how to use the unionFacts() function :

```
public FactMap extractFacts(set[str] projects) {
  FactMap result = ();
```

```
  for (p <- projects) {
    result = unionFacts(result, extractFacts(p));
  }
  return result;
}
```

Here is an example function that extracts the inheritance hierarchy from a given project:

```
TypeInfoRel getInheritance(str projectName){
  fm = extractFacts(projectName);
  return getExtends(fm) + getImplements(fm);
}
```

The following example shows how to link the JDT type bindings to your own parse tree nodes. Suppose you have a relation `nodeLocations` that links your parse tree nodes to their location, you can get the type information of each node (if any) as follows:

```
FactMap fm = extractFacts(myProject, myJavaFile);
BindingRel jdtTypeBindings = getTypeBindings(fm);

rel[node, Entity] myTypeBindings;
BindingRel unmatchedBindings;
<myTypeBindings, unmatchedBindings> =
      matchLocations(nodeLocations, jdtTypeBindings);
```

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 1.15. Labelled Graph Functions

Function	Description
`set[&T] bottom(lgraph[&T] G)`	Bottom nodes of a labelled graph
set[&T] predecessors(graph[&T], &T From)	Direct predecessors of node From
set[&T] predecessors(graph[&T], set[&T] FromSet)	Direct predecessors of all nodes in FromSet
`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
set[&T] successors(graph[&T], &T From)	Direct successors of node From
set[&T] successors(graph[&T], set[&T] FromSet)	Direct successors of all nodes in FromSet
`set[&T] top(lgraph[&T] G)`	Top nodes of a labelled graph

shortestPath not yet implemented for lgraph.

List

Table 1.16. List Operators

Operator	Description
List$_1$ == *List*$_2$	Yields `true` if both arguments have the same elements in the same order
List$_1$!= *List*$_2$	Yields `true` if both arguments have different elements
List$_1$ <= *List*$_2$	Yields `true` if both lists are equal or *List*$_1$ is a sublist of *List*$_2$
List$_1$ < *List*$_2$	Yields `true` if *List*$_1$ is a sublist of *List*$_2$
List$_1$ >= *List*$_2$	Yields `true` if both lists are equal or *List*$_2$ is a sublist of *List*$_1$
List$_1$ > *List*$_2$	Yields `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*	Yields `true` if *Elm* occurs as element in *List*
Elm notin *List*	Yields `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 1.17. 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) throws IndexOutOfBounds(int index)`	Delete nth element from list
`set[int] domain(list[&T] lst)`	Set of all legal index values for a list
`&T head(list[&T] lst) throws EmptyList`	Get the first element of a list
`list[&T] head(list[&T] lst, int n) throws IndexOutOfBounds(int index)`	Get the first n elements of a list
`&T getOneFrom(list[&T] lst) throws EmptyList`	Get an arbitrary element from a list
`list[&T] insertAt(list[&T] lst, int n, &T elm) throws IndexOutOfBounds(int index)`	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 unit)`	Multiply the elements of a list using given unit element
`set[list[&T]] permutations(list[&T] lst)`	All permutations of a list
`&T reducer(list[&T] lst, &T (&T, &T) fn, &T unit)`	Apply function `fn` to successive elements of a list

Function	**Description**
`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) throws IndexOutOfBounds(int index)`	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) throws EmptyList`	All but the first element of a list
`list[&T] tail(list[&T] lst, int len) throws IndexOutOfBounds(in index)`	Last n elements of a list
`tuple[&T, list[&T]] takeOneFrom(list[&T] lst) throws EmptyList`	Remove an arbitrary element from a list, returns the element and the modified list
`map[&A,&B] toMap(list[tuple[&A, &B]] lst) throws DuplicateKey`	Convert a list of tuples to a map; result must be a map
map[&A,set[&B]] toMap(list[tuple[&A, &B]] lst)	Convert a list of tuples to a map in which the first element of each tuple is associated with the set of second elements from all tuples with the same first element
`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 1.18. Operations on Locations

Operator	**Description**
Loc_1 == Loc_2	Yield `true` if both arguments are identical and `false` otherwise
Loc_1 != Loc_2	Yield `true` if both arguments are not identical and `false` otherwise
Loc_1 <= Loc_2	Yield `true` if Loc_1 is textually contained in or equal to Loc_2 and `false` otherwise
$Loc_1 < Loc_2$	Yield `true` if Loc_1 is strictly textually contained in Loc_2 and `false` otherwise
Loc_1 >= Loc_2	Yield `true` if Loc_1 textually encloses or is equal to Loc_2 and `false` otherwise
Loc_1 >= Loc_2	Yield `true` if Loc_1 textually encloses Loc_2 and `false` otherwise
Loc . *Field*	Retrieve one of the fields of location value *Loc*

The field names for locations are:

- `url`
- `offset`
- `length`
- `beginLine`, `beginColumn`

- `endLine`, `endColumn`.

Map

Table 1.19. Map Operators

Operator	**Description**
Map_1 == Map_2	Yield `true` if both arguments consist of the same pairs
Map_1 != Map_2	Yield `true` if both arguments have different pairs
Map_1 <= Map_2	Yield `true` if all pairs in Map_1 occur in Map_2 or Map_1 and Map_2 are equal
Map_1 < Map_2	Yield `true` if all pairs in Map_1 occur in Map_2 but Map_1 and Map_2 are not equal
Map_1 >= Map_2	Yield `true` if all pairs in Map_2 occur in Map_1 or Map_1 and Map_2 are equal
Map_1 > Map_2	Yield `true` if all pairs in Map_2 occur in Map_1but 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*	Yield `true` if *Key* occurs in a key:value pair in *Map*
Key_1 `notin` Map_2	Yield `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 1.20. Map Functions

Function	**Description**
`set[&K] domain(map[&K, &V] M)`	The domain (keys) of a map
`&K getOneFrom(map[&K, &V] M) throws emptyMap`	Arbitrary key of a map
`map[&V, &K] invertUnique(map[&K, &V] M) throws MultipleKey`	Map with key and value inverted; result should be a map
map[&V, set[&K]] invert(map[&K, &V] M)	Inverted map in which each value in the old map is associated with a set of key values from the old map
`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 1.21. 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`$_2$	*Explain*
`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 1.22. Node Functions

Function	Description
`int arity(node T)`	Number of children of a node
`list[value] getChildren(node T)`	The children of a node
`str getName(node T)`	The function name of a node
`node makeNode(str N, value V...)`	Create a node given its function name and arguments
node readAtermFromFile(str Name) throws IO(str msg)	*under discussion*

Real

Table 1.23. 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 1.24. Real Functions

Function	Description
`real arbReal()`	Arbitrary real value in the interval [0.0,1.0).
`real max(real n, real m)`	Largest of two reals
`int toInt(real d)`	Convert a real to integer. *Explain*
`str toString(real d)`	Convert a real to a string.

Relation

Relation are sets of tuples, therefore all set operators (see, Table 1.29, “Set Operators” (page 91) apply to relations as well

Table 1.25. Operations on Relations

Operator	Description
Rel_1 o Rel_2	Relation resulting from the composition of the two arguments
Set_1 x Set_2	Relation resulting from the Cartesian product of the two arguments
Rel +	Relation resulting from the transitive closure of *Rel*. Explain type constraints.
Rel *	Relation resulting from the reflexive transitive closure of *Rel*
Rel [Exp_1, Exp_2, ...]	Relation resulting from subscription of *Rel* with the index values of Exp_1, Exp_2, ... The result is a relation with all tuples that have these index values as first elements with the index values removed from the tuple. If the resulting tuple has only a single element, a set is returned instead of a relation. A wildcard _ as index value matches all possible values at that index position.
Rel [*Set*]	Relation resulting from subscription of *Rel* with all elements of *Set*.
Rel < $Index_1$, $Index_2$, ... >	Relation resulting from restricting *Rel* to the columns described by $Index_1$, $Index_2$, ...

Examples:

```
rascal> {<1,10>, <2,20>, <3,15>} o {<10,100>, <20,200>};
rel[int,int] : {<1,100>, <2,200>}

rascal> {1, 2, 3} x {9};
rel[int,int] :  {<1, 9>, <2, 9>, <3, 9>}

rascal> {<1,2>, <2,3>, <3,4>}+;
rel[int,int]: {<1,2>, <2,3>, <3,4>, <1, 3>, <2, 4>, <1, 4>}

rascal> {<1,2>, <2,3>, <3,4>}*;
rel[int,int]: {<1,2>, <2,3>, <3,4>, <1, 3>, <2, 4>, <1, 4>,
               <1, 1>, <2, 2>, <3, 3>, <4, 4>}

rascal> R = {<1,10>, <2,20>, <1,11>, <3,30>, <2,21>};
rel[int,int] : {<1,10>, <2,20>, <1,11>, <3,30>, <2,21>}

rascal> R[1];
set[int] : {10, 11}

rascal> R[{1}];
set[int] : {10, 11}
```

```
rascal> R[{1, 2}];
set[int] : {10, 11, 20, 21}

rascal> RR = {<1,10,100>,<1,11,101>,<2,20,200>,<2,22,202>,
              <3,30,300>};
rel[int,int,int]:{<1,10,100>,<1,11,101>,<2,20,200>,<2,22,202>,
                  <3,30,300>}

rascal> RR[1];
rel[int,int]: {<10,100>,<11,101>};

rascal> RR[1,_];
set[int]: {100,101}

rascal> RR<2,0,1>;
rel[int,int,int]: {<100,1,10>,<101,1,11>,<200,2,20>,<300,3,30>};

rascal> RR<2.0,1>[100];
rel[int,int]: {<1,10>}
```

Table 1.26. 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
`rel[&T,&T] ident(set[&T] S)`	The identity relation for set S.
`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 tuple 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:

```
rascal> id({1,2,3});
rel[int,int] : {<1,1>, <2,2>, <3,3>}

rascal> id({"mon", "tue", "wed"});
rel[str,str] : {<"mon","mon">, <"tue","tue">, <"wed","wed">}

rascal> invert({<1,10>, <2,20>});
rel[int,int] : {<10,1>,<20,2>}
```

```
rascal> compl({<1,10>});
rel[int,int] : {<1, 1>, <10, 1>, <10, 10>}

rascal> domain({<1,10>, <2,20>});
set[int] : {1, 2}

rascal> domain({<"mon", 1>, <"tue", 2>});
set[str] : {"mon", "tue"}

rascal> range({<1,10>, <2,20>});
set[int] : {10, 20}

rascal> range({<"mon", 1>, <"tue", 2>});
set[int] : {1, 2}

rascal> carrier({<1,10>, <2,20>});
set[int] : {1, 10, 2, 20}

rascal> domainR({<1,10>, <2,20>, <3,30>}, {3, 1});
rel[int,int] : {<1,10>, <3,30>}

rascal> rangeR({<1,10>, <2,20>, <3,30>}, {30, 10});
rel[int,int] : {<1,10>, <3,30>}

rascal> carrierR({<1,10>, <2,20>, <3,30>}, {10, 1, 20});
rel[int,int] : {<1,10>}

rascal> domainX({<1,10>, <2,20>, <3,30>}, {3, 1});
rel[int,int] : {<2, 20>}

rascal> rangeX({<1,10>, <2,20>, <3,30>}, {30, 10});
rel[int,int] : {<2, 20>}

rascal> carrierX({<1,10>, <2,20>, <3,30>}, {10, 1, 20});
rel[int,int] : {<3,30>}
```

RSF

Table 1.27. RSF Functions

Function	Description
`map[str, rel[str,str]] readRSF(str nameRSFFile) throws IO(str msg)`	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 1.28. Resource Functions

Function	Description
`set[loc] files(str project)`	The files contained in a project

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

Set

Table 1.29. 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	Set resulting from the difference of the two arguments
Set_1 * Set_2	Relation resulting from the product of the two arguments. It contains a tuple for each combination of values from both 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	Relation resulting from the natural join of the two arguments. It contains tuples that are the result from concatenating the elements from both arguments
Bool ? Set_1 : Set_2	If *Bool* is true then Set_1 else Set_2

Examples:

```
rascal> {1, 2, 3} + {4, 5, 6};
set[int] : {1, 2, 3, 4, 5, 6}

rascal> {1, 2, 3} + {1, 2, 3};
set[int] : {1, 2, 3}

rascal> {1, 2, 3, 4} - {1, 2, 3};
set[int] : {4}

rascal> {1, 2, 3} - {4, 5, 6};
set[int] : {1, 2, 3}

rascal> {1,2,3} * {4,5,6};
rel[int,int]: {<1,4>,<1,5>,<1,6>,<3,4>,<3,5>,
               <3,6>,<2,4>,<2,5>,<2,6>}

rascal> {1, 2, 3} & {4, 5, 6};
set[int] : { }
```

```
rascal> {1, 2, 3} & {1, 2, 3};
set[int] : {1, 2, 3}

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

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

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

rascal> 4 notin {1, 2, 3};
bool : true

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

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

rascal> {<1,2>, <10,20>} join {<2,3>};
rel[int, int, int, int]: {<1,2,2,3>,<10,20,2,3>}

rascal> {<1,2>} join {3, 4};
rel[int, int, int]: {<1,2,3>,<1,2,4>}
```

Table 1.30. Set Functions

Function	Description
`&T average(set[&T] st, &T zero)`	Average of the elements of a set
`&T getOneFrom(set[&T] st) throws emptySet`	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 set
`&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) throws emptySet`	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,set[&B]] toMap(rel[&A, &B] st)	Convert a set of tuples to a map in which the first element of each tuple is associated with the

Function	Description
	set of second elements from all tuples with the same first element
`map[&A,&B] toMapUnique(rel[&A, &B] st) throws DuplicateKey`	Convert a set of tuples to a map; the result should be a legal map
`str toString(set[&T] st)`	Convert a set to a string

Examples:

```
rascal> power({1, 2, 3, 4});
set[set[int]] : { {}, {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});
set[set[int]] : { {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});
int : 3

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

String

Table 1.31. 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 1.32. String Functions

Function	Description
`int charAt(str s, int i) throws IndexOutOfBounds(int index)`	Character at position i in string s.
`bool endsWith(str s, str suffix)`	Yields `true` if string s ends with given string suffix.
`str center(str s, int n)`	Center s in string of length n using spaces

Function	Description
`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)`	Length of string s.
`bool startsWith(str s, str prefix)`	Yields `true` if string s starts with the string prefix.
`int toInt(str s) throws IllegalArgument(value v)`	Convert string s to integer. Throws IllegalArgument when s cannot be converted.
`str toLowerCase(str s)`	Convert all characters in string s to lowercase.
`double toReal(str s) throws IllegalArgument(value v)`	Convert s to a real. Throws IllegalArgument when s cannot be converted.
`str toUpperCase(str s)`	Convert all characters in string s to uppercase.

Tuple

Table 1.33. 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 1.34. 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 1.35. 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
$Value_1$ <= $Value_2$	*Explain*
$Value_1$ < $Value_2$	
$Value_1$ >= $Value_2$	
$Value_1$ > $Value_2$	
Bool ? $Value_1$: $Value_2$	if *Bool* is true then $Value_1$ else $Value_2$

ValueIO

Table 1.36. ValueIO Functions

Function	Description
`value readValueFromBinaryFile(str namePBFFile) throws IO(str msg)`	Read a value from a binary file in PBF format
`value readValueFromTextFile(str namePBFFile) throws IO(str msg)`	Read a value from a text file
`void writeValueToBinaryFile(str namePBFFile, value val) throws IO(str msg)`	Write a value to a binary file in PBF format
`void writeValueToTextFile(str namePBFFile, value val) throws IO(str msg)`	Write a value to a binary file in PBF format

View (Eclipse only)

Table 1.37. 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`.

Table of Built-in Operators

Table 1.38. All Operators

Operator	Description	See Library Module/Section
Exp_1[$Name$ = Exp_2]	Field assignment	Non-Boolean Operator Expressions (page 56)
Exp . $Name$	Field selection	Non-Boolean Operator Expressions (page 56), Location (page 85), Tuple (page 94)
Exp < field, ... >	Field projection	Non-Boolean Operator Expressions (page 56), Relation (page 88)
Exp_1[Exp_2 , Exp_3, ...]	Index	Non-Boolean Operator Expressions (page 56), List (page 84), Map (page 86), Node (page 87), Relation (page 88), Tuple (page 94)
Exp ?	Isdefined: true if Exp has a defined value	Boolean Operator Expressions (page 59)
! Exp	Negation	Boolean Operator Expressions (page 59), Boolean (page 77)
- Exp	Negation	Integer (page 79), Real (page 87)
Exp +	Transitive closure	Relation (page 88)
Exp *	Reflexive transitive	Relation (page 88)
Exp @ $Name$	Attribute value	
Exp_1 [@ $Name$ = Exp_2]	Assign attribute value	
Exp_1 o Exp_2	Composition	Relation (page 88)
Exp_1 x Exp_2	Cartesian product	Relation (page 88)
Exp_1 / Exp_2	Division	Integer (page 79), Real (page 87)
Exp_1 % Exp_2	Modulo	Integer (page 79), Real (page 87)
Exp_1 * Exp_2	Multiplication/product	Integer (page 79), List (page 84), Real (page 87), Set (page 91)
Exp_1 & Exp_2	Intersection	Set (page 91)
Exp_1 + Exp_2	Addition/ concatenation/union	Integer (page 79), List (page 84), Map (page 86), Real (page 87), Set (page 91), String (page 93), Tuple (page 94)
Exp_1 - Exp_2	Subtraction/difference	Integer (page 79), List (page 84), Map (page 86), Real (page 87), Set (page 91)
Exp_1 join Exp_2	Join	Set (page 91)
Exp_1 in Exp_2	Element of	Boolean Operator Expressions (page 59), List (page 84), Map (page 86), Set (page 91)
Exp_1 notin Exp_2	Not element of	Boolean Operator Expressions (page 59), List (page 84), Map (page 86), Set (page 91)
Exp_1 <= Exp_2	Less than/sublist / subset	Boolean Operator Expressions (page 59), Boolean (page 77), Integer (page 79), List (page 84), Location (page 85), Map (page 86), Node (page 87), Real (page 87),Set (page 91), String (page 93), Tuple (page 94), Value (page 95)

Operator	Description	See Library Module/Section
$Exp_1 < Exp_2$	Less than/strict sublist/ strict subset	Boolean Operator Expressions (page 59), Boolean (page 77), Integer (page 79), List (page 84), Location (page 85), Map (page 86), Node (page 87), Real (page 87), Set (page 91), String (page 93), Tuple (page 94), Value (page 95)
$Exp_1 >= Exp_2$	Greater than/superlist/ superset	Boolean Operator Expressions (page 59), Boolean (page 77), Integer (page 79), List (page 84), Location (page 85), Map (page 86), Node (page 87), Real (page 87), Set (page 91), String (page 93), Tuple (page 94), Value (page 95)
$Exp_1 > Exp_2$	Greater than/strict superlist/ strict superset	Boolean Operator Expressions (page 59), Boolean (page 77), Integer (page 79), List (page 84), Location (page 85), Map (page 86), Node (page 87), Real (page 87), Set (page 91), String (page 93), Tuple (page 94), Value (page 95)
`Pat` := Exp	Match	Boolean Operator Expressions (page 59)
`Pat` != Exp	No Match	Boolean Operator Expressions (page 59)
$Exp_1 == Exp_2$	Equality	Boolean Operator Expressions (page 59), Boolean (page 77), Integer (page 79), List (page 84), Location (page 85), Map (page 86), Node (page 87), Real (page 87), Set (page 91), String (page 93), Tuple (page 94), Value (page 95)
Exp_1 != Exp_2	Inequality	Boolean Operator Expressions (page 59), Boolean (page 77), Integer (page 79), List (page 84), Location (page 85), Map (page 86), Node (page 87), Real (page 87), Set (page 91), String (page 93),Tuple (page 94), Value (page 95)
Exp_1 ? Exp_2	Ifdefined Otherwise	Boolean Operator Expressions (page 59)
Exp_1 ? Exp_2 : Exp_3	Conditional Expression	Boolean Operator Expressions (page 59), Boolean (page 77), Integer (page 79), List (page 84), Map (page 86), Node (page 87), Real (page 87), Set (page 91), String (page 93), Tuple (page 94), Value (page 95)
Exp_1 ==> Exp_2	Implication	Boolean Operator Expressions (page 59), Boolean (page 77)
Exp_1 <==> Exp_2	Equivalence	Boolean Operator Expressions (page 59), Boolean (page 77)
Exp_1 && Exp_2	Boolean and	Boolean Operator Expressions (page 59), Boolean (page 77)
Exp_1 || Exp_2	Boolean or	Boolean Operator Expressions (page 59), Boolean (page 77)

Table of Built-in Functions

Table 1.39. All Functions

Function	See Library Module
abs	Integer (page 79)
arbBool	Boolean (page 77)
arbInt	Integer (page 79)
arbReal	Real (page 87)
arity	Node (page 87)
assertEqual	UnitTest (page 95)
assertTrue	UnitTest (page 95)
average	List (page 84), Set (page 91)
bottom	Graph (page 78), Labelled Graph (page 83)
browse	View (Eclipse only) (page 95)
carrier	Relation (page 88)
carrierR	Relation (page 88)
carrierX	Relation (page 88)
center	String (page 93)
charAt ? remove	String (page 93)
complement	Relation (page 88)
currentTimeMillis	Benchmark (page 77)
delete	List (page 84)
domain	List (page 84), Map (page 86), Relation (page 88)
domainR	Relation (page 88)
domainX	Relation (page 88)
endsWith	String (page 93)
files	Resource (Eclipse only) (page 90)
fromInt	Boolean (page 77)
fromString	Boolean (page 77)
getChildren	Node (page 87)
getName	Node (page 87)
getOneFrom	List (page 84), Map (page 86), Set (page 91)
head	List (page 84)
ident	Relation (page 88)
insertAt	List (page 84)
invert	Map (page 86), Relation (page 88)
invertUnique	Map (page 86)
isEmpty	List (page 84), Map (page 86), Set (page 91), String (page 93)
left	String (page 93)
location ?name	Resource (Eclipse only) (page 90)

Function	**See Library Module**
`makeNode`	Node (page 87)
`mapper?`	List (page 84), Map (page 86), Set (page 91)
`max`	Integer (page 79), List (page 84), Real (page 87), Set (page 91)
`min`	Integer (page 79), List (page 84), Real (page 87), Set (page 91)
`multiply`	List (page 84), Set (page 91)
`permutations`	List (page 84)
`power`	Set (page 91)
`power1`	Set (page 91)
`predecessors`	Graph (page 78), Labelled Graph (page 83)
`println`	IO (page 80)
`projects`	Resource (Eclipse only) (page 90)
`range`	Map (page 86)
`rangeR`	Relation (page 88)
`reach`	Graph (page 78), Labelled Graph (page 83)
`reachR`	Graph (page 78), Labelled Graph (page 83)
`reachX`	Graph (page 78), Labelled Graph (page 83)
`readFile`	IO (page 80)
`readRSF`	RSF (page 90)
`readValueFromBinaryFile`	ValueIO (page 95)
`readValueFromTextFile`	ValueIO (page 95)
`reducer`	List (page 84), Set (page 91)
`references`	Resource (Eclipse only) (page 90)
`report`	UnitTest (page 95)
`reverse`	List (page 84), String (page 93)
`right`	String (page 93)
`root`	Resource (Eclipse only) (page 90)
`shortestPathPair`	Graph (page 78), Labelled Graph (page 83)
`show`	View (Eclipse only) (page 95)
`size`	List (page 84), Map (page 86), Set (page 91), String (page 93)
`slice`	List (page 84)
`sort`	List (page 84)
`startsWith`	String (page 93)
`successors`	Graph (page 78), Labelled Graph (page 83)
`sum`	List (page 84)
`tail`	List (page 84)
`takeOneFrom`	List (page 84), Set (page 91)
`toReal`	String (page 93)
`toInt`	Boolean (page 77), Real (page 87),String (page 93)
`toList`	Map (page 86), Set (page 91)

Function	**See Library Module**
`toLowerCase`	String (page 93)
`toMap`	List (page 84), Set (page 91)
toMapUnique	List (page 84),Set (page 91)
`top`	Graph (page 78), Labelled Graph (page 83)
`toReal`	Boolean (page 77), Integer (page 79)
`toRel`	Map (page 86)
`toSet`	List (page 84)
`toString`	Boolean (page 77), Integer (page 79), List (page 84), Map (page 86), Real (page 87), Set (page 91)
`toUpperCase`	String (page 93)
`writeValueToBinaryFile`	ValueIO (page 95)
`writeValueToTextFile`	ValueIO (page 95)

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. 19. 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.

Acknowledgements

Rascal has been designed and implemented by the authors but they have received strong support, encouragement, and help from the following individuals. We are very grateful to them.

Emilie Balland implemented the Rascal debugger in the Eclipse version of Rascal during her visit to CWI in the summer of 2009.

Bas Basten provided useful feedback on the design and contributed the Rascal/JDT interface that makes it easy to extract facts from Java source code.

Bob Fuhrer's inspiring IMP project motivated us to build Rascal on top of IMP. The PDB was designed and implemented during Jurgen's visit to IBM Research in 2007-2008.

Arnold Lankamp implemented a very efficient version of the Program Data Base (PDB), added a binary streaming format, implemented the SGLR invoker, and takes care of deployment issues.

Karel Pieterson used Rascal as topic for a usability study in his Master's thesis. The results will be included in the final version of this document.

Frank Tip and Yaroslav Usenko provided feedback on this manual and suggested several improvements.

Glossary of Terminology

Annotation
: An annotation is a (name, value) pair that can be attached to a datatype value. The value can be retrieved and changed via the name. One datatype value can have more than one annotation.

ASF
: Algebraic Specification Formalism. This a notation for describing rewrite rules and is mostly used for defining software analysis, fact extraction, and software transformation. It is the predecessor of Rascal.

ASF+SDF
: The combination of the formalisms ASF and SDF. ASF+SDF can describe both the syntax of a language and the operations on that language (checking, execution, analysis, transformation). It is the predecessor of Rascal.

AsFix
: ASF+SDF Fixed format. A dataformat used to represent parse trees. AsFix is a specialized view on ATerms. Important features are:

- The AsFix format is a full parse tree that contains all the original layout and comments from the original source code program that was parsed.
- The AsFix format is self-descriptive: each subtree contains information about the exact grammar production that has been used to parse the text that has resulted in that parse tree.
- The AsFix format does not contain source code coordinates per se, but a separate tool can easily compute these coordinates and add them to the parse tree in the form of annotations.

AsFix is the predecessor of the ParseTree format used by Rascal

ATerm
: Annotated terms. A dataformat used for the internal representation of all data. Distinguishing features are:

- ATerms are language-independent and can be processed by programs in any language.
- ATerms can be annotated with auxiliary information that does not affect the tree structure.
- ATerms preserve *maximal subterm sharing*. This means that common parts of the data are not duplicated but shared. This leads to considerable size-reduction of the data.

ATerms are the predecessors of the shared PDB implementation used in Rascal.

Backtracking
: Backtracking is a general algorithmic method for finding all (or some) solutions to some computational problem, in the case of Rascal for pattern matching. It incrementally builds candidate matches, and abandons each partial match ("backtracks") as soon as it determines that it cannot possibly be completed to a valid match.

Bag	A bag can contain arbitrary values, without any particular order, and possibly repeated elements. All elements of a bag have the same type. *Rascal does not (yet?) support bags.*
Comprehension	An algorithmic schema for computing lists, sets or relations.
Constraint solving	The process of finding solutions to a problem that is only specified by the boundary conditions (constraints) that a solution should satisfy.
Control flow	The order in which the statements in a program can be executed. Usually represented as a control flow graph.
Debugger	A debugger allows the step-by-step execution of a program.
Dataflow	The order in which values may be assigned to variables during program execution. Usually represented as a data flow graph.
Eclipse	Interactive Development Environment (IDE) for Java and other languages. The Rascal IDE is built on top of Eclipse and IMP.
Enumerator	A Rascal construct that enumerates all the elements of a given value.
Function (higher order)	A function is a computational entity with typed formal parameters and a typed result. When applied to actual parameters, the function is applied and the result replaces the function call. A higher-order function is a function that has one or more functions as parameter.
Graph	In mathematics a graph is a set of objects (nodes) that may be pairwise connected with links (edges). In Rascal, a graph is a binary relation in which each tuple represents an edge in the corresponding (mathematical) graph.
Interactive Development Environment (IDE)	A programming environment for the interactive design, development, debugging and testing of programs. Eclipse is an example of an IDE.
IDE Meta Platform (IMP)	IMP is an extension of Eclipse and is intended to make it easier to add support for new languages to Eclipse. The Rascal implementation is part of IMP.
Location	Data format to describe locations in source code.
List	A list can contain arbitrary values, with a particular order, and possibly repeated elements. All elements of a list have the same type.
Map	A map is a binary relation consisting of 2-tuples (pairs) of the form (key, value) that associate each key with a unique value.
Meta-Environment	The ASF+SDF Meta-Environment is the IDE for writing ASF+SDF specifications.
Meta-programming	The activity of writing programs that manipulate other programs as data.
Parser	A program that performs syntactic analysis on a given input text and builds a parse tree.

Parse table	A parse table is produced by a parse table generator and is an efficient representation of grammar that can be used by a parser.
Parse table generator	A parse table generator takes a grammar as input and converts it to a parse table to be used by a parser.
Parse tree	Tree-structured representation of a text that has been analyzed by a parser.
ParseTree	The ParseTree datatype is the parse tree format used in Rascal. It is the successor of AsFix.
Pattern	A structural description of values that is used for matching.
Pattern matching	Matching is an algorithmic method for determining the structural similarity of a pattern and a subject value. The pattern may contain variables that are bound to corresponding parts of the subject value when the match succeeds.
Program Data Base (PDB)	Subsystem of IMP that is responsible for implementing Rascal's datatypes.
Polymorphism	Functions and datatypes that can handle values of different types in a uniform manner.
Prettyprinter	A prettyprinter converts parse trees to text. The prettyprinter uses default rules to insert layout in a parse tree so that its corresponding text is presented in a uniform way. Optionally, user-defined formatting rules can replace this default behaviour.
Relation	A relation is a set of tuples. All tuples of a relation have the same type.
Rscript	A small scripting language for defining relational expressions. Used for the analysis of facts extracted from software. Predecessor of Rascal.
Relational Algebra	An algebra on relations that provides operators on relations that yield new relations. Relational algebra uses no variables and provides an operational view on relations.
Relational calculus	Relational calculus provides a more declarative and descriptive way to formulate operations on relations and depends on the use of variables.
Read-Eval-Print-Loop (REPL)	Command-line loop that reads user input, evaluates it and prints the result.
Rewrite rule	A rewrite rule consists of a pattern and a replacement. A rewrite rule is applied to all subparts of a datastructure and when the pattern matches the matching subpart is replaced. Pattern and replacement may contain variables and variables in the replacement are first replaced by their value before the replacement is applied.
SDF	Syntax Definition Formalism. A notation for describing the grammar of programming and application languages. SDF definitions can be imported in Rascal.
Set	A set can contain arbitrary values, without any particular order, and no repeated elements. All elements of a set have the same type.

Side-effect	In imperative programming languages like C and Java, assignments cause changes in a global state that are visible for other statements and procedures.
Syntax tree	Tree-structured representation of a text that has been analyzed by a parser.
Summary	An error or message summary. A dataformat for the internal representation of errors and messages. Summaries are produced by checker and compilers and are used by the GUI.
Term rewriting	Term rewriting is the act of applying a set of rewrite rules to a subject value. The rules are applied as long as any pattern in some rule matches. The result is the subject value with replacements made as prescribed by the matching rules.
Traversal	The process of visiting all the sub-elements of a larger datastructure. The `visit` expression and enumerators provide traversal functionality in Rascal.
Tuple	A tuple is an ordered list of values of possibly different type. An element of a tuple may have a label by which that element can be selected or modified.
Type checking	The activity to check that all entities in a program are declared and are used in a way that is consistent with their declared type.
Type inference	The activity to infer the type of program entities from their use.