Chapter 1. EASY Meta-Programming with Rascal

Paul Klint, Jurgen Vinju, Tijs van der Storm

Wed Jun 10 23:00:41 CEST 2009

Note

Rascal is a work in progress both regarding implementation and documentation. The current version of this article is a preview version only. The online documentation [http://www.meta-environment.org/doc/books//analysis/rascal-manual/rascal-manual.pdf] is always up-to-date and provides much more information.

1. A New Language for Meta-Programming

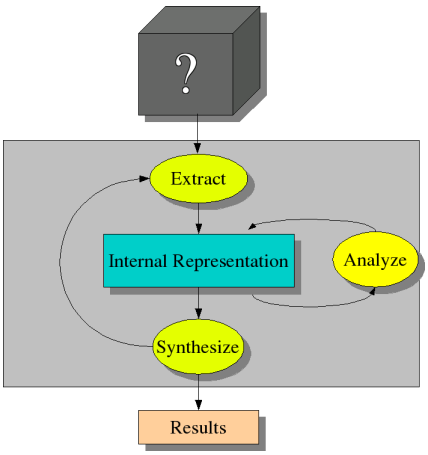
Meta-programs are programs that analyze, transform or generate other programs. Ordinary programs work on data; meta-programs work on programs. The range of programs to which meta-programming can be applied is large: from programs in standard languages like C and Java to domain-specific languages for 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.

1.1. 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”.

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.

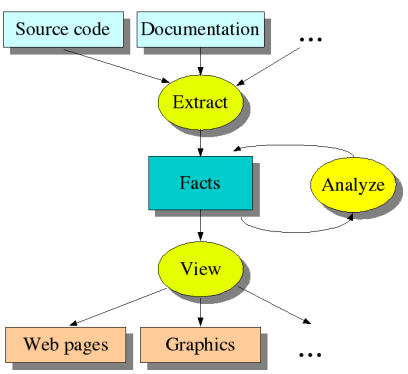
Example 1.2. A Forensic DSL compiler

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”.

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.

1.2. 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.

1.3. 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!

1.4. Aim and Scope of this Article

Aim. The aim of this article 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. This article 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.

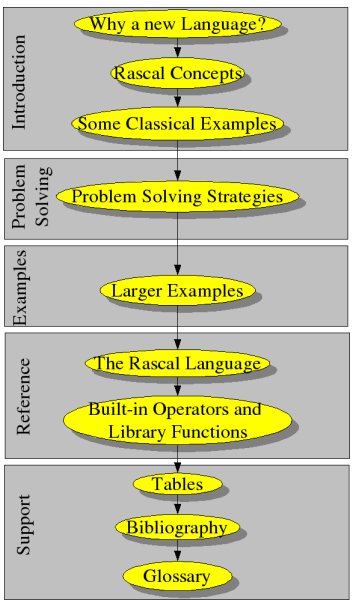
Scope. The scope of this article is limited to the Rascal language and its applications but does not address implementation aspects of the language.

1.5. Downloading, Installing and Running Rascal

See the online instructions [http://www.meta-environment.org/Meta-Environment/Rascal] for details about downloading, installing and running Rascal.

1.6. Reading Guide

Figure 1.3. Structure of the Rascal Description



The structure of the description of Rascal is shown in Figure 1.3, “Structure of the Rascal Description”. This article provides the first three parts:

- **Introduction**: gives a high-level overview of Rascal and consists of Section 1, “A New Language for Meta-Programming” and Section 2, “Rascal Concepts” . It also presents some simple examples in Section 3, “Some Classical Examples”.
- **Problem Solving**: describes the major problem solving strategies in Rascal's application domain, see Section 4, “Problem Solving Strategies”.
- **Examples**: gives a collection of larger examples, see Section 5, “Larger Examples”.

The other parts can be found online [http://www.meta-environment.org/doc/books//analysis/rascal-manual/rascal-manual.pdf]:

- **Reference**: gives a detailed description of the Rascal language, and all built-in operators and library functions.
- **Support**: gives tables with operators and library functions, a bibliography and a glossary that explains many concepts that are used in the descriptions of Rascal and tries to make them self-contained.

1.7. 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:

```
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.

2. Rascal Concepts

We give now an overview of the concepts on which Rascal is based.

2.1. 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.

2.2. 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”.

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`
`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])`

2.3. 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:.*$>/
```

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.

2.4. 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.

2.5. 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`.

2.6. 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.

2.7. 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");
}
```

2.8. 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).

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.

2.9. 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;
}
```

2.10. 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.

2.11. 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.

2.12. 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 Section 5.6, “Dataflow Analysis” for examples.

2.13. Other features

Some features we have not yet mentioned are:

- Rascal programs consist of modules that are organized in packages.
- Modules can import other modules. These can be Rascal modules or SDF modules (as shown above in Section 2.10, “Syntax Definition and Parsing”).
- 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.

2.14. 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 expression determine choices that drive the control structures. Rewrite rules form the only exception to the explicit control flow principle. Only local backtracking is provided (no surprise) in the context of boolean expressions and pattern matching; side effects are undone in case of backtracking.

3. Some Classical Examples

The following simple examples will help you to grasp the main features of Rascal quickly. You can also consult the online documentation [http://www.meta-environment.org/doc/books//analysis/rascal-manual/rascal-manual.pdf] for details of the language or specific operators or functions.

3.1. 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 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
```

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

3.2. 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: 25862324151116818064296435515361197996919763238912000000000
```

Indeed, Rascal has infinite length integers.

3.3. 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))));
ColoredTree: red(black(leaf(1), red(leaf(2),leaf(3))))
```

We define two operations on ColoredTrees, one to count the red nodes, and one to add 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: 6
```

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))))
```

3.4. 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"
```

4. 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.

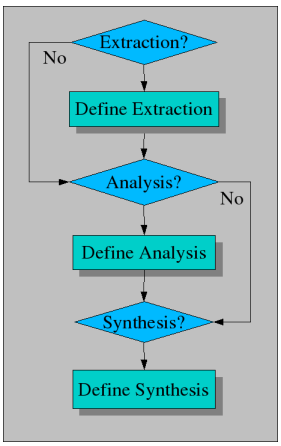
With these examples in mind, we can study the overall problem solving workflow as shown in Figure 1.4, “General 3-Phased Problem Solving Workflow”. It consists of three optional phases:

- Is **extraction needed** to solve the problem, then define the extraction phase, see Section 4.1, “Defining Extraction”.

- Is **analysis needed**, then define the analysis phase, see Section 4.2, “Defining Analysis”.

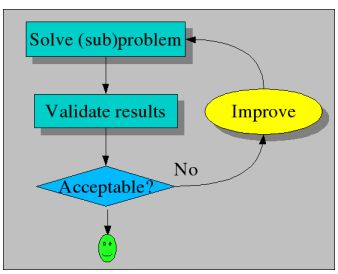
- Is **synthesis needed**, then define the synthesis phase, see Section 4.3, “Defining Synthesis”.

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”. 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” and amounts to the phases:

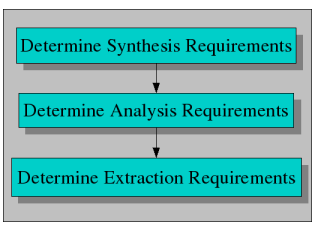
- **Requirements of the synthesis phase**. This amounts to making an inventory of the desired results of the whole project and may include generated source code, abstract models, or visualizations.

- **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”.

4.1. 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”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 is bitting 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.

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

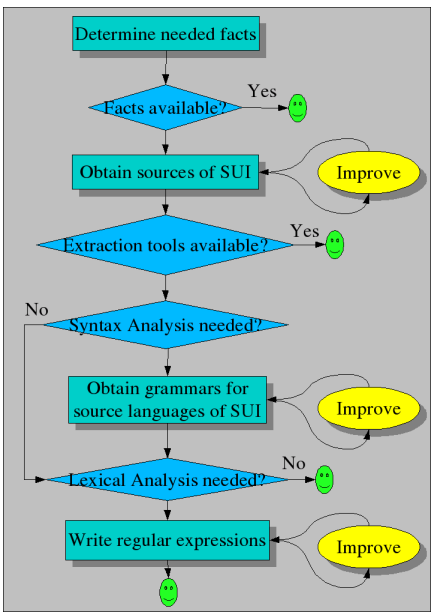
The strategy to validate extracted facts differ per case but here are three strategies:

- 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.

4.2. Defining Analysis

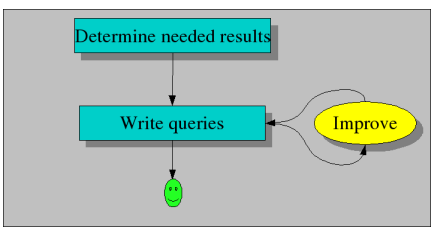
The analysis workflow is shown in Figure 1.8, “Analysis Workflow” and consists of two steps:

- 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.

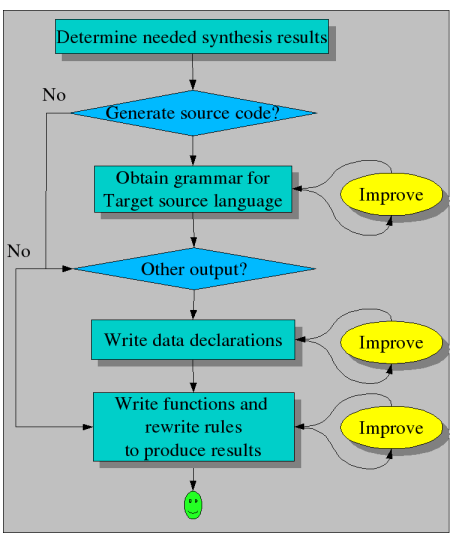
4.3. Defining Synthesis

Results are synthesized as shown in Figure 1.9, “Synthesis Workflow”. 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.

5. Larger Examples

Now we will have a closer look at some larger applications of Rascal. We start with a call graph analysis in Section 5.1, “Call Graph Analysis” and then continue with the analysis of the component structure of an application in Section 5.2, “Analyzing the Component Structure of an Application” and of Java systems in Section 5.3, “Analyzing the Structure of Java Systems”. Next we move on to the detection of uninitialized variables in Section 5.4, “Finding Uninitialized and Unused Variables in a Program”. As an example of computing code metrics, we describe the calculation of McCabe's cyclomatic complexity in Section 5.5, “McCabe Cyclomatic Complexity”. Several examples of dataflow analysis follow in Section 5.6, “Dataflow Analysis”. A description of program slicing concludes the chapter, see Section 5.7, “Program Slicing”.

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

5.1. 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?

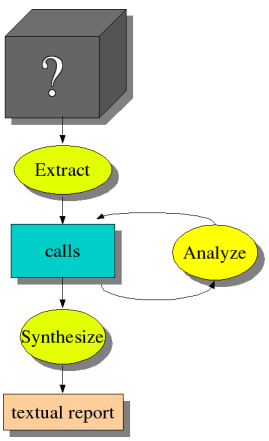
There are now two possibilities. Either you have this superb programming environment or tool suite that can immediately answer all these questions for you or you can use Rascal.

5.1.1. Preparations

To illustrate this process consider the workflow in Figure 1.10, “Workflow for analyzing mystery box”. First we have to extract the calls from the source code. Rascal is very

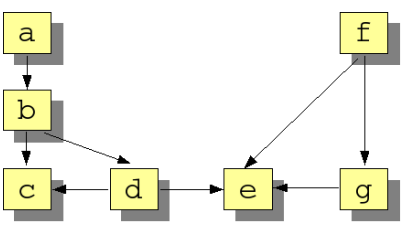
good at this, but to simplify this example we assume that this call graph has already been extracted. Also keep in mind that a real call graph of a real application will contain thousands and thousands of calls. Drawing it in the way we do later on in Figure 1.11, “Graphical representation of the `calls` relation” 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”. This section is intended to give you a first impression what can be done with Rascal.

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`:

Caution

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.

5.1.2. Questions

5.1.2.1. 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 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. In this example, `nCalls` will get the value `8`.

5.1.2.2. 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
```

5.1.2.3. 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. 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.

5.1.2.4. 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"}`.

5.1.2.5. 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">}
```

5.1.2.6. 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"}
```

5.1.2.7. 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"`.

5.1.2.8. 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”. 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.

5.2. 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:

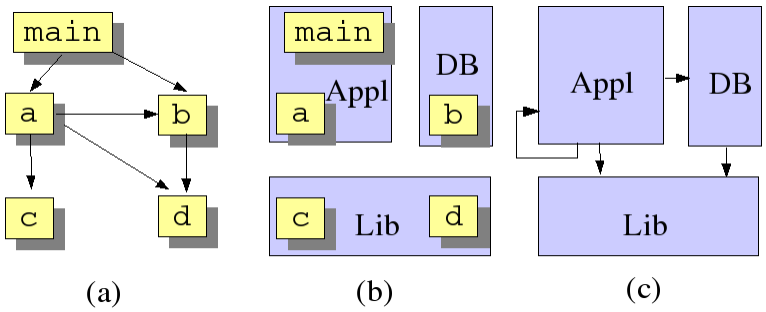
```
rascal> import demo::Lift;
ok

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`.”.

Figure 1.12. (a) `Calls`; (b) `PartOf`; (c) `ComponentCalls`.



5.3. 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 attributes of class C_1 is of type C_2.

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

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

import java.awt.Point;
import CH.ifa.draw.framework.*;
/**
 * A LocatorHandle implements a Handle by delegating the
 * location requests to a Locator object.
 */
public class LocatorHandle extends AbstractHandle {
    private Locator       fLocator;
    /**
     * Initializes the LocatorHandle with the
```

```
     * given Locator.
     */
    public LocatorHandle(Figure owner, Locator l) {
        super(owner);
        fLocator = l;
    }
    /**
     * Locates the handle on the figure by forwarding
     * the request to its figure.
     */
    public Point locate() {
        return fLocator.locate(owner());
    }
}
```

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

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

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

```
rel[class,class] USE = CALL + 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.

5.4. 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 Section 1.1, “EASY Programming” 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 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 path that can reach statement *S*. All is well if *all* these execution path contain a definition of *V*. However, if one or more of these path do *not* contain a definition of *V*, then *V* may be uninitialized in statement *S*. This can be formalized as follows:

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

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({ROOT}, DEFS[V], PRED)` 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({ROOT}, DEFS[V], PRED)` is a set that contains all nodes that are reachable from the `ROOT` (as well as all intermediate nodes on each path).

- Finally, `E in reachX({ROOT}, DEFS[V], PRED)` tests whether expression `E` can be reached from the `ROOT`.

- The net effect is that `UNINIT` will only contain pairs that satisfy the test just described.

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

```
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"}`.

5.5. 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`.

5.6. 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 (Section 5.6.1, “Dominators”): which nodes in the flow dominate the execution of other nodes?

- Reaching definitions (Section 5.6.2, “Reaching Definitions”): which definitions of variables are still valid at each statement?

- Live variables (Section 5.6.3, “Live Variables”): 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.

5.6.1. 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

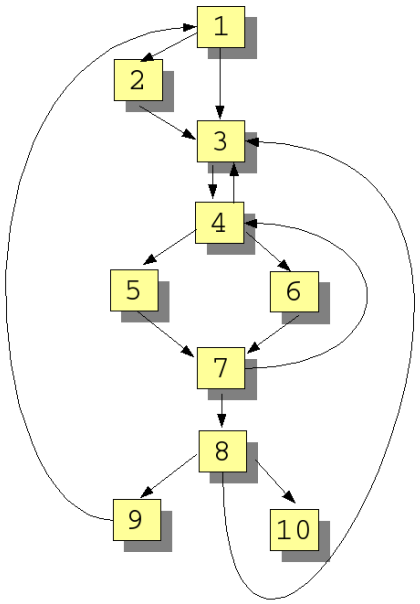
- S_i is not an initial node or equal to S.
- S_i cannot be reached from the initial node without going through S.

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 in Figure 1.13, “Flow graph”

Figure 1.13. Flow graph



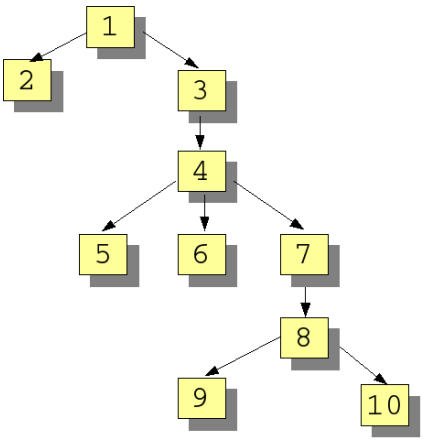
(a)

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”. 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

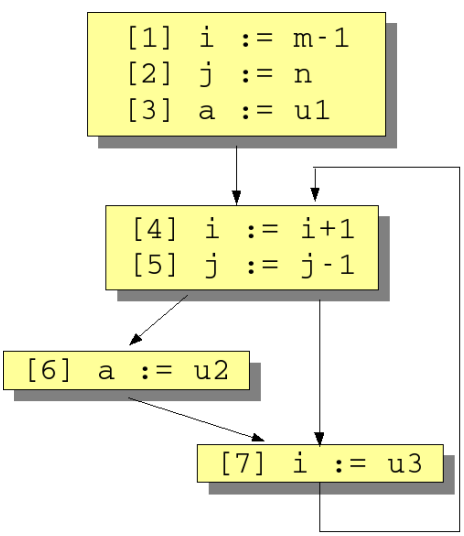


(b)

5.6.2. Reaching Definitions

We illustrate the calculation of reaching definitions using the example in Figure 1.15, “Flow graph for various dataflow problems” 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 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:

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

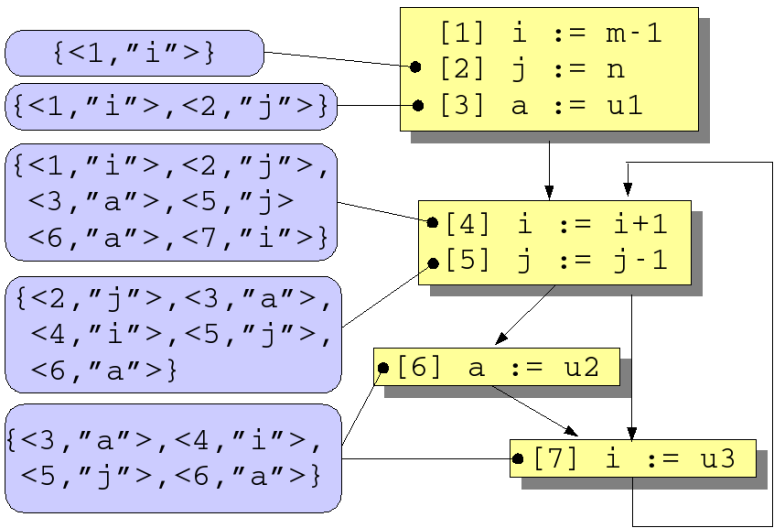
$OUT[S] = DEF[S] + (IN[S] - KILL[S])$

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 <- predecessor(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 predecessor to obtain the predecessor 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”



For our running example (Figure 1.16, “Reaching definitions for flow graph in Figure 1.15, “Flow graph for various dataflow problems”””) the results are as follows (see Figure 1.16, “Reaching definitions for flow graph in Figure 1.15, “Flow graph for various dataflow problems”””). 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””. We will use the function `reachingDefinitions` later on in Section 5.7, “Program Slicing” when defining program slicing.

5.6.3. 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$_{S' \text{ in successor of } S}$ IN[S']

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

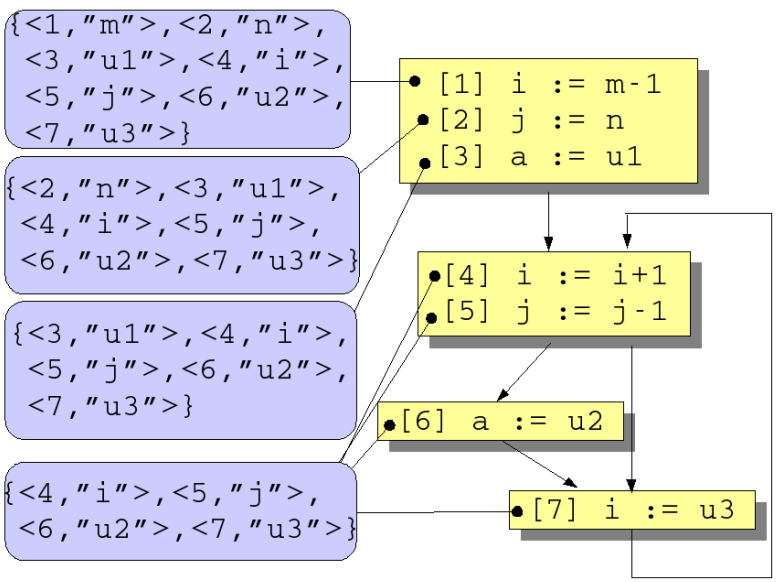
This can be expressed in Rascal as follows:

```
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 <- successor(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””.

Figure 1.17. Live variables for flow graph in Figure 1.15, “Flow graph for various dataflow problems”



5.7. 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 Section 5.6.2, “Reaching Definitions”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.

6. Concluding Remarks

Rascal and its IDE are in full development at the time of writing and a first prototype implementation is available for download. We have given here only a sketch of the language and its applications. The following topics have not been covered in the current version of this document:

- The use of SDF modules to parse source text.
- The extensive Rascal library that supports many operations on the basic datatypes including shortest path, reachability and bisimulation on graphs.
- The Rascal Eclipse JDT library that provides direct access to facts that have been extracted from Java source code.
- The Rascal IDE that is based on Eclipse and that provides, for instance, very good interactive debugging facilities and basic visualization tools.

We refer the interested reader to the online documentation [http://www.meta-environment.org/doc/books//analysis/rascal-manual/rascal-manual.pdf] for a more complete and up-to-date overview.

7. 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 is developing a 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.

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

8. 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.