SetlX — A Tutorial

Version 2.5.0

Karl Stroetmann
Duale Hochschule Baden-Württemberg Mannheim
karl.stroetmann@dhbw-mannheim.de

Tom Herrmann
Duale Hochschule Baden-Württemberg Stuttgart
setlx@randoom.org

September 24, 2016

Abstract

Galileo Galilei once said that mathematics is the alphabet used by God to describe the universe. In the late sixties, Jack Schwartz, a renowned professor of mathematics at the Courant Institute for Mathematical Sciences in New York, developed a programming language that uses this alphabet, the language called SETL [Sch70, SDSD86]. The most distinguishing feature of this language is the support it offers for sets and lists. As set theory is the language of mathematics, many mathematical algorithms that are formulated in terms of set theory have a straightforward implementation in SETL.

Unfortunately, at the time of its invention, SETL did not get the attention that it deserved. One of the main reasons was that SETL is an interpreted language and in the early days of computer science, the run time overhead of an interpreter loop was not yet affordable. More than forty years after the first conception of SETL, the efficiency of computers has changed dramatically and for many applications, the run time efficiency of a languange is no longer as important as it once was. After all, modern scripting languages like Python [vR95] or Ruby [FM08] are all interpreted and noticeably slower than compiled languages like C, but this fact hasn't impeded their success.

While teaching at the Baden-Württemberg Corporate State University, the first author had used SETL2[1] [Sny90] for several years in a number of introductory computer science courses. He had noticed that the adoption of SETL had made the abstract concepts of set theory more accessible to the students. Nevertheless, as SETL2 is more than 25 years old, it has a number of shortcomings. One issue is the fact that the syntax of SETL2 is quite dated and has proven difficult to master for students that are mainly acquainted with C and *Java*. Therefore, the original language SETL has been extended into the new language SETLX. The main features that have been changed or added are as follows:

- SETLX supports terms in a way similar to the language *Prolog* [SS94]. For example, SETLX supports matching. This makes SETLX well suited for symbolic computations.
- SETLX supports several ideas from functional programming. In particular, functions can be used as a primitive data type. Furthermore, SETLX supports *closures* [Lan64] and *memoization* [Mic68] of functions.
- SETLX has support for regular expressions.
- SETLX provides a limited way of *backtracking*. Like terms, this feature was also inspired from the language *Prolog*.
- SETLX supports object oriented programming concepts.
- Vectors and matrices are supported as basic data type. Various linear algebra functions have been added on top of these data types. For example, there are functions to invert a matrix or to compute the eigenvalues and eigenvectors of a matrix.
- SETLX provides a small set of graphical primitives that support the animation of algorithms.
- SETLX provides a modest plotting library that can be used to graph functions or to draw scatter plots.
- Lastly, while SETL2 has a syntax that is reminiscent of Algol, SETLX has a syntax that is more akin to languages like C or *Java*.

The language SETLX has been implemented by Tom Herrmann as part of his student research project. He still continues to supervise the addition of new features. Markus Jagiella has created the package that can be used to animate algorithms. Patrick Robinson added the package supporting linear algebra. Arne Röcke and Fabian Wohriska have added the plotting functionalities to the SETLX interpreter.

[1] SETL2 is an object oriented extension of SETL. It was created by Kirk Snyder at the Courant Institute of Mathematical Sciences at New York University.

Contents

1 Introduction **6**
- 1.1 Motivation . . . 6
- 1.2 Overview . . . 7
- 1.3 Downloading . . . 7

2 Whetting Your Appetite **9**
- 2.1 Getting Started . . . 9
  - 2.1.1 Rational Numbers and Floating Point Numbers . . . 11
- 2.2 Boolean Values . . . 13
  - 2.2.1 Boolean Operators . . . 13
  - 2.2.2 Quantifiers . . . 14
- 2.3 Sets . . . 15
  - 2.3.1 Operators on Sets . . . 17
  - 2.3.2 Set Comprehensions . . . 20
  - 2.3.3 Miscellaneous Set Functions . . . 20
- 2.4 Lists . . . 21
  - 2.4.1 Operators on Lists . . . 22
  - 2.4.2 Extracting Elements from a List . . . 23
- 2.5 Pairs, Relations, and Functions . . . 25
- 2.6 Procedures . . . 27
- 2.7 Strings . . . 27
  - 2.7.1 Literal Strings . . . 28
- 2.8 Terms . . . 29
- 2.9 Vectors and Matrices . . . 31

3 Statements **33**
- 3.1 Assignment Statements . . . 33
- 3.2 Functions . . . 34
  - 3.2.1 Calling a Procedure . . . 36
  - 3.2.2 Memoization . . . 37
- 3.3 Branching Statements . . . 39
  - 3.3.1 `if-then-else` Statements . . . 39
  - 3.3.2 `switch` Statements . . . 39
- 3.4 Matching . . . 41
  - 3.4.1 String Matching . . . 41
  - 3.4.2 List Matching . . . 43
  - 3.4.3 Set Matching . . . 43
  - 3.4.4 Term Matching . . . 44
  - 3.4.5 Term Decomposition via List Assignment . . . 46
- 3.5 Loops . . . 47

3.5.1 `while` Loops . . . 47
3.5.2 `do-while` Loops . . . 48
3.5.3 `for` Loops . . . 49

4 Regular Expressions 53
4.1 Using Regular Expressions in a `match` Statement . . . 53
4.1.1 Extracting Substrings . . . 54
4.1.2 Testing Regular Expressions . . . 55
4.1.3 Extracting Comments from a File . . . 55
4.1.4 Conditions in `match` Statements . . . 57
4.2 The `scan` Statement . . . 58
4.3 Additional Functions Using Regular Expressions . . . 61

5 Functional Programming and Closures 63
5.1 Introductory Examples . . . 63
5.1.1 Implementing Counters via Closures . . . 65
5.1.2 Lambda Definitions for Closures . . . 65
5.1.3 Closures as Return Values . . . 66
5.2 Closures in Action . . . 67
5.3 Decorators . . . 73

6 Exceptions and Backtracking 76
6.1 Exceptions . . . 76
6.1.1 Different Kinds of Exceptions . . . 78
6.2 Backtracking . . . 79
6.2.1 The 8 Queens Puzzle . . . 80
6.2.2 The Zebra Puzzle . . . 83

7 Vectors and Matrices 90
7.1 Vectors . . . 90
7.2 Matrices . . . 92
7.3 Numerical Methods for Matrices and Vectors . . . 94
7.3.1 Computing the Determinant . . . 94
7.3.2 Solving a System of Linear Equations . . . 94
7.3.3 The Singular Value Decomposition and the Pseudo-Inverse . . . 96
7.3.4 Eigenvalues and Eigenvectors . . . 96

8 Plotting 98
8.1 Plotting Curves . . . 98
8.1.1 Plotting the Parabola . . . 99
8.1.2 Plotting Parametric Curves . . . 102
8.2 Scatter Plots . . . 106
8.3 Plotting Statistical Charts . . . 108
8.3.1 Bar Charts . . . 108
8.3.2 Pie Charts . . . 110

9 Classes and Objects 113
9.1 Basic Examples . . . 113
9.1.1 Introducing Classes . . . 113
9.1.2 Simulating Inheritance . . . 118
9.2 A Case Study: Dijkstra's Algorithm . . . 119
9.3 Representing Complex Numbers as Objects . . . 124

10 Predefined Functions **129**
10.1 Functions and Operators on Sets and Lists . . . 129
10.2 Functions for String Manipulation . . . 132
10.3 Functions for Term Manipulation . . . 135
10.3.1 Library Functions for Term Manipulation . . . 137
10.4 Mathematical Functions . . . 137
10.5 Generating Random Numbers and Permutations . . . 141
10.5.1 `shuffle` . . . 142
10.5.2 `nextPermutation` . . . 142
10.5.3 `permutations` . . . 143
10.6 Type Checking Functions . . . 143
10.7 Debugging . . . 144
10.7.1 Library Functions for Debugging . . . 144
10.8 I/O Functions . . . 146
10.8.1 `appendFile` . . . 146
10.8.2 `ask` . . . 146
10.8.3 `deleteFile` . . . 147
10.8.4 `get` . . . 147
10.8.5 `load` . . . 147
10.8.6 `loadLibrary` . . . 147
10.8.7 `multiLineMode` . . . 148
10.8.8 `nPrint` . . . 148
10.8.9 `nPrintErr` . . . 148
10.8.10 `print` . . . 148
10.8.11 `printErr` . . . 148
10.8.12 `read` . . . 148
10.8.13 `readFile` . . . 149
10.8.14 `writeFile` . . . 149
10.9 Plotting Functions . . . 149
10.9.1 plot_createCanvas . . . 149
10.9.2 plot_addBullets . . . 150
10.9.3 plot_addGraph . . . 150
10.9.4 plot_addListGraph . . . 151
10.9.5 plot_addParamGraph . . . 152
10.9.6 plot_addBarChart . . . 152
10.9.7 plot_addPieChart . . . 153
10.9.8 plot_addLabel . . . 154
10.9.9 plot_defineTitle . . . 154
10.9.10 plot_exportCanvas . . . 154
10.9.11 plot_labelAxis . . . 155
10.9.12 plot_legendVisible . . . 155
10.9.13 plot_modScale . . . 156
10.9.14 plot_modScaleType . . . 156
10.9.15 plot_removeGraph . . . 157
10.9.16 plot_modSize . . . 157
10.10 Miscellaneous Functions . . . 157
10.10.1 abort . . . 158
10.10.2 `cacheStats` . . . 158
10.10.3 `clearCache` . . . 158
10.10.4 `compare` . . . 158
10.10.5 `getOsID` . . . 158
10.10.6 `getScope` . . . 159

10.10.7 `logo` 159
10.10.8 `now` 159
10.10.9 `run` 159
10.10.10 `sleep` 160

A Graphical Library Functions **163**

Chapter 1

Introduction

1.1 Motivation

Every year, dozens of new programming languages are proposed and each time this happens, the inventors of the new language have to answer the same question: There are hundreds of programming languages already, why do we need yet another one? Of course, the answer is always the same. It consists of an economical argument, a theological argument and a practical argument. For the convenience of the reader, let us brievely review these arguments as they read for SetlX.

1. Nothing less than the prosperity and welfare of the entire universe is at stake and SetlX provides the means to save it.

2. Programming in SetlX is the only way to guarantee redemption from the eternal hell fire that awaits those not programming in SetlX.

3. Programming in SetlX is fun!

The economical argument has already been discussed at length by Adams [Ada80], therefore we don't have to repeat it here. We deeply regret the fact that the philosophical background of the average computer scientist does not permit them to follow advanced theological discussions. Therefore, we have to refrain from giving a detailed proof of the second claim. Nevertheless, we hope the examples given in this tutorial will convince the reader of the truth of the third claim. One of the reasons for this is that SetlX programs are both very concise and readable. This often makes it possible to fit the implementation of complex algorithms in SetlX on a single slide because the SetlX program is no longer than the pseudocode that is usually used to present algorithms in lectures. The benefit of this is that instead of pseudocode, students have a running program that they can modify and test. This and the conciseness of SetlX programs was one of the reasons for the first author to adopt SetlX as a programming language in his various lectures on computer science and mathematics: It is often feasible to write a complete SetlX program in a few lines onto the blackboard, since SetlX programs are nearly as compact as mathematical formulæ.

SetlX is well suited to implement complex algorithms. This is achieved because SetlX provides a number of sophisticated builtin data types that enable the user to code at a very high abstraction level. These data types are sets, lists, first-order terms, and functions. As sets are implemented as ordered binary trees, sets of pairs can be used both as symbol tables and as priority queues. This enables very neat implementations of a number of graph theoretical algorithms. As SetlX is implemented in *Java* and the *Java virtual machine* is quite fast, various tests have shown that SetlX beats *Python* in terms of performance.

The purpose of this tutorial is to introduce the most important features of SetlX and to show, how the use of the above mentioned data types leads to programs that are both shorter and clearer than the

corresponding programs in other programming languages. This was the prime motivation of the first author to develop SetlX: It turns out that SetlX is very convenient as a tool to present algorithms at a high abstraction level in a class room. Furthermore, SetlX makes the abstract concepts of set theory tangible for students of computer science.

1.2 Overview

The remainder of this tutorial is structured as follows:

1. In the second chapter, we discuss the data types available in SetlX.
2. The third chapter provides the control structures.
3. The fourth chapter deals with *regular expressions*.
4. The fifth chapter discusses *functional programming* and *closures*.
5. The sixth chapter discusses the `try-catch` and `throw` mechanism and demonstrates the use of *backtracking*.
6. The seventh chapter discusses how vectors and matrices can be used and demonstrates the linear algebra package that is provided by SetlX.
7. The eighth chapter presents the functions that can be used for plotting.
8. Chapter number nine introduces classes and demonstrates that SetlX supports an object-oriented programming style.
9. The final chapter lists and explains all predefined functions.
10. The appendix discusses the graphical primitives provided by SetlX. These primitives can be used, for example, to animate algorithms.

This tutorial is not meant as an introduction to programming. It assumes that the reader has had some preliminary exposure to programming and has already written a few programs in either `C`, *Java*, or a similar language.

1.3 Downloading

The current distribution of SetlX can be downloaded from

`http://randoom.org/Software/SetlX`.

SetlX is written in *Java* and is therefore supported on a number of different operating systems. Currently, SetlX is supported on *Linux*, *OS X*, *Microsoft Windows*, and *Android*. The websites given above explains how to install the language on various platforms. The distribution contains the *Java* code and a development guide that gives an overview of the implementation. For those interested in peeking under the hood, the source code of SetlX is available at

`https://github.com/herrmanntom/setlX`.

Disclaimer

The development of SetlX is an ongoing project. Therefore some of the material presented in this tutorial might be out of date, while certain aspects of the language won't be covered. The current version of this tutorial is not intended to be a reference manual. The idea is rather to provide the reader with an introduction that is sufficient to get started.

Encouragement

The authors would be grateful for any kind of feedback. They can be contacted via email as follows:

Karl Stroetmann: karl.stroetmann@dhbw-mannheim.de

Tom Herrmann: setlx@randoom.org

Acknowledgements

We would like to acknowledge that Karl-Friedrich Gebhardt and Hakan Kjellerstrand have both read earlier drafts of this tutorial and have given valuable feedback that has helped to improve the current presentation.

Chapter 2

Whetting Your Appetite

This chapter contains a short overview of the data types supported by SetlX and tries to whet your appetite for the language by showing off some of the features that are unique to SetlX. Before we discuss the more elaborate data types, we introduce the basic data types for numbers and strings and show how to invoke the interpreter.

2.1 Getting Started

SetlX is an interpreted language. To start the interpreter, the file `setlX` has to be both executable and part of the search path. If these preconditions are satisfied, the command

```
setlX
```

launches the interpreter. The interpreter first prints the banner shown in Figure 2.1, followed by a prompt "=>". Commands are typed after the prompt. If the command is an assignment or an expression, then it has to be terminated by a semicolon. However, complex commands like, for example, branch statements, loops or class definitions are not terminated by a semicolon.

```
-====================================setlX============================v2.5.0=-

Welcome to the setlX interpreter!

Open Source Software from http://setlX.randoom.org/
(c) 2011-2016 by Herrmann, Tom

You can display some helpful information by using '--help' as parameter when
launching this program.

Interactive-Mode:
  The 'exit;' statement terminates the interpreter.

-================================Interactive=Mode==============================-

=>
```

Figure 2.1: The SetlX banner followed by a prompt.

The welcome banner of SetlX points out that we can call `setlX` with the parameter "`--help`".

Doing this yields the output shown in Figure 2.2 below.

```
-====================================setlX============================v2.5.0=-

File paths supplied as parameters for this program will be parsed and executed.
The interactive mode will be started if called without any file parameters.

Interactive-Mode:
  The 'exit;' statement terminates the interpreter.

Additional parameters:
  -l <path>, --libraryPath <path>
      Override SETLX_LIBRARY_PATH environment variable.
  -a, --noAssert
      Disables all assert functions.
  -n, --noExecution
      Load and check code for syntax errors, but do not execute it.
  -r, --predictableRandom
      Always use the same sequence of random numbers (useful for debugging).
  -p <argument> ..., --params <argument> ...
      Pass all following arguments to executed program via 'params' variable.
  -e <expression>, --eval <expression>
      Evaluates next argument as expression and exits.
  -x <statement>, --exec <statement>
      Executes next argument as statement and exits.
  -v, --verbose
      Display the parsed program before executing it.
  --doubleDefault
  --doubleScientific
  --doubleEngineering
  --doublePlain
      Sets how the exponent of a floating point number is printed.
  -h, --harshWelcome
      Interactive mode: Reduce welcome message to a bare minimum.
  -m, --multiLineMode
      Interactive mode: Input is only processed after an additional new line.
  --version
      Display interpreter version and terminate.
```

Figure 2.2: Calling `setlx` with the option "`--help`".

The SetlX interpreter can be used as a simple calculator: Typing

```
=> 1/3 + 2/5;
```

and then hitting the return key[1] will result in the following response:

```
~< Result:  11/15 >~
```

[1] In SetlX, statements can extend over many lines. If the user intends to use multi line statements, then she can start the interpreter using the commandline switch "`--multiLineMode`". In multi line mode, the return key needs to be hit twice to signal the end of the input. Instead of using a commandline switch, the user can also issue the command

```
multiLineMode(true);
```

to activate multiline mode. To switch back to single line mode, use the command "`multiLineMode(false);`".

After printing the result, the interpreter provides a new prompt so the next command can be entered. Incidently, the last example shows the first data type supported by SetlX: Rational numbers. These will be discussed in more detail next.

2.1.1 Rational Numbers and Floating Point Numbers

SetlX supports both rational numbers and floating point numbers. A rational number consists of a numerator and a denominator, both of which are integers. SetlX takes care to ensure that numerator and denominator are always reduced to lowest terms. Furthermore, if the denominator happens to be the value 1, only the numerator is printed. Therefore, after typing

```
1/3 + 2/3;
```

SetlX will respond:

```
~< Result:  1 >~
```

The precision of rational numbers is only limited by the available memory. For example, the command

```
50!;
```

computes the factorial of 50 and yields the result

```
~< Result:  30414093201713378043612608166064768844377641568960512000000000000 >~.
```

Therefore, as long as only rational numbers are used, there will be no rounding errors. Unfortunately, if a rational number q is the result of a computation involving n arithmetic operations, in the worst case the length of the nominator and denominator of the rational numbers q grows exponentially with n. For this reason, most calculations in engineering are done using floating point numbers. Since SetlX is based on *Java*, SetlX supports 64 bit floating point numbers according to the standard IEEE 754.

SetlX provides the following arithmetic operators. If not stated otherwise, these operators work for both rational numbers as well as floating point numbers:

1. "+" performs addition.
2. "-" performs subtraction.
3. "*" performs multiplication.
4. "/" performs division.
5. "\" performs integer division. The result is always an integer.
6. "%" computes the rest. This operator is only provided for rational numbers.

   For rational number x and y the operators "*", \" and "%" satisfy the equation

   $$x = (x \setminus y) * y + x \,\%\, y.$$

In order to use floating point numbers instead of rational numbers, the easiest way is to add `0.0` at the end of an expression because if an arithmetic expression contains a floating point value, the result is automatically converted to a floating point number. Therefore, the command

```
=> 1/3 + 2/5 + 0.0;
```

yields the answer:

```
~< Result:  0.7333333333333333 >~
```

Of course, the same result can also be achieved via the expression

```
1.0/3 + 2/5;
```

Since the precision of rational numbers in SetlX is not limited, we can do things like computing $\sqrt{2}$ to a hundred decimal places. Figure 2.3 on page 12 shows a program that uses the *Babylonian method* to compute the square root of 2. The idea is to compute $\sqrt{2}$ as the limit of the sequence $(b_n)_{n\in\mathbb{N}}$, where the numbers b_n are defined by induction on n as follows:

$$b_1 := 2 \quad \text{and} \quad b_{n+1} := \frac{1}{2} \cdot \left(b_n + \frac{2}{b_n}\right).$$

In order to print a rational number in decimal notation with a fixed number of places, the function `nDecimalPlaces`(x, n) has been used. The first argument x is the rational number to be printed, while the second argument n gives the number of places to be printed. To run this program from the command line, assume it is stored in a file with the name `sqrt.stlx` in the current directory. Then the command

```
setlX sqrt.stlx
```

loads and executes this program. Alternatively, the command can be executed interactively in the interpreter via the following command:

```
load("sqrt.stlx");
```

The output of the program is shown in Figure 2.4.

```
1  b := 2;
2  for (n in [1 .. 9]) {
3      b := 1/2 * (b + 2/b);
4      print(n + ": " + nDecimalPlaces(b, 100));
5  }
```

Figure 2.3: A program to calculate $\sqrt{2}$ to 100 places.

```
1: 1.5000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
2: 1.4166666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666
3: 1.4142156862745098039215686274509803921568627450980392156862745098039215686274509803921568627450980392
4: 1.4142135623746899106262955788901349101165596221157440445849050192000543718353892683589900431576443402
5: 1.4142135623730950488016896235025302436149819257761974284982894986231958242289236217849418367358303565
6: 1.4142135623730950488016887242096980785696718753772340015610131331132652556303399785317871612507104752
7: 1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276416016
8: 1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276415727
9: 1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276415727
```

Figure 2.4: The output produced by the program in Figure 2.3.

The previous program also demonstrates that SetlX supports strings. In SetlX, any sequence of characters enclosed in either double or single quote characters is a string. For example, the *hello world program* in SetlX is just

```
=> "Hello world!";
```

It yields the output:

```
~< Result:  "Hello world!" >~
```

However, this only works in interactive mode. If you want to be more verbose[2] or if you are not working interactively, you can instead write

[2] If you want to be much more verbose, you should program in either *Cobol* or *Java*.

```
=> print("Hello world!");
```

This will yield two lines of output:

```
Hello world!
~< Result: om >~
```

The first line shows the effect of the invocation of the function `print`, the second line gives the return value computed by the call of the function `print()`. As the function `print()` does not return a meaningful value, the return value is the *undefined value* that, following the tradition of SETL, is denoted as Ω. In SETLX, this value is written as `om`.

In order to assign the value of an expression to a variable, SETLX provides the *assignment operator* ":=". Syntactically, this operator is the only major deviation from the syntax of the programming language C. For example, the statement

```
x := 1/3;
```

binds the variable `x` to the fraction `1/3`.

2.2 Boolean Values

The Boolean values `true` and `false` represent truth and falsity, respectively. Boolean expressions can be constructed using the binary comparison operators "==", "!=", "<", ">", "<=", and ">=". Note that these operators are identical to the comparison operators used in the programming languages C and *Java*.

2.2.1 Boolean Operators

SETLX provides the following operators from propositional logic to combine Boolean expressions.

1. "&&" denotes the logical *and* (also known as *conjunction*), so the expression

   ```
   a && b
   ```

   is true if and only if both `a` and `b` are true.

2. "||" denotes the logical *or*, which is also known as *disjunction*. Therefore, the expression

   ```
   a || b
   ```

   is true if either `a` or `b` or both are true.

3. "!" denotes the logical *not*, (also known as *negation*). Therefore, the expression

   ```
   !a
   ```

   is true if and only if `a` is false.

   The first three Boolean operators work exactly as they work in C or *Java*.

4. "=>" denotes the logical *implication*. Therefore, the expression

   ```
   a => b
   ```

   is true if `a` is false or `b` is true. Hence the expression

   ```
   a => b
   ```

   has the same truth value as the expression

   ```
   !a || b.
   ```

5. "<==>" denotes the logical *equivalence*, so the expression

   ```
   a <==> b
   ```

   is true if either a and b are both true or a and b are both false. Therefore, the expression

   ```
   a <==> b
   ```

   has the same value as the expression

   ```
   (a && b) || (!a && !b).
   ```

6. "<!=>" denotes the logical *antivalence*, so the expression

   ```
   a <!=> b
   ```

   is true if the truth values of a and b are different. Therefore, the expression

   ```
   a <!=> b
   ```

   is equivalent to the expression

   ```
   !(a <==> b).
   ```

The operators "==" and "!=" can be used instead of the operators "<==>" and "<!=>". However, note that the precedence of these operators differ. While the precedence of "<==>" and "<!=>" is lower than the precedence of any other operator, the precedence of "==" and "!=" is lower than the precedence of the arithmetical operators like "+" and "*", but higher than the precedence of the logical operators "&&" and "||". Therefore, we recommend the use of "<==>" and "<!=>" when comparing Boolean expressions.

2.2.2 Quantifiers

In addition to the propositional operators, SetlX supports both the universal quantifier "forall" and the existential quantifier "exists". For example, to test whether the formula

$$\forall x \in \{1, \cdots, 10\} : x^2 \leq 2^x$$

is true, we can evaluate the following expression:

```
forall (x in {1..10} | x ** 2 <= 2 ** x);
```

This expression checks whether x^2 is less than or equal to 2^x for all x between 1 and 10. Syntactically, a forall expression is described by the following grammar rule:

expr → "forall" "(" *var* "in" *expr* "|" *cond* ")"

Here, *var* denotes a variable and *expr* is an expression that evaluates to a set s (or a list or a string), while *cond* is a Boolean expression. The forall expression evaluates to true if for all elements of s the condition *cond* evaluates to true. There is a generalization of the grammar rule that allows to check several variables simultaneously, so we can write an expression like the following:

```
forall (x in {1..10}, y in [20..30] | x < y).
```

If instead we want to know whether the formula

$$\exists x \in \{1, \cdots, 10\} : 2^x < x^2$$

is true, we have to write:

```
exists (x in {1..10} | 2 ** x < x ** 2);
```

This expression checks whether there exists a natural number $x \in \{1, \cdots, 10\}$ such that $2^x < x^2$.

Important: The quantifiers forall and exists expressions do not create their own local scope. For

example, the sequence of statements

```
exists ([x, y] in {[a,b] : a in {1..10}, b in {1..10}} | 3*x - 4*y == 5);
print("x = $x$, y = $y$");
```

will print

```
x = 3, y = 1
```

since for `x = 3` and `y = 1` the equation

```
3*x - 4*y == 5
```

is true. This Example shows that the values of the variables `x` and `y` is available outside of the existentially quantified expression. The execution of a `forall` or an `exists` expression stops as soon as the truth value of the statement is known. This feature is quite useful if the actual value of the variable in an `exists` statement is needed. It is also useful for a `forall` statement in case that the `forall` statement fails. For example, the expression

```
forall (n in [1..10] | n**2 <= 2**n);
```

evaluates to `false` and, furthermore, it assigns the value 3 to the variable `n`, since 3 is the first integer in the list `[1..10]` such that the expression $n^2 \leq 2^n$ is false. In the case that an expression of the form

`forall (`x `in` s `|` $f(x)$`)`

is true, the variable x will be set to the undefined value `om`. Similarly, if an expression of the form

`exists (`x `in` s `|` $f(x)$`)`

is false, the variable x is set to the undefined value `om`.

2.3 Sets

The most interesting data type provided by SetlX is the *set* type. According to Georg Cantor, who invented set theory, The definition of the notion of a set runs as follows:

> *A set is a gathering together into a whole of definite, distinct objects of our perception or of our thought — which are called elements of the set [Can95].*

This definition has two important implications:

1. Since the elements of a set are distinct, a set contains an object as most once. Hence, an object cannot occur multiple times in a set.
2. A set is defined by its elements. Two sets are identical if and only if they have the same elements.

To create a simple set containing the numbers 1, 2, and 3, we can write:

```
s := {1, 2, 3};
```

This creates a set containing the numbers 1, 2, and 3 as elements. In general, if we want to combine the objects

$$o_1,\ o_2,\ o_3,\ \cdots,\ o_n$$

into a set, we just have to surround these objects by the curly braces "{" and "}" as follows:

$$\{o_1,\ o_2,\ o_3,\ \cdots,\ o_n\}$$

As two sets are equal if they contain the same elements, the order in which the elements are listed

does not matter. For example, the comparison

```
{1,2,3} == {2,3,1}
```

yields the result

```
true.
```

This behaviour can lead to results that are sometimes surprising for the novice. For example, the command

```
print({3,2,1});
```

yields the following output:

```
{1, 2, 3}
```

The reason is, that SetlX does not remember the order of the insertion of different elements into a set, SetlX just remembers the elements and sorts them internally to maximize the efficiency of looking them up.

To check whether a set contains a given entity as an element, SetlX provides the binary infix operator "`in`". For example, the command

```
2 in {1,2,3};
```

yields the result `true`, as 2 is indeed an element of the set $\{1, 2, 3\}$, while

```
4 in {1,2,3};
```

returns `false`.

The data type of a set would be quite inconvenient to use if we could only create sets by explicitly listing all their elements. Fortunately, there are some more powerful expressions to create sets. The operator that is most straightforward to use is the *range operator* that can create a set containing all the integers in a given range. For example, to create the set containing the integers from 1 up to 16, we can write

```
{1..16};
```

This expression returns the result

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}.
```

In general, for integers a and b such that $a \leq b$, the expression

```
{a..b}
```

will generate the set of all integers starting from a up to and including the number b. Mathematically, the semantics of this operator is given by the formula

$$\{a..b\} = \{a + n \mid a \in \mathbb{Z} \wedge a + n \leq b\}.$$

Here $\mathbb{Z}$ denotes the set of all integers. If b is less than a, then the expression

```
{a..b}
```

denotes the empty set $\{\}$.

While successive elements of a set created by the expression "$\{a..b\}$" differ by 1, there is a variant of the range operator that allows us to specify the size of the difference between successive elements. For example, the expression

```
{1,3..10};
```

yields the set

```
{1, 3, 5, 7, 9}.
```

In general, when a set definition of the form

$$\{a,b..c\}$$

is evaluated, there are two cases.

1. $a < b$. In this case, the *step size* s is defined as

   $$s := b - a.$$

   Then, we have

   $$\{a,b..c\} := \{a + n \cdot s \mid n \in \mathbb{N}_0 \wedge a + n \cdot s \leq c\}.$$

2. $a > b$. In this case, the *step size* s is defined as

   $$s := a - b.$$

   Then, we have

   $$\{a,b..c\} := \{a + n \cdot s \mid n \in \mathbb{N} \wedge a + n \cdot s \geq c\}.$$

   For example, we have

   $$\{10,\ 8..1\} = \{10,\ 8,\ 6,\ 4,\ 2\}.$$

2.3.1 Operators on Sets

The following operators act on sets:

1. "+" is used to compute the *union* of two sets.

   In mathematics, the union of two sets s_1 and s_2 is written as $s_1 \cup s_2$. It is defined as

   $$s_1 \cup s_2 := \{x \mid x \in s_1 \vee x \in s_2\}.$$

2. "*" computes the *intersection* of two sets.

   In mathematics, the intersection of two sets s_1 and s_2 is written as $s_1 \cap s_2$. It is defined as

   $$s_1 \cap s_2 := \{x \mid x \in s_1 \wedge x \in s_2\}.$$

3. "-" computes the *difference* of two sets.

   In mathematics, the difference of two sets s_1 and s_2 is often written as $s_1 \backslash s_2$. It is defined as

   $$s_1 \backslash s_2 := \{x \in s_1 \mid x \notin s_2\}.$$

4. "><" computes the *Cartesian product* of two sets.

   In mathematics, the Cartesian product of two sets s_1 and s_2 is written as $s_1 \times s_2$. It is defined as the set of all pairs $[x_1, x_2]$ such that x_1 is an element of s_1 and x_2 is an element of s_2:

   $$s_1 \times s_2 := \{[x_1, x_2] \mid x_1 \in s_1 \wedge x_2 \in s_2\}.$$

5. "** 2" computes the *Cartesian product* of a set with itself, i.e. we have

   $$s ** 2 := s \times s.$$

6. "2 ** " computes the *power set* of a given set. For a given set s, the power set of s is defined as the set of all subsets of s. In mathematics, the power set of a given set s is written as 2^s and the formal definition is

   $$2^s := \{m \mid m \subseteq s\}.$$

   For example, the expression

```
2 ** {1,2,3}
```

yields the result

```
{{}, {1}, {1, 2}, {1, 2, 3}, {1, 3}, {2}, {2, 3}, {3}}.
```

7. "%" computes the *symmetric difference* of two sets.

   In mathematics, the symmetric difference of two sets s_1 and s_2 is often written as $s_1 \triangle s_2$. It is defined as

   $$s_1 \triangle s_2 := (s_1 \backslash s_2) \cup (s_2 \backslash s_1).$$

Therefore, the commands

```
s1 := { 1, 2 }; s2 := { 2, 3 };
print("s1 +  s2 = $s1 +  s2$");
print("s1 -  s2 = $s1 -  s2$");
print("s1 *  s2 = $s1 *  s2$");
print("s1 **  2 = $s1 ** 2 $");
print("2  ** s2 = $2  ** s2$");
print("s1 >< s2 = $s1 >< s2$");
print("s1 %  s2 = $s1 %  s2$");
```

will produce the follwing results:

```
s1 +  s2 = {1, 2, 3}
s1 -  s2 = {1}
s1 *  s2 = {2}
s1 **  2 = {[1, 1], [1, 2], [2, 1], [2, 2]}
2  ** s2 = {{}, {2}, {2, 3}, {3}}
s1 >< s2 = {[1, 2], [1, 3], [2, 2], [2, 3]}
s1 %  s2 = {1, 3}
```

For sets, the exponentiation operator "**" is only defined if either the base of the exponentiation is a set and the exponent is the number 2 or the base is the number 2 and the exponent is a set. Therefore, if s is a set, expressions like

3 ** s and s ** 3

are undefined. Also note the use of *string interpolation* in the `print` statements given above: Inside a string, any expression enclosed in $-symbols is evaluated and the result of this evaluation is inserted into the string. String interpolation will be discussed in more detail later when we explain how strings are supported in SetlX.

For all of the binary operators discussed above there is a variant of the assignment operator that incorporates the binary operator. For example, the command

```
s += {x};
```

adds the element `x` to the set `s`, while the command

```
s -= {x};
```

removes the element `x` from the set `s`. Of course, these modified assignment operators also when these operators act on numbers, i.e. the statement

```
x += 1;
```

increments the value of the variable `x` by 1.

In addition to the basic operators discussed above, SetlX provides a number of more elaborate operators for sets. One of these operators is the cardinality operator "#", which computes the number

of elements of a given set. The cardinality operator is used as a unary prefix operator, for example we can write:

```
# {5,7,13};
```

Of course, in this example the result will be 3. We can sum the elements of a set using the prefix operator "+/". For example, in order to compute the sum

$$\sum_{i=1}^{6^2} i,$$

we can create the set {1..6**2} containing all the numbers to sum and then use the command

```
+/ {1..6**2};
```

to calculate the actual sum. For arguments that are numbers, "**" is the exponentiation operator. Therefore, the expression given above first computes the set of all numbers from 1 up to the number 36 and then returns the sum of all these numbers. There is also a binary version of the operator "+/" that is used as an infix operator. For a number x and a set s, the expression

```
x +/ s
```

will insert x into the set s if the set s is empty. After that, it returns the sum of all elements in the resulting set. This is useful as the expression "+/ {}" is undefined. So if we want to compute the sum of all numbers in s but we are not sure whether s might be empty, we can use the expression

```
0 +/ s
```

since inserting 0 into the empty set guarantees that the result is 0 in case the set s is empty. If s is a set whose elements are sets themselves, the expression

```
+/ s
```

computes the union of the sets that are elements of the set s. For example, the expression

```
+/ { {1,2,3}, {3,4,5}, {5,6,7} }
```

yields the result

```
{1, 2, 3, 4, 5, 6, 7}.
```

There is a similar operator for multiplying the elements of a set: It is the operator "*/". For example, in order to compute the factorial[3] $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$ we can use the expression

```
*/ {1..n}.
```

Again, there is also a binary version of this operator. Since the number 1 is the neutral element for multiplication, the first argument to this operator will most often be one. Therefore, the expression

```
1 */ s
```

multiplies all the numbers from a the set s and this expression will return 1 if the set s happens to be empty.

If s is a set whose elements are sets themselves, the expression

```
*/ s
```

computes the intersection of the sets that are elements of the set s. For example, the expression

```
*/ { {1,2,3}, {2,3,4} }
```

yields the result

[3] The factorial operator "!" is a builtin postfix operator. Computing the factorial of a number using this operator is more efficient than first building a set and then using the operator "*/".

{2, 3}.

2.3.2 Set Comprehensions

We can use *set comprehension* to build sets. For example, the command

```
{ a * b:  a in { 1..3 }, b in { 1..3 } };
```

computes the set of all products $a*b$ where both a and b run from 1 to 3. The command will therefore compute the set

{1, 2, 3, 4, 6, 9}.

In general, a *set comprehension expression* has the form

{ *expr* : x_1 in s_1, ⋯, x_n in s_n | *cond* }.

Here, *expr* is some expression containing the variables x_1, ⋯, x_n, while s_1, ⋯, s_n are either sets or lists (to be discussed later), and *cond* is an expression returning a Boolean value. The set comprehension expression given above evaluates *expr* for all possible combinations of values $x_1 \in s_1$, ⋯, $x_n \in s_n$ such that *cond* is true and it will add the corresponding value of *expr* into the resulting set. The expression *cond* is optional. If it is missing, it is implicitly taken to be always true.

We are now ready to demonstrate some of the power that comes with sets. The following two statements compute the set of all prime numbers smaller than 100:

```
s := {2..100};
s - { p * q :  p in s, q in s };
```

The expression given above computes the set of prime numbers smaller than 100 correctly, as a prime number is any number bigger than 1 that is not a proper product. Therefore, if we subtract the set of all proper products from the set of numbers, we get the set of prime numbers. Obviously, this is not an efficient way to calculate primes, but efficiency is not the point of this example. It rather shows that SetlX is well suited to execute a mathematical definition as it is.

Let us look at another example of set comprehension: The expression

```
{ p: p in {2..100} | { t: t in {2..p-1} | p % t == 0 } == {} }
```

yields the set of all prime numbers less than 100. The subexpression

```
{ t: t in {2..p-1} | p % t == 0 }
```

computes the set of all those natural numbers less than p that divide p evenly. If this set is empty, then, by the definition of a prime number, p is prime.

The syntax for building set comprehensions is a little bit more general than discussed above. Provided s_1, ⋯, s_n are sets of *lists*, where all of the lists in the set s_i have the same length m_i, a set comprehension expression can take the form

{ *expr* : $[x_1, x_2, \cdots, x_{m_1}]$ in s_1, ⋯, $[z_1, z_2, \cdots, z_{m_n}]$ in s_n | *cond* }.

This kind of set comprehension expressions will be discussed further when discussing lists.

2.3.3 Miscellaneous Set Functions

In addition to the operators provided for sets, SetlX has a number of functions targeted at sets. The first of these functions is arb. For a set s, an expression of the form

arb(s)

yields some element of the set s. The element returned is not specified, so the expression

```
arb({1,2,3});
```

might yield either 1, 2, or 3. The function `from` has a similar effect, so

```
from(s)
```

also returns some element of the set s. In addition, this element is removed from the set. Therefore, if the set `s` is defined via

```
s := {1,2,3};
```

and we execute the statement

```
x := from(s);
```

then some element of `s` is assigned to `x` and this element is removed from the set `s`. For example, after this assignment, we possibly have

```
x = 3   and   s := {1,2}.
```

Of course, we could just as well have

```
x = 1   and   s := {2,3}
```

as a result of the assignment "`x := from(s);`".

For a set s, the expression

```
first(s)
```

returns the *first* element of the set s, while

```
last(s)
```

yields the *last* element. Internally, sets are represented as ordered binary trees. As long as the sets contain only numbers, the ordering of these numbers is the usual ordering on numbers. Therefore, in this case the *element* is just the smallest number and the *last element* is the biggest number. If the sets contain sets (or lists), these sets (or lists) are themselves compared lexicographically. Things get a little bit more complicated if a set is heterogeneous and contains both number and sets. Programs should not rely on the ordering implemented in these cases.

For a given set s, the expression

```
rnd(s)
```

yields a random element of the set s. In contrast to the expression "`arb(`s`)`", this expression will in general return different results on different invocations. The function `rnd` also works for lists, so for a list l the expression `rnd(`l`)` returns a random element of l. The function `rnd()` is also supported for integer numbers, for example the expression

```
rnd(5)
```

computes a random non-negative number less or equal than 5. Essentially it is equal to the expression `rnd({0..5})`. While we are at it: The function `random` computes pseudo-random values in the interval $[0, 1]$. This function does not take an argument. Therefore, the expression

```
random()
```

might output a value like `0.5171688974345957`.

2.4 Lists

Besides sets, SetlX also supports lists. Syntactically, the main difference between sets and lists is that the curly braces "{" and "}" are substituted with the square brackets "[" and "]". Semantically, while a set is an unordered collection of elements that can contain any element at most once, a list is an ordered collection of elements that can contain an element multiple times. For example

```
[3,4,7,2,5,3]
```

is a typical list. Note that the element 3 occurs twice in this list. The easiest way to construct a list is by defining the list explicitly. For example, the assignment

```
l := [1,4,7,2,4,7]
```

defines the list `[1,4,7,2,4,7]` and assigns it to the variable `l`. Another way to construct lists is via the range operator "`..`": For example, the expression

```
[1..10]
```

returns the list

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
```

The prefix operators "`+/`" and "`*/`" work for lists in the same way that they work for sets: "`+/ l`" computes the sum of all elements of the list `l`, while "`*/ l`" computes the product of the elements of `l`. For example, the expression

```
+/ [1..100]
```

yields the result 5050.

List comprehension works similar to set comprehension, the expression

`[` *expr* `:` x_1 `in` s_1`,` $\cdots$`,` x_n `in` s_n `|` *cond* `]`

picks up all lists

$$[x_1, \cdots, x_n] \in s_1 \times \cdots \times s_n$$

such that *cond* is true and evaluates *expr* for the corresponding values of the variables x_1, $\cdots$, x_n. The value of *expr* is then inserted into the resulting list. Here, the s_i denote either sets or lists. For example, the following expression computes all primes up to 100:

```
[ p :  p in [2..100] | { x :  x in {1..p} | p % x == 0 } == {1, p} ].
```

Here, the idea is that a number p is prime if the set of its divisors, which is the set of all numbers x such that $p \,\%\, x = 0$, only contains the number 1 and the number p.

The syntax for building lists via comprehensions is a little bit more general than discussed above. Provided s_1, $\cdots$, s_n are sets or lists, such that all of the elements of s_i are lists that have the same length m_i, a list can be defined as

`[` *expr* `:` $[x_1, x_2, \cdots, x_{m_1}]$ `in` s_1`,` $\cdots$`,` $[z_1, z_2, \cdots, z_{m_n}]$ `in` s_n `|` *cond* `]`.

For example, assume that `r1` and `r2` are sets containing *pairs* of elements, where a *pair* is just a list of length two. In this case, `r1` and `r2` could be viewed as binary relations. Then, mathematically the *composition* of `r1` and `r2` is defined as

$$\texttt{r1} \circ \texttt{r2} := \{[x, z] \mid \exists y : [x, y] \in \texttt{r1} \wedge [y, z] \in \texttt{r2}\}.$$

In SetlX, this composition can be computed as

```
{ [x,z]: [x,y] in r1, [y,z] in r2 }.
```

2.4.1 Operators on Lists

Currently, there are four operators for lists. These are discussed below.

1. The operator "`+`" concatenates its arguments. For example

   ```
   [1 ..  3] + [5 ..  10];
   ```

   yields

```
[1, 2, 3, 5, 6, 7, 8, 9, 10].
```

2. The operator "*" takes a list as its first argument, while the second argument needs to be a natural number or 0. An expression of the form

   ```
   l * n
   ```

   concatenates `n` copies of the list `l`. Therefore, the expression

   ```
   [1, 2, 3] * 3
   ```

   yields the result

   ```
   [1, 2, 3, 1, 2, 3, 1, 2, 3].
   ```

3. The cardinality operator "#" is a prefix operator that works for lists the same way it works for sets, i.e. it returns the number of elements of the list given as argument. For example,

   ```
   # [ 7, 4, 5 ];
   ```

   yields the result 3.

4. The infix operator "><" takes two lists l_1 and l_2. The length of l_1 has to be the same as the length of l_2. The operator *zips* the lists given as arguments to produce a list of pairs, where the first component of each pair is from the list l_1, while the second component is the corresponding element from the list l_2. For example, the expression

   ```
   [1, 2, 3] >< ["a", "b", "c"];
   ```

   returns the list

   ```
   [[1, "a"], [2, "b"], [3, "c"]].
   ```

   Formally, the behaviour of the operator "><" on lists l_1 and l_2 is specified as follows:

   l_1 `><` l_2 `== [ [`l_1`[`i`], `l_2`[`i`]] :` i `in [1 ..` `#`l_1`] ]`.

   Here, l`[`i`]` denotes the i-th element of the list l and the lists l_1 and l_2 must have the same length.

2.4.2 Extracting Elements from a List

As the elements of a list are ordered, it is possible to extract an element from a list with respect to its position. In general, for a list l and a positive natural number n, the expression

l`[`n`]`

selects the n-th element of l. Here, counting starts with 1, so $l[1]$ is the first element of the list l. For example, after the assignment

```
l := [99, 88, 44];
```

the expression

```
l[2]
```

yields the result 88. The right hand side of an extraction expression can also be a *slicing operator*: The expression

l`[`a`..`b`]`

extracts the sublist of l that starts at index a and ends at index b. For example, after defining l via the assignment

```
l := ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j" ];
```

the expression

```
l[5..8]
```

yields the sublist `["f", "g", "h"]`. The expression

l`[`a`..`b`]`

returns the empty list if $a > b$ and both a and b are positive.

There are two more variants of the slicing operator. The expression

l`[`a`..]`

returns the sublist of l that starts with the element at index a, and the expression

l`[..`b`]`

computes the sublist of l that starts at the first element and includes all the elements up to the element at index b.

The index operator also works with negative indices. For a nonempty list l, the expression

l`[-1]`

selects the last element of l, while

l`[-2]`

would select the second to last element, provided the list has at least two elements. In general, for a list l of length n and an index $i \in \{1, \cdots, n\}$, the expression

l`[-`i`]`

selects the element with index $n + 1 - i$, i.e. for $i \in \{1, \cdots, n\}$ the expressions

l`[-`i`]` and l`[`$n + 1 - i$`]`

are equivalent. Negative indices also work with the slicing operator. For example, for a list l containing at least two elements, the expression

l`[2..-2]`

returns the list that results from l by chopping of both the first and the last element of l. The variants

`l[`a`..]` and `l[..`b`]`

also work with negative values of a and b. For example, the expression

l`[..-2]`

returns a list containing all elements of l but the last.

After a list is created, we can add new elements to the list by assigning to the list as the following example demonstrates. After defining `l` as

```
l := ["a", "b", "c"];
```

we can use the assignment

```
l[5] := "e";
```

to assign a fifth element to `l`. Here, you might wonder how it is possible to add a fifth element, as the list `l` does not have a fourth element. However, after the assignment shown above, we have

```
l == ["a", "b", "c", om, "e"].
```

Hence, in this case the fourth element is simply the undefined value `om`.

When indexing a list, the number 0 is not allowed as an index. Therefore, the expression

```
l[0]
```

will yield the following error message:

```
Error in "l[0]":
Index '0' is invalid.
```

However, the index 0 is allowed as the second index in a range operator. For example, the expression `l[1..0]` yields the empty list.

Lists of variables can be used on the left hand side of an assignment. This feature enables the *simultaneous assignment* of several variables. The statement

```
[x,y] := [1,2];
```

sets the variable x to 1 and y to 2. A more interesting example is the following assignment:

```
[y,x] := [x,y];
```

This assignment swaps the values of the variables x and y.

2.5 Pairs, Relations, and Functions

In SETL, a pair of the form $\langle x, y\rangle$ is represented as the list $[x, y]$. A set of pairs can be regarded as a binary relation and the notion of a relation is a generalization of the notion of a function. If r is a binary relation, we define the *domain* and *range* of r as the set containing the first and second component of the pairs, that is we have

`domain(r) = { x : [x,y] in r }` and `range(r) = { y : [x,y] in r }`.

Furthermore, if r is a binary relation such that

$$[x, y_1] \in r \wedge [x, y_2] \in r \rightarrow y_1 = y_2$$

holds for all x, y_1, and y_2, then r is called a *map* and represents a function. Therefore, a map is a set of `[key, value]` pairs such that the keys are *unique*, i.e. every key is associated with exactly one value. If a binary relation r is a map, SETLX permits us to use the relation r as a function: If r is a map and $x \in \mathit{domain}(r)$, then $r[x]$ denotes the unique element y such that $\langle x, y\rangle \in r$:

$$r[x] := \begin{cases} y & \text{if the set } \{y \mid [x, y] \in r\} \text{ contains exactly one element } y; \\ \Omega & \text{otherwise.} \end{cases}$$

The program shown below in Figure 2.5 provides a trivial example demonstrating how maps can be used as functions in SETLX.

```
1  r := { [n, n*n] : n in {1..5} };
2  print( "r[3]      = $r[3]$"      );
3  print( "domain(r) = $domain(r)$" );
4  print( "range(r)  = $range(r)$"  );
5  r[2] := 17;
6  print( "r         = $r$" );
```

Figure 2.5: Binary relations as functions.

The program computes the map `r` that represents the function $x \mapsto x * x$ on the set

$$\{n \in \mathbb{N} \mid 1 \leq n \wedge n \leq 5\}.$$

In line 2, the relation r is evaluated at $x = 3$. This is done using square brackets. Next, *domain*(r) and *range*(r) are computed. Finally, the assignment in line 5 changes the relation for the argument 2 to yield the value 17 instead of 4. Hence, we get the following result:

```
r[3]      = 9
domain(r) = {1, 2, 3, 4, 5}
range(r)  = {1, 4, 9, 16, 25}
r         = {[1, 1], [2, 17], [3, 9], [4, 16], [5, 25]}
```

It is a natural question to ask what happens if r is a binary relation and we try to evaluate the expression $r(x)$ but the set { y : [x, y] in r } is either empty or contains more than one element. The program shown in Figure 2.6 on page 26 answers this question.

```
1  r := { [1, 1], [1, 4], [3, 3] };
2  print( "r[1] = ", r[1] );
3  print( "r[2] = ", r[2] );
4  print( "{ r[1], r[2] } = ", { r[1], r[2] } );
5  print( "r{1} = ", r{1} );
6  print( "r{2} = ", r{2} );
```

Figure 2.6: A binary relation that is not a map.

If the set { y :[x, y] in r } is either empty or has more than one element, then the expression r[x] is undefined. In mathematics, an undefined value is sometimes denoted as Ω. In SetlX, the undefined value is printed as "om". If we try to add the undefined value to a set m, then m is not changed. Therefore, line 4 of the program shown in Figure 2.6 prints the empty set, as both r[1] and r[2] are undefined.

We can use the notation r{x} instead of r[x] to avoid undefined values. For a binary relation r and an object x, the expression $r\{x\}$ is defined as follows:

```
r{x} := { y : [x, y] in r }.
```

Therefore, r{x} is the set of all y such that there is an x such that the pair [x, y] is an element of r. Hence, the program shown in Figure 2.6 yields the following results:

```
r[1] = om
r[2] = om
{ r[1], r[2] } = {}
r{1} = {1, 4}
r{2} = {}
```

Binary relations can also be used to represent functions accepting more than one argument. For example, the statements

```
r := {};
r[1,2] := 3;
r[2,1] := 4;
```

define a function that maps the pair $[1, 2]$ to the value 3, while the pair $[2, 1]$ is mapped to the value 4. If we print r after these statements have been executed, then the set

```
{[[1, 2], 3], [[2, 1], 4]}
```

is stored in the variable r.

Remark: This section has shown that binary relations can be used to represent the data type of a *dictionary*. Other programming languages, for example *Perl* [WS92], provide *associative arrays* to

represent dictionaries. In most of the scripting languages providing associative arrays, these associative arrays are implemented as *hash tables*. In the programming language SETL, sets and relations were also represented via hash tables. In contrast, the sets (and therefore the dictionaries) in SETLX are implemented as *red-black trees*. Although implementing sets as red-black trees is slightly slower than a hash table based implementation, the advantage of using red-black trees is that they support a number of operations that are not available when using hash-tables. The efficient implementation of the functions `first` and `last` would be impossible if sets had been represented as hash tables.

2.6 Procedures

Although functions can be represented as binary relations, this is not the preferred way to represent functions. After all, using a relation to represent a function has a big memory footprint and also requires to compute all possible function values regardless of their later use. Therefore, the preferred way to code a function is to use a *procedure*. For example, Figure 2.7 defines a procedure to compute all prime numbers up to a given natural number n. The idea is to take the set s of all numbers in the range from 2 upto n and then to subtract the set of all non-trivial products from this set. This will leave us with the set of all prime numbers less or equal to n as a natural number is prime if and only if it is not a non-trivial product.

```
1  primes := procedure(n) {
2      s := { 2..n };
3      return s - { p*q : [p, q] in s >< s };
4  };
```

Figure 2.7: A procedure to compute the prime numbers.

In Figure 2.7, the block starting with "`procedure` {" and ending with the closing brace "}" defines a function. This function is then assigned to the variable `primes`. Therefore, a function is just another kind of a value. Conceptually, the type of functions is not different from the type of sets or strings: A function can be assigned to a variable, it can be used as an argument to another function and it can also be returned from another function. To summarize, functions are *first class citizens* in SETLX. The ramifications of this fact will be explained in Chapter 5 on functional programming and *closures*.

Remark: A function defined using the keyword `procedure` is **not** able to read or write variables that are defined outside of the function definition. If you need access to variables defined outside of the function, you have to define a *closure* instead. This is done by substituting the keyword `procedure` with the keyword `closure`. Closures are discussed in more detail in Chapter 5 later.

2.7 Strings

Any sequence of characters enclosed in either double quotes or single quotes is considered a string. Strings can be concatenated using the infix operator "+", so

```
"abc" + "xyz"
```

yields the string `"abcxyz"` as a result. In order to concatenate multiple instances of the same string, we can use the infix operator "*". For a string s and a natural number n, the expression

s * n

returns a string that consists of n copies of the string s. For example, the expression

```
"abc" * 3
```

yields the result

```
"abcabcabc".
```

This also works when the position of the string and the number are exchanged. Therefore,

```
3 * "abc"
```

also yields `"abcabcabc"`.

In order to extract the i-th character of the string s, we can use the expression

s`[`i`]`.

As for lists, this also works for negative values of the index i. For example, the expression s`[-1]` returns the last character of s.

There is also a slicing operator. This operator works similar to lists, for example, if s has the value `"abcdef"`, then

```
s[2..5];
```

yields the result `"bcde"`, while

```
s[2..];
```

yields `"bcdef"` and

```
s[..5];
```

gives `"abcde"`. In the expressions

s`[`a`..`b`]`, s`[`a`..]`, and s`[..`b`]`

the indices a and b can be negative numbers. This works in the same way as it works for lists. For example, if `s` is a string, then `s[2..-2]` is the string that results from `s` by removing the first and the last character.

SetlX provides *string interpolation*: If a string contains a substring enclosed in "$"-symbols, then SetlX parses this substring as an expression, evaluates this expression, and then substitutes the result back into the string. For example, if the variable `n` has the value 6, the command

```
print("$n$!  = $n!$");
```

will print

```
6!  = 720.
```

In order to insert a literal "$"-symbol into a string, the "$"-symbol has to be escaped with a backslash. For example, the command

```
print("A single \$-symbol.");
```

prints the text:

```
A single $-symbol.
```

2.7.1 Literal Strings

Sometimes it is necessary to turn off any kind of preprocessing when using a string. This is achieved by enclosing the content of the string in single quotes. These strings are known as *literal* strings. For example, after the assignment

```
s := '\n';
```

the string `s` contains exactly two characters: The first character is the backslash "\", while the second character is the character "n". If instead we write

```
s := "\n";
```

then the string s contains just one character, which is the newline character. SetlX supports the same escape sequences as the language C.

As we have just seen, there is no replacement of escape sequences in a literal string. There is no string interpolation either. Therefore, the statement

```
print('$1+2$');
```

prints the string "$1+2$", where we have added the opening and closing quotes to delimit the string, they are not part of the string. Later, literal strings will come in very handy when dealing with *regular expressions*.

Remark: It should be noted that SetlX does not have a special data type to support single characters. Instead, in SetlX single characters are represented as strings of length one.

2.8 Terms

SetlX provides *first order terms* similar to the terms available in the programming language *Prolog* [SS94]. Terms are built from *functors* and *arguments*. To distinguish functors from function symbols, a functor is prefixed with the operator "@". For example, the expression

```
@f(1,"x")
```

is a term with functor f and two arguments. The functor is just a name, it is not a function that can be evaluated. To demonstrate the usefulness of terms, consider implementing *ordered binary trees* in SetlX. There are two types of ordered binary tress:

1. The empty tree represents the empty set. We use the functor Nil to represent an empty binary tree, so the term

   @Nil()

   codes the empty tree.

2. A non-empty binary tree has three components:

   (a) The *root* node,

   (b) the left subtree, and

   (c) the right subtree.

   The root node stores one element k and all elements in the left subtree l have to be less than k, while all elements in the right subtree are bigger than k. Therefore, a non-empty binary tree can be represented as the term

   @Node(k, l, r),

   where k is the element stored at the root, l is the left subtree and r is the right subtree.

For example, the term

```
@Node(2, @Node(1,@Nil(),@Nil()), @Node(3,@Nil(), @Nil()))
```

is a typical binary tree. At the root, this tree stores the element 2, the left subtree stores the element 1 and the right subtree stores the element 3.

There are two functions to decompose a term and there is one function that constructs a term.

1. If t is a term, then the expression

   fct(t)

returns the functor of the term t as a string. For example, the expression

```
fct(@Node(3,@Nil(),@Nil()))
```

yields the result "Node".

2. If t is a term, then the expression

   `args`(t)

   returns the list of arguments of the term t. For example, the expression

   ```
   args(@Node(3,@Nil(),@Nil()))
   ```

   yields the result

   ```
   [3, @Nil(), @Nil()].
   ```

3. The function `makeTerm`(f, l) constructs a term t such that

   `fct`$(t) = f$ and `args`$(t) = l$

   holds. For example,

   ```
   makeTerm("Node", [ makeTerm("Nil",[]), makeTerm("Nil",[]) ])
   ```

   constructs the term

   ```
   @Node(3, @Nil(), @Nil()).
   ```

   Note that we must not use the operator "@" when using the function `makeTerm`. Of course, the term `@Node(3, @Nil(), @Nil())` can also be given directly as an expression: The statement

   ```
   a := @Node(3,@Nil(),@Nil());
   ```

   assigns the term `@Node(3,@Nil(),@Nil())` to the variable a.

```
1  insert := procedure(m, k1) {
2      switch {
3          case fct(m) == "Nil" :
4               return @Node(k1, @Nil(), @Nil());
5          case fct(m) == "Node":
6               [ k2, l, r ] := args(m);
7               if (k1 == k2) {
8                   return @Node(k1, l, r);
9               } else if (compare(k1, k2) < 0) {
10                  return @Node(k2, insert(l, k1), r);
11              } else {
12                  return @Node(k2, l, insert(r, k1));
13              }
14     }
15 };
```

Figure 2.8: Inserting an element into a binary tree.

Figure 2.8 shows how terms can be used to implement binary trees. In this example, we define a function with the name `insert`. This function takes two arguments. The first argument `m` is supposed to be a term representing an ordered binary tree. The second argument `k1` denotes the element that is to be inserted into the binary tree `m`. The implementation needs to distinguish two cases:

1. If the binary tree `m` is empty, then the function returns the tree

   `@Node(k1, @Nil(), @Nil())`.

   This is a binary tree containing the number `k1` at its root, while both subtrees are empty. In order to check whether `m` is indeed empty we use the functor of the term `m`. We test in line 3 whether this functor is `"Nil"`.

2. If the binary tree `m` is nonempty, the functor of `m` is `"Node"`. In this case, we need to extract the arguments of this functor, which is done in line 6: The first argument `k2` is the element stored at the root, while the arguments `l` and `r` correspond to the left and the right subtree of `m` respectively. The predefined function `compare` is used to compare the element `k1` that is to be inserted into the tree with the element `k2`, which is the element at the root of the tree.

   The expression `compare`$(k1, k2)$ returns -1 if `k1` is less than `k2`, it returns $+1$ is `k1` is greater than `k2`, and it returns 0 if `k1` and `k2` are the same. The function `compare` is implemented for all data types. However, it should be noted that the comparison operators "<", ">", "<=", and ">=" are only implemented for numbers and sets.

This example uses a number of features of SetlX that have not been introduced. The discussion of the control structure `switch` will be given in the next chapter.

The present discussion of terms is not complete and will be continued in the next chapter when we discuss matching.

2.9 Vectors and Matrices

SetlX has some support for both vectors and matrices. Suppose we have to solve the following system of linear equations:

$$\begin{aligned} 1 \cdot x + 2 \cdot y + 3 \cdot z &= 1, \\ 2 \cdot x + 3 \cdot y + 1 \cdot z &= 2, \\ 3 \cdot x + 1 \cdot y + 2 \cdot z &= 3. \end{aligned}$$

In order to solve this system of equations we define the matrix

$$\mathrm{a} := \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix} \quad \text{and the vectors} \quad \vec{b} := \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad \vec{s} := \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Then, the solution $\vec{s}$ can be calculated via the formula

$$\vec{s} = \mathrm{a}^{-1} \cdot \vec{b},$$

where a^{-1} denotes the *inverse* of the matrix a. In order to carry out this calculation in SetlX, we run the commands shown in Figure 2.9 on page 31. We discuss these statements line by line.

```
1  a := la_matrix([[1, 2, 3], [2, 3, 1], [3, 1, 2]]);
2  b := la_vector([1, 2, 3]);
3  s := a**-1 * b;
4  print(s);
```

Figure 2.9: Solving a system of linear equations using matrix inversion.

1. Line 1 defines the matrix a using the predefined function `la_matrix`. Note that the matrix a is defined as a list of its rows, where each row is represented as a list of numbers.

2. Line 2 defines the vector $\vec{b}$ using the predefined function `la_vector`.

3. In line 3, the expression `a**-1` computes the inverse of the matrix a. This inverse is then multiplied with the vector $\vec{b}$ to yield the solution $\vec{s}$.

4. In line 4, printing the variable `s` yields the following result:

```
< 0.9999999999999998 > < 1.1102230246251565E-16 > < 5.551115123125783E-17 >
```

This shows that the solution to the system of equations that was given initially is approximately

$$x \approx 1, \quad y \approx 0, \quad \text{and} \quad z \approx 0.$$

Instead of computing the inverse of the matrix a we could have solved the system of equations using the command `la_solve` as follows:

```
s := la_solve(a, b);
```

In that case, SetlX computes the solution

```
< 1.0 > < 0.0 > < 0.0 >.
```

The solution computed by `la_solve` is more precise than the solution computed using the inverse matrix. Presumable the method `solve` from the Jama library that is working behind the scenes of the function `la_solve` is using a final iteration step to increase the precision of the result.

Chapter 7 discusses the data types of vectors and matrices in more detail.

Chapter 3

Statements

This section discusses the various possibilities to write statements and the features offered by SetlX to steer the control flow in a program. Besides the assignment statement, SetlX supports the following means to control the order of statement execution:

1. branching statements like `if-then-else`, `switch`, and `match`,
2. the looping statements `for` and `while` together with `break` and `continue`,
3. the `try-catch` statement to deal with exceptions,
4. the `backtrack` statement to support a limited form of backtracking.

3.1 Assignment Statements

The most basic command is the assignment statement. In contrast to the programming languages `C` and *Java*, SetlX uses the operator ":=" to assign a value to a variable. For example, the statement

```
x := 2/3;
```

binds the variable `x` to the fraction $\frac{2}{3}$. SetlX supports simultaneous assignments to multiple variables via lists. For example, the statement

```
[x, y, z] := [1, 2, 3];
```

simultaneously binds the variables `x` to 1, `y` to 2, and `z` to 3. This feature can be used to swap the values of two variables. For example, the statement

```
[x, y] := [y, x];
```

swaps the values of x and y. If we do not need to assign all the values of a list, we can use the underscore "_" as a so called *anonymous variable*. For example, if the list `l` happens to have the value `[1,2,3]`, then the statement

```
[x, _, z] := l;
```

assigns the number `1` to the variable `x` and the variable `z` is is set to 3.

The assignment operator can be combined with any of the operators "+", "-", "*", "/", "%", and "\". For example, the statement

```
x += 1;
```

increments the value of the variable `x` by one, while the statement

```
x *= 2;
```

doubles the value of `x`. Finally, assignment statements can be chained. For example, the statement

```
a := b := 3;
```

assigns the value 3 to both `a` and `b`.

3.2 Functions

The code shown in figure 3.1 on page 34 shows a simple program to compute *prime numbers*. It defines two functions. The function `factors` takes a natural number `p` as its first argument and computes the set of all *factors* of `p`. Here, a number `f` is a factor of `p` iff dividing `p` by `f` leaves no remainder, that is the expression `p % f` is equal to `0`. The second function `primes` takes a natural number `n` as its sole argument and computes the set of all those numbers `p` less or equal to `n` that have only the trivial factors `1` and `p`. These numbers are, by definition, prime numbers.

Note that, as SetlX is a functional language, the functions that are defined by the keyword `procedure` are assigned to variables. As already mentioned in the previous chapter, these functions can be used like any other values.

```
1  factors := procedure(p) {
2      return { f : f in { 1 .. p } | p % f == 0 };
3  };
4  primes := procedure(n) {
5      return { p : p in { 2 .. n } | factors(p) == { 1, p } };
6  };
7  print(primes(100));
```

Figure 3.1: A naive program to compute prime numbers.

A simplified grammar rule for the definition of a function can be given as follows:

fctDef → `VAR` “`:=`” “`procedure`” “(” *paramList* “)” “{” *block* “}” “;”

The meaning of the symbols used in this grammar rule are as follows:

1. `VAR` identifies a variable. This variable is bound to the definition of the function.

2. *paramList* is a list of the formal parameters of the function. In Ebnf-notation the grammar rule for *paramList* is given as

   $$\begin{array}{lcl} \textit{paramList} & \rightarrow & \big(\textit{paramSpec}\ (\text{“,”}\ \textit{paramSpec})^*\ (\text{“,”}\ \text{“*”}\ \texttt{VAR})?\big)? \\ & \mid & \text{“*”}\ \texttt{VAR} \end{array}$$

   Therefore, a *paramList* is a possibly empty list of parameter specifications that are separated by a comma “,”. Optionally, as the last element, a parameter list can contain a variable that is prefixed with the operator “*”. This is used when a function takes a variable number of arguments and will be explained later. Furthermore, a parameter list can consist of a single variable prefixed with the operator “*”.

   A parameter specification is either just a variable, or it is a variable with the assignment of a default value, or it is a variable preceded by the token “`rw`”:

   $$\begin{array}{lcl} \textit{paramSpec} & \rightarrow & \texttt{VAR} \\ & \mid & \texttt{VAR}\ \text{“:=”}\ \textit{expr} \\ & \mid & \text{“rw”}\ \texttt{VAR} \end{array}$$

If the parameter is preceded by the keyword `rw`, then this parameter is a *read-write parameter*, which means that the function can change the value of the variable given as argument and this change will then be visible outside of the function. Therefore, parameters prefixed with the keyword `rw` have a *call by name* semantics. Parameters not specified as read-write parameters have a strict *call by value* semantics, and hence changes to those parameters will not be visible outside the function.

3. *block* is a sequence of statements.

Note that the definition of a function has to be terminated by the symbol ";". The reason is that the function is part of an assignment and every assignment is terminated with a semicolon. Let us inspect a few examples of function definitions. Figure 3.2 shows several functions definition.

```
1  f := procedure(a, b, c) {
2      return a + b + c;
3  };
4  print(f(1,2,3));
5
6  g := procedure(a, b := 2, c := 3) {
7      return a + b + c;
8  };
9  print(g);
10 print(g(1));
11 print(g(1, 3));
12 print(g(1, 3, 5));
13
14 h := procedure(a := 1, b := 2, c := 3, *rest) {
15     s := a + b + c;
16     for (x in rest) {
17         s += x;
18     }
19     return s;
20 };
21 print(h);
22 print(h(1));
23 print(h(2, 4));
24 print(h(2, 4, 6));
25 print(h(1, 2, 3, 4, 5, 6));
```

Figure 3.2: Functions with and without default arguments.

1. The function f takes three arguments. It adds these arguments and returns the resulting sum. As none of the arguments has a default assigned, f can only be called with three arguments.

2. The function g also takes three arguments, but this time both the second and the third argument have a default value. Therefore, g can be called with either one argument, two arguments, or three arguments.

3. The function h takes four arguments. The first three arguments are regular arguments and each of them has a default value. The remaining argument `rest` is prefixed with a "*". Therefore, this argument is a list that collects all arguments beyond the third argument. Hence, the function

h can be called with any number of arguments. For example, if we call h as

```
h(1, 2, 3, 4, 5, 6)
```

then `a` is set to `1`, `b` is set to `2`, `c` is set to `3`, and `rest` is set to the list `[4, 5, 6]`. Later, when we discuss *closures* we will see that having an argument prefixed with the "`*`"-operator enables us to build so called *decorator functions*. This is a very powerful idea that has been borrowed from the programming language *Python*.

There is a variant syntax for defining a function which is appropriate if the definition of the function is just a single expression. For example, the function mapping x to the square $x * x$ can be defined as

```
f := x |-> x * x;
```

A definition of this form is called a *lambda definition*. The example given above is equivalent to defining `f` as follows:

```
f := procedure(x) { return x * x; };
```

We see that using a lambda definition is shorter as we do not have to use the keywords "`procedure`" and "`return`". If the function takes more than one arguments and we want to use a lambda definition, the argument list has to be enclosed in square brackets. For example, the function `hyp` that computes the length of the hypotenuse of a rectangular triangle can be defined as follows:

```
hyp := [x, y] |-> sqrt(x*x + y*y);
```

The syntax for a lambda definition is given by the follwing grammar rule:

fctDef → `VAR` "`:=`" *lambdaParams* "`|->`" *expr* "`;`"

Here, *lambdaParams* is either just a single parameter or a list of parameters, where the parameters are enclosed in square brackets and are separated by commas, while *expr* denotes an expression.

Lambda definitions are handy if we don't bother to give a name to a function. For example, the code in figure 3.3 defines a function map. This function takes two arguments: The first argument l is a list and the second argument f is a function that is to be applied to all arguments of this list. In line 4, the function map is called with a function that squares its argument. Therefore, the assignment in line 4 computes the list of the first 10 square numbers. Note that we did not had to name the function that did the squaring. Instead, we have used a lambda definition.

```
1  map := procedure(l, f) {
2      return [ f(x) : x in l ];
3  };
4  t := map([1 .. 10], x |-> x * x);
```

Figure 3.3: An example of a lambda definition in use.

Of course, it would be much easier to build the list of the first 10 squares using the following statement:

```
t := [x*x : x in [1..10]];
```

3.2.1 Calling a Procedure

There are two ways to call a procedure. If a procedure has n arguments, then one way to call f is to write

$$f(a_1, \cdots, a_n)$$

where a_1, $\cdots$, a_n are the arguments. However, there is another way to call a procedure: If l is a list

of the form

$$l = [a_1, \cdots, a_n],$$

then the expression

$$f(*l)$$

is equivalent to the expression

$$f(a_1, \cdots, a_n).$$

Later, when we discuss functional programming in Chapter 5, we will see that this can be very convenient. The reason is that this technique permits us to write second order functions that take procedures as inputs and that then modify these procedures. Even though we will not know the number of parameters that the procedures given as input take, we will still be able to modify the procedure given as argument.

3.2.2 Memoization

The function $\mathit{fib} : \mathbb{N} \to \mathbb{N}$ computing the *Fibonacci numbers* is defined recursively by the following set of recurrence equations:

$$\mathit{fib}(0) = 0, \quad \mathit{fib}(1) = 1, \quad \text{and} \quad \mathit{fib}(n+2) = \mathit{fib}(n+1) + \mathit{fib}(n).$$

These equations are readily implemented as shown in Figure 3.4. However, this implementation has a performance problem which can be easily seen when tracing the computation of $\mathit{fib}(4)$.

```
1  fibonacci := procedure(n) {
2      if (n in [0,1]) {
3          return n;
4      }
5      return fibonacci(n-1) + fibonacci(n-2);
6  };
```

Figure 3.4: A naive implementation of the Fibonacci function.

```
1  fibonacci := procedure(n) {
2      if (n in [0,1]) {
3          result := n;
4      } else {
5          result := fibonacci(n-1) + fibonacci(n-2);
6      }
7      print("fibonacci($n$) = $result$");
8      return result;
9  };
```

Figure 3.5: Tracing the computation of the Fibonacci function.

In order to trace the computation, we change the program as shown in Figure 3.5. If we evaluate the expression `fibonacci(4)`, we get the output shown in Figure 3.6. This output shows that the expression `fibonacci(2)` is evaluated twice. The reason is that the value of `fibonacci(2)` is needed in the equation

`fibonacci(4)` := `fibonacci(3)` + `fibonacci(2)`

to compute `fibonacci(4)`, but then in order to compute `fibonacci(3)`, we have to compute `fibonacci(2)` again. The trace also shows that the problem gets aggravated the longer the computation runs. For example, the value of `fibonacci(1)` has to be computed three times.

```
=> fibonacci(4);

fibonacci(1) = 1
fibonacci(0) = 0
fibonacci(2) = 1
fibonacci(1) = 1
fibonacci(3) = 2
fibonacci(1) = 1
fibonacci(0) = 0
fibonacci(2) = 1
fibonacci(4) = 3
~< Result: 3 >~

=>
```

Figure 3.6: Output of evaluating the expression `fibonacci(4)`.

In order to have a more efficient computation, it is necessary to memorize the values of the function `fibonacci` once they are computed. Fortunately, SetlX offers *cached functions*. If a function f is declared as a cached function, then every time the function f is evaluated for an argument x, the computed value $f(x)$ is memorized and stored in a table. The next time the function f is used to compute $f(x)$, the interpreter first checks whether the value of $f(x)$ has already been computed. In this case, instead of computing $f(x)$ again, the function returns the value stored in the table. This technique is known as *memoization*. Memoization is directly supported in SetlX via cached functions. Figure 3.7 shows an implementation of the Fibonacci function as a cached function. If we compare the program in Figure 3.7 with our first attempt shown in Figure 3.4, then we see that the only difference is that instead of the keyword "`procedure`" we have used the keyword "`cachedProcedure`" instead.

```
1  fibonacci := cachedProcedure(n) {
2      if (n in [0,1]) {
3          return n;
4      }
5      return fibonacci(n-1) + fibonacci(n-2);
6  };
```

Figure 3.7: A cached implementation of the Fibonacci function.

Warning: A function should only be declared as a `cachedProcedure` if it is guaranteed to always produce the same result when called with the same argument. Therefore, a function should not be declared as a `cachedProcedure` if it does one of the following things:

1. The function makes use of random numbers.
2. The function reads input either from a file or from the command line.

To further support cached procedures, SetlX provides the function `cacheStats`, which is called with a single argument that must be a cached function. For example, if we define the function `fibonacci` as shown in Figure 3.7 and evaluate the expression `fibonacci(100)`, then the expression `cacheStats(fib)` gives the following result:

```
~< Result:  ["cache hits", 98], ["cached items", 101] >~
```

This tells us that the cache contains 101 different argument/value pairs, as the cache now stores the values

`fibonacci(`n`)` for all $n \in \{0, \cdots, 100\}$.

Furthermore, we see that 98 of these 101 argument/value pairs have been used more than once in order to compute the values of `fibonacci` for different arguments.

In order to prevent memory leaks, SetlX provides the function `clearCache`. This function is invoked with one argument which must be a cached function. Writing

```
clearCache(f)
```

clears the cache for the function f, that is all argument/value pairs stored for f will be removed from the cache.

3.3 Branching Statements

Like most modern languages, SetlX supports `if-then-else` statements and `switch` statements. A generalization of `switch` statements, the so called `match` statements, are also supported. We begin our discussion with `if-then-else` statements.

3.3.1 `if-then-else` Statements

In order to support branching, SetlX supports `if-then-else` statements. The syntax is similar to the corresponding syntax in the programming language `C`. However, braces are required. For example, figure 3.8 on page 39 shows a recursive function that computes the binary representation of a natural number. Here, the function `str` is a predefined function that converts its argument into a string.

```
1  toBin := procedure(n) {
2      if (n < 2) {
3          return str(n);
4      }
5      [r, n] := [n % 2, n \ 2];
6      return toBin(n) + str(r);
7  };
```

Figure 3.8: A function to compute the binary representation of a natural number.

As in the programming languages `C` and *Java*, the `else` clause is optional.

3.3.2 `switch` Statements

Figure 3.9 shows a function that takes a list of length 3. The function sorts the resulting list. In effect, the function `sort3` implements a *decision tree*. This example shows how `if-then-else` statements can be cascaded.

```
1   sort3 := procedure(l) {
2       [ x, y, z ] := l;
3       if (x <= y) {
4           if (y <= z) {
5               return [ x, y, z ];
6           } else if (x <= z) {
7               return [ x, z, y ];
8           } else {
9               return [ z, x, y ];
10          }
11      } else if (z <= y) {
12          return [z, y, x];
13      } else if (x <= z) {
14          return [ y, x, z ];
15      } else {
16          return [ y, z, x ];
17      }
18  };
```

Figure 3.9: A function to sort a list of three elements.

Figure 3.10 on page 40 shows an equivalent program that uses a `switch` statement instead of an `if-then-else` statement to sort a list of three elements. Note that this implementation is much easier to understand than the program using `if-then-else` statements. [1]

```
1   sort3 := procedure(l) {
2       [ x, y, z ] := l;
3       switch {
4           case x <= y && y <= z: return [ x, y, z ];
5           case x <= z && z <= y: return [ x, z, y ];
6           case y <= x && x <= z: return [ y, x, z ];
7           case y <= z && z <= x: return [ y, z, x ];
8           case z <= x && x <= y: return [ z, x, y ];
9           case z <= y && y <= x: return [ z, y, x ];
10          default: print("Impossible error occurred!");
11      }
12  };
```

Figure 3.10: Sorting a list of 3 elements using a `switch` statement.

The grammar rule describing the syntax of `switch` statements is as follows:

stmnt → "`switch`" "{" *caseList* "}"

where the syntactical variable *caseList* is defined via the rule:

caseList → ("`case`" *boolExpr* "`:`" *block*)*("`default`" "`:`" *block*)?

[1] However, we should also mention that the version using `switch` is less efficient than the version using `if-then-else`. The reason is that some of the tests are redundant. This is most obvious for the last case in line 9 since at the time when control arrives in line 9 it is already known that z must be less or equal than y and that, furthermore, y must be less or equal than x, since all other cases have already been covered.

Here, *boolExpr* is a Boolean expression and *block* represents a sequence of statements.

In contrast to the programming languages C and *Java*, the `switch` statement in SetlX doesn't have a fall through. Therefore, we don't need a `break` statement in the block of statements following the Boolean expression. There are two other important distinction between the `switch` statement in *Java* and the `switch` statement in SetlX:

1. In SetlX, the keyword switch is not followed by a value.
2. The expressions following the keyword `case` have to produce Boolean values when evaluated.

There is a another type of a branching statement that is much more powerful than the `switch` statement. This branching statement is the *matching statement* and is discussed in the next section.

3.4 Matching

The most powerful branching construct is *matching*. Although the syntax for the matching statement is always the same, there are really four different variants of matching. The reason is that there four different data types that support matching. Matching has been implemented for *strings*, *lists*, *sets*, and *terms*. We discuss *string matching* first.

3.4.1 String Matching

Many algorithms that deal with a given string s have to deal with two cases: Either the string s is empty or it is nonempty and in that case has to be split into its first character c and the remaining characters r, that is we have

$$s = c + r \quad \text{where } c = s[1] \text{ and } r = s[2..].$$

In order to facilitate algorithms that perform this kind of case distinction, SetlX provides the `match` statement. Consider the function[2] `reverse` shown in Figure 3.11. This function reverses its input argument, so the expression

```
reverse("abc")
```

yields the result `"cba"`. In order to reverse a string s, the function has to deal with two cases:

```
1  reverse := procedure(s) {
2      match (s) {
3          case []   : return s;
4          case [c|r]: return reverse(r) + c;
5          default   : abort("type error in reverse($s$)");
6      }
7  };
```

Figure 3.11: A function that reverses a string.

1. The string s is empty. In this case, we can just return the string s as it is. This case is dealt with in line 3. There, we have used the pattern "`[]`" to match the empty string. Instead, we could have used the empty string itself. Using the pattern "`[]`" will prove beneficial when dealing with lists because it turns out that the function `reverse` as given in Figure 3.11 can also be used to reverse a list.

[2] There is also a predefined version of the function `reverse` which does exactly the same thing as the function in Figure 3.11. However, once we define the function `reverse` as in Figure 3.11, the predefined function gets overwritten and is no longer accessible.

2. If the string s is not empty, then it can be split up into a first character c and the remaining characters r. In this case, we reverse the string r and append the character c to the end of this string. This case is dealt with in line 4.

The `match` statement in the function `reverse` has a default case in line 5 to deal with those cases where s is neither a string nor a list. In those cases, the function is aborted with an error message.

The previous example shows that the syntax for the match statement is similar to the syntax for the switch statement. The main difference is that the keyword is now "`match`" instead of "`switch`" and that the cases no longer contain Boolean values but instead contain *patterns* that can be used

- to check whether a string has a given form and
- to extract certain components (like the first character or everything but the first character).

Basically, for strings the function `reverse` given above is interpreted as if it had been written in the way shown in Figure 3.12 on page 42. This example shows that the use of the `match` statement can make programs more compact while increasing their legibility.

```
1  reverse := procedure(s) {
2      if (s == "") {
3          return s;
4      } else if (isString(s)) {
5          c := s[1];
6          r := s[2..];
7          return reverse(r) + c;
8      } else {
9          abort("type error in reverse($s$)");
10     }
11 };
```

Figure 3.12: A function to reverse a string that does not use matching.

To explore string matching further, consider a function `reversePairs` that interchanges all pairs of characters, so for example we have

`reversePairs("abcd") = "badc"` and `reversePairs("abcde") = "badce"`.

This function can be implemented as shown in Figure 3.13 on page 43. Notice that we can match the empty string with the pattern "`[]`" in line 3. The pattern "`[c]`" matches a string consisting of a single character c. In line 5 the pattern `[a,b|r]` extracts the first two characters from the string s and binds them to the variables a and b. The rest of the string is bound to r.

String Decomposition via Assignment

Assignment can be used to decompose a string into its constituent characters. For example, if s is defined as

```
s := "abc";
```

then after the assignment

```
[u, v, w] := s;
```

the variables u, v, and w have the values

u = "a", v = "b", and w = "c".

```
1  reversePairs := procedure(s) {
2      match (s) {
3          case []     : return s;
4          case [c]    : return c;
5          case [a,b|r]: return b + a + reversePairs(r);
6
7      }
8  };
```

Figure 3.13: A function to exchange pairs of characters.

Therefore, for strings, list assignment can be seen as a lightweight alternative to matching. However, string decomposition via assignment only works if the list on the left hand side has the same length as the string.

3.4.2 List Matching

As strings can be regarded as lists of characters, the matching of lists is very similar to the matching of strings. The function `reverse` shown in Figure 3.11 on page 41 can also be used to reverse a list. If the argument s of `reverse` is a list instead of a string, we match the empty list with the pattern "`[]`", while the pattern "`[c|r]`" matches a non-empty list: The variable c matches the first element of the list, while the variable r matches the remaining elements.

List assignment is another way to decompose a list that is akin to matching. If the list l is defined via

```
l := [1..3];
```

then after the assignment

```
[x, y, z] := l;
```

the variables x, y, and y have the values $x = 1$, $y = 2$, and $z = 3$. Of course, this only works if the number of variables on the left hand side of the assignment is equal to the length of the list on the right hand side.

3.4.3 Set Matching

As sets are quite similar to lists, the matching of sets is closely related to the matching of lists. Figure 3.14 shows the function `setSort` that takes a set of numbers as its argument and returns a sorted list containing the numbers appearing in the set. In the `match` statement, we match the empty set with the pattern "`{}`", while the pattern "`{x|r}`" matches a non-empty set: The variable x matches the first element of the set, while the variable r matches the set of all the remaining elements.

```
1  setSort := procedure(s) {
2      match (s) {
3          case {}   : return [];
4          case {x|r}: return [x] + setSort(r);
5      }
6  };
```

Figure 3.14: A function to sort a set of numbers.

Of course, in SetlX sorting a set into a list is trivial, as a set is represented as an ordered binary tree and therefore is already sorted. For this reason, we could also transform a set s into a sorted list by using the expression

```
[] + s.
```

The reason is that in this case the set `s` is first transformed into a list and this list is then appended to the empty list. In general, for a list l and a set s the expression l `+` s creates a new list containing all elements of l. Then, the elements of s are appended to this list. Similarly, for a set s and a list l the expression s `+` l creates a new set containing the elements of the set s and the list l.

3.4.4 Term Matching

The most elaborate form of matching is the matching of terms. This kind of matching is similar to the kind of matching provided in the programming languages *Prolog* and ML [MTH90]. Figure 3.15 shows an implementation of the function `insert` to insert a number into an ordered binary tree. This implementation uses term matching instead of the functions "`fct`" and "`args`" that had been used in the previous implementation shown in Figure 2.8 on page 30. In line 3 of Figure 3.15, the `case` statement checks whether the term m is identical to the empty binary tree, which is represented by the term `@Nil()`. This is more straightforward than testing that the functor of m is "`Nil`", as it was done in line 3 of Figure 2.8. However, the real benefit of matching shows in line 5 of Figure 3.15 since the case statement in this line does not only check whether the functor of the term m is "`Node`" but also assigns the subterms of m to the variables `k2`, `l`, and `r`, respectively. Compare this with line 5 and line 6 of Figure 2.8 where we had to use a separate statement in line 6 to extract the arguments of the term m.

```
1  insert := procedure(m, k1) {
2      match (m) {
3          case @Nil() :
4               return @Node(k1, @Nil(), @Nil());
5          case @Node(k2, l, r) | k1 == k2:
6               return @Node(k1, l, r);
7          case @Node(k2, l, r) | compare(k1, k2) < 0:
8               return @Node(k2, insert(l, k1), r);
9          case @Node(k2, l, r):
10              return @Node(k2, l, insert(r, k1));
11         default: abort("Error in insert($m$, $k1$)");
12     }
13 };
```

Figure 3.15: Inserting an element into a binary tree using matching.

In general, a `case` statement that is part of a `match` statement has the form

`case` *pattern* `|` *condition*`:` *stmnt*

Here, *pattern* is a term, *condition* is a Boolean condition, and *stmnt* is a statement. When this `case` statement is processed, the term t in the statement

`match` (t) $\{ \cdots \}$

is *matched* against *pattern*. This will be explained in more detail in the next paragraph. If this *matching* succeeds, the variables in *pattern* are bound to the corresponding subterms in t. Next, it is checked whether *condition* is true. If it is, the *match* succeeds and the statement *stmnt* is executed. Otherwise,

the statement is skipped and control proceeds to the next case.

Let us now explain the details how a term t is matched against a pattern p. The pattern p is either a variable or it is a term. If p is a variable, the match succeeds and the term t is assigned to the variable p, i.e. the statement

$$p \texttt{ := } t\texttt{;}$$

is executed. If p is a term, then t and p can be written as

$$t = f(t_1, \cdots, t_n) \quad \text{and} \quad p = g(p_1, \cdots, p_m).$$

Then, matching t against p proceeds as follows:

1. If the function symbols f and g are different, the match fails.
2. If the numbers n and m are different, the match fails.
3. If the match hasn't failed yet, the subterms t_1, $\cdots$, t_n are matched recursively against the patterns p_1, $\cdots$, p_n. The match only succeeds if all of the recursive matches succeed. If it does not succeed, then the assignments that have been made during the recursive matching are undone.

The following more complex example will serve to elucidate matching further. The function `diff` shown in Figure 3.16 on page 46 is supposed to be called with two arguments:

1. The first argument t is a term that is interpreted as an arithmetic expression.
2. The second argument x is a term that is interpreted as the name of a variable.

The function `diff` interprets the term t as a mathematical function and takes the *derivative* of this function with respect to the variable x. For example, in order to compute the derivative of the function

$$x \mapsto x^x$$

we can invoke `diff` as follows:

```
diff(parseTerm("x ** x"), parseTerm("x"));
```

Here the function `parseTerm` transforms the string "`x ** x`" into a term. The form of this term will be discussed in more detail later. This function is a part of the package `termUtilities`. Therefore, to use this function we have to load the corresponding library first using the statement

```
loadLibrary("termUtilities");
```

For the moment, let us focus on the `match` statement in Figure 3.16. Consider line 5: If the term that is to be differentiated has the form `t1 + t2`, then both `t1` and `t2` have to be differentiated separately and the resulting terms have to be added. For a more interesting example, consider line 9. This line implements the product rule:

$$\frac{\mathrm{d}}{\mathrm{d}x}\big(t_1 \cdot t_2\big) = \frac{\mathrm{d}\,t_1}{\mathrm{d}x} \cdot t_2 + t_1 \cdot \frac{\mathrm{d}\,t_2}{\mathrm{d}x}.$$

Note how the pattern

```
t1 * t2
```

in line 9 extracts the two factors t_1 and t_2 from a term that happens to be a product. Further, note that in line 18, 19, 21, and 22 we had to prefix the function symbols "`exp`" and "`ln`" with the character "`@`" in order to convert these function symbols into functors.

Note that the example makes extensive use of the fact that terms are *viral* when used with the arithmetic operators "`+`", "`-`", "`*`", "`/`", "`\`", and "`%`": If one operand of these operators is a term, the operator automatically yields a term. For example, if `x` is a term, then

```
1  loadLibrary("termUtilities");
2
3  diff := procedure(t, x) {
4      match (t) {
5          case t1 + t2:
6               return diff(t1, x) + diff(t2, x);
7          case t1 - t2:
8               return diff(t1, x) - diff(t2, x);
9          case t1 * t2:
10              return diff(t1, x) * t2 + t1 * diff(t2, x);
11         case t1 / t2:
12              return ( diff(t1, x) * t2 - t1 * diff(t2, x) ) / (t2 * t2);
13         case f ** 0:
14              return 0;
15         case f ** n | isNumber(n):
16              return n * diff(f, x) * f ** (n-1);
17         case f ** g:
18              return diff(@exp(g * @ln(f)), x);
19         case @ln(a):
20              return diff(a, x) / a;
21         case @exp(a):
22              return diff(a, x) * @exp(a);
23         case var | var == x :
24              return 1;
25         case var | isVariable(var) :
26              return 0;
27         case t | isNumber(t):
28              return 0;
29     }
30 };
```

Figure 3.16: A function to perform symbolic differentiation.

```
x + 2
```

is also a term. Note also that terms are not viral inside function symbols like "exp". Therefore, this function symbol has to be prefixed by the operator "@" to turn it into a functor.

Line 27 shows how a condition can be attached to a pattern: The pattern

```
case t:
```

would match anything. However, we want to match only numbers here. Therefore, we have attached the condition `isNumber(t)` via the condition operator "|" to this pattern and hence have written

```
case t | isNumber(t):
```

in order to ensure that `t` is indeed a number.

3.4.5 Term Decomposition via List Assignment

As a syntactical convenience, terms can be decomposed via list assignment. For example, after the assignment

```
[x,y,z] := @f(1, @g(2), {2,3});
```

the variables x, y, and z have the values

$$x = 1, \quad y = @g(2), \quad \text{and} \quad z = \{2, 3\}.$$

Of course, the function `args` achieves a similar effect. We have that

```
args(@f(1, @g(2), {2,3})) = [1, @g(2), {2, 3}].
```

3.5 Loops

SetlX offers three different kinds of loops: `for` loops, `while` loops, and `do-while` loops. The `while` loops are the most general loops. Therefore, we discuss them first.

3.5.1 `while` Loops

The syntax of `while` loops in SetlX is similar to the syntax of `while` loops in the programming language `C`. The only difference is that in SetlX, the body of a `while` loop has to be enclosed in curly braces even if it only consists of a single line. To demonstrate a `while` loop, let us implement a function testing the *Collatz conjecture*: Define the function

$$f : \mathbb{N} \to \mathbb{N}$$

recursively as follows:

1. $f(n) := 1$ if $n = 1$,
2. $f(n) := \begin{cases} f(n/2) & \text{if } n \,\%\, 2 = 0; \\ f(3 \cdot n + 1) & \text{otherwise.} \end{cases}$

The Collatz conjecture claims that $f(n) = 1$ for all $n \in \mathbb{N}$. If we assume the Collatz conjecture is true, then f is well-defined. Otherwise, if the Collatz conjecture is not true, for certain values of n the function $f(n)$ is undefined. Figure 3.17 shows an implementation of the function f in SetlX.

```
1  f := procedure(n) {
2      if (n == 1) {
3          return 1;
4      }
5      while (n != 1) {
6          if (n % 2 == 0) {
7              n /= 2;
8          } else {
9              n := 3 * n + 1;
10         }
11     }
12     return n;
13 };
```

Figure 3.17: A program to test the Collatz conjecture.

The function f is implemented via a `while` loop. This loop runs as long as n is different from the number one. Therefore, if the Collatz conjecture is true, the `while` loop will eventually terminate for all values of n.

The syntax of a `while` loop is given by the following grammar rule:

statement → "`while`" "(" *boolExpr* ")" "{" *block* "}" .

Here, *boolExpr* is a Boolean expression returning either `true` or `false`. This condition is called the *guard* of the `while` loop. The syntactical variable *block* denotes a sequence of statements. Note that, in contrast to the programming languages `C` and *Java*, the block of statements always has to be enclosed in curly braces, even if it consists only of a single statement. The semantics of a `while` loop is the same as in `C`: The loop is executed as long as the guard is true. In order to abort an iteration prematurely, SetlX provides the command `continue`. This command aborts the current iteration of the loop and proceeds with the next iteration. In order to abort the loop itself, the command `break` can be used. Figure 3.18 shows a function that uses both a `break` statement and a `continue` statement. This function will print the number 1. Then, when n is incremented to 2, the `continue` statement in line 6 is executed so that the number 2 is not printed. In the next iteration of the loop, the number n is incremented to 3 and printed. In the final iteration of the loop, n is incremented to 4 and the `break` statement in line 9 terminates the loop.

```
1  testBreakAndContinue := procedure() {
2      n := 0;
3      while (n < 10) {
4          n := n + 1;
5          if (n == 2) { continue; }
6          if (n == 4) { break;    }
7          print(n);
8      }
9  };
```

Figure 3.18: This function demonstrates the semantics of `break` and `continue`.

3.5.2 `do-while` Loops

Similar to the language `C`, SetlX supports the `do-while` loop. The difference between a `do-while` loop and an ordinary `while` loop is that sometimes the body of a loop needs to execute at least once, regardless of the condition controlling the loop. For example, imagine you want to implement the following guessing game: The computer generates a random natural number between 0 and 100 inclusive and the player has to guess it. Every time the player enters some number, the computer informs the player whether the number was too big, too small, or whether the player has correctly guessed the secret number. In this guessing game, the player always has to enter at least one number. Therefore, the most natural way to implement this game is to use a `do-while` loop. Figure 3.19 shows an implementation of the guessing game. The implementation works as follows:

1. In line 2, the secret number is generated using the function `rnd` which picks a random element from the set given as argument. This number has to be guessed by the player.

2. The variable `count` is used to count the number of guessing attempts. This variable is initialized in line 3.

3. Since the user has to enter at least one number, we use a `do-while` loop that loops as long as the user has not yet guessed the secret number.

   In line 6, the user is asked to guess the secret number. If this number is either too small or too big, an appropriate message is printed. The loop terminates in line 14 if the number that has been guessed is identical to the secret number.

```
1  guessNumber := procedure() {
2      secret := rnd(100);
3      count  := 0;
4      do {
5          count += 1;
6          x := read("input a number between 0 and 100 inclusively: ");
7          if (x < secret) {
8              print("sorry, too small");
9          } else if (x > secret) {
10             print("sorry, too big");
11         } else {
12             print("correct!");
13         }
14     } while (x != secret);
15     print("number of guesses: $count$");
16 };
```

Figure 3.19: Implementing the guessing game in SetlX.

The syntax of a `do-while` loop is given by the following grammar rule:

statement → "`do`" "{" *block* "}" "`while`" "(" *boolExpr* ")" ";" .

Here, *boolExpr* is a Boolean expression returning either `true` or `false`. This expression is called the *guard* of the `do-while` loop. The syntactical variable *block* denotes a sequence of statements. Note that, in contrast to the programming languages `C` and *Java*, the block of statements always has to be enclosed in curly braces, even if it consists only of a single statement. Furthermore, note that a `do-while` loop has to be terminated with a semicolon. This is a notable difference to `while` loops and `for` loops, which are not terminated by a semicolon.

The semantics of a `do-while` loop is the same as in `C`: The body of the loop is executed once. Then, if the guard is false, execution terminates. Otherwise, the body is executed again and again as long as the guard is true. In order to abort an iteration of the loop prematurely, the commands `continue` and `break` can be used. These commands work in the same way as in a `while` loop.

3.5.3 `for` Loops

In order to either perform a group of commands a predefined number of times, or to iterate over a set, list or string, a `for` loop should be used. Figure 3.20 on page 50 shows some SetlX code that prints a multiplication table. The output of this program is shown in Figure 3.21 on page 50.

In the program in Figure 3.20, the printing is done in the two nested `for` loops that start in line 8 and line 9, respectively. In line 8, the counting variable `i` iterates over all values from 1 to 10. Similarly, the counting variable `j` in line 9 iterates over the same values. The product `i * j` is computed in line 10 and printed without a newline using the function `nPrint`. The function `rightAdjust`(n) turns the number n into a string by padding the number with blanks from the left so that the resulting string always has a length of 4 characters.

The general syntax of a `for` loop is given by the following Ebnf rule:

statement → "`for`" "(" *iterator* ("`,`" *iterator*)* ("|" *condition*)? ")" "{" *block* "}".

Here an iterator is either a *simple iterator* or a *tuple iterator*. A *simple iterator* has the form

x "`in`" s

```
1  rightAdjust := procedure(n) {
2      switch {
3          case n < 10 : return "   " + n;
4          case n < 100: return  "  " + n;
5          default:      return   " " + n;
6      }
7  };
8  for (i in [1 .. 10]) {
9      for (j in [1 .. 10]) {
10         nPrint(rightAdjust(i * j));
11     }
12     print();
13 }
```

Figure 3.20: A simple program to generate a multiplication table.

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Figure 3.21: Output of the program in Figure 3.20.

where s is either a set, a list, or a string and x is the name of a variable. This variable is bound to the elements of s in turn. For example, the statement

```
for (x in [1..10]) { print(x); }
```

will print the numbers from 1 to 10. If s is a string, then the variable x iterates over the characters of s. For example, the statement

```
for (c in "abc") { print(c); }
```

prints the characters `"a"`, `"b"`, and `"c"`.

The iterator of a `for` loop can also be a *tuple iterator*. The simplest form of a tuple is

$[x_1, \cdots, x_n]$ "`in`" s.

Here, s must be either a set or a list whose elements are lists of length n. Figure 3.22 on page 51 shows a procedure that computes the relational product of two binary relations r_1 and r_2. In set theory, the relational product $r_1 \circ r_2$ is defined as

$$r_1 \circ r_2 := \{\langle x, z\rangle \mid \langle x, y\rangle \in r_1 \wedge \langle y, z\rangle \in r_2\}.$$

The `for` loop in line 3 iterates over the two relations `r1` and `r2`. The condition "`y1 == y2`" selects those pairs of numbers such that the second component of the first pair is identical to the first component of the second pair.

```
1  product := procedure(r1, r2) {
2      r := {};
3      for ([x, y1] in r1, [y2, z] in r2 | y1 == y2) {
4          r += { [x, z] };
5      }
6      return r;
7  };
```

Figure 3.22: A program to compute the relational product of two binary relations.

The `for` loop in line 3 can be simplified as shown in Figure 3.23 on page 51. Here, we have forced the second component of the pair `[x,y]` to be equal to the first component of the pair `[y,z]` by using the same variable name for both components.

```
1  product := procedure(r1, r2) {
2      r := {};
3      for ([x, y] in r1, [y, z] in r2) {
4          r += { [x, z] };
5      }
6      return r;
7  };
```

Figure 3.23: A program to compute the relational product of two binary relations.

Of course, in SetlX the relational product can be computed more easily via set comprehension. Figure 3.24 on page 51 shows an implementation of the function `product` that is based on set comprehension. In general, most occurrences of `for` loops can be replaced by equivalent set comprehensions. Our experience shows that the resulting code is often both shorter as well as easier to understand.

```
1  product := procedure(r1, r2) {
2      return { [x, z] : [x, y] in r1, [y, z] in r2 };
3  };
```

Figure 3.24: Computing the relational product via set comprehension.

Iterators can be even more complex, since the tuples can be nested, so a `for` loop of the form

```
for ([[x,y],z] in s) { ··· }
```

is possible. However, as this feature is rarely needed, we won't discuss it in more detail.

A `for` loop creates a *local scope* for its iteration variables. This means that changes to an iteration variable that occur in the `for` loop are not visible outside of the `for` loop. Therefore, the last line of the program shown in Figure 3.25 prints the message

```
x = 1.
```

```
1  x := 1;
2  for (x in "abc") {
3      print(x);
4  }
5  print("x = $x$");
```

Figure 3.25: A program illustrating the scope of a `for` loop.

Chapter 4

Regular Expressions

Regular expressions are a very powerful tool when processing strings. Therefore, most modern programming languages support them. SetlX is no exception. As the SetlX interpreter is implemented in *Java*, the syntax of the regular expressions supported by SetlX is the same as the syntax of regular expressions in *Java*. For this reason, instead of discussing every detail of the syntax of regular expressions, we refer the reader to the documentation of the *Java* class `java.util.regex.Pattern`, which can be found at

http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html.

This documentation contains a concise description of the syntax and semantics of regular expressions. For a more general discussion of the use of regular expression, the book of Friedl [Fri06] is an excellent choice. Here, we will confine ourselves to show how regular expressions can be used in SetlX programs. SetlX provides two control structures that make use of regular expressions. The first is the `match` statement and the second is the `scan` statement.

4.1 Using Regular Expressions in a `match` Statement

Instead of the keyword "`case`", a branch in a `match` statement can begin with the keyword "`regex`". Figure 4.1 shows the definition of a function named `classify` that takes a string as its argument and tries to classify this string as either a word or a number.

```
1  classify := procedure(s) {
2      match (s) {
3          regex '0|[1-9][0-9]*': print("found an integer");
4          regex '[a-zA-Z]+'    : print("found a word");
5          regex '\s+'          : // skip white space
6          default              : print("unkown: $s$");
7      }
8  };
```

Figure 4.1: A simple function to recognize numbers and words.

Note that we have specified the regular expressions using literal strings, i.e. strings enclosed in single quote characters. This is necessary, since the regular expression in line 5

```
'\s+'
```

contains a backslash character. If we had used double quotes, it would have been necessary to escape

the backslash character with another backslash and we would have had to write

```
"\\s+"
```

instead. Invoking the function `classify` as

```
classify("123");
```

prints the message "`found an integer`", while invoking the function as

```
classify("Hugo");
```

prints the message '`found a word`". Finally, calling

```
classify("0123");
```

prints the answer "`unkown: 0123`". The reason is that the string "0123" does not comply with the regular expression `'0|[1-9][0-9]*'` because this regular expression does not allow a number to start with the character "0" followed by more digits.

4.1.1 Extracting Substrings

Often, strings are structured and the task is to extract substrings corresponding to certain parts of a string. This can be done using regular expressions. For example, consider a phone number in the format

```
+49-711-6673-4504.
```

Here, the substring "49" is the country code, the substring "711" is the area code, the substring "6673" is the company code, and finally "4504" is the extension. The regular expression

```
\+([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)
```

specifies this format and the different blocks of parentheses correspond to the different codes. If a phone number is given and the task is to extract, say, the country code and the area code, then this can be achieved with the SetlX function shown in Figure 4.2.

```
1  extractCountryArea := procedure(phone) {
2      match (phone) {
3          regex '\+([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)' as [_, c, a, _, _]:
4              return [c, a];
5          default: abort("The string $phone$ is not a phone number!");
6      }
7  };
```

Figure 4.2: A function to extract the country and area code of a phone number.

Here, the regular expression to recognize phone numbers has several parts that are enclosed in parentheses. These parts are collected in a list of the form

$$[s, p_1, \cdots, p_n].$$

The first element s of this list is the string that matched the regular expression. The remaining elements p_i correspond to the different parts of the regular expression: p_1 corresponds to the first group of parentheses, p_2 corresponds to the second group, and in general p_i corresponds to the i-th group. In line 3 this list is then matched against the pattern

```
[_, c, a, _, _].
```

Therefore, if the match is successful, the variable c will contain the country code and the variable a is set to the area code. The groups of regular expressions that are not needed are matched against the anonymous variable "_" and are therefore effectively ignored.

If a regular expression contains nested groups of parentheses, then the order of the groups is determined by the left parenthesis of the group. For example, the regular expression

```
'((a+)(b*)c?)d'
```

contains three groups:

1. the first group is '((a+)(b*)c?)',
2. the second group is '(a+)', and
3. the third group is '(b*)'.

4.1.2 Testing Regular Expressions

In real life applications, regular expressions can get quite involved and difficult to comprehend. The function testRegexp shown in Figure 4.3 can be used to test a given regular expression: The function takes a regular expression re as its first argument, while the second argument is a string s. The function tests whether the string s matches the regular expression re. If this is the case, the function testRegexp returns a list that contains all the substrings corresponding to the different parenthesized groups of the regular expression re. For example, invoking this function as

```
testRegexp('(a*)(a+)b', "aaab");
```

yields the result

```
["aaab", "aa", "a"].
```

Here, the first element of the list is the string that was matched by the regular expression, the second element "aa" corresponds to the regular subexpression '(a*)', and the last element "a" corresponds to the regular subexpression '(a+)'.

```
1  testRegexp := procedure(re, s) {
2      match (s) {
3          regex re as l: return l;
4          default : print("No match!");
5      }
6  };
```

Figure 4.3: Testing regular expressions.

4.1.3 Extracting Comments from a File

In this section we present an example of the match statement in action. The function printComments shown in Figure 4.4 attempts to extract all those C-style comments from a file that are contained in a single line. The regular expression '\s*(//.*)' in line 5 matches comments starting with the string "//" preceded by optional white space, while the regular expression

```
'\s*(/\*([^*]|\*+[^*/])*\*+/)\s*'
```

in line 6 matches comments that start with the string "/*", preceded by optional white space and end with the string "*/" followed by optional trailing white space. This regular expression is quite difficult to read for three reasons:

1. We have to precede the symbol "*" with a backslash in order to prevent it from being interpreted as a regexp quantifier.

2. We have to ensure that the string between "/*" and "*/" does not contain the substring "*/". The regular expression

   ```
   ([^*]|\*+[^*/])*
   ```

   specifies this substring: This substring can have an arbitrary number of parts that satisfy the following specification:

   (a) A part may consists of any character different from the character "*". This is specified by the regular expression `'[^*]'`.

   (b) A part may be a sequence of "*" characters that is neither followed by the character "/" nor the character "*". These parts are matched by the regular expression `\*+[^*/]`.

   Concatenating any number of these parts will never produce a string containing the substring "*/".

3. Lastly, we have to ensure that we can match a string of the form "***···***/" that ends the comment. The problem here is that we need not only be able to recognize the string "*/" but also have to deal with the case that this string is preceded by an arbitrary number of "*"-characters, since the regular expression `([^*]|\*+[^*/])*` does not accept any trailing "*"-characters.

```
1  printComments := procedure(file) {
2      lines := readFile(file);
3      for (l in lines) {
4          match (l) {
5              regex '\s*(//.*)'                        as c: print(c[2]);
6              regex '\s*(/\*([^*]|\*+[^*/])*\*+/)\s*' as c: print(c[2]);
7          }
8      }
9  };
10
11 for (file in params) {
12     printComments(file);
13 }
```

Figure 4.4: Extracting comments from a given file.

If the code shown in Figure 4.4 is stored in the file `find-comments.stlx`, then we can invoke this program as

```
setlx find-comments.stlx --params file.stlx
```

The option "`--params`" creates the global variable `params` that contains a list containing all the remaining arguments given to the program. In this case, there is just one argument, which is the string "`file.stlx`". Therefore, `params` is a list of length one containing the file name as a string. Hence the `for` loop in line 11 will call the function `printComments` with the given string. If we invoke the program using the command

```
setlx find-comments.stlx --params *.stlx
```

instead, then the function `printComments` will be called for all files ending in "`.stlx`".

4.1.4 Conditions in `match` Statements

A clause of a `match` statement can contain an optional Boolean condition that is separated from the regular expression by the condition operator "|". Figure 4.5 shows a program to search for *palindromes* in a given file.

1. Line 2 reads the file using the function `readFile`. This function returns a list of strings where each string corresponds to a line of the file.

   These lines are then concatenated by the function `join` into a single string. In this string, the different lines are separated by the newline character "\n".

   Finally the resulting string is split into a list of words using the function `split`. The second argument of this function is a regular expression that specifies that the string should be split at every non-empty sequence of characters that are not letters.

2. The `for`-loop in line 4 iterates over all these words and the `match`-statement in line 5 tests every word.

3. Line 6 shows how a condition can be attached to a `regex` clause. The regular expression

   ```
   '[a-zA-Z]+'
   ```

   matches any number of letters, hence it recognizes words. However, the string `c[1]` corresponding to the match is only added to the set of palindromes if the predicate `isPalindrome[c[1])` yields `true`. The function `isPalindrome`(s) checks whether s is a palindrome that has a length of at least 3.

```
1  findPalindrome := procedure(file) {
2      all := split(join(readFile(file), "\n"), '[^a-zA-Z]+');
3      palindromes := {};
4      for (s in all) {
5          match (s) {
6              regex '[a-zA-Z]+' as c | isPalindrome(c[1]):
7                    palindromes += { c[1] };
8              regex '.|\n':
9                    // skip everything else
10         }
11     }
12     return palindromes;
13 };
14
15 isPalindrome := procedure(s) {
16     if (#s < 3) {
17         return false;
18     }
19     return reverse(s) == s;
20 };
```

Figure 4.5: A program to find palindromes in a file.

```
1  printComments := procedure(file) {
2      s := join(readFile(file), "\n");
3      scan (s) {
4          regex '//[^\n]*'                as c: print(c[1]);
5          regex '/\*([^*]|\*+[^*/])*\*+/' as c: print(c[1]);
6          regex '.|\n'                        : // skip every thing else
7      }
8  };
```

Figure 4.6: Extracting comments using the match statement.

4.2 The `scan` Statement

The program to extract comments that was presented in the previous subsection is quite unsatisfactory as it will only recognize those strings that span a single line. Figure 4.6 on page 58 shows a function that instead extracts all comments from a given file. The function `printComments` takes a string as argument. This string is interpreted as the name of a file. In line 2, the function `readFile` reads this file. This function produces a list of strings. Each string corresponds to a single line of the file without the trailing line break. The function `join` joins all these lines into a single string. As the second argument of `join` is the string "\n", a newline is put in between the lines that are joined. The end result is that the variable `s` contains the content of the file as one string. This string is then scanned using the `scan` statement in line 3. The general form of a scan statement is as follows:

```
scan (s) {
   regex r_1 as l : b_1
   ⋮
   regex r_n as l : b_n
}
```

Here, s is a string to be analyzed, r_1, $\cdots$, r_n denote regular expressions, while b_1, $\cdots$, b_n are lists of statements. The `scan` statement works as follows:

1. All the regular expressions r_1, $\cdots$, r_n are tried in parallel to match a prefix of the string s.
   (a) If none of these regular expression matches, the scan statement is aborted with an error message.
   (b) If exactly one regular expression r_i matches, then the corresponding statements b_i is executed and the prefix matched by r_i is removed from the beginning of s.
   (c) If more than one regular expression matches, then there is a conflict which is resolved in two steps:
      i. If the prefix matched by some regular expression r_j is longer than any other prefix matched by another regular expression r_i, then the regular expression r_j wins, the list of statements b_j is executed and the prefix matched by r_j is removed from s.
      ii. If there are two (or more) regular expressions r_i and r_j that both match a prefix of maximal length, then the regular expression with the lowest index wins, i.e. if $i < j$, then b_i is executed.

   Finally, the prefix of s that has been matched is removed from s.
2. Step 1 is repeated as long as the string s is not empty. Therefore, a `scan` statement can be viewed as a `while` loop combined with a `match` statement.

The clauses in a scan statement can also have Boolean conditions attached. This works the same way as it works for a match statement.

The scan statement provides a functionality that is similar to the functionality provided by tools like lex [Les75] or *JFlex* [Kle09]. In order to support this claim, we present an example program that computes the marks of an exam. Assume the results of an exam are collected in a text file like the one shown in Figure 4.7 on page 59. The first line of this file shows that this is an exam about algorithms. The third line tells us that there are 6 exercises in the given exam and the remaining lines list the points that have been achieved by individual students. A hyphen signals that the student did not work on the corresponding exercise. In order to calculate marks, we first have to sum up all the points. From this sum, the mark can then be calculated.

```
1  Exam:  Algorithms
2
3  Exercises:          1. 2. 3. 4. 5. 6.
4  Magnus Peacemaker:   8  9  8  -  7  6
5  Dewy Dullamore:      4  4  2  0  -  -
6  Alice Airy:          9 12 12  9  9  6
7  Jacob James:         9 12 12  -  9  6
```

Figure 4.7: Typical results from an exam.

Figure 4.8 shows a program that performs this calculation. We discuss this program line by line.

1. The function evalExam takes two arguments: The first is the name of a file containing the results of the exam and the second argument is the number of points needed to get the top mark. This number is a parameter that is needed to calculate the mark of a student.

2. Line 2 creates a string containing the content of the given file.

3. We will use two states in our scanner:

   (a) The first state is identified by the string "normal". In line 3, the variable state is initialized to this value.

   (b) The second state is identified by the string "printBlanks". We enter this state when we have read the name of a student. This state is needed to read the white space between the name of a student and the first number following the student.

   In state "normal", all white space is discarded, but in state "printBlanks" white space is printed. This is necessary to format the output neatly.

   Line 3 therefore initializes the variable state to "normal".

4. Line 4 shows the scan statement. Here, we have added the string "using map" after "scan(all)". This enables us to retrieve line and column information via the variable map. How this is done is shown in line 24 and 25.

5. The regular expression in line 5 is needed to skip the header of the file. The two header lines can be recognized by the fact that there is a single word followed by a colon ":". In contrast, the names of students always consist of two words separated by a single white space character.

6. The regular expression in line 5 matches the name of a student. Notice that we have used "\s" to specify the blank between the first name and the family name. Furthermore, the family name is allowed to contain the hyphen character "-". The student's name is printed and the number of points for this student is set to 0. Furthermore, after the name of a student has been seen, the state is changed to the state "printBlanks".

```
1  evalExam := procedure(file, maxPoints) {
2      all   := join(readFile(file), "\n");
3      state := "normal";
4      scan (all) using map {
5          regex '[a-zA-Z]+:.*\n': // skip header lines
6          regex '[A-Za-z]+\s[A-Za-z\-]+:' as [ name ]:
7                nPrint(name);
8                state     := "printBlanks";
9                sumPoints := 0;
10         regex '[ \t]+' as [ whiteSpace ] | state == "printBlanks":
11               nPrint(whiteSpace);
12               state := "normal";
13         regex '[ \t]+' | state == "normal":
14               // skip white space between results
15         regex '0|[1-9][0-9]*' as [ number ]:
16               sumPoints += int(number);     // add results
17         regex '-': // skip exercises that have not been done
18         regex '\n' | sumPoints != om:       // calculate mark
19               print(mark(sumPoints, maxPoints));
20               sumPoints := om;
21         regex '[ \t]*\n' | sumPoints == om: // skip empty lines
22         regex '.|\n' as [ c ]:              // error handling
23               print("unrecognized character: $c$");
24               print("line:   ", map["line"]);
25               print("column: ", map["column"]);
26     }
27 };
28 mark := procedure(p, m) {
29     return 7.0 - 6.0 * p / m;
30 };
```

Figure 4.8: A program to compute marks for an exam.

7. The regular expression in line 10 matches the white space following the name of a student. Notice that this regular expression has a condition attached to it: The condition is the formula

   `state == "printBlanks"`.

   The white space matched by the regular expression is then printed and the state is switched back to `"normal"`.

   In effect, this rule will guarantee that the output is formatted in the same way as the input, as the white space following a student's name is just copied verbatim to the output.

8. Line 13 skips over white space that is encountered in state `"normal"`. This is the white space separating different numbers in the list of numbers following a students name.

9. The clause in line 15 recognizes strings that can be interpreted as numbers. These strings are converted into numbers with the help of the conversion function `int` and then this number is added to the number of points achieved by this student.

10. Line 17 skips over hyphens as these correspond to those exercises that have not been attempted by the particular student.

11. Line 18 checks whether the end of a line listing the points of a particular student has been reached. This is the case if we encounter a newline and the variable `sumPoints` is not undefined. In this case, the mark for this student is computed and printed. Furthermore, the variable `sumPoints` is set back to the undefined status.
12. Any empty lines are skipped in line 21.
13. Finally, if we encounter any remaining character, then there is a syntax error in our input file. In this case, line 22 recognizes this character and produces an error message. This error message specifies the line and column of the character. This is done with the help of the variable `map` that has been declared in line 4 via the `using` directive. `map` is effectively a dictionary that supports the keys `char`, `line`, and `column`.
    (a) `char` counts the characters encountered, hence this variable is incremented for every character that is read. This variable has the value 1 for the first character.
    (b) `line` counts the line numbers. The line number is incremented once a newline character is read. This variable has the value 1 for the first line.
    (c) `column` counts the characters after the last line break, i.e. this variable is incremented for every character that is read and it resets to 1 after a newline character has been read.

4.3 Additional Functions Using Regular Expressions

There are four predefined functions that use regular expressions.

1. The function

    `matches`(s, r)

    takes a string s and a regular expression r as its arguments. It returns `true` if the string s is in the language described by the regular expression r. For example, the expression

    ```
    matches("42", '0|[1-9][0-9]*');
    ```

    returns `true` as the string `"42"` can be interpreted as a number and the regular expression `'0|[1-9][0-9]*'` describes natural numbers in the decimal system.

    There is a variant of `matches` that takes three arguments. It is called as

    `matches`$(s, r,$ `true`$)$.

    In this case, r should be a regular expression containing several *groups*, i.e. there should be several subexpressions in r that are enclosed in parentheses. Then, if r matches s, the function `matches` returns a list of substrings of s. The first element of this list is the string s, the remaining elements are the substrings corresponding to the different groups of r. For example, the expression

    ```
    matches("+49-711-6673-4504", '\+([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)', true)
    ```

    returns the list

    ```
    ["+49-711-6673-4504", "49", "711", "6673", "4504"].
    ```

    If `matches` is called with three arguments where the last argument is `true`, an unsuccessful match returns the empty list.

2. The function

    `replace(`s`,` r`,` t`)`

    receives three arguments: the arguments s and t are strings, while r is a regular expression. The

function looks for substrings in s that match r. These substrings are then replaced by t. For example, the expression

```
replace("+49-711-6673-4504", '[0-9]{4}', "XXXX");
```

returns the string

```
"+49-711-XXXX-XXXX".
```

3. There is a variant to the function `replace(`s`, `r`, `t`)` that replaces only the first substring in s that matches r. This variant is called `replaceFirst` and is called as

   `replaceFirst(`s`, `r`, `t`)`.

   For example, the expression

   ```
   replaceFirst("+49-711-6673-4504", '[0-9]{4}', "XXXX");
   ```

   returns the string

   ```
   "+49-711-XXXX-4504".
   ```

4. The function

   `split(`s`, `r`)`

   takes a string s and a regular expression r as its inputs. It returns a list of substrings of s. To be more precise, s is split into substrings at those positions that match the regular expression r. The parts of s that are matched by r are not returned. A common way to invoke the function is given by the following example:

   ```
   l := split("1; 2; 3", ';\s*');
   ```

   Here, the string `"1, 2, 3"` is split at every position where there the character ";" is followed by white space. As a result, we have

   ```
   l = ["1", "2", "3"].
   ```

   If the previous example is changed into

   ```
   l := split("1; 2; 3;", ';\s*');
   ```

   i.e. the string is terminated with a ";", then the result changes to

   ```
   l = ["1", "2", "3", ""].
   ```

   Note that the function

   `join(`l`, `s`)`

   is an inverse of the function `split`. It takes a list of strings l and a string s and joins the items of l into a single result string by separating them with the string s. For example,

   ```
   join(["a", "b", "c"], ";")
   ```

   yields the string "`a;b;c`".

Chapter 5

Functional Programming and Closures

We have already mentioned that SetlX is a full-fledged functional language: Functions can be used both as arguments to other functions and as return values. There is no fundamental difference between the type of a function and, say, the type of a rational number, as both can be assigned to variables, converted to strings, used as arguments of other functions, or returned from a function.

Additionaly, SetlX supports *closures* in a way similar to languages like *Scheme* [SS75]. In order to introduce closures, we will first present some simple examples. After that, we show a more complex case study: We discuss a program that transforms a regular expression into a non-deterministic finite state machine.

5.1 Introductory Examples

In this section we will first introduce the basic ideas of functional programming. After that, we discuss the notion of a closure.

Using functions as arguments to other functions is something that is not often seen in conventional programming languages. Figure 5.1 on page 64 shows the implementation of the function `reduce` that takes two arguments:

1. The first argument is a list l.
2. The second argument is a binary function f. This function is expected to take two arguments.

The function `reduce` combines successive arguments of the list l using the function f and thereby reduces the list l into a single value. For example, if the function f is the function `add` that just adds its arguments, then `reduce`$(l, \texttt{add})$ computes the sum of the elements of l. Therefore, in line 12 the invocation of `reduce(l, add)` computes the sum of all elements of the list l. Since this list contains the natural number from 1 up to 36, reduce effectively computes the sum

$$\sum_{i=1}^{36} i.$$

Note that the expression "`+/ l`" is another way to compute this sum. If instead the second arguments of the function `reduce` is the function `multiply` that returns the product of its arguments, then `reduce`$(l, \texttt{multiply})$ computes the product of all elements of the list l and hence is equivalent to the expression "`*/ l`". Finally, it should be noted that in the case that the list l is empty, `reduce` returns the value `om`.

Notice that the function `reduce` is a so called higher order function because one of its arguments is itself a function. While the example given above might seem artificial, there are common applications

```
reduce := procedure(l, f) {
    match (l) {
        case []     : return;
        case [x]    : return x;
        case [x,y|r]: return reduce([f(x,y) | r], f);
    }
};
add      := [x, y] |-> x + y;
multiply := [x, y] |-> x * y;

l := [1 .. 36];
x := reduce(l, add     );
y := reduce(l, multiply);
```

Figure 5.1: Implementing a second order function.

```
sort := procedure(l, cmp) {
    if (#l < 2) { return l; }
    m := #l \ 2;
    [l1, l2] := [l[.. m], l[m+1 ..]];
    [s1, s2] := [sort(l1, cmp), sort(l2, cmp)];
    return merge(s1, s2, cmp);
};
merge := procedure(l1, l2, cmp) {
    match ([l1, l2]) {
        case [[], _] : return l2;
        case [_, []] : return l1;
        case [[x1|r1], [x2|r2]] :
            if (cmp(x1, x2)) {
                return [x1 | merge(r1, l2, cmp)];
            } else {
                return [x2 | merge(l1, r2, cmp)];
            }
    }
};
less    := procedure(x, y) { return x < y; };
greater := procedure(x, y) { return y < x; };
l  := [1,3,5,4,2];
s1 := sort(l, less   );
s2 := sort(l, greater);
```

Figure 5.2: A generic sort function.

of this scheme. One example is sorting. Figure 5.2 on page 64 shows a generic implementation of the algorithm *merge sort*. Here, the function `sort` takes two arguments:

1. The first argument l is the list to be sorted.
2. The second argument `cmp` is a binary function that compares two elements and returns either `true` or `false`. If we call the function `sort` with second argument `less`, where the function

`less` is defined in line 20, then the resulting list will be sorted ascendingly. If instead we use the function `greater` defined in line 21 as the second argument, then the list l will be sorted descendingly.

The second argument `cmp` of the procedure `sort` enables us to sort a list that contains elements that are not numbers. In order to do so, we just have to implement an appropriate version of the function `cmp`.

5.1.1 Implementing Counters via Closures

The concept of a closure is non-trivial, therefore we introduce the idea using some simple examples. Figure 5.3 shows the function `createCounter`. This function initializes the variable `count` to a given value and then returns a procedure that increments this value. Note that this procedure is not defined via the keyword `procedure` but rather via the keyword `closure`. The reason is that this function needs access to the variable `count` that is defined outside of the closure `counter`. By defining the function `counter` as a `closure` rather than as a `procedure`, the function `counter` has access to the variable `count`: It can both read its value and can even change the value. When the function `counter` is defined, the variable `count` is safely tucked away together with the function `counter`. Therefore, although the variable `count` goes out of scope once the function `createCounter` terminates, the function `counter` still has access to a copy of this variable. Hence, when we call the function `createCounter` in line 9, the variable `ctr0` is assigned a version of the function counter where the variable `count` initially has the value 0. Later, when we call the function `ctr0`, in line 12, this counter is incremented 3 times. Therefore, after line 12 is executed, the variables `u`, `v`, and `w` have the values 1, 2, and 3, respectively.

```
1   createCounter := procedure(i) {
2       count   := i;
3       counter := closure() {
4                       count += 1;
5                       return count;
6                   };
7       return counter;
8   };
9   ctr0 := createCounter(0);
10  ctr9 := createCounter(9);
11
12  u := ctr0(); v := ctr0(); w := ctr0();
13  x := ctr9(); y := ctr9(); z := ctr9();
```

Figure 5.3: Creating a counter as a closure.

You should notice that the function `ctr9` created in line 10 gets its own copy of the variable `count`. In the case of the function `ctr9`, this copy of `count` is initialized with the value 9. Therefore, after we have called the function `ctr9` in line 13, the variables `x`, `y`, and `z` will have the values 10, 11, and 12.

5.1.2 Lambda Definitions for Closures

There is a short-hand notation for the definition of a closure that can be used if the closure only contains a single `return` statement. This short-hand notation relates to closures as lambda definitions relate to procedures. This short-hand notation is called a *lambda closure*. Here is a simple example: The following two lines define a function `f` that computes the cube of its argument.

```
c := 3;
```

```
f := x |=> x ** c;
```

These two lines are equivalent to the following two lines:

```
c := 3;
f := closure(x) { return x ** c; };
```

In general, the expression

```
x |=> expr
```

is equivalent to the following expression:

```
closure(x) { return expr; }
```

If there is more than one argument to a closure, the arguments have to be enclosed in square brackets. For example,

```
f := [x,y] |=> sqrt(x ** c + y ** c);
```

is equivalent to

```
f := closure(x,y) { return sqrt(x ** c + y ** c); };
```

5.1.3 Closures as Return Values

So far, the functions we have discussed did not return functions. The next example will change that. In mathematics, given a function

$$f : \mathbb{N} \to \mathbb{N}$$

mapping natural numbers into natural numbers, the *discrete derivative* of f is again a function mapping natural numbers to natural numbers and is denoted as Δf. For $n \in \mathbb{N}$, the value $(\Delta f)(n)$ is defined as

$$(\Delta f)(n) := f(n+1) - f(n).$$

For example, given the function $f : \mathbb{N} \to \mathbb{N}$ defined as $f(n) := 2^n$, we have

$$(\Delta f)(n) = 2^{n+1} - 2^n = 2^n = f(n).$$

To give another example, the discrete derivative of the identity function is the function that maps every natural number to the number 1, since $(n+1) - n = 1$.

```
1  delta := procedure(f) {
2      return n |=> f(n+1) - f(n);
3  };
4  g := n |-> n;
5  h := n |-> 2 ** n;
6  deltaG := delta(g);
7  deltaH := delta(h);
8
9  print([ deltaG(n) : n in [1 .. 10]]);
10 print([ deltaH(n) : n in [1 .. 10]]);
```

Figure 5.4: Computing the discrete derivative of a given function.

Figure 5.4 shows the implementation of the function `delta` that takes a function `f` as input and then returns the discrete derivative of `f`. In line 6 and 7, this function is applied to the identity function and

to the function $n \mapsto 2^n$, respectively. The resulting functions `deltaG` and `deltaH` are then applied to the sequence of natural numbers from 1 to 10. In the first case, the result is a list containing the number one ten times, while in the second case the the result is a list of the first ten powers of two.

Note that we had to define the return value of the function `delta` as a closure. The reason is that otherwise the value `f` referring to the function would not be in the local scope of the function that is returned by `delta` and hence `f` would be undefined.

5.2 Closures in Action: Generating Finite State Machines from Regular Expressions

In this section we demonstrate a non-trivial application of closures. We present a program that takes a regular expression r as input and converts this regular expression into a *non-deterministic finite state machine* using the *Thompson construction* [HMU06]. To this end, we first define a *alphabet* as a finite set of characters. Then, for the purpose of this section, *regular expressions over an alphabet* Σ are defined inductively as follows.

1. Every character $c \in \Sigma$ is a regular expression matching exactly this character and nothing else.
2. If r_1 and r_2 are regular expressions, then $r_1 \cdot r_2$ is a regular expression. If r_1 matches the string s_1 and r_2 matches the string s_2, then $r_1 \cdot r_2$ matches the string $s_1 s_2$, where $s_1 s_2$ is understood as the concatenation of s_1 and s_2.
3. If r_1 and r_2 are regular expressions, then $r_1 \mid r_2$ is a regular expression. The regular expression $r_1 \mid r_2$ matches any string that is matched by either r_1 or r_2.
4. If r is a regular expression, then r^* is a regular expression. This regular expression matches the empty string and, furthermore, matches any string s that can be decomposed as

   $$s = t_1 t_2 \cdots t_n$$

   where each of the substrings t_i is matched by the regular expression r.

For the purpose of the program we are going to develop, composite regular expressions will be represented by terms. In detail, we have the following representation of regular expressions.

1. A regular expression r of the form $r = c$ where $c \in \Sigma$ is represented the character c.
2. A regular expression r of the form $r = r_1 \cdot r_2$ is represented by the term

   ```
   @Cat(r1, r2)
   ```

   where `r1` is the representation of r_1 and `r2` is the representation of r_2. The name `Cat` reminds us of concatenation.
3. A regular expression r of the form $r = r_1 \mid r_2$ is represented by the term

   ```
   @Or(r1, r2)
   ```

   where `r1` is the representation of r_1 and `r2` is the representation of r_2.
4. A regular expression r of the form $r = r_0^*$ is represented by the term

   ```
   @Star(r0)
   ```

   where `r0` is the representation of r_0.

The *Thompson construction* of a non-deterministic finite state machine from a given regular expression r works by induction on the structure of r. Given a regular expression r, we define a non-deterministic finite state machine $A(r)$ by induction on r. For the purpose of this section, a finite state machine A

is a 4-tuple

$$A = \langle Q, \delta, q_0, q_f \rangle$$

where Q is the set of states, δ is the transition function and maps a pair $\langle q, c \rangle$ where q is a state and c is a character to a set of new states. If we denote the set of characters as Σ, then the function δ has the signature

$$\delta : Q \times \Sigma \to 2^Q.$$

The understanding of $\delta(q, c)$ is that if the finite state machine A is in the state q and reads the character c, then it can switch to any of the states in $\delta(q, c)$. Finally, q_0 is the start state and q_f is the accepting state. In the following, we abbreviate the notion of a *finite state machine* as fsm.

1. For a letter c the fsm $A(c)$ is defined as

   $$A(c) = \langle \{q_0, q_1\}, \{\langle q_0, c \rangle \mapsto q_1\}, q_0, q_1 \rangle.$$

   The set of states is given as $\{q_0, q_1\}$, on reading the character c the transition function maps the state q_0 to q_1, the start state is q_0 and the accepting state is q_1. This fsm is shown in Figure 5.5. The start state q_0 is marked by an incoming arrow, while the accepting state q_1 is marked by a doubled boundary.

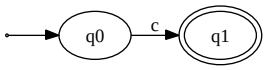

Figure 5.5: The finite state machine $A(c)$.

```
1   genCharNFA := procedure(c, rw ctr) {
2       q0 := getNewState(ctr);
3       q1 := getNewState(ctr);
4       delta := closure(q, d) {
5                    if (q == q0 && d == c) {
6                        return { q1 };
7                    } else {
8                        return {};
9                    }
10               };
11      return [ {q0, q1}, delta, q0, q1 ];
12  };
```

Figure 5.6: Generating the fsm to recognize the character c.

Figure 5.6 shows the function `genCharNFA` that takes a character c and generates the fsm $A(c)$. The code corresponds to the diagram shown in Figure 5.5.

(a) We generate two new states q_0 and q_1 using the function `getNewState`. The implementation of this function is shown in Figure 5.13 on page 72. Every time this function is called, it creates a new unique string that is interpreted as a state.

(b) The transition function checks whether the state q given as input is equal to the start state q_0 and whether, furthermore, the character d that is read is identical to the character c. If this is the case, the fsm switches into the state q_1 and therefore the set of possible next states is the singleton set $\{q_1\}$. Since this is the only transition of the fsm, in all other cases the set of next states is empty.

2. In order to compute the fsm $A(r_1 \cdot r_2)$ we have to assume that the sets of states of the fsms $A(r_1)$ and $A(r_2)$ are disjoint. This is true since the function `getNewState` does create a new state every time it is called. Let us assume that the fsms $A(r_1)$ and $A(r_2)$ have the following form:

   (a) $A(r_1) = \langle Q_1, \delta_1, q_1, q_2 \rangle$,

   (b) $A(r_2) = \langle Q_2, \delta_2, q_3, q_4 \rangle$, where

   (c) $Q_1 \cap Q_2 = \{\}$.

   Using $A(r_1)$ and $A(r_2)$ we construct the fsm $A(r_1 \cdot r_2)$ as

   $$\langle Q_1 \cup Q_2, \Sigma, \{\langle q_2, \varepsilon\rangle \mapsto q_3\} \cup \delta_1 \cup \delta_2, q_0, q_4 \rangle$$

   The notation $\{\langle q_2, \varepsilon\rangle \mapsto q_3\} \cup \delta_1 \cup \delta_2$ specifies that the transition function δ contains all the transitions from the transition functions δ_1 and δ_2. Additionally, there is an ε-transition from q_2 to q_3. Formally, the transition function could also be specified as follows:

   $$\delta(q, c) := \begin{cases} \{q_3\} & \text{if } q = q_2 \text{ and } c = \varepsilon, \\ \delta_1(q, c) & \text{if } q \in Q_1 \text{ and } \langle q, c\rangle \neq \langle q_2, \varepsilon\rangle, \\ \delta_2(q, c) & \text{if } q \in Q_2. \end{cases}$$

   Figure 5.7 depicts the fsm $A(r_1 \cdot r_2)$.

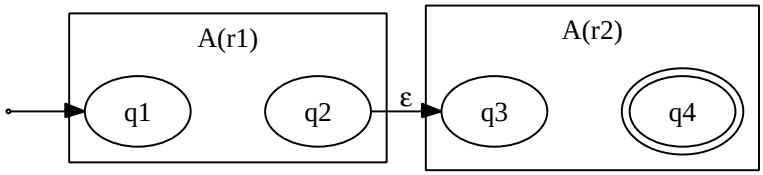

Figure 5.7: The finite state machine $A(r_1 \cdot r_2)$.

   Figure 5.8 on page 70 shows the implementation of the function `catenate`. This function takes two finite state machines `f1` and `f2` and concatenates them in the way depicted in Figure 5.7. This function can be used to compute $A(r_1 \cdot r_2)$, since we have

   $$A(r_1 \cdot r_2) = \texttt{catenate}\big(A(r_1), A(r_2)\big).$$

3. In order to define the fsm $A(r_1 + r_2)$ we assume that we have already computed the fsms $A(r_1)$ and $A(r_2)$ and that their sets of states are disjoint. If $A(r_1)$ and $A(r_2)$ are given as

   $$A(r_1) = \langle Q_1, \delta_1, q_1, q_3 \rangle \quad \text{and} \quad A(r_2) = \langle Q_2, \delta_2, q_2, q_4 \rangle$$

   then the fsm $A(r_1 + r_2)$ can be defined as

   $$\langle \{q_0, q_5\} \cup Q_1 \cup Q_2, \{\langle q_0, \varepsilon\rangle \mapsto q_1, \langle q_0, \varepsilon\rangle \mapsto q_2, \langle q_3, \varepsilon\rangle \mapsto q_5, \langle q_4, \varepsilon\rangle \mapsto q_5\} \cup \delta_1 \cup \delta_2, q_0, q_5 \rangle.$$

   This finite state machine is shown in Figure 5.9. In addition to the states of $A(r_1)$ and $A(r_2)$ there are two additional states:

   (a) q_0 is the start state of the fsm $A(r_1 + r_2)$,

   (b) q_5 is the accepting state of $A(r_1 + r_2)$.

   In addition to the transitions of the fsms $A(r_1)$ and $A(r_2)$ we have four ε-transitions.

   (a) There are ε-transitions from q_0 to the states q_1 and q_2.

   (b) There are ε-transitions from q_3 and q_4 to q_5.

```
1   catenate := procedure(f1, f2) {
2       [m1, delta1, q1, q2] := f1;
3       [m2, delta2, q3, q4] := f2;
4       delta := closure(q, c) {
5                       if (q == q2 && c == "") {
6                           return { q3 };
7                       } else if (q in m1) {
8                           return delta1(q, c);
9                       } else if (q in m2) {
10                          return delta2(q, c);
11                      } else {
12                          return {};
13                      }
14                  };
15      return [ m1 + m2, delta, q1, q4 ];
16  };
```

Figure 5.8: The function to compute $A(r_1 \cdot r_2)$

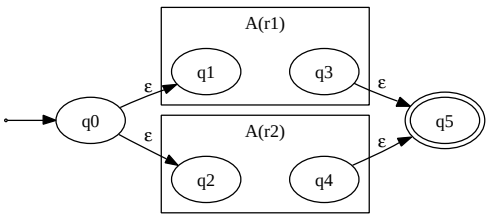

Figure 5.9: The finite state machine $A(r_1 + r_2)$.

Figure 5.10 on page 71 shows the implementation of the function `disjunction`. This function takes two finite state machines `f1` and `f2` and combines them in the way depicted in Figure 5.9. This function can be used to compute $A(r_1 + r_2)$, since we have

$$A(r_1 + r_2) = \texttt{disjunction}\big(A(r_1), A(r_2)\big).$$

4. In order to define $A(r^*)$ we assume $A(r)$ is given as

   $$A(r) = \langle Q, \delta, q_1, q_2 \rangle.$$

   Then $A(r^*)$ is defined as

   $$\langle \{q_0, q_3\} \cup Q, \{\langle q_0, \varepsilon\rangle \mapsto q_1, \langle q_2, \varepsilon\rangle \mapsto q_1, \langle q_0, \varepsilon\rangle \mapsto q_3, \langle q_2, \varepsilon\rangle \mapsto q_3\} \cup \delta, q_0, q_3 \rangle.$$

   Figure 5.11 depicts the fsm $A(r^*)$. In addition to the states of the fsm $A(r)$ there are two new states:

   (a) q_0 is the start state of $A(r^*)$,

   (b) q_3 is the accepting state of $A(r^*)$.

   Furthermore, there are four additional ε-transitions.

   (a) There are two ε-transitions form the start state q_0 to q_1 and q_3.

   (b) From q_2 there is an ε-transition back to q_1 and also an ε-transition to q_3.

```
1  disjunction := procedure(f1, f2, rw ctr) {
2      [m1, delta1, q1, q3] := f1;
3      [m2, delta2, q2, q4] := f2;
4      q0 := getNewState(ctr);
5      q5 := getNewState(ctr);
6      delta := closure(q, c) {
7                   if (q == q0 && c == "") {
8                       return { q1, q2 };
9                   } else if (q in { q3, q4 } && c == "") {
10                      return { q5 };
11                  } else if (q in m1) {
12                      return delta1(q, c);
13                  } else if (q in m2) {
14                      return delta2(q, c);
15                  } else {
16                      return {};
17                  }
18              };
19      return [ { q0, q5 } + m1 + m2, delta, q0, q5 ];
20  };
```

Figure 5.10: The function to compute $A(r_1 + r_2)$.

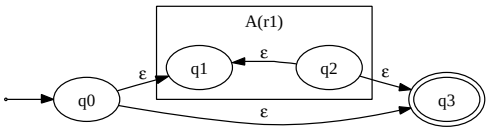

Figure 5.11: The finite state machine $A(r^*)$.

Figure 5.12 on page 72 shows the implementation of the function `kleene`. This function takes a finite state machine `f` and transforms it in the way depicted in Figure 5.11. This function can be used to compute $A(r^*)$, since we have

$$A(r^*) = \texttt{kleene}\big(A(r)\big).$$

Figure 5.13 shows the procedure `getNewState` that is used to create new states. Since the parameter `ctr` of this procedure is preceded by the keyword `rw`, this parameter has a *call-by-name* semantics and therefore the procedure `getNewState` is able to increment this variable. Hence, the states generated by `getNewState` are unique.

Finally, Figure 5.14 shows how to generate a finite state machine from a given regular expression r. The function `regexp2NFA`(r) is defined by recursion on r:

1. If r is a single character c, then the fsm $A(c)$ is computed using the function `genCharNFA`(c).

2. If $r = r_1 \cdot r_2$, the function `regexp2NFA` recursively computes finite state machines $A(r_1)$ and $A(r_2)$. These fsms are then combined using the function `catenate`$\big(A(r_1), A(r_2)\big)$.

3. If $r = r_1 + r_2$, the function `regexp2NFA` recursively computes finite state machines $A(r_1)$ and $A(r_2)$. These fsms are then combined using the function `disjunction`$\big(A(r_1), A(r_2)\big)$.

4. If $r = r_0^*$, the function `regexp2NFA` recursively computes the finite state machines $A(r_0)$. This fsm is then transformed using the function `kleene`$\big(A(r_0)\big)$.

```
kleene := procedure(f, rw ctr) {
    [m, delta0, q1, q2] := f;
    q0 := getNewState(ctr);
    q3 := getNewState(ctr);
    delta := closure(q, c) {
                if (q == q0 && c == "") {
                    return { q1, q3 };
                } else if (q == q2 && c == "") {
                    return { q1, q3 };
                } else if (q in m) {
                    return delta0(q, c);
                } else {
                    return {};
                }
            };
    return [ { q0, q3 } + m, delta, q0, q3 ];
};
```

Figure 5.12: The function to compute $A(r^*)$.

```
getNewState := procedure(rw ctr) {
    ctr += 1;
    return "q" + ctr;
};
```

Figure 5.13: A function to generate unique states.

The reader should note that we have made heavy use of closures to implement the functions `genCharNFA`, `catenate`, `disjunction`, and `kleene`.

```
regexp2NFA := procedure(r, rw ctr) {
    match (r) {
        case c | isString(c):
            return genCharNFA(c, ctr);
        case @Cat(r1, r2):
            return catenate(regexp2NFA(r1, ctr), regexp2NFA(r2, ctr));
        case @Or(r1, r2):
            return disjunction(regexp2NFA(r1, ctr), regexp2NFA(r2, ctr), ctr);
        case @Star(r0):
            return kleene(regexp2NFA(r0, ctr), ctr);
    }
};
```

Figure 5.14: Generating a non-deterministic fsm from a regular expression.

Let us take a step back and consider what we have achieved so far. We have implemented a function for converting regular expressions to non-deterministic finite state machines. It is rather straightforward to implement a function that scans a given string using a non-deterministic finite state

machine. If we would implement this function, we would then have implemented our own support for regular expressions.

5.3 Decorators

The programming language *Python* has a nice feature called *decorators* that makes it easy to modify existing functions. For example, a *trace decorator* is a function that can modify a given function f so that every invocation of the function f and every value returned by the function f are automatically traced. SETLX supports similar techniques as *Python* and, therefore, it is easy to implement decorators in SETLX, too. In this section, we show how a simple albeit powerful trace decorator can be implemented in just a few lines of SETLX code.

```
1  gcd := procedure(n, m) {
2      if (m == 0) {
3          return n;
4      }
5      return gcd(m, n % m);
6  };
7  myTracer := tracer();
8  gcd := myTracer.trace(gcd, "gcd");
9  n   := gcd(36, 27);
10 gcd := myTracer.untrace("gcd");
```

Figure 5.15: Using the trace decorator.

Figure 5.16 on page 74 shows the class `tracer`. While classes will be discussed in full detail in a later chapter, it is convenient to use them for this example. This class `tracer` implements a trace decorator. Before we discuss the details of its implementation, lets us show how this decorator is used. Figure 5.15 shows the implementation of the function `gcd` computing the greatest common divisor of two natural numbers using the Euclidean algorithm. Suppose we want to trace every invocation of the function `gcd`. We do so by first creating an object of class `tracer` in line 7 of Figure 5.15. Then the line

```
gcd := myTracer.trace(gcd, "gcd");
```

activates tracing for the function `gcd`. Note that the first argument of the method `trace` is the function we want to trace, while the second argument is the name of the function. After executing this assignment statement, the function `gcd` will be traced. For example, the call `gcd(36, 27)` produces the following output:

```
calling gcd(36, 27)
calling gcd(27, 9)
calling gcd(9, 0)
gcd(9, 0) = 9
gcd(27, 9) = 9
gcd(36, 27) = 9
~< Result: 9 >~
```

This shows that every time the function `gcd` is called, a statement of the form

```
calling gcd(m, n)
```

is printed. This is true even for the recursive invocations of `gcd`. Furthermore, every time the function `gcd` returns a result, this is also printed together with the arguments used to compute this result.

To stop tracing the function gcd, we simply issue the statement:

```
gcd := myTracer.untrace("gcd");
```

After this assignment, the function gcd is restored to its original state and will not print anything when called.

```
1  class tracer() {
2      mStoredProcedures := {};
3
4      trace := procedure(function, functionName) {
5          mStoredProcedures[functionName] := function;
6          tracedFunction := closure(*args) {
7              argsString := join(args, ", ");
8              print("calling $functionName$($argsString$)");
9              result := function(*args);
10             print("$functionName$($argsString$) = $result$");
11             return result;
12         };
13         return tracedFunction;
14     };
15     untrace := procedure(functionName) {
16         return mStoredProcedures[functionName];
17     };
18 }
```

Figure 5.16: A class providing a trace decorator.

Let us proceed to discuss how the class tracer is implemented in SetlX. Figure 5.16 shows the implementation of this class. We discuss this implementation line by line.

1. Line 2 defines the member variable mStoredProcedures. This variable will store a map that is represented as a binary relation. The keys of this maps are procedure names, while the values entered into this map are the procedures themselves. Initially, this map is empty. Every time a procedure is traced, the old value of the procedure is stored in this map. This is needed in order to implement the method untrace that restores a given procedure to its original state.

2. The workhorse of the class tracer is the method trace defined in line 4. This method takes two arguments: The first argument is a procedure, while the second argument is the name of the procedure. The function will return a new procedure that behaves like the given procedure but, furthermore, traces the invocation of the original procedure as discussed previously in the example. The procedure trace works as follows:

   (a) In line 5, the old value of the procedure to be traced is saved in the dictionary

   mStoredProcedures.

   This value is needed to restore the procedure to its original value when we do not want to trace the procedure any more.

   (b) Next, line 6 defines a closure called tracedFunction. This is a function called with a variable number of arguments. The reason is that we do not know how many arguments the procedure function that is given as argument to the method trace actually takes. In the example discussed above, we had traced the function gcd that takes two arguments.

We could just as well have traced a procedure taking only one argument as we could have traced a procedure taking nine arguments.

(c) In line 7 and 8, the closure `tracedFunction` constructs a string containing the arguments and the name of the function and then prints this string. This `tracedFunction` refers to `function` and `functionName`. As these variables are only defined in the method `trace`, the function `tracedFunction` needs to be defined as a `closure` so that it can access these variables.

(d) Line 9 is the most important line of the whole implementation: It is here that the original procedure is called with the given arguments. Note how prefixing the list of arguments `args` with the operator "`*`" makes it possible to call this function without knowing how many arguments the procedure actually takes.

(e) Line 10 prints the result that has been computed.

(f) This result is then returned in line 11.

(g) Finally, the method `trace` returns the newly created closure. By assigning this closure to the name of the original procedure we can thus trace this procedure.

3. As a convenience, the class `tracer` also provides the method `untrace` in line 16. This procedure looks up the original procedure stored under the given name in the dictionary `mStoredProcedures` and returns it. Hence, assigning the value returned by this procedure to the name of a procedure stops tracing this procedure.

Chapter 6

Exceptions and Backtracking

In the first section of this chapter we will discuss *exceptions* as a means to deal with error situations. The second section will introduce a mechanism that supports *backtracking*. This mechanism is quite similar to the exception handling discussed in the first subsection. Indeed, we will see that the backtracking mechanism provided in SetlX is really just a special case of exception handling.

6.1 Exceptions

If we issue the assignment

```
y := x + 1;
```

while the variable `x` is undefined, SetlX reports the following error:

```
Error in "y := x + 1":
Error in "x + 1":
'om + 1' is undefined.
```

Here, evaluation of the expression `x + 1` has raised an *exception* as it is not possible to add a number to the undefined value `om`. This exception is then propagated to the enclosing assignment statement. SetlX offers to handle exceptions like the one described above. The mechanism is similar to the way exceptions are treated in *Java* and uses the keywords "`try`" and "`catch`". If we have a sequence of statements *stmntList* and we suspect that something might go wrong with these statements, then we can put the list of statements into a `try`/`catch`-block as follows:

```
try {
   stmntList
} catch (e) { errorCode }
```

If the execution of *stmntList* executes without errors, then the `try`/`catch`-block does nothing besides the execution of *stmntList*. However, if one of the statements in *stmntList* raises an exception `e`, then the execution of the statements in *stmntList* is aborted and instead the statements in *errorCode* are executed. These statements can use the variable `e` that contains the exception that has been raised.

Typically, exception handling is necessary when processing user input. Consider the program shown in Figure 6.1 on page 77. The function `findZero` implements the bisection method which can be used to find a zero of a function that has a sign change in a given interval. The first argument of `findZero` is a function f, while the arguments a and b are the left and right boundary of the interval where the zero of f is sought. Therefore, a has to be less than b. Finally, for the bisection algorithm implemented in the function `findZero` to work, the function f needs to have a sign change in the interval $[a, b]$, i.e. the sign of $f(a)$ needs to be different from the sign of $f(b)$.

```
1  findZero := procedure() {
2      try {
3          s := read("Please enter a function: ");
4          f := eval("x |-> " + s);
5          a := read("Enter left  boundary: ");
6          b := read("Enter right boundary: ");
7          z := findZero(f, a, b);
8          print("zero at z = $z$");
9      } catch (e) {
10         print(e);
11         print("Please try again.\n");
12         findZero();
13     }
14 };
15 bisection := procedure(f, a, b) {
16     if (a > b) {
17         throw("Left boundary a has to be less than right boundary b!");
18     }
19     [ fa, fb ] := [ f(a), f(b) ];
20     if (fa * fb > 0) {
21         throw("Function f has to have a sign change in [a, b]!");
22     }
23     while (b - a >= 10 ** -12) {
24         c := 1/2 * (a + b);
25         fc := f(c);
26         if ((fa < 0 && fc < 0.0) || (fa >= 0 && fc >= 0)) {
27             a := c; fa := fc;
28         } else {
29             b := c; fb := fc;
30         }
31     }
32     return 1/2 * (a + b);
33 };
```

Figure 6.1: The bisection method for finding a zero of a function.

The function `findZero` asks the user to input the function f together with the left and right boundary of the interval $[a, b]$. The idea is that the user inputs a term describing the function value of f for a given value of x. For example, in order to compute the zero of the function

$$x \mapsto x * x - 2$$

the user has to provide the string "`x*x-2`" as input when prompted by the function `read` in line 3. The string `s` that is input by the user is then converted into the string

```
x |-> x*x-2
```

in line 4 and the resulting string is then evaluated. In this way, the variable `f` in line 4 will be assigned the function mapping `x` to `x*x-2` just as if the user had written

```
f := x |-> x*x-2;
```

in the command line.

There are a couple of things that can go wrong with the function `findZero`. First, the string `s` that is input by the user might not be a proper function and then we would presumably get a parse error in line 4. In this case, the function `parse` invoked in line 4 will raise an exception. Next, if the user enters a string of the form

```
x ** y
```

then, since the variable `y` is undefined, the evaluation of the function would raise an exception when SetlX tries to evaluate an expression of the form

```
x ** om.
```

Furthermore, either of the two conditions

$$a < b \quad \text{or} \quad f(a) * f(b) \leq 0$$

might be violated. In this case, we raise an exception in line 17 or line 21 of the function `bisection`. This is done using the function `throw`. Since we don't want to abort the program on the occurrence of an exception, the whole block of code from line 3 up to line 8 is enclosed in a `try/catch`-block. In case there is an exception, the value of this exception, which is a string containing an error message, is caught in the variable `e` in line 9. To continue our program, we print the error message in line 10 and then invoke the function `findZero` recursively so that the user of the program gets another chance to enter valid input.

6.1.1 Different Kinds of Exceptions

SetlX supports two different kinds of exceptions:

1. *User generated* exceptions are generated by the user via a `throw` statement. In general, the statement

   `throw(`e`)`

   raises a *user generated* exception with the value e. Here, e can be any value. However, typically e will be a string.

2. *Language generated* exceptions are the result of error conditions arising in the program.

While all kinds of exceptions can be caught with a `catch` clause, most of the time it is useful to distinguish between the different kinds of exceptions. This is supported by offering two variants of `catch`:

1. `catchUsr` only catches user generated exceptions. For example, the statement

   ```
   try { throw(1); } catchUsr(e) { print(e); }
   ```

   prints the number 1, but assuming that the variable `y` is undefined, the statement

   ```
   try { x := y + 1; } catchUsr(e) { print("caught " + e); }
   ```

   will not print anything but instead this command raises an exception.

2. `catchLng` only catches language generated exceptions. Therefore, the statement

   ```
   try { x := y + 1; } catchLng(e) { print("caught " + e); }
   ```

   prints the text

   ```
   caught Error:  'om + 1' is undefined.
   ```

   On the other hand, the exception thrown in the statement

   ```
   try { throw(1); } catchLng(e) { print(e); }
   ```

is a user generated exception that is not caught by `catchLng`. Instead, the exception is propagated outside of this statement. Hence, the net effect of this statement is to raise an exception.

Being able to distinguish between user generated and language generated exceptions is quite valuable and we strongly advocate that user generated exceptions should only be catched using a `catchUsr` clause. The reason is, that a simple `catch` clause which the user intends to catch a user generated exceptions might, in fact, catch other exceptions and thus mask real errors. In general, the SetlX interpreter implements a *fail fast* strategy: Once an error is discovered, the execution of the program is aborted. The interpreter does a lot of effort to detect errors as early as possible. However, this strategy is thwarted if unrestricted `catch` clauses are used, because then language generated exceptions are effectively silenced. As a result, programming mistakes might go undetected and result in bugs that are very difficult to locate.

6.2 Backtracking

One of the distinguishing features of the programming language *Prolog* is the fact that *Prolog* supports *backtracking*. However, on closer inspection of the programs found in popular text books, e.g. "*The Art of Prolog*" [SS86] and "*Prolog Programming for Artificial Intelligence*" [Bra90], it becomes obvious that very few programs actually make use of backtracking in its most general form. The first author of this tutorial has developed *Prolog* based software in an industrial environment for more than 10 years. His personal experience suggests that *Prolog* programs that use backtracking in an unrestricted fashion tend to be very hard to maintain. Therefore, it is our believe that the use of backtracking should follow the *generate and test* paradigm:

1. Assume our goal is to find some x such that x is the solution of a given problem.
2. In order to find possible values for x, we invoke a *generating function* that provides us with *solution candidates*, i.e. the generating function produces values of x that might possibly be solutions to the given problem. However, the set of values produced by the generating function is only a superset of the set of solutions, i.e. some of those values do not actually solve the given problem. Although it is perfectly acceptable that most of the values generated do not solve the problem, it is required that the generating function will never miss a solution.
3. In order to check which of the generated values are indeed solutions, these values have to be *tested*. If a test fails for a given value x, the program backtracks to step 2 where the next value to be tested is generated.

In order to support the generate and test paradigm, SetlX implements backtracking only in a very restricted form. Thus, we avoid the pitfalls that accompany an unrestricted use of backtracking. Backtracking is implemented via the keywords "`check`" and "`backtrack`". A block of the form

```
check {
   stmntList
}
```

is more or less[1] converted into a block of the following form:

```
try {
   stmntList
} catch (e) {
   if (e != "fail") { throw(e); }
}
```

[1] Technically, instead of the string `"fail"`, SetlX generates a unique exception, which is referred to as a *fail-exception*. This exception can only be catched using `check`.

Here, *stmntList* might contain the keyword "backtrack". This keyword is translated into the command `throw("fail")`.

In the rest of this section we show how to solve two classical logical puzzles using backtracking. As a first example, we solve the 8 queens puzzle. We will see that this puzzle can be solved using a straightforward recursive approach. The second example is the zebra puzzle. In order to solve this puzzle succinctly, we need to make use of *higher-order programming*, i.e. we solve the problem by writing a function that, instead of solving the problem directly, will first generate some SetlX code. This code is then executed and thereby the problem is solved. Hence, the zebra puzzle provides the opportunity to show of the power of the `eval` function provided by SetlX.

6.2.1 The 8 Queens Puzzle

The program in Figure 6.2 on page 80 solves the *8 queens puzzle*. This problem asks to position 8 queens on a chessboard such that no queen can attack another queen. In chess, a queen can attack all those positions that are either on the same row, on the same column, or on the same diagonal as the queen. The details of the program in Figure 6.2 are as follows.

```
1  solve := procedure(queens, n) {
2      if (#queens == n) {
3          return queens;
4      }
5      for (x in {1..n} - {i : i in queens}) {
6          check {
7              testNext(queens, x);
8              return solve(queens + [x], n);
9          }
10     }
11     backtrack;
12 };
13 testNext := procedure(queens, x) {
14     m := #queens;
15     if (exists (i in {1..m} | i-queens[i] == m+1-x || i+queens[i] == m+1+x)) {
16         backtrack;
17     }
18 };
```

Figure 6.2: Solving the 8 queens puzzle using `check` and `backtrack`.

1. The procedure `solve` has two parameters.

   (a) The first parameter `queens` is a list of positions of queens that have already been placed on the board. It can be assumed that the queens already positioned in `queens` do not attack each other.

   Technically, `queens` is a list of integers. If `queens[`i`]` $= k$, then the queen in row i is placed in column k. For example,

   `queens` $=$ `[4, 8, 1, 3, 6, 2, 7, 5]`

   is a solution of the 8 queens puzzle. This solution is depicted in Figure 6.3 on page 81.

   (b) The second parameter `n` is the size of the board.

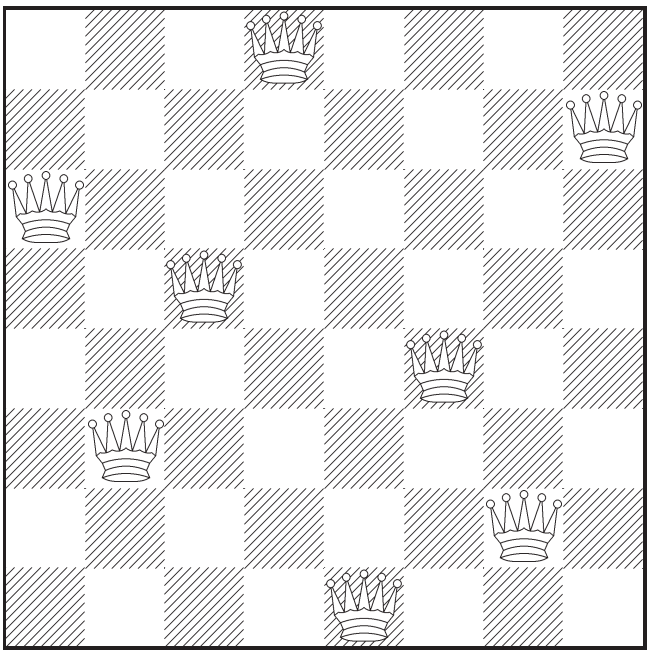

Figure 6.3: A solution of the 8 queens puzzle.

In order to solve the 8 queens puzzle, the procedure `solve` can be called as

```
solve([], 8).
```

Taking the parameter `queens` to be the empty list assumes that initially no queen has been placed on the chess board. Therefore, the assumption that the queens already positioned in the list `queens` do not attack each other is trivially satisfied.

2. In line 2 it is checked, whether the list `queens` already specifies the positions of `n` queens, that is we check whether `queens` is a list of length `n`. If this is the case, then because of the assumption that the queens specified in `queens` do not attack each other, the problem is solved and therefore the list `queens` is a solution that is returned.

3. Otherwise, we find a position `x` for the next queen in line 5. Of course, there is no point in trying to position the next queen into a column that has already been taken by one of the queens in the list `queens`. Therefore, the number of column positions available for the next queen is given by the set

   ```
   {1 .. n} - {i : i in queens}.
   ```

   Note that we had to convert the list `queens` into the set `{i : i in queens}` in order to be able to subtract the set of columns specified in `queens` from the set of all possible columns.

4. Once we have decided to position the next queen in column `x`, we have to test whether a queen that is put into that position can be attacked by another queen which happens to be on the same diagonal. This test is performed in line 7 with the help of the function `testNext`.

5. If this test succeeds, we add a queen in position `x` to the list `queens` and recursively try to solve the resulting instance of the problem.

6. On the other hand, if the call to `testNext` in line 7 fails, we have to try the remaining values for `x`. The function `testNext` does not return a Boolean value to indicate success or failure so

at this point you might well ask how we know that the call to `testNext` has failed. The answer is that the function `testNext` includes a call to `backtrack` if it is not possible to place the next queen in column `x`. Technically, calling `backtrack` raises an exception that is caught by the `check` statement in line 6. After that, the `for` loop in line 5 proceeds and picks the next candidate for `x`.

7. During the recursive invocation of the procedure `solve` in line 8, we might discover that the list `queens + [x]` can not be completed into a solution of the 8 queens puzzle. In this case, it is the function `solve` that backtracks in line 11. This happens when the `for` loop in line 5 is exhausted and we have not found a solution. Then control reaches line 11, where the backtrack statement signals that the list `l` could not be completed into a solution to the `n` queens puzzle.

8. The function `testNext` in line 13 has two parameters: The first parameter is the list of already positioned queens while the second parameter specifies the column of the next queen. The function checks whether the queen specified by `x` is on the same diagonal as any of the queens in `queens`.

   In order to understand the calculation in line 15 we have to realize that the Cartesian coordinates of the queens in column `x` are

   $$\langle \texttt{\#queens} + 1, \texttt{x} \rangle.$$

   A diagonal is specified as the equation of a line with slope either $+1$ or -1. The i-th queen in `queens` has the coordinates

   $$\langle \texttt{i}, \texttt{queens}[\texttt{i}] \rangle.$$

   Therefore, it is on the same ascending diagonal as the queen specified by `x` if we have

   $$\texttt{i-queens[i]} = \texttt{\#queens+1-x},$$

   while it is one the same descending diagonal if we have.

   $$\texttt{i+queens[i]} = \texttt{\#queens+1+x}.$$

It is easy to change the program in Figure 6.2 such that all solutions are completed. Figure 6.4 on page 83 shows how this is done.

1. We have added a function `allSolutions`. This function gets the parameter `n`, which is the size of the board. The function returns the set of all solutions of the n queens puzzle. To do so, it first initializes the variable `all` to the empty set. The solutions are then collected in this set.

2. The new version of the function `solve` gets `all` as an additional parameter. Note that this parameter is specified in line 2 as an `rw` parameter, so the value of `all` can actually be changed by the procedure `solve`.

3. The important change in the implementation of `solve` is that instead of returning a solution, a newly found solution is added to the set `all` in line 10. After that, the function `solve` backtracks to look for more solutions. Since the function `solve` is recursive, this backtracking asks the `for`-loop in line 13 to look for another solution.

4. Note that we have to enclose the call to `solve` in line 4 in a `check` statement. The reason is that the function `solve` will never return anything. Instead, it will continue to add solutions to the set `all`. If this is no longer possible, the call to `solve` will backtrack in line 19. The `check` statement in the procedure `allSolutions` is meant to catch the corresponding fail-exception.

5. The implementation of the function `testNext` has not changed from the previous program.

```
1  allSolutions := procedure(n) {
2      all := {};
3      check {
4          solve([], n, all);
5      }
6      return all;
7  };
8  solve := procedure(queens, n, rw all) {
9      if (#queens == n) {
10         all += { queens };
11         backtrack;
12     }
13     for (x in {1 .. n} - {i : i in queens}) {
14         check {
15             testNext(queens, x);
16             solve(queens + [x], n, all);
17         }
18     }
19     backtrack;
20 };
```

Figure 6.4: Computing all solutions of the n queens puzzle.

This program finds all 92 solutions of the 8 queens puzzle.

The keyword `check` can be used with an additional optional branch. In this case the complete `check` block has the form

```
check {
   stmntList
} afterBacktrack { body }
```

Here, *body* is a list of statements that is executed if there is a call to `backtrack` in *stmntList*. For example, the code

```
check {
    print(1);
    backtrack;
    print(2);
} afterBacktrack {
    print(3);
}
```

prints the number 1 and 3.

6.2.2 The Zebra Puzzle

There are many different versions of the *zebra puzzle*. The version below is taken from *Wikipedia*. The puzzle is the following:

1. There are five houses.
2. The Englishman lives in the red house.

3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.
16. Who drinks water?
17. Who owns the zebra?

In order to solve the puzzle, we also have to know the following facts:

1. Each of the five houses is painted in a different color.
2. The inhabitants of the five houses are of different nationalities,
3. they own different pets,
4. they drink different beverages, and
5. they smoke different brands of cigarettes.

In order to solve this problem we use a generic approach and first implement a general function that can be used to solve *constraint satisfaction problems*. Figure 6.5 on page 85 shows the implementation of the function `generateAndTest` that can be used to solve a constraint satisfaction problem via backtracking. This function is called with three arguments.

1. The first argument `constraints` is a list of constraints. Each of the elements in this list are, in turn, lists of either two or three items. These lists have the form

   ```
   [var, assign, condition],
   ```

   where the last item `condition` may be missing. These items have the following interpretation.

   (a) `var` is a string that is interpreted as the name of a variable.

   (b) `assign` is a string that can be interpreted as a SetlX statement. Typically, this will be an assignment statement. Another common case will the statement "`skip`" that does nothing.

   (c) `condition` is a string that can be parsed as a logical expression in SetlX. If the condition is missing it is implicitly interpreted as the trivial condition `true`.

For example, the constraint "*The Englishman lives in the red house.*" can be modeled by the constraint

```
["english", "red := english", true].
```

This constraint introduces the two variables "`english`" and "`red`". Due to the assignment

"`red := english`",

these two variables are required to have the same value. Since there are no further constraints, the condition has the value `true`. Hence, we could also have dropped this part of the constraint and thereby shorten it to

```
["english", "red := english"].
```

2. `values` is a list of the possible values for those variables that are introduced via the first argument `constraints`.
3. `result` is a string that, when evaluated as an expression, returns the solution of the constraint satisfaction problem.

```
1  generateAndTest := procedure(constraints, values, result) {
2      stmnt := "procedure() {";
3      n     := #constraints;
4      for (i in [1..n]) {
5          if (#constraints[i] == 3) {
6              [var, assign, condition] := constraints[i];
7          } else {
8              [var, assign] := constraints[i];
9              condition := true;
10         }
11         stmnt += "for ($var$ in $values$) {";
12         if (assign != "skip") {
13             stmnt += assign + ";";
14         }
15         stmnt += "check {";
16         if (condition != true) {
17             stmnt += "if (!($condition$)) { backtrack; }";
18         }
19     }
20     stmnt += "return $result$;";
21     stmnt += "}" * (2*n+1);
22     return eval(stmnt+"()");
23 };
```

Figure 6.5: The function `generateAndTest`.

It is easiest to understand how the arguments of `generateAndTest` are used by inspecting an example. Suppose we are looking for natural numbers $a, b, c \in \{1, \cdots, 5\}$ such that both

$$b = a + 1 \quad \text{and} \quad a^2 + b^2 = c^2$$

holds. Then we can define the parameters of the function `generateAndTest` as follows:

1.
```
constraints := [ ["a", "b := a + 1", "b <= 5"],
                 ["c", "skip", "a**2 + b**2 == c**2"] ];
```

2. `values := {1..5};`
3. `result := "[a, b, c]";`

Calling `generateAndTest` with these arguments will return the value `[3,4,5]`. It is easy to check that the assignments $a := 3$, $b := 4$, and $c := 5$ solve the given problem.

In order to understand the mechanics of `generateAndTest` we have to know that this function works in two steps.

1. In the first step, the function generates a string that can be parsed and evaluated as a SetlX procedure. This string is called `stmnt` in Figure 6.5 on page 85. Figure 6.6 on page 86. shows how this string looks for the example discussed above. From the example we can see that every constraint is translated into a `for` loop. This for loop iterates over all elements of the argument `values`. Inside this loop we first have the assignment `assign`, which is then followed by the `condition`. Note, however, that this condition is encapsulated in a `check` statement. If the `condition` is not satisfies, we have to `backtrack` so that the for loop can try the next value for the given variable. We can also see that the second contsraint is nested in the first constraint.
2. In the second step, the generated procedure is evaluated. This happens through the call of the function `eval` in line 22 of Figure 6.5.

The kind of programming that is exhibited here is often called *higher-order programming*.

```
1  procedure() {
2    for (a in [1, 2, 3, 4, 5]) {
3      b := a + 1;
4      check {
5        if (!(b <= 5)) { backtrack; }
6        for (c in [1, 2, 3, 4, 5]) {
7          check {
8            if (!(a**2 + b**2 == c**2)) { backtrack; }
9            return [a,b,c];
10         }
11       }
12     }
13   }
14 }
```

Figure 6.6: The statement generated by `generateAndTest`.

We proceed to discuss the details of the generation of the string `stmnt` that is later evaluated.

1. The string `stmnt` that is to contain the definition of the generated code is created in line 2.
2. In line 3 we define `n` to be the number of elements of the list `constraints`.
3. Therefore, the `for` loop in line 4 effectively iterates over all constraints in the list `constraints`.
4. Since there maybe constraints where the `condition` is missing, the `if` statement in line 5 checks whether the ith constraint is a list of two or three elements. Then, the variables `var`, `assign`, and `condition` are set to the first, second, and third element of the ith constraint. In case there are only two elements, the variable `condition` is set to the trivial test `true`.
5. Next, we generate the `for` loop that tries the different elements of `values` for the variable `var` of the ith constraint.

6. If there is are real assignment, i.e. if `assign` is different from the string "`skip`", this assignment is the first statement in the `for` loop that is generated.

7. Finally, if the test `condition` is non-trivial, an `if` statement is added that checks if `condition` is satisfies and that backtracks if the condition is violated.

8. Since the generated procedure takes no arguments, we can evaluate it by appending the string "`()`" to the string `stmnt` and then calling the function `eval` on the resulting string.

Finally, we are ready to solve the zebra puzzle. Figure 6.7 on page 88 shows how to set the parameters of the function `generateAndTest` to solve the zebra puzzle. The idea is to model the colors of the houses, the nationalities of their inhabitants, the beverages, the cigarette brand, and the pets by numbers from the set $\{1, 2, 3, 4, 5\}$. For example, if the variable "`red´´`" has the value 3, then this is interpreted as the fact that the third house is red. Let us discuss the details of this program line by line.

1. In line 1 we define the auxiliary function `nextTo`. This function checks whether the houses given to it as arguments are next to each other. Now a is next to b if either a is immediately to the left of b or if a is immediately to the right of b. In the first case we have that $b = a + 1$, in the second case we have $a = b + 1$.

2. The function `distinct`, which is given in line 2 checks whether the five arguments given to it are all distinct. This is the case if and only if the set $\{a, b, c, d, e\}$ contains exactly five elements.

3. Since we model the houses by the number from 1 to 5, the set of possible values for all variables occurring in the zebra puzzle is the set $\{1, 2, 3, 4, 5\}$. This explains line 3.

4. Next, the different statements of the zebra puzzle are translated into a list of constraints. For example, the constraint that the Englishman lives in the red house is translated into the list

   ```
   ["english", "red := english"]
   ```

   This constraint introduces the variable "`english`" and furthermore guarantees that the variable "`red`" has the same value as the variable "`english`".

   As another example, let us explore how the statement

   "Kools are smoked in the house next to the house where the horse is kept."

   is translated into a constraint in line 26. Since the variable "`kools`" has already been introduced in line 18, we only need to introduce the variable "`horse`" to model this statement. The fact that the horse is in the house next to the house where Kools are smoked is specified via the condition

   ```
   nextTo(horse, kools).
   ```

5. Some of the constraints have a condition that uses the function `distinct`. Let us explore the use of this function for the nationalities. Line 33 introduces the variable "`japanese`". Since this is the last variable that introduces a nationality, we can now evaluate the condition that the nationalities are all different via the use of the function `distinct`. Since the constraint in line 33 also introduces the variable "`parliaments`", which is the last of the cigarette brands, we also have attached the condition that all cigarette brands are different,

   The general idea is to attach a condition as soon as all variables mentioned in this condition have been introduced and hence are defined. As another example, line 40 introduces the variable "`water`". Since this is the last of the beverages, the condition

   ```
   "distinct(water, tea, milk, orange, coffee)"
   ```

   is attached to the constraint introducing the variable "`water`".

```
1  nextTo   := procedure(a, b)          { return b == a+1 || a == b+1;  };
2  distinct := procedure(a, b, c, d, e) { return #{a, b, c, d, e} == 5; };
3  values := [1..5];   // 1. There are five houses
4  constr :=
5  [ //  2. The Englishman lives in the red house.
6    ["english", "red := english"],
7    //  3. The Spaniard owns the dog.
8    ["spaniard", "dog := spaniard"],
9    //  4. Coffee is drunk in the green house.
10   ["coffee", "green := coffee"],
11   //  5. The Ukrainian drinks tea.
12   ["ukrainian", "tea := ukrainian"],
13   //  6. The green house is immediately to the right of the ivory house.
14   ["ivory", "skip", "green == ivory + 1"],
15   //  7. The Old Gold smoker owns snails.
16   ["oldGold", "snails := oldGold"],
17   //  8. Kools are smoked in the yellow house.
18   ["kools", "yellow := kools"],
19   //  9. Milk is drunk in the middle house.
20   ["milk", "skip", "milk == 3"],
21   // 10. The Norwegian lives in the first house.
22   ["norwegian", "skip", "norwegian == 1"],
23   // 11. Chesterfields are smoked next to the fox.
24   ["chesterfields", "skip"],
25   ["fox", "skip", "nextTo(chesterfields, fox)"],
26   // 12. Kools are smoked in the house next to the horse.
27   ["horse", "skip", "nextTo(horse, kools)"],
28   ["zebra", "skip", "distinct(fox, horse, snails, dog, zebra)"],
29   // 13. The Lucky Strike smoker drinks orange juice.
30   ["luckies", "orange := luckies"],
31   ["water", "skip", "distinct(water, tea, milk, orange, coffee)"],
32   // 14. The Japanese smokes Parliaments.
33   ["japanese", "parliaments := japanese",
34    "distinct(norwegian, ukrainian, english, spaniard, japanese) &&
35     distinct(kools, chesterfields, oldGold, luckies, parliaments)"],
36   // 15. The Norwegian lives next to the blue house
37   ["blue", "skip",
38    "nextTo(norwegian, blue) && distinct(yellow, blue, red, ivory, green)"],
39   // 16. Who drinks water?
40   ["water", "skip", "distinct(water, tea, milk, orange, coffee)"],
41   // 17. Who owns the zebra?
42   ["zebra", "skip", "distinct(fox, horse, snails, dog, zebra)"]
43 ];
44 result := "[zebra, water]";
45 [zebra, water] := generateAndTest(constr, values, result);
```

Figure 6.7: Coding the zebra problem.

6. Since we are only interested in the house where the zebra lives and the house where the inhabitant drinks water, the variable result is defined as the string

```
"[zebra, water]"
```

in line 44.

7. Finally, line 45 calls the function `generateAndTest` to solve the puzzle.

When the program runs we find out that the zebra is in house number 5 while the inhabitant of the first house drinks water.

Chapter 7

Vectors and Matrices

Certain applications, for example data mining and machine learning, require an efficient support of both vectors and matrices. Although vectors and matrices could easily be implemented in SetlX, this would not be very efficient because of the overhead of the interpreter loop. Therefore, SetlX supports both vectors and matrices natively. This support is based on the *Java* library *Jama*. This library is the result of a joint effort of Mathworks and the National Institute of Standards and Technology. *Jama* has been integrated into SetlX by Patrick Robinson. This library provides the basic means for computations involving matrices and vectors. In the following exposition we assume that the reader has some familiarity with linear algebra.

7.1 Vectors

SetlX supports real valued vectors of arbitrary dimensions. Conceptually, a vector can be viewed as a list of floating point numbers. A vector is constructed from a list of numbers via the function `la_vector` as follows:

```
v := la_vector([1/2, 1/4, 1/5]);
```

When executed, this statement yields the result

```
<<0.5 0.25 0.2>>.
```

This result shows that the fractions in the argument list have been converted to floating point numbers. Conceptually, the vector `v` is a column vector. Therefore, mathematically `v` would be written as

$$\begin{pmatrix} 0.5 \\ 0.25 \\ 0.2 \end{pmatrix}.$$

Instead of using the function `la_vector`, we could also have used the command

```
v := <<1/2 1/4 1/5>>;
```

to define the vector `v`. This shows that the operators "<<" and ">>" can be used to define a vector. **Note** that the components of a vector are **not** separated by the character "`,`" but **rather** are separated by white space.

SetlX support the basic arithmetic operations that are defined for vectors. If `a` and `b` are two vectors, then

```
a + b
```

computes the sum of `a` and `b`, while

```
a - b
```

computes their difference. Concretely, if we have

```
a = <<a_1 ··· a_n>>   and   b = <<b_1 ··· b_n>>,
```

then "a+b" and "a-b" are defined componentwise:

```
a + b := <<(a_1+b_1) ··· (a_n+b_n)>>   and   a - b := <<(a_1-b_1) ··· (a_n-b_n)>>.
```

For example, if we define

```
a := <<1 2 3>>;  b := <<4 5 6>>;
```

then the expressions "a + b" and "a - b" yield the results

```
<<5.0 7.0 9.0>>   and   <<-3.0 -3.0 -3.0>>,
```

respectively. Additionally, the shortcut assignment operators "+=" and "-=" are available for vectors. They work as expected. Note that for the expressions "a + b" and "a - b" to be defined, a and b have to have **the same number of elements**.

Vectors support *scalar multiplication*. If α is a real number and

```
v = <<v_1 ··· a_n>>
```

is an n-dimensional number, then the scalar products α * v and v * α are both defined as the vector

```
<<(α * v_1) ··· (α * v_n)>>.
```

For example, if a is defined as above, then the expression

```
1/2 * a
```

yields the vector

```
<<0.5 1.0 1.5>>.
```

It does not matter whether we multiply the scalar from the left or from the right, so the expression

```
a * (1/2)
```

yields the same result. Note that we had to put the fraction 1/2 in parenthesis. The reason is that the expression "a * 1/2" is parsed as "(a * 1) / 2" and division of a vector by a scalar is not defined. If v is a vector and n is a number, then the assignment statement

```
v *= n;
```

is equivalent to the statement

```
v = v * n;
```

For two n-dimensional vectors

$$\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad \text{and} \quad \vec{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix},$$

the *dot product*, which is also known as the *scalar product*, is defined as

$$\vec{x} \cdot \vec{y} := \sum_{i=1}^{n} x_i \cdot y_i.$$

Students often confuse scalar multiplication and the scalar product. Therefore, we have decided to use the same operator for both products: If a and b are two vectors of the same dimension, the expression

```
a * b
```

yields their scalar product. For example, using the definition of a and b given above, the expression "a * b" yields the result 32 since

$$1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32.$$

Finally, SetlX supports the *cross product*. For two three-dimensional vectors

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \text{and} \quad \vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix},$$

the cross product $\vec{x} \times \vec{y}$ is defined as

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \times \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_2 \cdot y_3 - x_3 \cdot y_2 \\ x_3 \cdot y_1 - x_1 \cdot y_3 \\ x_1 \cdot y_2 - x_2 \cdot y_1 \end{pmatrix}.$$

If a and b are both 3-dimensional vectors, then the expression

```
a >< b
```

computes the cross product of a and b. For example, if a and b are the vectors defined at the beginning of this section, then the expression

```
a >< b
```

yields the result

```
<<-3.0 6.0 -3.0>>
```

since we have

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \times \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 2 \cdot 6 - 3 \cdot 5 \\ 3 \cdot 4 - 6 \cdot 1 \\ 1 \cdot 5 - 2 \cdot 4 \end{pmatrix} = \begin{pmatrix} -3 \\ 6 \\ -3 \end{pmatrix}.$$

Note that the cross product of two vectors is only defined iff both a and b are three-dimensional. There are generalisations of the cross product for n-dimensional vectors. However, in that case the cross product is no longer a binary operator but rather takes $n - 1$ arguments. These generalisations are not supported in SetlX.

Vectors provide the same access operations as list. Therefore, to extract the i-th component of a vector v we can use the expression "v[i]". Furthermore, the operator "#" returns the dimension of a given vector. Therefore, given the vector a defined above, the expression #a returns the value 3.

7.2 Matrices

SetlX supports real valued matrices. The function la_matrix can be used to construct a matrix from a list of list of numbers. For example, the assignment

```
a := la_matrix([[1,2],[3,4]]);
```

constructs a matrix that can be written as

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

We can see that the inner lists used as an argument to the function la_matrix correspond to the rows of the resulting matrix. In SetlX, the matrix is printed as

```
<< <<1.0 2.0>> <<3.0 4.0>> >>.
```

Note that a matrix is written as a list of vectors where instead of the square brackets "[" and "]" the

tokens "<<" and ">>" are used as opening and closing delimiters. However, whereas the elements of a list are separated by commas, the row vectors making up a matrix are separated by white space. Instead of using the function `la_matrix` we could have defined the matrix `a` using the following command:

```
a := << <<1.0 2.0>> <<3.0 4.0>> >>;
```

The function `la_matrix` can also be called with a single vector as its argument. If v is an n-dimensional vector, then

`la_matrix(`v`)`

interprets this vector as a column vector and turns it into an $n \times 1$ matrix. For example, the statement

```
a := la_matrix(<<1 2 3>>);
```

yields the result

```
<< <<1.0>> <<2.0>> <<3.0>> >>.
```

This result shows that `a` is a matrix that consists of three rows that are themselves vectors of length 1.

Similar to vectors, matrices can be added and subtracted using the operators "+" and "−". If we define

```
a := << <<1 2>> <<3 4>> >>;   and   b := << <<5 6>> <<7 8>> >>;
```

then the expressions "`a + b`" and "`a - b`" work componentwise and yield the results

```
<< <<6.0 8.0>> <<10.0 12.0>> >>   and   << <<-4.0 -4.0>> <<-4.0 -4.0>> >>.
```

In addition to "+" and "−", the assignment operators "+=" and "-=" are supported for matrices and work as expected.

Furthermore, matrices support scalar multiplication in the same way as vectors. For example, if the matrix `a` is defined as abobe, then the expressions "`2 * a`" and "`a * 2`" both yield the result

```
<< <<2.0 4.0>> <<6.0 8.0>> >>.
```

Next, SetlX supports matrix multiplication. If

$\mathtt{a} = (\mathtt{a}_{i,j})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$ is an $m \times n$ matrix

and

$\mathtt{b} = (\mathtt{b}_{j,l})_{\substack{1 \leq j \leq n \\ 1 \leq l \leq k}}$ is an $n \times k$ matrix,

then the matrix product `a * b` is defined as the matrix

$$\left(\sum_{j=1}^{n} \mathtt{a}_{i,j} \cdot \mathtt{b}_{j,l}\right)_{\substack{1 \leq i \leq n \\ 1 \leq l \leq k}}.$$

For example, given the definitions of `a` and `b` shown above, the expression "`a * b`" yields the result

```
<< <<19.0 22.0>> <<43.0 50.0>> >>.
```

Furthermore, if `a` is an $m \times n$ matrix and `v` is an n dimensional vector, then the expression

```
a * v
```

is computed as a matrix multiplication, where v is interpreted as an $n \times 1$ matrix. In this case, the resulting $m \times 1$ matrix is then automatically converted back into an m dimensional vector.

In addition to matrix multiplication, SetlX also support exponentiation of a square matrix by an integer number. For example,

```
a ** 2;
```

returns the square of `a`, while

```
a ** -1
```

return the *inverse* of a matrix, provided the matrix is not *singular*. If the matrix a is singular, evaluation of the expression a ** n raises an exception if the exponent n is negative.

Matrices can be *transposed* via the postfix operator "!". For example, using the definition of the matrix a shown above, the expression "a!" yields

```
<< <<1.0 3.0>> <<2.0 4.0>> >>.
```

If a is an $m \times n$ matrix, the expression "#a" yields the dimension m. In order to compute the dimension n, we can use the expression

```
#a[1].
```

The reason is that a[1] returns the first row of the matrix a as a list and the length of this list is the dimension n. In order to access the element in row i and column j of matrix a we can use the expression

```
a[i][j].
```

7.3 Numerical Methods for Matrices and Vectors

In this section we will discuss the numerical methods that are provided. These methods are inherited from *Jama*, which is a *Java* matrix package. The names of all methods inherited from *Jama* start with "la_". This is short for linear algebra.

7.3.1 Computing the Determinant

If a is a square matrix, the expression

```
la_det(a)
```

computes the *determinant* of a. For example, if we define

```
a := << <<1 2>> <<3 4>> >>;
```

then the expression "la_det(a)" yields the result -2. The determinant can be used to check whether a matrix is invertible as a matrix is invertible if and only if the determinant is different from 0. However, note that due to rounding errors the result of the expression "la_det(a)" might be a small non-zero number even if the matrix a is really singular.

7.3.2 Solving a System of Linear Equations

A system of linear equations of the form

$$\mathtt{a} \cdot \mathtt{x} = \mathtt{b}$$

where a is a square matrix $n \times n$ matrix and b is an n-dimensional vector can be solved using the expression

```
la_solve(a, b).
```

For example, to solve the system of equations

$$\begin{aligned} 1 \cdot x + 2 \cdot y &= 5 \\ 3 \cdot x + 4 \cdot y &= 6 \end{aligned}$$

we define

```
a := << <<1 2>> <<3 4>> >>;   and   b := <<5 6>>;
```

Then, the solution is computed via the expression

```
la_solve(a, b).
```

Note that this expression throws an exception if the given system of equations is not solvable.

Note that it is also posible to solve a system of equations that is *overdetermined*. For example, assume that the system of equations is given as

$$\begin{aligned} 1 \cdot x + 2 \cdot y &= 1 \\ 2 \cdot x + 3 \cdot y &= 2 \\ 3 \cdot x + 4 \cdot y &= 3 \end{aligned}$$

In order to compute the solution $\vec{x}$ that minimizes the error $\|a \cdot \vec{x} - \vec{b}\|_2$ we can define `a` and `b` via

```
a := << <<1 2>> <<2 3>> <<3 4>> >>;   and   b := <<1 2 3>>;
```

and then call the function `la_solve` as

```
la_solve(a, b);
```

In this case, `la_solve` computes the result

```
<<0.9999999999999982 1.4430967688835178E-15>>
```

which is pretty close to the exact result $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

When solving a system of linear equations, care has to be taken that the solution is well defined. To this end, SetlX provides the function `la_cond` that computes the condition number of a given matrix. Let us describe the problem via an example borrowed from

http://nm.mathforcollege.com/mws/gen/04sle/mws_gen_sle_spe_adequacy.pdf.

Assume the matrix a and the vector $\vec{b}$ are defined as

$$a := \begin{pmatrix} 1.000 & 2.000 \\ 2.000 & 3.999 \end{pmatrix} \quad \text{and} \quad \vec{b} := \begin{pmatrix} 4.000 \\ 7.999 \end{pmatrix}.$$

It can easily be verified that the solution $\vec{x}$ to the system of equations

$$\vec{b} = a \cdot \vec{x}$$

is given as

$$\vec{x} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

However, let us assume that the vector $\vec{b}$ is distorted into the vector $\vec{c}$ that is given as

$$\vec{c} := \begin{pmatrix} 4.001 \\ 7.998 \end{pmatrix}.$$

If we solve the system

$$\vec{c} = a \cdot \vec{x},$$

the solution for $\vec{x}$ becomes

$$\vec{x} = \begin{pmatrix} -3.999 \\ 4.000 \end{pmatrix}.$$

We see that a tiny change in the right hand side of the system of equations has caused a big change in the solution $\vec{x}$. The reason is that the matrix `a` is *ill-conditioned*. We can verify this using the function `la_cond`: The expression

```
la_cond(<< <<1.000 2.000>> <<2.000 3.999>> >>);
```

yields the result

```
24992.000959995028.
```

This shows that small errors in the right hand side of a system of linear equations involving the matrix a are multiplied by a factor of nearly $25\,000$. Our recommendation is that if you ever have to solve a system of equations, you should first check whether the system is well-conditioned by computing the condition number of the matrix associated with the system. This condition number tells us by how much an error in the right hand side of the system of equations is magnified when we compute the solution. In particular, if the condition number is so big that it will magnify known uncertainties in the right hand side of the equations to an extend that the result is meaningless, then it is not possible to solve the given system of equations in a meaningful way.

7.3.3 The Singular Value Decomposition and the Pseudo-Inverse

The function `la_svd` can be used to compute the *singular value decomposition* of a given matrix a. For a given $m \times n$ matrix a, the expression

```
la_svd(a)
```

returns a list of the form

```
[u, s, v]
```

where u is an orthogonal $m \times m$ matrix, s is an $m \times n$ diagonal matrix, and v is an $n \times n$ orthogonal matrix such that

$$a = u \cdot s \cdot v^{\mathrm{T}},$$

where v^{T} denotes the transpose of the matrix v. In practice, the singular value decomposition is used to compute the *pseudo-inverse* of a singular matrix. However, this can be done directly as for a matrix a the call

```
la_pseudoInverse(a);
```

computes the pseudo-inverse of a.

7.3.4 Eigenvalues and Eigenvectors

In order to compute the eigenvalues and eigenvectors of a square matrix, SetlX provides the functions

```
la_eigenValues   and   la_eigenVectors.
```

Both of these function take a single argument a, where a must be a square matrix. The function `la_eigenValues` returns a list of the eigenvalues of a. The function call `la_eigenVectors(a)` returns a list of the eigenvectors of the matrix a. For example, in order to compute the eigenvalues and eigenvectors of the matrix

$$\mathtt{a} := \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}$$

we can execute the following commands:

```
1  a        := << <<1 2>> <<2 3>> >>;
2  [l1, l2] := la_eigenValues(a);
3  [x1, x2] := la_eigenVectors(a);
```

Then, the eigenvalues will be computed as

```
l1 = -0.23606797749978958   and   l2 = 4.23606797749979
```

In this case, the exact values of these eigenvalues are

$$\texttt{l1} = 2 - \sqrt{5} \quad \text{and} \quad \texttt{l2} = 2 + \sqrt{5}.$$

The eigenvectors are computed as

```
x1 = <<-0.8506508083520399 0.5257311121191336>>   and
x2 = <<0.5257311121191336 0.8506508083520399>>.
```

The eigenvectors computed are *normalized*, that is the length of these vectors is 1. Furthermore, as the matrix `a` happens to be symmetric, i.e. we have `a = a!`, the eigenvectors are orthogonal to each other.

Observe that not every square $n \times n$ matrix is *diagonalizable* and hence has n different eigenvectors. In case the matrix is not diagonalizable in the real numbers, the functions `la_eigenValues` and `la_eigenVectors` both throw an exception. Note, however, that every *symmetric* matrix is diagonalizable. Therefore, if the matrix `a` is symmetric, neither of the two calls

```
la_eigenValues(a)   or   la_eigenVectors(a)
```

will throw an exception.

Chapter 8

Plotting

SetlX provides a small plotting library which is based on JFreeChart. The SetlX-interface to *JFreeChart* has been implemented by Arne Röcke and Fabian Wohriska. The plotting library in SetlX supports three different kinds of plots:

1. The library can be used to plot curves. Essentially, when it comes to curves, there are two kinds of curves:

   (a) Curves that are given as the *graph of a function*. In this case, the y-coordinate is given as an expression involving the x-coordinate. For example, the *parabola* satisfies the equation $y = x^2$.

   (b) Curves that are given via a *parametric equations*. For example, a *circle* of radius 1 can be specified via the parametric equations

   $$x = \cos(t) \quad \text{and} \quad y = \sin(t), \quad \text{where } t \in [0, 2 \cdot \pi].$$

   These equations specify a circle because $x^2 + y^2 = \cos^2(x) + \sin^2(x) = 1$.

   Both of these types of curves can be plotted by SetlX.

2. The library can be used for *scatter plots*.

3. The library can be used to plot *statistical charts* such as *bar charts* or *pie charts*.

We will discuss the plotting of curves, scatter plots, and charts in different sections.

It should be noted that the figures presented in this tutorial had to be scaled down to fit the paper size. This scaling has resulted in a considerable loss of their artistic quality.

8.1 Plotting Curves

In this section, we discuss the plotting of mathematical curves. The section is structured as follows.

1. First, we discuss curves that are given as the graph of a function. As an example, we show how to plot the parabola.

2. Next, we discuss curves that are specified via parametric equations. As a simple example, we show how to plot a circle. A second example discusses the plotting of a *hypotrochoid*.

8.1.1 Plotting the Parabola

In order to start drawing, we first have to create a canvas on which we can draw something. This is achieved with the command

```
c := plot_createCanvas();
```

Note that all functions related to plotting start with the prefix "`plot_`". The function

"`plot_createCanvas`"

takes one optional argument, which is a string. This string adds a title to the canvas. For example, the command

```
c := plot_createCanvas("The Parabola");
```

creates a canvas with the title "*The Parabola*" and returns a reference to this canvas. However, as long as nothing is painted on the canvas, the canvas remains invisible. Let us draw a parabola on the canvas created previously. This is done using the command

```
plot_addGraph(c, "x*x", "f(x) = x*x", [0, 0, 255]);
```

The resulting figure is shown in Figure 8.1 on page 100. The arguments provided to the function `plot_addGraph` are interpreted as follows:

1. `c` refers to the canvas on which the specified function is to be drawn.
2. `"x*x"` specifies the function. The string given as argument is evaluated for different values of the variable x and the resulting function is then plotted.
3. `"f(x) = x*x"` is the string that is used as the legend of the plot. This string is printed verbatim below the plot.
4. `[0, 0, 255]` specifies the color used to plot the specified function. This color is specified via the RGB color model. Hence, in this case the parabola is plotted using the color blue.

   This parameter is optional. If it is not specified, the curve will be plotted in black.
5. There is a final optional parameter to the function `plot_addGraph` that is not used in this example. This parameter needs to be a Boolean value. If this value is `true`, then the area under the curve is filled with the specified color.

In Figure 8.1 the parabola is plotted for arguments ranging from $x = -10$ up to $x = 10$. Suppose that we would prefer to draw the parabola in the region where x runs from -4 to $+4$. We can use the function `plot_modscale` to achieve this as follows:

```
plot_modScale(c, [-4, 4], [-1,17]);
```

The parameters used by `plot_modScale` are as follows:

1. `c` specifies the canvas whose scale is to be modified.
2. `[-4, 4]` specifies that the x-coordinate is to be varied between -4 and $+4$.
3. `[-1, 17]` specifies that the y-coordinate extends from -1 to $+17$.

The resulting parabola is shown in Figure 8.2 on page 101.

Let us assume we want both the x-axis and the y-axis to be visible. For this purpose, the function `plot_addListGraph` comes in handy. To plot the x-axis we use the command

```
plot_addListGraph(c, [[-4,0], [4,0]], "x-axis", [0,0,0]);
```

Here, the parameters are as follows:

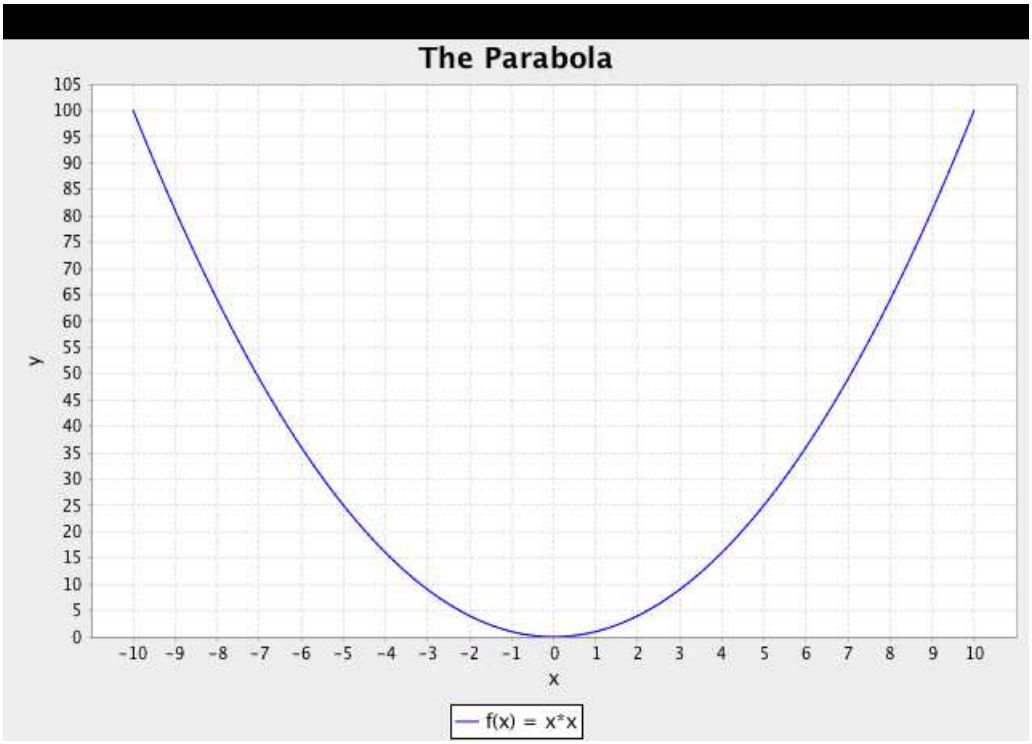

Figure 8.1: A plot of the parabola $x \mapsto x^2$.

1. c specifies the canvas where we want to draw the line.
2. `[[-4,0], [4,0]]` is a list of points. Each point is specified as a pair of x- and y-coordinates. These points are then connected by straight lines. In this case, as there is only one pair of points there is only one line.
3. "x-axis" specifies the legend that is used for this line.
4. `[0,0,0]` specifies the color of the line. This parameter is optional. If the color is not specified, then black is the default.

Let us add the y-axis using the command

```
plot_addListGraph(c, [[0,-1], [0,17]], "y-axis");
```

Furthermore, let us add a tangent to the parabola at the point $\langle 1, 1 \rangle$. Since the tangent line at this point is given by the formula

$$x \mapsto 2 \cdot (x - 1) + 1,$$

the tangent line passes the points $\langle -4, -9 \rangle$ and $\langle 4, 7 \rangle$, Therefore, we can plot the tangent using the command

```
plot_addListGraph(c, [[-4,-9], [4,7]], "x |-> 2*(x-1)+1");
```

Finally, let us mark the point $\langle 1, 1 \rangle$ with a tiny red box using the command

```
plot_addBullets(c, [[1,1]], [255,0,0], 3.0);
```

This command accepts four parameters.

1. c specifies the canvas.

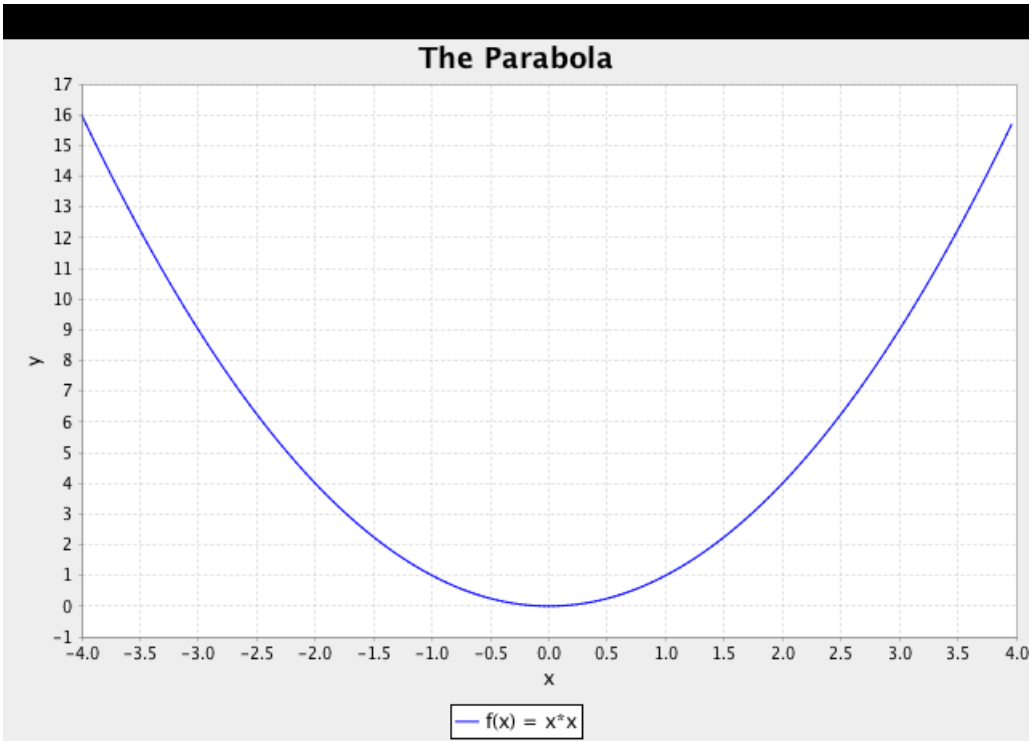

Figure 8.2: A plot of the parabola $x \mapsto x^2$.

2. `[[1,1]]` is a list of points. Each point itself is specified as a pair of x- and y- coordinates. The function plots a square at the location of each of the points.

3. `[255,0,0]` specifies the color.

4. The last parameter specifies the size of each box. This parameter is optional and defaults to 5.0. Note that this parameter has to be specified as a floating point number.

The resulting graphics is shown in Figure 8.3 on page 102.

The program used to create this plot showing the parabola is shown in Figure 8.4 on page 102. This program contains three lines the we have not yet discussed.

1. Line 2 contains a call to the function `sleep`. This call is necessary if we want to run this program from the command line. The reason is that the function `plot_createCanvas` runs concurrently to the rest of the program. However, the invocation of the function `plot_addGraph` in line 3 uses the canvas created in line 1. If the creation of this canvas has not been finished, the *Java* runtime system that is lying beneath SetlX might create a *concurrent modification exception*. By calling `sleep` the execution of the main thread of the program is halted for 100 milliseconds. This should be more than enough for the function `plot_createCanvas` to finish its work.

2. Line 9 exports the plot into a png-file named `parabola.png`. This file is saved into the current directory.

3. Finally, we have to prevent the SetlX interpreter from exiting after it has finished displaying the parabola. This is achieved via the invocation of `get` in line 10. The command `get` waits for the user to enter any string. Therefore, as long as the user does not press ⟨`Enter`⟩, the window showing the parabola remains visible.

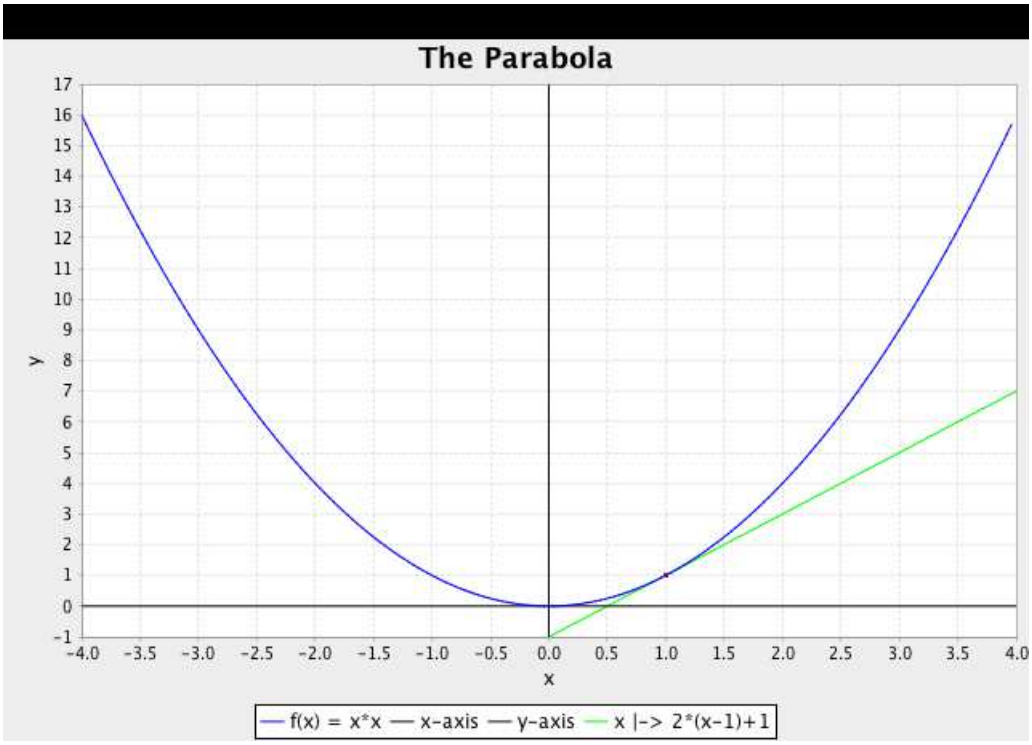

Figure 8.3: A plot of the parabola $x \mapsto x^2$ and a tangent touching it.

```
1  c := plot_createCanvas("The Parabola");
2  sleep(100);
3  plot_addGraph(c, "x*x", "f(x) = x*x", [0, 0, 255]);
4  plot_modScale(c, [-4, 4], [-1,17]);
5  plot_addListGraph(c, [[-4, 0], [4, 0]], "x-axis", [0,0,0]);
6  plot_addListGraph(c, [[ 0,-1], [0,17]], "y-axis");
7  plot_addListGraph(c, [[-4,-9], [4, 7]], "x |-> 2*(x-1)+1", [0,255,0]);
8  plot_addBullets(c, [[1,1]], [255,0,0], 3.0);
9  plot_exportCanvas(c, "parabola");
10 get("Press Enter to continue");
```

Figure 8.4: SETLX commands to plot the parabola.

8.1.2 Plotting Parametric Curves

Next, we show how to plot a parametric curve. As a first example, we show how to plot a circle of radius 1. Figure 8.5 on page 103 shows a SETLX program that can be used to plot a circle. The resulting circle is shown in Figure 8.6 on page 103.

We proceed to discuss the program shown in Figure 8.5 line by line.

1. Line 1 creates the canvas and equips it with a title.

2. Line 2 waits for the canvas to be created.

3. According to the Pythagorean theorem the points $\langle x, y \rangle$ on a unit circle satisfy the equation

```
c := plot_createCanvas("The Unit Circle");
sleep(100);
interval := [0, 2*mathConst("Pi")];
blue     := [0, 0, 255];
plot_addParamGraph(c, "cos(x)", "sin(x)", "circle", interval, blue);
plot_addListGraph(c, [[-2, 0], [2, 0]], "x-axis", [0,0,0]);
plot_addListGraph(c, [[ 0,-2], [0, 2]], "y-axis");
plot_modScale(c, [-1.2, 1.2], [-1.2, 1.2]);
plot_modSize(c, [600, 600]);
plot_exportCanvas(c, "circle");
get("Press Enter to continue");
```

Figure 8.5: The unit circle.

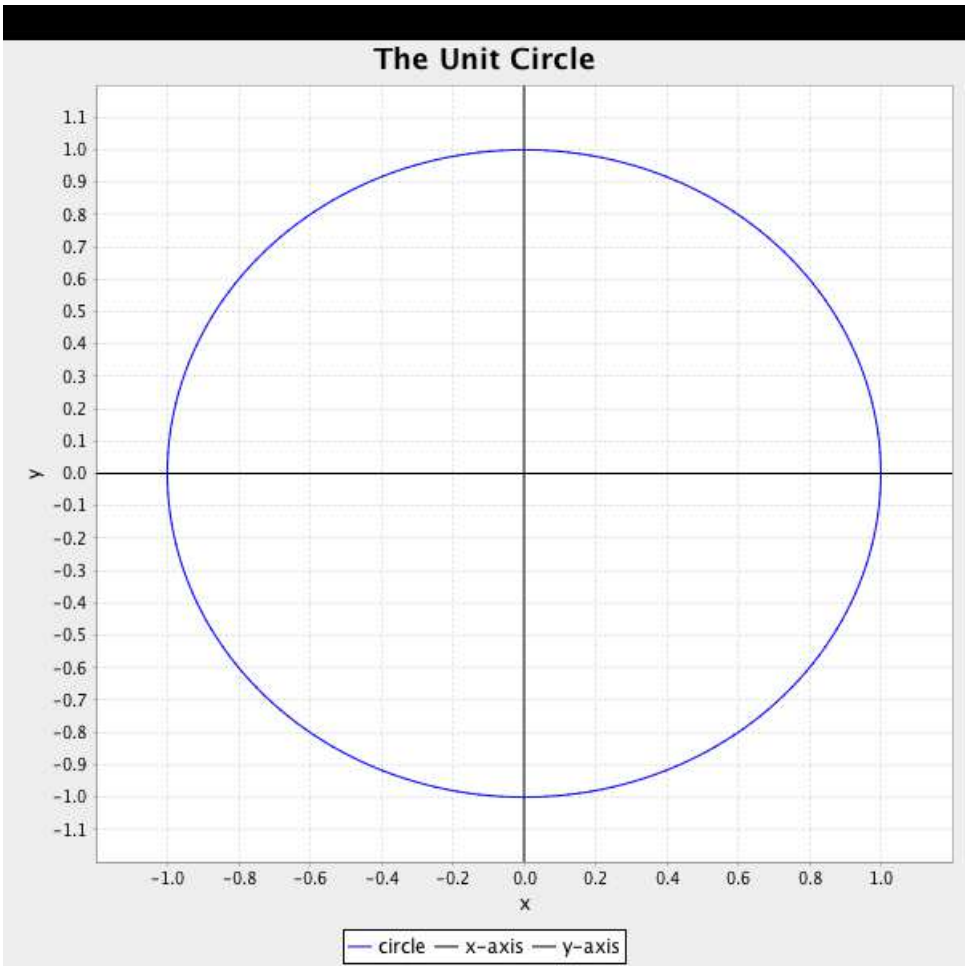

Figure 8.6: A plot of the unit circle satisfying $x^2 + y^2 = 1$.

$$x^2 + y^2 = 1^2.$$

Since we have $\sin^2(\varphi) + \cos^2(\varphi) = 1$, the set of all points on the unit circle is given as

$$\{\langle \cos(\varphi), \sin(\varphi)\rangle : \varphi \in [0, 2 \cdot \pi]\}.$$

Therefore, we specify the variable `interval` as the pair `[0, 2*mathConst("Pi")]` since this variable is used to specify the range over which the parameter has to vary.

4. Line 4 specifies the color.
5. Line 5 calls the function `addParamGraph` to do the actual plotting. The parameters to this function are as follows:
   (a) `c` is the canvas on which the curve is to be plotted.
   (b) `"cos(x)"` specifies the function computing the x-coordinate.
   (c) `"sin(x)"` specifies the function computing the y-coordinate.
   (d) `"circle"` specifies the legend.
   (e) `interval` specifies the range over which the variable `x` that is used in the expressions `"cos(x)"` and `"sin(x)"` has to vary.
   (f) `blue` specifies the color. This argument is optional.
6. Line 6 plots the x-axis.
7. Line 7 plots the y-axis.
8. Line 8 specifies that the region to be plotted ranges from $x = -1.2$ to $x = 1.2$ horizontally and form $y = -1.2$ to $y = 1.2$ vertically.
9. Line 9 changes the size of the plot. By default, a canvas is 800 pixels wide and 600 pixels high. Plotting a circle on a canvas with these dimensions would distort the circle and make it look like an *ellipse*. In order to plot a proper circle we therefore have to ensure that the canvas is quadratic. Hence we change the plotting area to be 600 pixels wide and 600 pixels high.
10. Line 10 saves the generated plot into a file.
11. Line 11 prevents the program from finishing prematurely.

As another example let us write a program for plotting *hypotrochoids*. These curves are described by the equations

$$x = (R-r)\cdot\cos(\varphi) + d\cdot\cos\Big(\varphi\cdot\frac{R-r}{r}\Big) \quad \text{and} \quad y = (R-r)\cdot\sin(\varphi) - d\cdot\sin\Big(\varphi\cdot\frac{R-r}{r}\Big).$$

Here, φ has to run form 0 up to some number Φ such that Φ is a multiple of $2\cdot\pi$ and, furthermore,

$$\Phi\cdot\frac{R-r}{r}$$

has to be a multiple of $2\cdot\pi$, too. Figure 8.7 on page 105 shows a SETLX program that can be used to plot a hypotrochoid. Figure 8.8 on page 106 shows the hypotrochoid where $R = 5$, $r = 3$ and $d = 5$.

```
period := procedure(bigR, smallR) {
    for (i in [1 ..10000]) {
        if (isInteger(i * (bigR - smallR) / (smallR* 2))) {
            return i;
        }
    }
    return 1000;
};
hypotrochoid := procedure(bigR, smallR, d) {
    c := plot_createCanvas("The Hypotrochroid R=$bigR$, r=$smallR$, d=$d$.");
    sleep(100);
    interval := [0, 2.01*mathConst("Pi")*period(bigR, smallR)];
    red      := [255, 0, 0];
    sum      := (bigR - smallR + d) + 0.5;
    xParam   := "(bigR-smallR) * cos(x) + d * cos(x*(bigR-smallR)/smallR)";
    yParam   := "(bigR-smallR) * sin(x) - d * sin(x*(bigR-smallR)/smallR)";
    plot_addParamGraph(c, xParam, yParam, "hypotrochoid", interval, red);
    plot_addListGraph(c, [[-sum,   0], [sum,   0]], "x-axis", [0,0,0]);
    plot_addListGraph(c, [[   0,-sum], [  0, sum]], "y-axis");
    plot_modScale(c, [-sum, sum], [-sum, sum]);
    plot_modSize(c, [600, 600]);
    plot_exportCanvas(c, "hypotrochoid");
    get("Press Enter to continue");
};
```

Figure 8.7: A program for plotting Hypotrochoids.

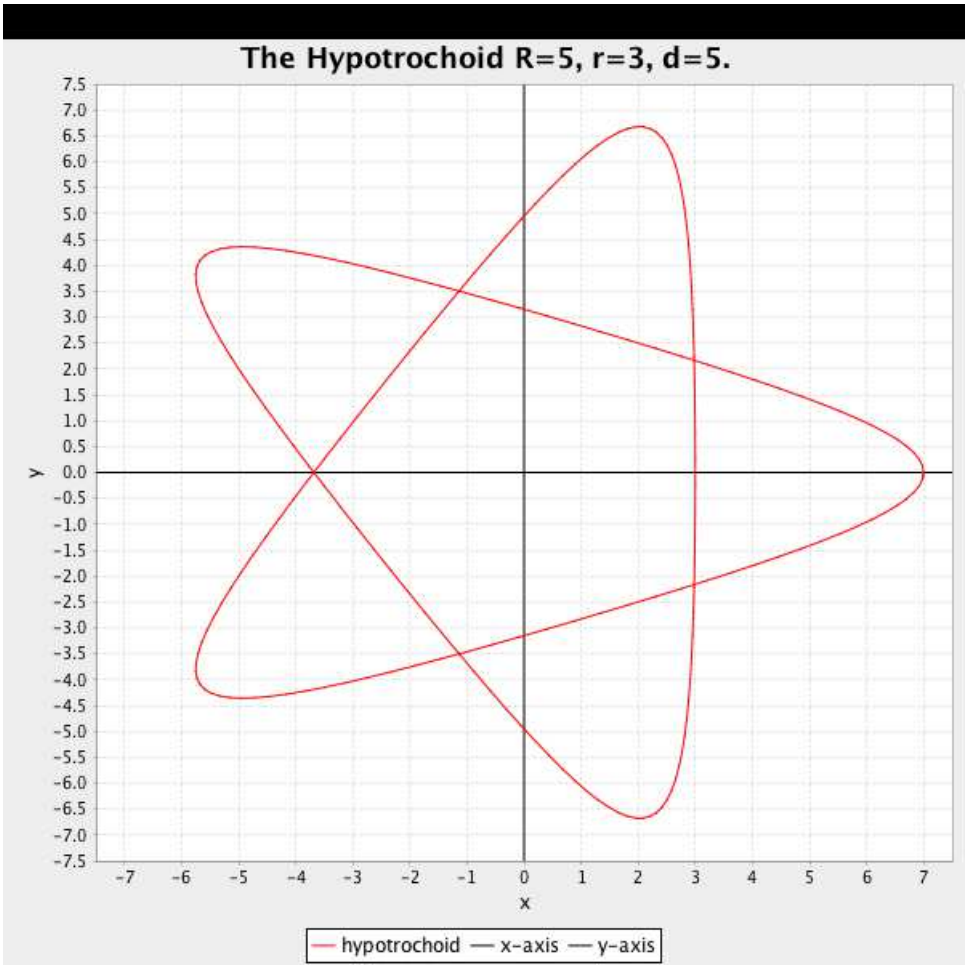

Figure 8.8: A plot of the hypotrochoid satisfying $R = 5$, $r = 3$, $d = 5$.

8.2 Scatter Plots

In order to generate a scatter plot, we can use the function `plot_addBullets`. Figure 8.9 on page 107 shows the implementation of the function `fatherAndSons` that reads a file containing the heights of fathers and their sons and that converts these data into the scatter plot shown in Figure 8.10 on page 108. We proceed to discuss the details of the implementation of the function `fatherAndSons`.

1. Line 2 reads the file "`fathers-and-sons.txt`" and converts this file into a list of strings that is called `data`. Each string corresponds to one line in the file. The first line in the file has the form

   ```
   65.04851 59.77827
   ```

   This line states that the first father-son pair in the data set consists of a father who is 65.04851 inches tall, while his son is only 59.77827 inches tall.[1].

[1] At this point you are probably wondering how it is possible that the height is measured with a precision of 5 digits. Of course, when measuring body heights this is precision is not achievable. The data set I am using here is part of the R project and can be found in the package

```
UsingR
```

2. Line 3 splits every string in the list `data` into a list of two strings. These two strings are the strings representing the height of the father and the height of his son. Therefore, the variable `fsStr` is a list of pairs of strings. These pairs correspond to the lines in the file "`fathers-and-sons.txt`".
3. Line 4 converts these strings into floating point values. This is done using the conversion function `double` that takes a string and converts it into a floating point number.
4. Line 5 creates the canvas.
5. Line 6 takes the pairs of floating point numbers and plots them as boxes of size 2.0. Note that the size has to be specified as a floating point number: Specifying the height as an integer will result in an error message.

   The function `plot_addBullets` takes a list of pairs of the form $[x, y]$ where x specifies the x-coordinate and y specifies the y-coordinate of a data point. These data points are then plotted one by one.
6. As the heights vary between 58 inches and 80 inches, line 7 modifies the scale accordingly.
7. Line 8 modifies the size of the canvas so that the width and the height are the same.
8. Line 9 changes the labeling of the x-axis and of the y-axis using the function `plot_labelAxis`. This function takes three arguments.
   (a) The first argument is the canvas.
   (b) The second argument is the label of the x-axis.
   (c) The third argument is the label of the y-axis.
9. Line 10 saves the scatter plot into a file. The file is saved as a `.png` file.

```
1  fatherAndSons := procedure() {
2      data  := readFile("fathers-and-sons.txt");
3      fsStr := [ split(line, " "): line in data ];
4      fs    := [ [double(x), double(y)] : [x,y] in fsStr ];
5      c     := plot_createCanvas("Heights of Fathers vs. Heights of Sons.");
6      plot_addBullets(c, fs, [0,0,255], 2.0);
7      plot_modScale(c, [58, 80], [58, 80]);
8      plot_modSize(c, [1000, 1000]);
9      plot_labelAxis(c, "heights of fathers in inches",
10                       "heights of sons in inches"    );
11     plot_exportCanvas(c, "fathers-and-sons");
12 };
```

Figure 8.9: How to generate a scatter plot from data stored in a file.

in the variable `father.son`. My best guess is that the data set has suffered from some kind of unit conversion.

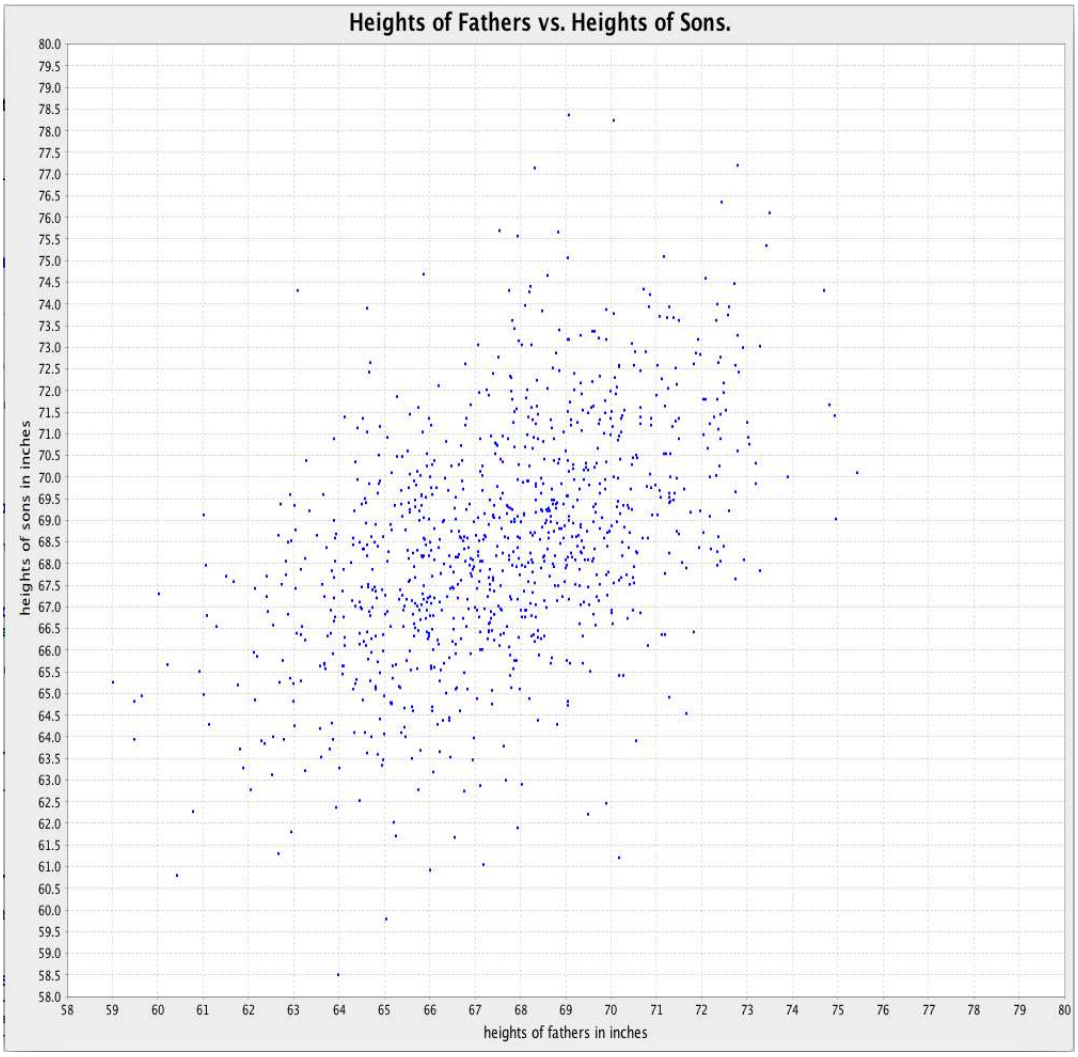

Figure 8.10: A scatter plot showing the heights of fathers and their sons.

8.3 Plotting Statistical Charts

SetlX is able to draw *bar charts* and *pie charts*. We will present each of these charts in a separate subsection.

8.3.1 Bar Charts

Table 8.1 on page 109 shows the population of different regions of the world for various years. This data is more easily comprehensible when presented as a bar char. Figure 8.11 presents a SetlX-program that converts these data into a bar chart. We proceed to discuss this program line by line. The bar chart produced by this program is shown in Figure 8.12 on page 110.

1. Line 1 to line 8 define the data that is to be displayed. The data is stored as a list of lists. The first list specifies the years corresponding to the population counts. The remaining lists each specify the population counts for a specific geographical region.

Region \ Year	1900	1950	1999	2008	2010
Northern America	82	172	307	337	345
Oceania	6	13	30	34	37
Latin America	74	167	511	577	590
Europe	408	547	729	732	738
Africa	133	221	767	973	1,022
Asia	947	1,402	3,634	4,054	4,164
World	1,650	2,521	5,978	6,707	6,896

Table 8.1: The world population in millions of people.

```
1  data := [ ["Year"           , 1900, 1950, 1999, 2008, 2010],
2            ["Northern America",   82,  172,  307,  337,  345],
3            ["Oceania"         ,    6,   13,   30,   34,   37],
4            ["Latin America"   ,   74,  167,  511,  577,  590],
5            ["Europe"          ,  408,  547,  729,  732,  738],
6            ["Africa"          ,  133,  221,  767,  973, 1022],
7            ["Asia"            ,  947, 1402, 3634, 4054, 4164],
8            ["World"           , 1650, 2521, 5978, 6707, 6896] ];
9  worldPopulation := procedure(data) {
10     c       := plot_createCanvas();
11     years   := data[1][2..];
12     regions := [line[1]: line in data[2..]];
13     for (i in [1..#years]) {
14         population := [region[i+1] : region in data[2..]];
15         plot_addBarChart(c, population, regions, "Population in $years[i]$");
16     }
17 };
18 worldPopulation(data);
```

Figure 8.11: Program to plot the world population as a bar chart.

2. The function `worldPopulation` has one parameter. This parameter is of the form of the data specified in line 1 to line 8.
3. Line 10 creates the canvas.
4. Line 11 extracts the years.
5. Line 12 extract the regions from the first column of `data`.
6. Since our intention is to present the data of the different regions combined by the years, we select these data in the list `population` in line 14. Here, `region` iterates over the different regions in `data`, while `region[i+1]` selects the data for the ith year in the table.
7. The function `plot_addBarChart` plots the bar chart. The arguments to this function are as follows:
   (a) `c` specifies the canvas.
   (b) `population` is a list of numbers. For every number in this list, the function `plot_addBarChart` plots a line of the corresponding height into each of the groups. The groups are labeled by

the list `regions`. The lines plotted by one invocation of `plot_addBarChart` all have the same unique color. This color is determined automatically.

(c) `regions` is a list of strings specifying a label for each group.

(d) The last argument specifies a legend for the data plotted by this instance of `plot_addBarChart`.

In order to understand what is plotted where, it is easiest to compare the variable `data` defined in Figure 8.11 to the plot in Figure 8.12

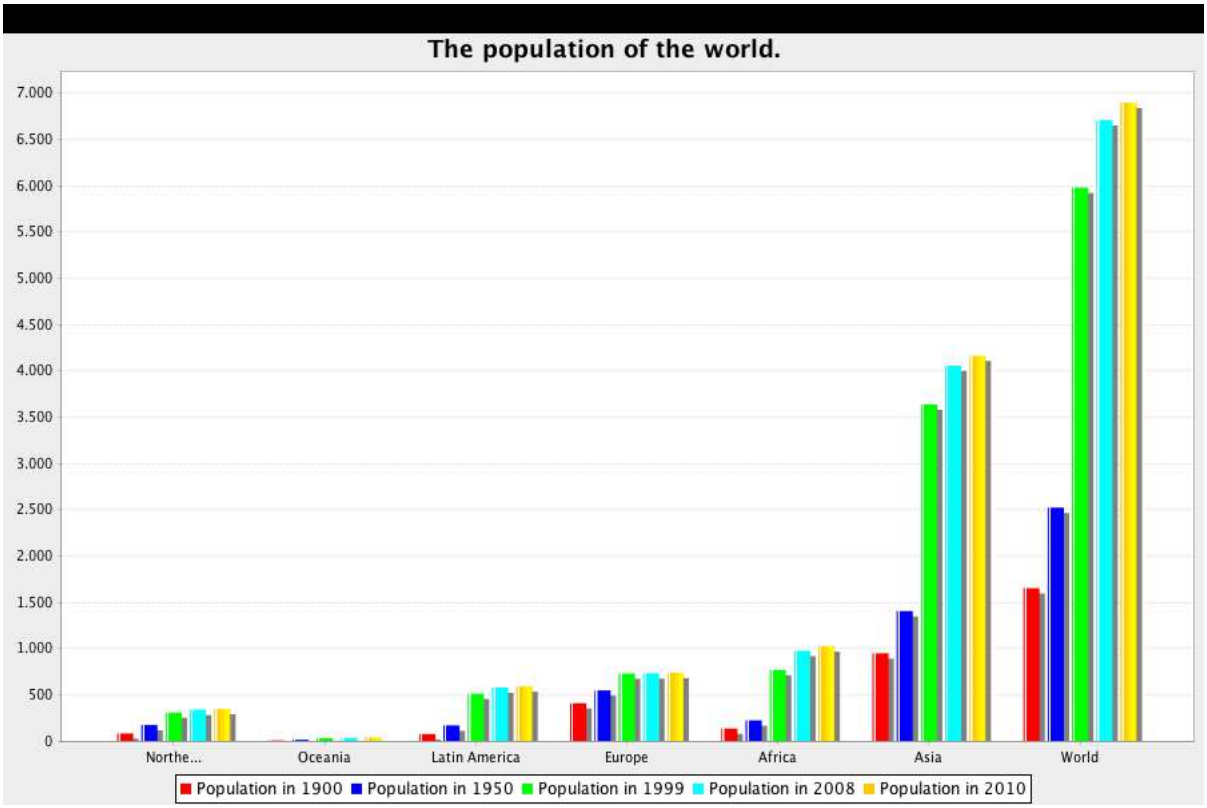

Figure 8.12: A Bar chart showing the world population in millions in different regions.

8.3.2 Pie Charts

Table 8.2 on Page 111 shows the distribution of different races in the Texan population and in the population of Texan prisons. In order to further our understanding of the data, let us visualize it via two pie charts. The visualization is achieved using the SetlX-program shown in Figure 8.13 on page 111. The resulting pie charts are shown in Figure 8.14 on page 112. We proceed to discuss the SetlX-program.

1. The data are defined in line 1 to 5 as a list of list. Each list in `data` corresponds to a different race.
2. The procedure `prisonPopulation` takes this data as its sole input.
3. Line 7 and 8 create two different canvases for the two pie charts.
4. Line 9 extracts the different races stored in the table `data`.

```
1  data := [ ["white", 10933313, 38396],
2            ["black",  2404566, 51649],
3            ["latino", 6669666, 35028],
4            ["other",   844275,   582]
5      ];
6  prisonPopulation := procedure(data) {
7      c1 := plot_createCanvas("Race disdribution in Texas.");
8      c2 := plot_createCanvas("Race disdribution in Texan prisons.");
9      races      := [line[1]: line in data];
10     population := [line[2]: line in data];
11     inmates    := [line[3]: line in data];
12     plot_addPieChart(c1, population, races);
13     plot_addPieChart(c2, inmates,    races);
14     plot_exportCanvas(c1, "race-population");
15     plot_exportCanvas(c2, "race-incarceration");
16 };
```

Figure 8.13: A program to generate two pie charts.

5. Line 10 is defined as list containing the number of people of the various races living in Texas.
6. Line 11 is the corresponding list containing the number of prison inmates of each race.
7. Line 12 draws the pie chart showing the Texan race distribution via a call of the function `plot_addPieChart`.
   (a) The first parameter specifies the canvas on which the pie chart is to be drawn.
   (b) The second parameter is a list of numbers.
   (c) The final parameter is a list of labels.
8. Line 13 draws the pie chart showing the race distribution of Texan inmates.
9. The last two lines save the generated charts as `png` files.

Race	Population	Prison Population
white	10933313	38396
black	2404566	51649
latino	6669666	35028
other	844275	582

Table 8.2: The race distribution in the Texan population vs. the Texan prison population.

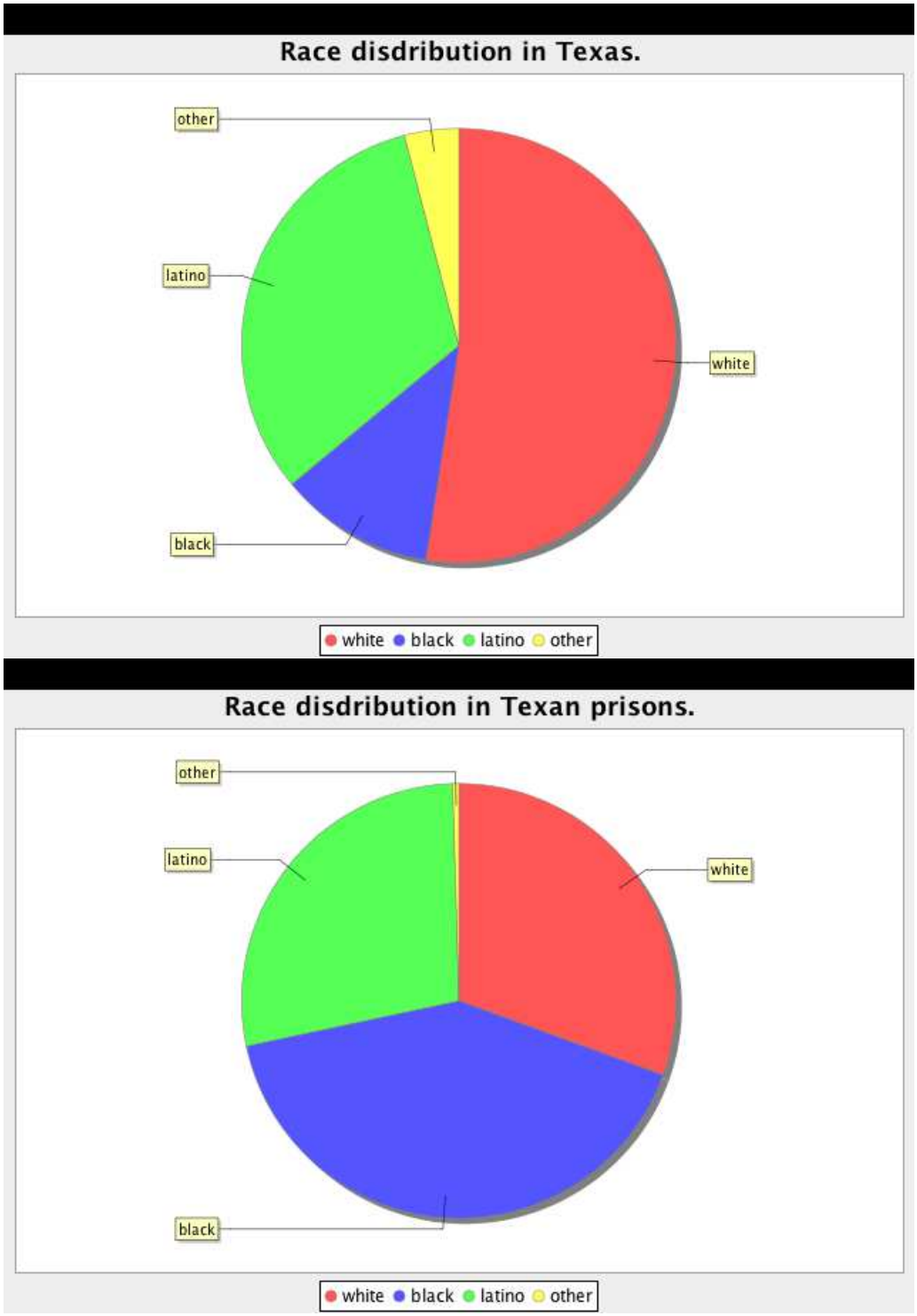

Figure 8.14: The race distribution in Texas and in Texan prisons.

Chapter 9

Classes and Objects

These days, everything is *object-oriented*. After having spotted object-oriented toasters on the loose, we have decided that SetlX should not be an exception. Therefore, SetlX supports some basic means for object-oriented programming. Most importantly, SetlX supports classes. In SetlX a *class* is an agglomeration of variables and functions. These variables are referred to as *member variables*, while the functions are called *methods*. In this chapter we demonstrate how classes can be defined and used in SetlX. However, this chapter is not intended to be an introduction into object-oriented programming. Instead, we assume that the reader has had some previous exposure to object-oriented programming via a programming language like *Java*, C++, or *Python*. In the first section of this chapter, we give some simple examples of class definitions. The programs presented in that section are not intended to compute anything useful. Their sole purpose is to demonstrate the basic features of object-oriented programming available in SetlX. The second and third section will then discuss more interesting examples: The second section shows how to implement weighted directed graphs via classes and the final section discusses the implementation of complex numbers. That section will also discuss some of the more advanced features of object-oriented programming available in SetlX. In particular, we will discuss operator overloading.

9.1 Basic Examples

In this section, we introduce the basic means by which object-oriented programming is supported in SetlX. In the first subsection, we introduce classes. The second subsection shows how to simulate inheritance.

9.1.1 Introducing Classes

Our first example deals with the representation of points in a plane. One way to specify a point p is to specify both its x and its y-coordinate. This leads to the class definition shown in Figure 9.1 on page 114. We discuss the details of this definition next.

1. Line 1 defines the class `point` using the keyword `class`. Note that line 1 simultaneously defines a constructor for this class. This constructor takes two arguments `x` and `y`. These two arguments are interpreted as the x and y-coordinate of the point to construct.

   The general form of a class definition is as follows:

   ```
   class name(x_1, ···, x_n) {
       member-and-method-definitions
   }
   ```

   Here *name* is the name of the class that is defined, $x_1, \cdots, x_n$ are the formal parameters of the

```
1   class point(x, y) {
2       mX := x;
3       mY := y;
4
5       getX  := procedure()  { return mX;             };
6       getY  := procedure()  { return mY;             };
7       setX  := procedure(x) { this.mX := x;          };
8       setY  := procedure(y) { this.mY := y;          };
9       toStr := procedure()  { return "<$mX$, $mY$>"; };
10
11      distance := procedure(p) {
12          return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
13      };
14  }
```

Figure 9.1: The class `point`.

constructor of this class, and *member-and-method-definitions* is a list of the definitions of all member variables and methods.

The definition of a class is not terminated with the character ";". The reason is that, unlike most procedure definitions, a class definition is not part of an assignment. Actually, a procedure definition does not contain an assignment either. The reason that most procedure definitions are followed by a semicolon is the fact that most procedure definitions are part of an assignment. In that case the semicolon following the procedure definition is syntactically not part of the procedure definition but rather terminates the assignment.

2. In line 2 and line 3 we define the member variables `mX` and `mY`. These two member variables store the x and y-coordinates of the point that is constructed.

   It is our convention to start member variables with a lower case letter "`m`". This letter is intended as an abbreviation of member. The letter following the letter "`m`" will always be capitalized. However, this convention is only our recommendation for naming member variables and you are free to choose any valid variable name to designate a member variable.

3. Line 5 to 8 define getter and setter methods to access and write the member variables `mX` and `mY`. Note that, in order to **change** the value of a member variable, it is necessary to prefix this member variable with the keyword "`this.`". However, this prefixing is not necessary in order to read a member variable, as can be seen in the implementation of the methods `getX` and `getY`.

4. Line 9 defines the method `toStr` that transforms a given object of class `point` into a string.

5. Finally, we define a method called `distance` that takes a point `p` as its argument. This method computes the distance of the given point to the point `p` using the Pythagorean theorem.

So far, we have only shown how to define a class but we have not yet seen how to use a class. Therefore, assume that the code shown in Figure 9.1 has been loaded into the interpreter. Then we can define an object of class `point` by writing

```
origin := point(0, 0);
```

and the statement

```
print(origin.toStr());
```

would result in the following output:

```
<0, 0>
```

It is instructive to print the object `origin` itself. The statement

```
print(origin);
```

yields the output that is shown in Figure 9.2 on page 116. Note that we have formatted this output in order to render it more readable. Closer inspection of the output reveals that an object is just a collection of all its members. As SETLX is a functional language, there is really no distinction between a member variable and a method. For example, in line 5 the variable `toStr` is just a member variable which happens to be bound to a procedure. Note that the object `origin` also has a method `getClass`. The definition of this method is shown in line 12 – 23. This method is generated automatically for every class that is defined. Technically, this method returns the class definition. Practically , this method can used to check whether two objects have the same class. For example, after defining

```
p1 := point(1, 2);
p2 := point(2, 1);
```

`p1` and `p2` are different objects but the test

```
p1.getClass() == p2.getClass()
```

will return `true`. If we only are interested whether the object `p1` is of class `point` we can write the test as

```
p1.getClass() == point.
```

This works, since after defining the class `point`, the variable `point` is bound to the class object shown in Figure 9.3 on page 116.

At first, the fact that all methods are stored as part of the object `origin` might seem quite wasteful. However, this fact enables us to change these methods dynamically. For example, we can realize that the x and y-coordinates of the particular object `origin` really are fixed. Therefore, the getter methods can be simplified and the setter methods can even be eliminated. Assuming `origin` is defined as

```
origin := point(0, 0);
```

we could then write the code shown in Figure 9.4. This code would make it impossible to call the setter methods for the object `origin`.

On the other hand, if we choose not to change methods on a per-object basis, we could just as well declare these methods to be static. Then, the resulting code would look as shown in Figure 9.5 on page 117. The only difference to our first implementation of the class `point` shown in Figure 9.1 on page 114 is the `static` block that starts in line 5 and ends in line 15. If we now define `origin` as

```
origin := point(0, 0);
```

and print `origin`, then the resulting output would be as shown in Figure 9.6 on page 118. Note that in this case only the member variables are stored in the object `origin`. This saves some space. Fortunately, we can still overwrite the static methods on a per-object basis, i.e. the assignments

```
origin.getX := procedure() { return 0; };
origin.getY := procedure() { return 0; };
origin.setX := om;
origin.setY := om;
```

will create member variables `getX`, `getY`, `setX`, and `setY` that are attributes of the object `origin`. The static variables of the same name are unchanged but an expression of the form

```
origin.getX()
```

```
object<
    { distance := procedure(p) {
                      return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
                  };
      getX := procedure() { return mX; };
      getY := procedure() { return mY; };
      mX := 0;
      mY := 0;
      setX := procedure(x) { this.mX := x; };
      setY := procedure(y) { this.mY := y; };
      toStr := procedure() { return "<$mX$, $mY$>"; };
      getClass := [] |-> class (x, y) {
          mX := x;
          mY := y;
          distance := procedure(p) {
              return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
          };
          getX := procedure() { return mX; };
          getY := procedure() { return mY; };
          setX := procedure(x) { this.mX := x; };
          setY := procedure(y) { this.mY := y; };
          toStr := procedure() { return "<$mX$, $mY$>"; };
      }
    }
>
```

Figure 9.2: The output of the command "`print(origin);`".

```
class (x, y) {
    mX := x;
    mY := y;
    distance := procedure(p) {
        return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
    };
    getX := procedure() { return mX; };
    getY := procedure() { return mY; };
    setX := procedure(x) { this.mX := x; };
    setY := procedure(y) { this.mY := y; };
    toStr := procedure() { return "<$mX$, $mY$>"; };
}
```

Figure 9.3: The class object corresponding to the class `point`.

invokes the function bound to the member variable `getX` and does no longer invoke the function bound to the static variable `getX`. Hence, the static member functions are effectively hidden.

Going back to the code in Figure 9.5, note that in order to invoke one of these static methods we still need an object of class `point`. The reason is that otherwise the variables `mX` and `mY`, which are used in these methods, would be undefined.

The last remark might confuse people accustomed to a programming language like *Java*. Let us

```
1  origin.getX := procedure() { return 0; };
2  origin.getY := procedure() { return 0; };
3  origin.setX := om;
4  origin.setY := om;
```

Figure 9.4: Changing the methods attached to origin.

```
1  class point(x, y) {
2      mX := x;
3      mY := y;
4
5    static {
6      getX  := procedure()  { return mX;             };
7      getY  := procedure()  { return mY;             };
8      setX  := procedure(x) { this.mX := x;          };
9      setY  := procedure(y) { this.mY := y;          };
10     toStr := procedure()  { return "<$mX$, $mY$>"; };
11
12     distance := procedure(p) {
13         return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
14     };
15   }
16 }
```

Figure 9.5: The class point implemented using static methods.

therefore elaborate: If we use the definition of the class point shown in Figure 9.5 on page 117 and define the object origin as

```
origin := point(0,0);
```

then we can issue the command

```
print(point.getX);
```

and get the result

```
procedure() { return mX; }.
```

This shows that the method getX is indeed a property of the class point. However, we cannot invoke this method on the class point because this method needs access to the member variable mX and this member variable is only available in objects of class point, but it is not an attribute of the class point itself, since only members defined as static are attributes of a class. Therefore, the command

```
point.getX();
```

yields the output

```
~< Result:  om >~
```

indicating that mX is undefined.

Of course, classes can have static variables. These are then accessible by means of the class name. This way, we can simulate global variables in SetlX. For example, we can define the class universal as shown in Figure 9.7. Then, we can always access the universal constant using the expression

```
1  object<
2      { mX := 0;
3        mY := 0;
4        getClass := [] |-> class (x, y) {
5            mX := x;
6            mY := y;
7          static {
8            distance := procedure(p) {
9                return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
10           };
11           getX := procedure() { return mX; };
12           getY := procedure() { return mY; };
13           setX := procedure(x) { this.mX := x; };
14           setY := procedure(y) { this.mY := y; };
15           toStr := procedure() { return "<$mX$, $mY$>"; };
16         }
17       }
18     }
19 >
```

Figure 9.6: Output of "`print(origin);`".

```
universal.sAnswer
```

Note that we can both read and write to this variable. We have started the name of the variable "`sAnswer`" with a lower case "s" since this is our naming convention for static variables.

```
1  class universal() {
2    static {
3      sAnswer := 42;
4    }
5  }
```

Figure 9.7: Defining global variables as static members of a class.

9.1.2 Simulating Inheritance

Continuing the example of the class `point` shown in Figure 9.1, let us assume that some points have a color as an additional attribute. If we were programming in *Java* and wanted to support both points with and without a color, we could create a class `coloredPoint` that extends the class `point`. The mechanics are a little different in SetlX. Figure 9.8 on page 119 shows how to implement both ordinary points and colored points in SetlX.

1. Line 1 defines the class `point`. The definition is the same as in Figure 9.1.
2. Line 14 defines the class `color`. This class stored the color by its rgb value: The member variable `mR` specifies the luminosity of the red color component, `mG` gives the luminosity of the green color component, and `mB` defines the blue color component.

   Note that it is not necessary to implement getters and setters for these member variables since in SetlX all member variables are accessible in the methods of the class.

```
1  class point(x, y) {
2      mX := x;
3      mY := y;
4
5      getX  := procedure()  { return mX;             };
6      getY  := procedure()  { return mY;             };
7      setX  := procedure(x) { this.mX := x;          };
8      setY  := procedure(y) { this.mY := y;          };
9      toStr := procedure()  { return "<$mX$, $mY$>"; };
10     distance := procedure(p) {
11         return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
12     };
13 }
14 class color(r, g, b) {
15     mR := r; mG := g; mB := b;
16 }
17 coloredPoint := procedure(x, y, c) {
18     p := point(x, y);
19     p.mColor := c;
20
21     p.toStr := procedure() {
22         return "<$mX$, $mY$>: $mColor.mR$, $mColor.mG$, $mColor.mB$";
23     };
24     return p;
25 };
```

Figure 9.8: Representing colored points.

3. Line 17 defines the procedure `coloredPoint`. This procedure takes three arguments:
   (a) x and y specify the x and y-coordinate of the given point and
   (b) c specifies the color of the point.

   The procedure `coloredPoint` constructs a colored point. In order to do so, it takes the following steps:
   (a) Line 18 creates an ordinary `point` p that has the specified coordinates.
   (b) Line 19 adds the member variable `mColor` to the `point` p.
   Note that in SetlX it is possible to add member variables dynamically to a given object.
   (c) Line 21 changes the method `toStr` for the object p. The new implementation return both the coordinates and the color information.
   (d) Finally, the object p is returned.

   Note that the procedure `coloredPoint` is effectively a constructor for objects that have class `point` but also have a color attribute. It could be called a *factory method*.

9.2 A Case Study: Dijkstra's Algorithm

In this section we show how to represent a weighted directed graph $G = \langle V, E, \texttt{length}\rangle$ as a class. Here, V is the set of nodes, E is the set of edges, where an edge is a pair of nodes, and

$$\texttt{length} : E \to \mathbb{N}$$

is a function assigning a *length* to every edge. We will present Dijkstra's algorithm for computing distances in a weighted directed graph. Figure 9.9 on page 120 shows a weighted directed graph. The set of nodes V of this graph is given as

$$V = \{1, 2, 3, 4, 5\},$$

the set of edges is

$$E = \{\langle 1, 5\rangle, \langle 1, 3\rangle, \langle 5, 2\rangle, \langle 3, 2\rangle, \langle 2, 4\rangle, \langle 3, 4\rangle\},$$

and the function `length` is given as the relation

$$\texttt{length} = \big\{ \big\langle \langle 1, 5\rangle, 3\big\rangle, \big\langle \langle 1, 3\rangle, 5\big\rangle, \big\langle \langle 5, 2\rangle, 1\big\rangle, \big\langle \langle 3, 2\rangle, 2\big\rangle, \big\langle \langle 2, 4\rangle, 4\big\rangle, \big\langle \langle 3, 4\rangle, 7\big\rangle \big\}.$$

For example, $\texttt{length}(\langle 1, 5\rangle) = 3$ and $\texttt{length}(\langle 1, 3\rangle) = 5$. In this graph, the shortest path from the node 1 to the node 4 is given by the list $[1, 5, 2, 4]$ and therefore the distance between these nodes is $3 + 1 + 4 = 8$.

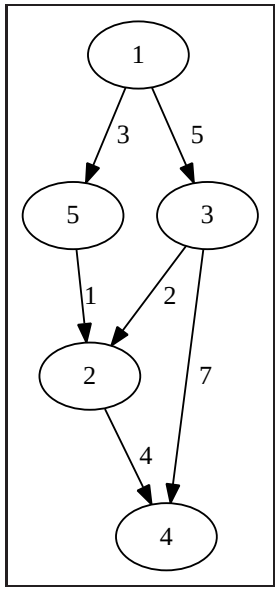

Figure 9.9: A simple weighted directed graph.

In order to compute the distances from a given node to all other nodes we first have to decide how to represent a weighted directed graph. In our implementation of Dijkstra's algorithm we will assume that the set of nodes V is given as a set of consecutive natural numbers of the form

$$V = \{1, 2, \cdots, n-1, n\}.$$

Therefore, there is no need to store the set V. All we have to store is the number of nodes n. The set of edges E and the function `length` will be stored using (a variant of) an *adjacency list representation*. To this end, we define a binary relation edges so that for every node $x \in V$ the expression $\texttt{edges}\,[x]$ is a set of the form

$$\texttt{edges}\,[x] = \{[y_1, w_1], \cdots [y_n, w_n]\},$$

so that there is an edge form x to y_i for all $i = 1, \cdots, n$ and, furthermore, the length of this edge is w_i, i.e. we have

$$\texttt{length}([x, y_i]) = w_i \quad \text{for all } i = 1, \cdots, n.$$

Stated differently, if x and y are nodes in V such that $\langle x, y \rangle$ is an edge in E of length w, then we have

$$[y, w] \in \texttt{edges}[x].$$

For example, the relation edges for the graph shown in Figure 9.9 can be defined in SetlX as follows:

```
edges := { [1, {[5,3]], [3,5]}],
           [5, {[2,1]}],
           [3, {[2,2], [4,7]}],
           [2, {[4,4]}],
           [4, {}]
         };
```

```
1  class graph(numberNodes, edges) {
2      mNumberNodes := numberNodes;
3      mEdges       := edges;
4
5    static {
6      shortestPath := procedure(source) {
7          oo   := mathConst("Infinity");
8          dist := { [x, oo]: x in [1 .. mNumberNodes] };
9          dist[source] := 0;
10         fringe  := { [0, source] };
11         visited := {};
12         while (fringe != {}) {
13             [d, u] := first(fringe);
14             fringe -= { [d, u] };
15             for ([v,l] in mEdges[u] | !(v in visited)) {
16                 if (d + l < dist[v]) {
17                     dvOld := dist[v];
18                     dvNew := d + l;
19                     dist[v] := dvNew;
20                     fringe -= { [dvOld, v] };
21                 }
22                 fringe += { [dist[v], v] };
23             }
24             visited += { u };
25         }
26         return dist;
27     };
28   }
29 }
```

Figure 9.10: A class to represent graphs with an implementation of Dijkstra's algorithm.

Now we are ready to present Dijkstra's algorithm in SetlX. Figure 9.10 on page 121 shows the implementation of the class graph that is used to represent a weighted directed graph. We discuss the implementation line by line.

1. Line 1 starts to define the class graph and the constructor for this class. In SetlX, every class has exactly one constructor and the definition of the class and the constructor are the same.

   In this case, the constructor takes two arguments.

(a) The first argument `numberNodes` specifies the number of nodes. Implicitly, this argument also specifies the set of nodes of the graph since we have agreed that the set of nodes V has the form

$$V = \{1, 2, \cdots, n-1, n\}.$$

Therefore, since `numberNodes` is the number of nodes, we must have $n =$ `numberNodes`.

(b) The second argument `edges` specifies both the set of edges and the function `length` in the way discussed above.

2. Line 2 and 3 define and assign the member variables `mNumberNodes` and `mEdges`. Every object o of class `graph` will therefore have the two attributes `mNumberNodes` and `mEdges`. For a given object o, these attributes can be accessed as o.`mNumberNodes` and o.`mEdges` respectively. These variables store the number of nodes and the edges in the way discussed previously.

3. Line 5 declares the beginning of a static block which ends in line 28. In this case, the static block contains only the definition of the variable `shortestPath`. Of course, if o is an object of class `graph` then o.`shortestPath` will access the function stored in the static variable `shortestPath`. We defer the discussion of the implementation of this function.

At this point, it should be noted that the function `shortestPath` accesses the variables `mNumberNodes` and `mEdges` and these variables are not static. The reason the function `shortestPath` is defined `static` is the fact that the implementation of this function is always the same, no matter what the graph is. Therefore, there is no need to store this implementation on a per object basis. Hence, it is sufficient to provide this function as a static variable of the class `graph`. Of course, calling this function as

`graph.shortestPath`(s)

wouldn't make any sense as then the variables `mNumberNodes` and `mEdges` would both be undefined. In order to call this function, we need to have an object g of class `graph`. Then it is possible to write

g.`shortestPath`(s)

as in this case the function `shortestPath` has access to the member variables `mNumberNodes` and `mEdges` that are stored in the object g.

Let us now briefly explain the implementation of the function `shortestPath`. This function gets a single node `source` as its argument and it will compute the distance of all other nodes from the node `source`. This distance will be represented as a binary relation that is stored in the variable `dist`. The elements of this binary relation will be pairs of the form

$$[x, d]$$

where x is a node and d is the distance of this node. When given the graph g depicted in Figure 9.10, the expression

g.`shortestPath`(1)

will yield the result

`{[1, 0], [2, 4], [3, 5], [5, 3], [4, 8]}`.

This elements of this binary relation are to be interpreted as follows:

1. The distance of the node 1 from the node 1 is obviously 0.

2. The distance of the node 2 from the node 1 is 4. The path $[1, 5, 2]$ exhibits this length.

3. $\vdots$

4. The distance of the node 4 from the node 1 is 8. The path $[1, 5, 2, 4]$ is the shortest path connecting the node 1 with the node 4 and has length 8.

The relation `dist`, which is computed and returned by the function `shortestPath`, is initialized in line 8 to assign the distance ∞ for every node x of the graph. This distance can be seen as an upper estimate of the minimum length of a path connecting the `source` to the node x. Since we do not yet know whether the node x is reachable from the node `source` at all, we initialize the distance to ∞. Initially, the only node that is definitely reachable from the node `source` is the node `source` itself. Therefore, `dist[source]` is set to 0 in line 9.

The algorithm to compute all distances maintains two data structures.

1. `fringe` is the set of those nodes x that have already been encountered when traversing the graph, but which have not been fully explored in the sense that there might exist nodes y such that there is an edge $\langle x, y\rangle$ and the node y has not been visited.

   However, instead of just storing these nodes, for every node y the set `fringe` stores a pair of the form

   $$[w, y],$$

   where w is the best estimate of the distance of the node y from the node `source`. Initially, we have only visited the node `source` and its distance from `source` is obviously 0. Therefore, `fringe` is initialized as

   ```
   { [0, source] }
   ```

   in line 13. Since a node u with distance d is stored as the pair $[d, u]$, the set `fringe` can be used as a priority queue. The reason is that in SetlX every set is stored as an ordered binary tree. Therefore, the elements of a set are ordered and it is easy to pick the smallest element of a set using the function `first`. Since, furthermore, lists (and pairs in particular) are ordered lexicographically, the first element in the set `fringe` is the pair $[d, u]$ with the smallest value of d, i.e. the node u in the set `fringe` that is closest to the `source`.

2. `visited` is the set of nodes that have been *visited* in the following sense: A node x is said to have been *visited* if all of the neighbours of x, i.e. all nodes y such that there is an edge $\langle x, y\rangle$ have been inspected and have been assigned a distance. It may be the case that the distance assigned to y will improve later, but once a node x is added to the set `visited`, the distance of x from `source` is known. Initially, the set `visited` is empty.

The workhorse of the function `shortestPath` is the `while`-loop in line 12. As long as there are nodes u in the `fringe` we try to improve the estimate `dist`$[v]$ for all nodes v such that there is an edge $\langle u, v\rangle$ from u to v. The idea is that if

$$\texttt{dist}[u] + l < \texttt{dist}[v], \quad \text{where } l \text{ is the length of the edge } \langle u, v\rangle$$

then we can improve our estimate of `dist`$[v]$ by assigning

```
dist[v] := dist[u] + l;
```

After updating the distance of node v, we have to add this node to the `fringe` so that we can in turn deal with those nodes that are reachable from node v. The thing that makes Dijkstra's algorithm fast is the fact that once a node is visited it can be shown that its distance can not be improved further. A proof of this fact can be found in a paper of Dijkstra [Dij59].

Remark: At this point you might feel a little bit disappointed because the previous example has not made much use of the object-oriented features of SetlX. However, that is exactly the point of this example: Of course we could have implemented graph nodes as object of some class and do the same for edges. However, the resulting program would then suffer from code bloat. We do not think that object-oriented programming should be used for the sake of object-oriented programming but rather

see it as a convenient method to encapsulate data, nothing more. In particular, an object-oriented program is not, in general, better than a program that does not use object-oriented features. We therefore advocate to use classes when this is either necessary or convenient but not for the sake of object-oriented programming.

9.3 Representing Complex Numbers as Objects

Our next example deals with *complex numbers*. Remember that complex numbers have the form

$$z = x + y \cdot i, \quad \text{where } x, y \in \mathbb{R} \text{ and } i \cdot i = -1.$$

Assume that we have to perform a number of intricate calculations using complex numbers. One way we could do this is by representing a complex number $x + y \cdot i$ as a pair $[x, y]$. Then we would have to implement functions to add, subtract, multiply, and divide complex numbers. For example, methods to add and multiply two complex numbers would have the form shown in Figure 9.11 on page 124.

```
1  addComplex := procedure(z1, z2) {
2      [x1, y1] := z1;
3      [x2, y2] := z2;
4      return [x1 + x2, y1 + y2];
5  };
6  multiplyComplex := procedure(z1, z2) {
7      [x1, y1] := z1;
8      [x2, y2] := z2;
9      return [x1*x2 - y1*y2, x1*y2 + x2*y1];
10 };
```

Figure 9.11: Representing complex numbers as pairs.

While it is certainly possible to represent complex numbers in this way, this method is not very convenient. One reason is that a even a moderately sized formula involving complex numbers would be more or less unreadable when implemented using functions like `addComplex` and `multiplyComplex`. Fortunately, there is a better way to represent complex numbers in SetlX. Figure 9.12 shows how we can define the class `complex` that represents complex numbers. In fact, `complex` is both a class and a *constructor* for this class. This constructor takes two arguments:

1. `real` is the real part of the complex number that is represented, while
2. `imag` is the imaginary part.

The class complex has two *member variables*. These member variables are called `mReal` and `mImag`. These member variables are defined and initialized in lines 2 and 3. Therefore every object o of class complex stores two numbers that can be accessed as

`o.mReal` and `o.mImag`.

In order to work with complex numbers we need methods to perform the basic arithmetic operations. The class `complex` in Figure 9.12 provides the implementation of functions to add, subtract, multiply, and divide complex numbers. As these functions are not properties of a specific complex number but rather apply to all complex numbers, these functions are defined as *static members* of the class `complex`. We refer to these functions as methods and discuss them next.

1. First, line 5 defines a method called `sum`. Despite the looks of it, this method takes two arguments. The first argument is implicit and it is called `this`. This argument is a reference to an object of

```
1  class complex(real, imag) {
2      mReal := real;
3      mImag := imag;
4    static {
5      sum :=
6          that |-> complex(mReal + that.mReal, mImag + that.mImag);
7      difference :=
8          that |-> complex(mReal - that.mReal, mImag - that.mImag);
9      product := procedure(that) {
10         return complex(mReal * that.mReal - mImag * that.mImag,
11                        mReal * that.mImag + mImag * that.mReal);
12     };
13     quotient := procedure(that) {
14         denom := that.mReal * that.mReal + mImag * mImag;
15         real  := (mReal * that.mReal + mImag * that.mImag) / denom;
16         imag  := (mImag * that.mReal - mReal * that.mImag) / denom;
17         return complex(real, imag);
18     };
19     power := procedure(that) {
20         return exp(that * log(this));
21     };
22     f_exp := procedure() {
23         r := exp(mReal);
24         return complex(r * cos(mImag), r * sin(mImag));
25     };
26     f_log := procedure() {
27         logR := log(mReal * mReal + mImag * mImag) / 2;
28         return complex(logR, atan2(mImag, mReal));
29     };
30     f_str := [] |-> mReal + " + " + mImag + " * i";
31     equals := procedure(that) {
32         return mReal == that.mReal && mImag == that.mImag;
33     };
34   }
35 }
```

Figure 9.12: The class `complex` implementing complex numbers.

class `complex`. The second argument, which we have called `that`, also references an object of class `complex`. Now in order to add the complex numbers `this` and `that` we could just return a new complex number as follows:

```
complex(this.mReal + that.mReal, this.mImag + that.mImag);
```

However, as `this` is always implicit, the actual implementation of the method `sum` is even simpler and we can instead write

```
complex(mReal + that.mReal, mImag + that.mImag);
```

to create a complex number that is the sum of the complex number `this` and `that`.

In order to invoke the method `sum`, let us first create two complex numbers `z1` and `z2` as follows:

```
z1 := complex(1,2);
z2 := complex(3,1);
```

Now we have z1 $= 1 + 2 \cdot i$ and z2 $= 3 + 1 \cdot i$. In order to add these numbers, we can execute the assignment

```
z3 := z1.sum(z2);
```

This will set z3 to the complex number $4 + 3 \cdot i$. Fortunately, as SetlX supports operator overloading, we can write

```
z3 := z1 + z2;
```

in order to compute the sum of z1 and z2.

Note: For operator overloading to work, we have to use the same method name that is used when SetlX evaluates the corresponding operator. For the operator "+" performing addition, the corresponding name is sum. Table 9.1 lists the names corresponding to the operators provided by SetlX. Note that certain operators cannot be overloaded because they are implicitly reduced to other operators. In particular

(a) `a <==> b` is reduced to `a == b`,

(b) `a <!=> b` is reduced to `!(a == b)`,

(c) `a != b` is reduced to `!(a == b)`,

(d) `a <= b` is reduced to `a == b || a < b`,

(e) `a >= b` is reduced to `a == b || b < a`, and

(f) `a > b` is reduced to `b < a`.

Furthermore, the operators "+/" and "*/" are reduced to applications of the operators "+" and "*". Therefore, it does not make sense to overload these operators on their own. They are implicitly overloaded when the operators "+" and "*" are overloaded.

Operator	Name	Operator	Name
+	sum	&&	conjunction
-	difference	||	disjunction
*	product	=>	implication
/	quotient	==	equals
%	modulo	**	power
\	integerDivision	<	lessThan
><	cartesianProduct		

Table 9.1: Operator names.

In case you want to find the names of these operators by yourself and do not have access to this manual, then there is a neat trick to figure out these names. Define the class empty as

```
class empty() {}
```

Then, after defining

```
a := empty();
```

the evaluation of the expression

```
a == a
```

will result in the error message

```
Error in "a == a":
Member 'equals' is undefined in 'object<{ class () {} }>'.
```

This message shows that the operator == has the name "equals". Evaluating the expression

```
a <= a;
```

yields the error message

```
Error in "a <= a":
Error in substitute comparison "(a == a) || (a < a)":
Member 'equals' is undefined in 'object< class ()   >'.
```

This error message shows that the expression "a <= b" is reduced to the expression "(a == b) || (a < b)".

Finally, note that Table 9.1 only lists the binary operators. There are three unary operators that can be overloaded, too.

(a) The operator "-" has the name "minus" when used as a unary prefix operator.
It should be noted that SETLX does **not** support the operator "+" as a unary prefix operator.

(b) The operator "!" has the name "not" when used as a unary prefix operator. This same operator has the name "factorial" when used as a unary postfix operator.

2. The implementation of the method difference in line 7 and 8 is similar to the implementation of sum.

3. The method product in line 9 takes two objects of class complex and computes their product according to the formula

$$(x_1 + y_1 \cdot i) \cdot (x_2 + y_2 \cdot i) = x_1 \cdot x_2 - y_1 \cdot y_2 + (x_1 \cdot y_2 + x_2 \cdot y_1) \cdot i.$$

Instead of the "|->"-notation used to define the methods sum and difference we have defined these methods via the keyword "procedure". Defining this method via "procedure" is more convenient as the implementation of the method requires more than one line. As a rule of thumb, we recommend using the "|->"-notation only in cases where the body of a function fits into a single line.

4. The method quotient in line 13 implements the division of complex numbers according to the formula

$$\frac{x_1 + y_1 \cdot i}{x_2 + y_2 \cdot i} = \frac{x_1 \cdot x_2 + y_1 \cdot y_2}{x_2 \cdot x_2 + y_2 \cdot y_1} + \frac{y_1 \cdot x_2 - x_1 \cdot y_2}{x_2 \cdot x_2 + y_2 \cdot y_1} \cdot i.$$

5. The method power raises the complex number this to the power of the argument that. This is done with the help of the function exp and log via the equation

$$z_1^{z_2} = \texttt{exp}(z_2 \cdot \texttt{log}(z_1)).$$

Of course, for this to work we need to implement the functions exp and log for complex numbers.

6. Line 22 defines the function "f_exp". Note that this function is not called "exp" but is rather called "f_exp". In SETLX, if we want to redefine a predefined function like the function "exp" we have to put the string "f_" in front of the function name. Later, when we invoke the function, the prefix "f_" is dropped. For example, if we want to compute the exponential of the imaginary unit i we can define

```
i := complex(0,1);
```

and can then invoke the exponential function as

```
exp(i);
```

The exponential of a complex number $z = x + y \cdot i$ is computed according to Euler's formula as

$$\exp(x + y \cdot i) = \exp(x) \cdot \big(\cos(y) + \sin(y) \cdot i\big).$$

7. Line 22 defines a function to compute the natural logarithm of a complex number z. In order to compute the natural logarithm of the complex number $z = x + y \cdot i$ we rewrite z in polar coordinates so that $z = r \cdot e^{\varphi \cdot i}$. Here, r and φ are given as

   $$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \varphi = \texttt{atan2}(y, x).$$

   Then we have

   $$\log(z) = \log\big(r \cdot e^{\varphi \cdot i}\big) = \log r + \varphi \cdot i.$$

8. Line 30 defines a function that converts a complex number into a string. If a class C provides a method with the name "`f_str`", then every time that an object of class C needs to be converted into a string this function is called implicitly.

9. Finally, we define the method `equals` in line 31. This method is automatically invoked when two objects of class `complex` are compared using either the operator "==" or the operator "!=".

10. Since it is not possible to define a linear order on the set of complex numbers that is compatible with the arithmetic operations, it does not make sense to define the operator "<" for complex numbers. Therefore, this operator has not been overloaded.

Chapter 10

Predefined Functions

This chapter lists the predefined functions. The chapter is divided into eight sections.

1. The first section lists all functions that are related to sets and lists.
2. The second section lists all functions that are related to strings.
3. The third sections lists all functions that are used to work with terms.
4. The following sections lists all mathematical functions, e.g. functions like `sqrt` or `exp`.
5. The next section list all functions that are used to test whether an object has a given type.
6. Section six lists the functions that support interactive debugging.
7. Section seven discusses the functions related to I/O.
8. Section eight discusses the plotting functions.
9. The last sections lists all those procedures that did not fit in any of the previous sections.

10.1 Functions and Operators on Sets and Lists

Most of the operators and functions that are supported on sets and lists have already been discussed. For the convenience of the reader, this section describes all operators and functions, even those that have already been discussed.

1. +: For sets, the binary operator "+" computes the union of its arguments. For lists, this operator appends its arguments.

   If the first argument of the binary operator "+" is a set and the second argument is a list, then the elements of the list are added to the set. If, instead, the first argument is a list and the second argument is a set, then the set is converted into a list and this list is appended to the first list.

   This makes it trivial to implement a sorting procedure as shown below:

   ```
   mySort := l |-> [] + ({} + l);
   ```

   When the function `mySort` is called with a list `l`, the following happens:

   (a) The elements of the list `l` are inserted into a set.

   (b) The elements of this set, which is ordered since all sets are ordered in SetlX, are inserted into a list.

2. *: If both arguments are sets, the operator "*" computes the intersection of its arguments.

   If one argument is a list l and the other argument is a number n, then the list l is appended to itself n times. Therefore, the expression

   ```
   [1, 2, 3] * 3
   ```

   yields the list

   ```
   [1, 2, 3, 1, 2, 3, 1, 2, 3]
   ```

   as a result. Instead of being a list, the argument l can also be a string. In this case, the string l is replicated n times.

3. -: The operator "-" computes the set difference of its arguments. This operator is only defined for sets.

4. %: The operator "%" computes the *symmetric difference* of its arguments. This operator is not defined for lists.

   Of course, all of the operators discussed so far are also defined on numbers and have the obvious meaning when applied to numbers.

5. **: The operator "**" computes the power set of the set s if it is used in an expression of the form

   `2 **` s.

   If, instead, the operator is used in an expression of the form

   ```
   s ** 2
   ```

   where s is a set, then it returns the Cartesian product of s with itself.

   The operator "**" is not defined for lists.

6. +/: The operator "+/" computes the sum of all the elements in its argument. These elements need not be numbers. They can also be sets, lists, or strings. For example, if s is a set of sets, then the expression

   `+/` s

   computes the union of all sets in s. If s is a list of lists instead, the same expression builds a new list by concatenating all lists in l.

7. */: The operator "*/" computes the product of all the elements in its argument. These elements might be numbers or sets. In the latter case, the operator computes the intersection of all elements.

8. ><: If the arguments of the operator "><" are sets, then this operator computes the *Cartesian product* of its arguments. For example, the expression

   ```
   {1,2,3} >< {8,9}
   ```

   yields the set

   ```
   {[1,8], [2,8], [3,8], [1,9], [2,9], [3,9]}
   ```

   If, instead, both arguments of this operator are lists, then the expression

   ```
   l1 >< l2
   ```

   zips the lists `l1` and `l2` into a single list, i.e. the first element of `l1` is paired with the first element of `l2`, the second element of `l1` is paired with the second element of `l2`, and, in general, the i-th element of `l1` is paired with the i-th element of `l2`. The resulting pairs are collected in a

list, which is the result. For example, the expression

```
[1,2] >< [3,4]
```

yields the result

```
[[1, 3], [2, 4]].
```

Formally, we can define the function `zip` as follows:

```
zip := [l1, l2] |-> [ [l1[i], l2[i]] :  i in [1 ..  #l1] ];
```

Then, if the lists `l1` and `l2` have the same length, we have that

```
zip(l1, l2) == l1 >< l2.
```

If, instead, the lists `l1` and `l2` do not have the same length, then the expression `l1 >< l2` raises an error.

9. `arb`: The function `arb`(s) picks an arbitrary element from the sequence s. The argument s can either be a set, a list, or a string.

   Note that an arbitrary element is not the same as a random element. Instead, the element returned by `arb`(s) will be either the first or the last element of the set s. However, this fact is considered an implementation secret and your programs should not rely on this behaviour.

10. `collect`: The function `collect`(l) takes a list l of arbitrary elements. The purpose of this function is to count the number of occurrences of these elements and to compute a binary relation l such that for every element x in the list l, the binary relation r contains a pair of the form $[x, c]$, where c is the number of occurrences of x in the list l. For example, the expression

    ```
    r := collect(["a", "b", "c", "a", "b", "a"]);
    ```

    sets the variable `r` to the value

    ```
    {["a", 3], ["b", 2], ["c", 1]}.
    ```

    Note that this value can be used as a binary relation.

11. `first`: The function `first`(s) picks the first element from the sequence s. The argument s can either be a set, a list, or a string. For a set s, the first element is the element that is smallest with respect to the function `compare` discussed in the last section of this chapter.

12. `last`: The function `last`(s) picks the last element from the sequence s. The argument s can either be a set, a list, or a string. For a set s, the last element is the element that is greatest with respect to the function `compare` discussed in the last section of this chapter.

13. `from`: The function `from`(s) picks an arbitrary element from the sequence s. The argument s can either be a set, a list, or a string. This element is removed from s and returned. This function returns the same element as the function `arb` discussed previously.

14. `fromB`: The function `fromB`(s) picks the first element from the sequence s. The argument s can either be a set, a list, or a string. This element is removed from s and returned. This function returns the same element as the function `first` discussed previously.

15. `fromE`: The function `fromB`(s) picks the last element from the sequence s. The argument s can either be a set, a string, or a list. This element is removed from s and returned. This function returns the same element as the function `last` discussed previously.

16. `domain`: If r is a binary relation, then the equality

    ```
    domain(r) = { x :[x,y] in R }
    ```

holds. For example, we have

```
domain({[1,2],[1,3],[5,7]}) = {1,5}.
```

17. `max`: If s is a set or a list containing only numbers, the expression `max`(s) computes the biggest element of s. For example, the expression

    ```
    max({1,2,3})
    ```

    returns the number 3. If s contains elements that are not numbers, then evaluating the expression `max`(s) raises an error.

18. `min`: If s is a set or a list containing only numbers, the expression `min`(s) computes the smallest element of s. For example, the expression

    ```
    min({1,2,3})
    ```

    returns the number 1. If s contains elements that are not numbers, then evaluating the expression `min`(s) raises an error.

19. `pow`: If s is a set, the expression `pow`(s) computes the power set of s. The power set of s is defined as the set of all subsets of s. For example, the expression

    ```
    pow({1,2,3})
    ```

    returns the set

    ```
    {{}, {1}, {1, 2}, {1, 2, 3}, {1, 3}, {2}, {2, 3}, {3}}.
    ```

20. `range`: If r is a binary relation, then the equality

    ```
    range(r) = { y :[x,y] in R }
    ```

    holds. For example, we have

    ```
    range({[1,2],[1,3],[5,7]}) = {2,3,7}.
    ```

21. `reverse`: If l is a list or string, then `reverse`(l) returns a list or string that contains the elements of l in reverse order. For example,

    ```
    reverse([1,2,3])
    ```

    returns the list

    ```
    [3,2,1].
    ```

22. `sort`: If l is a list or string, then `sort`(l) sorts l into ascending order. For example,

    ```
    sort([3,2,1])
    ```

    returns the list

    ```
    [1,2,3].
    ```

10.2 Functions for String Manipulation

SetlX provides the following functions that are related to strings.

1. +: The operator "+" concatenates strings.

2. *: the operator "*" repeats a string a given number of times. For example, both of the expressions

   ```
   3 * "abc"   and   "abc" * 3
   ```

yield the result

```
"abcabcabc".
```

3. `char`: For a natural number $n \in \{0, 1, 2, \cdots, 127\}$, the expression

   `char`(n)

   computes the character with the ASCII code n. For example, `char`(65) yields the string `"A"`.

   Note that `char`(n) is undefined if either $n < 0$ or $n > 127$.

4. `endsWith`: The function `endsWith` is called as

   `endsWith`(s, t).

   Here, both s and t have to be strings. The function succeeds if the string t is a suffix of the string s, i.e. if there is a string r such that the equation

   $$s = r + t$$

   holds.

5. `eval`: The function `eval` is called as

   `eval`(s).

   Here, s has to be a string that can be parsed as a SetlX expression. This expression is then evaluated in the current variable context and the result of this evaluation is returned. Note that s can describe a SetlX expression of arbitrary complexity. For example, the statement

   ```
   f := eval("procedure(x) { return x * x; }");
   ```

   has the same effect as the following statement:

   ```
   f := procedure(x) { return x * x; };
   ```

   Defining a function f via `eval`(s) is useful because the string s can be the result of an arbitrary computation.

6. `execute`: The function `execute` is called as

   `execute`(s).

   Here, s has to be a string that can be parsed as a SetlX statement. This statement is then executed in the current variable context and the result of this evaluation is returned. Note that s can describe a SetlX expression of arbitrary complexity. For example, the statement

   ```
   execute("f := procedure(x) { return x * x; };");
   ```

   has the same effect as the following statement:

   ```
   f := procedure(x) { return x * x; };
   ```

7. `matches`: The function `matches` is called as

   `matches`(s, r).

   It returns `true` if the regular expression r matches the string s. This function can be called with an optional third argument, which must be a Boolean value. In this case, if the last argument is `true` and the regular expression r contains *capturing groups*, i.e. if parts of the regular expression are enclosed in parentheses, then the substrings of the string s corresponding to these groups are captured and the function returns a list of strings: The first element of this list is the string s, and the remaining elements are those substrings of s that correspond to the different capturing groups. Chapter 4 contains examples demonstrating the use of this function.

8. `join`: The function `join` is called as

   `join`(s, t).

   Here, s is either a set or a list. First, the elements of s are converted into strings. Then these elements are concatenated using the string t as separator. For example, the expression

   ```
   join([1,2,3], "*")
   ```

   yields the string `"1*2*3"`.

   The function `join` comes in handy to generate comma separated values.

9. `replace`: The function `replace` is called as

   `replace`(s, r, t).

   Here, s is a string, r is a regular expression, and t is another string. Any substring u of the string s that is matched by the regular expression r is replaced by the string t. The string resulting from this replacement is returned. The string s itself is not changed. Chapter 4 contains examples demonstrating the use of this function.

10. `replaceFirst`: The function `replaceFirst` is called as

    `replaceFirst`(s, r, t).

    Here, s is a string, r is a regular expression, and t is another string. The first substring u of the string s that is matched by the regular expression r is replaced by the string t. The string resulting from this replacement is returned. The string s itself is not changed. Chapter 4 contains examples demonstrating the use of this function.

11. `split`: The function `split` is called as

    `split`(s, t).

    Here, s and t have to be strings. t can either be a single character or a regular expression. The call `split`(s, t) splits the string s at all occurrences of t. The resulting parts of s are collected into a list. If t is the empty string, the string s is split into all of its characters. For example, the expression

    ```
    split("abc", "");
    ```

    returns the list `["a", "b", "c"]`. As another example,

    ```
    split("abc xy z", " +");
    ```

    yields the list

    ```
    ["abc", "xy", "z"].
    ```

    Note that we have used the regular expression "+" to specify one or more blank characters.

    Certain *magic* characters, i.e. all those characters that serve as operator symbols in regular expressions have to be escaped if they are intended as split characters. Escaping is done by prefixing two backslash symbols to the respective character as in the following example:

    ```
    split("abc|xyz", "\\|");
    ```

    The function `split` is very handy when processing comma separated values from CSV files.

12. `str`: The function `str` is called as

    `str`(a)

    where the argument a can be anything. This function computes the string representation of a.

For example, after defining the function `f` as

```
f := procedure(n) { return n * n; };
```

the expression `str(f)` evaluates to the string

```
"procedure(n) { return n * n; }".
```

13. `toLowerCase`: Given a string s, the expression `toLowerCase`(s) converts all characters of s to lower case.

14. `toUpperCase`: Given a string s, the expression `toUpperCase`(s) converts all characters of s to upper case.

15. `trim`: Given a string s, the expression `trim`(s) returns a string that is the result of removing all leading or trailing white space characters from the string s. For example, the expression

    ```
    trim(" abc xyz\n")
    ```

    returns the string `"abc xyz"`.

10.3 Functions for Term Manipulation

The following functions support terms.

1. `args`: Given a term t that has the form

   $$F(s_1, \cdots, s_n),$$

   the expression `args`(t) returns the list

   $$[s_1, \cdots, s_n].$$

2. `evalTerm`: This function is called as

   `evalTerm`(t).

   Here, t has to be a term that represents a SETLX expression. This expression is then evaluated in the current variable context and the result of this evaluation is returned. For convenience, the term t can be produced by the function `parse`. For example, the statement

   ```
   f := evalTerm(parse("procedure(x) { return x * x; }"));
   ```

   has the same effect as the following statement:

   ```
   f := procedure(x) { return x * x; };
   ```

   The function `evalTerm` is an advanced feature of SETLX that allows for self modifying programs. This idea is that a function definition given as a string can be transformed into a term. This term can then be manipulated using the facilities provided by the `match` statement and the modified term can finally be evaluated using `evalTerm`.

3. `fct`: Given a term t that has the form

   $$F(s_1, \cdots, s_n),$$

   the expression `fct`(t) returns the functor F.

4. `getTerm`: The function `getTerm` is called as

   `getTerm`(v).

   It returns a term representing the value v. For example, the expression

```
canonical(getTerm(procedure(n) { return n * n; }))
```

produces the following output:

```
@@@procedure([@@@parameter("n", "nil")],
             @@@block(
                 [@@@return(@@@product(@@@variable("n"), @@@variable("n")))]
             )
            )
```

The funny looking function symbols that are preceded by three occurrences of the character "@" are the functors that are used by SetlX internally.

However, there is one twist when evaluating getTerm(v): If v is either a set or a list, then getTerm(v) transforms the elements of v into terms and returns the resulting set or list.

5. makeTerm: Given a functor F and a list $l = [s_1, \cdots, s_n]$, the expression

   makeTerm$(F, [s_1, \cdots, s_n])$

   returns the term

   $F(s_1, \cdots, s_n)$.

6. canonical: Given a term t, the expression canonical(t) returns a string that is the canonical representation of the term t. The point is, that all operators in t are replaced by functors that denote these operators internally. For example, the expression

   ```
   canonical(parse("x+2*y"));
   ```

   yields the string

   ```
   @@@sum(@@@variable("x"), @@@product(2, @@@variable("y")))
   ```

   This shows that, internally, variables are represented using the functor "@@@variable" and that the operator "+" is represented by the functor "@@@sum".

7. parse: Given a string s that is a valid expression in SetlX, the expression

   parse(s)

   tries to parse the string s into a term. In order to visualize the structure of this term, the function canonical discussed above can be used.

8. parseStatements: Given a string s, the expression

   parseStatements(s)

   tries to parse the string s as a sequence of SetlX statements. In order to visualize the structure of this term, the function canonical discussed above can be used. For example, the expression

   ```
   canonical(parseStatements("x := 1; y := 2; z := x + y;"));
   ```

   yields the following term (which has been formatted for enhanced readability):

   ```
   @@@block([@@@assignment(@@@variable("x"), 1),
             @@@assignment(@@@variable("y"), 2),
             @@@assignment(@@@variable("z"), @@@sum(@@@variable("x"), @@@variable("y")))
            ])
   ```

10.3.1 Library Functions for Term Manipulation

Beside the predefined functions discussed above, the library "termUtilities" implements three functions that process terms. This library is loaded via the command

```
loadLibrary("termUtilities");
```

After loading this library, the following functions will be available.

1. toTerm(t): This function takes a term t that has been produced by a call to the function parse. This term is then simplified into a more natural structure. For example, if we issue the command

   ```
   t := parse(exp(x));
   ```

   and then inspect the term t using the command

   ```
   canonical(t);
   ```

   the result is

   ```
   @@@call(@@@variable("exp"), [@@@variable("x")], "nil").
   ```

   For many applications, the structure of this term is too complicated. We can simplify the term t using the function toTerm. After the statement

   ```
   s := toTerm(t);
   ```

   we can check that the term s is indeed simpler using the command

   ```
   canonical(s);
   ```

   In this case, the result is

   ```
   @exp(@@@variable("x")).
   ```

2. parseTerm(t): Usually, the function toTerm is used to simplify the result of the predefined function parse. The function parseTerm combines these two functions. It is defined as follows:

   ```
   parseTerm := s |-> toTerm(parse(s));
   ```

3. fromTerm(t): The function fromTerm takes a term t and translates it back into the internal structure required by the function evalTerm. This function can be seen as an inverse of the function toTerm, i.e. we have

   ```
   fromTerm(toTerm(t)) = t
   ```

   provided t is a term produced by the function parse.

10.4 Mathematical Functions

The function SetlX provides the following mathematical functions.

1. The operators "+", "-", "*", "/", and "%" compute the sum, the difference, the product, the quotient, and the remainder of its operands. The remainder $a \,\%\, b$ is defined so that it satisfies

   $$0 \leq a \,\%\, b \quad \text{and} \quad a \,\%\, b < b.$$

2. The operator "\" computes integer division of its arguments. For two integers a and b, this is defined such that

   $$a = (a \backslash b) * b + a \,\%\, b$$

   holds.

3. `abs`(x) calculates the absolute value of the number x.

4. `ceil`(x) calculates the *ceiling function* of x. Mathematically, `ceil`(x) is defined as the smallest integer that is bigger than or equal to x:

   $$\texttt{ceil}(x) := \min\{n \in \mathbb{Z} \mid x \leq n\}.$$

   For example, we have

   ```
   ceil(2.1) = 3.
   ```

5. `floor`(x) calculates the *floor function* of x. Mathematically, `floor`(x) is defined as the largest integer that is less than or equal to x:

   $$\texttt{floor}(x) := \max\{n \in \mathbb{Z} \mid n \leq x\}.$$

   For example, we have

   ```
   floor(2.9) = 2.
   ```

6. `mathConst`(*name*) can be used to compute mathematical constants. At the moment, and π, Euler's number e, and infinity are supported. Therefore, the expression

   ```
   mathConst("pi")
   ```

   yields the result 3.141592653589793, while

   ```
   mathConst("e")
   ```

   yields 2.718281828459045. To get a number of infinite size, we can write

   ```
   mathConst("infinity").
   ```

   SetlX supports a little bit of arithmetic using ∞. For example, the expression

   ```
   mathConst("infinity") - 42
   ```

   evaluates to infinity, while

   ```
   1 / mathConst("infinity")
   ```

   evaluates to 0. However, expressions like

   ```
   mathConst("infinity") - mathConst("infinity");
   ```

   do not have a meaningful interpretation and result in an error message.

7. `nextProbablePrime`(n) returns the next probable prime number that is greater than n. Here, n needs to be a natural number. The probability that the result of `nextProbablePrime`(n) is not a prime number is less than 2^{-100}. For example, to find the smallest prime number greater than 1000 we can use the expression

   ```
   nextProbablePrime(1000).
   ```

8. `int`(s) converts the string s into a number. The function `int` can also be called if s is already a number. In this case, if s is an integer number, it is returned unchanged. Otherwise, the floating point part is truncated. Therefore, the expression

   ```
   int(2.9)
   ```

   returns the integer 2.

9. `rational`(s) converts the string s into a rational number. For example the expression

   ```
   rational("2/7")
   ```

will return the rational number $2/7$. The function `rational` can also be invoked on floating point numbers. For example, the expression

```
rational(mathConst("e"))
```

yields the result $6121026514868073/2251799813685248$. The expression `rational`$(q)$ returns q if q is already a rational number.

10. `double`(s) converts the string s into a floating point number. For example the expression

    ```
    double("0.5")
    ```

    will return the floating point number 0.5. The function `double` can also be invoked on rational numbers. For example, the expression

    ```
    double(2/7)
    ```

    yields 0.2857142857142857. The expression `double`(r) returns r if r is already a floating point number.

    The implementation of floating point numbers is based on the *Java* class `Double`. Therefore, floating point numbers satisfy the specification given in the IEEE standard 754 for the arithmetic of floating point numbers.

11. Furthermore, the trigonometric functions `sin`, `cos`, `tan` and the associated inverse trigonometrical functions `asin`, `acos`, and `atan` are supported. For example, the expression

    ```
    sin(mathConst("pi")/2);
    ```

    yields 1.0.

12. The functions `atan2`(y, x) and `hypot`(x, y) are useful when converting a point $\langle x, y\rangle$ from Cartesian coordinates into polar coordinates. If a point p has Cartesian coordinates $\langle x, y\rangle$ and polar coordinates $\langle r, \varphi\rangle$, then

    $$\varphi = \texttt{atan2}(y, x) \quad \text{and} \quad r = \texttt{hypot}(x, y).$$

    Therefore, $\texttt{hypot}(x, y) = \sqrt{x^2 + y^2}$, while as long as both $x > 0$ and $y > 0$ we have

    $$\texttt{atan2}(y, x) = \texttt{atan}(\tfrac{y}{x}).$$

13. `exp`(x) computes the exponential function of x, i.e. `exp`(x) computes e^x where e is Euler's number. Therefore, the expression

    ```
    exp(1)
    ```

    yields 2.718281828459046.

14. `expm1`(x) computes $e^x - 1$. This function is useful when computing values of e^x where the absolute value of x is very small.

15. `log`(x) computes the natural logarithm of x. This function is the inverse function of the exponential function `exp`. Therefore, we have the equation

    $$\texttt{log}(\texttt{exp}(x)) = x.$$

    This equation is valid as long as there is no overflow in the computation of `exp`(x).

16. `log1p`(x) computes `log`$(x + 1)$, where `log` denotes the natural logarithm. This function should be used to compute $\log(y)$ for values of y such that $y - 1$ is small.

17. `log10`(x) computes the base 10 logarithm of x.

18. `sqrt`(x) computes the square root of x, i.e. it computes $\sqrt{x}$. Therefore, as long as x is small enough so that there is no overflow when computing $x*x$ large, the equation

    `sqrt`$(x*x) = x$

    is valid.

19. `cbrt`(x) computes the cube root of x, i.e. we have `cbrt`$(x) = \sqrt[3]{x}$. Therefore, as long as x is not too large, the equation

    `cbrt`$(x*x*x) = x$

    is valid.

20. `round`(x) rounds the number x to the nearest integer.

21. `nDecimalPlaces`(q, n) takes a positive rational number q and a positive natural number n as arguments. It converts the rational number q into a string that denotes the value of q in decimal floating point notation. This string contains n digits after the decimal point. Note that these digits are truncated, there is no rounding involved. For example, the expression

    `nDecimalPlaces(2/3,5)`

    yields the string "`0.66666`", while

    `nDecimalPlaces(1234567/3,5)` returns "`411522.33333`".

22. `ulp`(x) returns the difference between the floating point number x and the smallest floating point number bigger than x. For example, when working with 64 bits floating point numbers (which is the default), we have

    `ulp`$(1.0) =$ `2.220446049250313E-16`.

    `ulp` is the abbreviation for *unit in the last place*.

23. `signum`(x) computes the sign function of x. If x is positive, the result is 1.0, if x is negative, the result is -1.0. If x is zero, `signum`(x) is also zero.

24. `sinh`(x) computes the hyperbolic sine of x. Mathematically, the hyperbolic sine of x is defined as

    $$\texttt{sinh}(x) := \tfrac{1}{2} \cdot \left(e^x - e^{-x}\right).$$

25. `cosh`(x) computes the hyperbolic cosine of x. Mathematically, the hyperbolic cosine of x is defined as

    $$\texttt{cosh}(x) := \tfrac{1}{2} \cdot \left(e^x + e^{-x}\right).$$

26. `tanh`(x) computes the hyperbolic tangent of x. Mathematically, the hyperbolic tangent of x is defined as

    $$\texttt{tanh}(x) := \frac{\texttt{sinh}(x)}{\texttt{cosh}(x)}.$$

27. `isPrime`(x) tests whether its argument is a prime number. Currently, this function has a naive implementation and is therefore not efficient.

28. `isProbablePrime`(x) tests whether its argument is a prime number. The test used is probabilistic. If x is indeed prime, the test `isProbablePrime`(x) will always succeed. If x is not prime, the predicate `isProbablePrime`(x) might nevertheless return `true`. However, the probability that this happens is less than 2^{-30}.

    It should be noticed that the implementation of `isProbablePrime` is based on random numbers. Therefore, in rare cases different invocations of `isProbablePrime` might return different results.

10.5 Generating Random Numbers and Permutations

In order to generate random numbers, SetlX provides the functions `rnd` and `random`. The easiest to use of these function is `random`. The expression

`random()`

generates a random number x such that

$0 \leq x \quad$ and $\quad x \leq 1$

holds. This will be a floating point number with 64 bits. For debugging purposes it might be necessary to start SetlX with the parameter

`--predictableRandom`.

In this case, the sequence of random numbers that is generated by the function `random` and `rnd` is always the same. This is useful when debugging a program working with random numbers. The function

`resetRandom()`

provides another way to reset the random number generator to its initial state.

The function `random` can also be called with an argument. The expression

`random`(n)

is equivalent to the expression

n `* random()`.

The function `rnd` has a number of different uses.

1. If l is either a list or a set, then the expression

   `rnd`(l)

   returns a random element of l.

2. If n is an natural number, then

   `rnd`(n)

   returns a natural number k such that we have

   $0 \leq k$ and $k \leq n$.

3. If n is a negative number, then

   `rnd`(n)

   returns a negative number k such that we have

   $n \leq k \quad$ and $\quad k \leq 0$.

4. In order to generate random rational numbers, the function `rnd` has to be called with two arguments. The expression

   `rnd`$(a/b,\ n)$

   is internally translated into the expression

   `rnd`$(n-1)$ `* a / (`b `* (`$n-1$`))`.

   Therefore, if a is positive, the number returned is a random number between 0 and the fraction a/b and there will be n different possibilities, so the parameter n is used to control the granularity

of the results. For example, the expression

```
rnd(1/2,6)
```

can return any of the following six fractions:

$$0,\ 1/10,\ 2/10,\ 3/10,\ 4/10,\ 5/10,$$

while the expression `rnd(1/2,101)` could return 101 different results, namely any fraction of the form

$$a/100 \quad \text{such that } a \in \{0, 1, \cdots 100\}.$$

Of course, the user will never see a result of the form $50/100$ as this is immediately reduced to $1/2$.

The second parameter also works if the first argument is a natural number. For example, the expression `random(1,11)` returns one of the following eleven fractions:

$$0,\ 1/10,\ 2/10,\ 3/10,\ 4/10,\ 5/10,\ 6/10,\ 7/10,\ 8/10,\ 9/10,\ 10/10.$$

10.5.1 `shuffle`

The function `shuffle` shuffles a list or string randomly. For example, the expression

```
shuffle("abcdef")
```

might yield the result `"dfbaec"`, while

```
shuffle([1,2,3,4,5,6,7])
```

might yield the list `[1, 7, 4, 6, 3, 5, 2]`.

10.5.2 `nextPermutation`

The function `nextPermutation` can be used to enumerate all possible permutations of a given list or string. For example, the function `printAllPermutations` shown in Figure 10.1 prints all permutations of a given list or string. Evaluation of the expression

```
printAllPermutations([1,2,3])
```

yields the following output:

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

```
1  printAllPermutations := procedure(l) {
2      while (l != om) {
3          print(l);
4          l := nextPermutation(l);
5      }
6  };
```

Figure 10.1: Printing all permutations of a given list or string.

10.5.3 permutations

Given a set, list or string l, the expression

permutations(l)

returns a set of all permutations of l. For example, the expression

```
permutations("abc")
```

returns the set

{"abc", "acb", "bac", "bca", "cab", "cba"}.

Likewise, the expression

```
permutations({1, 2, 3})
```

yields the result

{[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]}.

10.6 Type Checking Functions

SetlX provides the following functions to check the type of a given function.

1. `isBoolean` tests whether its argument is a Boolean value.
2. `isDouble` tests whether its argument is a floating point number.
3. `isError` tests whether its argument is an error object.

   For example, consider the following code:

   ```
   try { throw("foo"); } catch (e) { print(isError(e)); }
   ```

   Here, the exception e that gets caught has been thrown by the user and therefore is not considered an error object. Hence, this program fragment prints `false`. However, the program fragment

   ```
   try { om+1; } catch (e) { print(isError(e)); }
   ```

   will indeed print `true`, as the evaluation of `om+1` raises an exception that signifies an error.
4. `isInfinite` tests whether its argument is a floating point number that represents either positive or negative infinity.
5. `isInteger` tests whether its argument is an integer number.
6. `isList` tests whether its argument is a list.
7. `isMap` tests whether its argument is a binary relation that maps every element in its domain *uniquely* into an element of its range. Therefore, a binary relation r is not a map if it contains two pairs

   $$[x, y_1] \quad \text{and} \quad [x, y_2]$$

   such that $y_1 \neq y_2$.
8. `isNumber` tests whether its argument is a number.
9. `isProcedure` tests whether its argument is a procedure.
10. `isRational` tests whether its argument is a rational number.

11. `isSet` tests whether its argument is a set.
12. `isString` tests whether its argument is a string.
13. `isTerm` tests whether its argument is a term.
14. `isObject` tests whether its argument is an object.

10.7 Debugging

SetlX supports a few functions to help with debugging:

1. `trace`: This function is called as either

   `trace(true)` or `trace(false)`.

   The expression `trace(true)` switches tracing on: Afterwards, all assignments are written to the terminal window running SetlX. To switch tracing of, call `trace(false)`.

2. `stop`: The function `stop` is called as

   `stop(`*id*`)`.

   Here, the optional parameter *id* is a string that reminds you of the location in the program where it was inserted. When executed, this function produces a prompt. Confirm with no input to continue the execution, or enter comma separated variable names to display their current value. Enter 'All' to display all variables in the current scope. This function always returns zero, therefore it is often possible to call this function inside expressions.

3. `assert`: The function `assert` is called as

   `assert(`*condition*, *msg*`)`.

   Here, *condition* is an expression that has to yield either `true` or `false` when evaluated. If the *condition* is true, the function `assert` has no further effect. However, if the evaluation of *condition* yields the value `false`, then the execution of the program is terminated and the error message *msg* is printed.

   SetlX has an option "`--noAssert`". The short form of this option is "`-a`". If this option is activated, the function `assert` is not evaluated. Hence, in order to speed up the program, this option can be used to switch off all assertions.

10.7.1 Library Functions for Debugging

The library `debugUtilities` offers more sophisticated procedures for tracing than the predefined procedure `trace` which either traces everything or nothing. The library `debugUtilities` has to be loaded using the command

```
loadLibrary(debugUtilities);
```

In order to trace a specific procedure, we first need to create a tracer using the command

```
myTracer := tracer();
```

Then, a procedure with the name f can be traced via the command

```
f := myTracer.trace(f, "f");
```

To deactivate tracing of the function f, the command

```
f := myTracer.untrace("f");
```

can be used. This is explained in more detail in Section 5.3.

Besides tracing, the library `debugUtilities` also supports a very simple form of *profiling*, i.e. the measurement of execution times of a procedure. To this end, the library "`debugUtilities`" contains the class `profiler`. This class offers the function `profile`. This function is called as follows:

p`.profile(`f`, "`f`");`

Here, p is an object of the class `profiler` and f is the name of a function. An example will clarify the idea. Figure 10.2 on page 145 shows a naive computation of the Fibonacci numbers that makes use of the method `profile`.

```
1   loadLibrary("debugUtilities");
2
3   fibonacci := procedure(n) {
4       if (n in [0,1]) {
5           return n;
6       }
7       return fibonacci(n-1) + fibonacci(n-2);
8   };
9
10  p := profiler();
11  fibonacci := p.profile(fibonacci, "fibonacci");
12
13  for (n in [10 .. 32]) {
14      start := p.mTimes["fibonacci"];
15      fibonacci(n);
16      stop  := p.mTimes["fibonacci"];
17      print("$n$: $stop - start$");
18  }
```

Figure 10.2: Profiling the Fibonacci function.

1. First, we need to load the library `debugUtilities` in line 1.
2. Line 3 – 8 provide the implementation of the function `fibonacci`. This function is called with an integer argument n and `fibonacci(`n`)` computes the n-th Fibonacci number. Since we want to demonstrate profiling, the implementation given here is, on purpose, inefficient.
3. Line 10 creates a profiler `p`.
4. Line 11 initiates profiling of the function `fibonacci`.
5. In line 13 – 18 we compute the n-th Fibonacci number for $n \in \{20, \cdots, 32\}$.
6. Every time the function `fibonacci` is called, the accumulated running time of all invocations of the function `fibonacci` is stored in the dictionary `p.mTimes` under the key `"fibonacci"`. Hence the expression

   `p.mTimes["fibonacci"]`

   in line 14 yields the time spent so far to compute the Fibonacci numbers up to the $n-1$-th Fibonacci number in milliseconds, while the same expression in line 16 yields the time spent to compute all Fibonacci numbers up to the n-th Fibonacci number. Therefore, the difference

   `stop` $-$ `start`

is the time to compute the n-th Fibonacci number.

7. The output of this program is shown in Figure 10.3. This output clearly shows that the time needed to compute the n-th Fibonacci number increases exponentially with n.

```
20: 1005
21: 1952
22: 2802
23: 5604
24: 9547
25: 12699
26: 21346
27: 36260
28: 60400
29: 102612
30: 172454
31: 288334
32: 482399
```

Figure 10.3: Output of the program shown in Figure 10.2

Besides the procedure `profile`, the class `profiler` also implements the procedure `unprofile` that stops profiling of its argument. It is called as

p`.unprofile("`f`");`

Here, p needs to be the same instance of the class `profiler` that has been used to initiate profiling of the function f.

10.8 I/O Functions

This section lists all functions related to input and output.

10.8.1 `appendFile`

The function `appendFile` is called as

`appendFile`(*fileName*, l).

Here, *fileName* is a string denoting the name of a file, while l is a list of strings. If file a with the specified name does not exist, it is created. Then the strings given in l are appended to the file. Each string written to the specified file is automatically terminated by a newline character.

10.8.2 `ask`

The function `ask` is called as

`ask`(*question*, *answerList*).

Here, *question* is a string that is printed and *answerList* is a list of all possible answers that the user can choose from. For example, imagine you are programming some support application for the US judicial system and your program has to ask your customer a difficult question. The call

```
ask("How would you like to die?", ["electrocution", "intoxination"]);
```

prints the following message to the screen:

```
How would you like to die?
1) electrocution
2) intoxination
Please enter a number between 1 and 2:
```

If the user would now presses the digit 1 and hits the enter key, then the function `ask` will return the string `"electrocution"`.

10.8.3 `deleteFile`

The function `deleteFile` is called as

`deleteFile`(*fileName*).

Here, *fileName* is a string denoting the name of a file. If a file with the specified name does exist, it is deleted. In this case the function returns `true`. If a file with the specified name does not exist, the function returns `false` instead.

10.8.4 `get`

The function `get` is called as

`get`(s).

Here, s is a string that is used for a prompt. This argument is optional. The function prints s and then returns the string that the user has supplied. If no string is supplied, `get` uses the prompt `": "`.

10.8.5 `load`

The function `load` is called as

`load`(*file*).

Here, *file* has to be a string that denotes a file name, including the extension of the file. Furthermore, *file* is expected to contain valid SetlX commands. These commands are then executed. In general, most of the commands will be definitions of functions. These function can then be used interactively.

10.8.6 `loadLibrary`

The function `loadLibrary` is called as

`loadLibrary`(*file*).

Here, *file* has to be a string that denotes a file name, excluding the extension of the file. This file is assumed to be located in the directory that is specified by the environment variable

`SETLX_LIBRARY_PATH`.

There are three different ways to set this variable.

1. The variable can be set a in file like "`.profile`" or "`.bashrc`", or something similar. Depending on the type of shell you are using, these files are automatically executed when a new shell is started.
2. The variable can be set manually in the shell.
3. The variable can be set using the option "`--libraryPath`".

The specified *file* is expected to contain valid SetlX commands.

10.8.7 multiLineMode

The function multiLineMode is called as

multiLineMode(*flag*).

Here, flag should be an expression that evaluates to a Boolean value. The function either activates or deactivates *multi line mode*. If multi line mode is activated, then in a shell the next input expression is only evaluated after the user hits the return key twice. Multi line mode makes it possible to enter statements spanning several lines interactively. However, normally this mode is inconvenient, as it requires the user to press the return key twice in order to evaluate an expression. Therefore, by default multi line mode is not active. Multi line mode can also be activated using the option "--multiLineMode" when starting SetlX.

10.8.8 nPrint

The function nPrint is called as

nPrint($a_1, \cdots, a_n$).

It takes any number of arguments and prints these arguments onto the standard output stream. In contrast to the function print, this function does not append a newline to the printed output.

10.8.9 nPrintErr

The function nPrintErr is called as

nPrintErr($a_1, \cdots, a_n$).

It takes any number of arguments and prints these arguments onto the standard error stream. In contrast to the function printErr, this function does not append a newline to the printed output.

10.8.10 print

The function print is called as

print($a_1, \cdots, a_n$).

It takes any number of arguments and prints these arguments onto the standard output stream. After all arguments are printed, this function appends a newline to the output.

10.8.11 printErr

The function printErr is called as

printErr($a_1, \cdots, a_n$).

It takes any number of arguments and prints these arguments onto the standard error stream. After all arguments are printed, this function appends a newline to the output.

10.8.12 read

The function read is called as

read(s).

Here, s is a string that is used for a prompt. This argument is optional. The function prints s and then returns the string that the user has supplied. However, leading and trailing white space is removed from the string that has been read. If the string can be interpreted as a number, this number is

returned instead. Furthermore, this function keeps prompting the user for input until the user enters a non-empty string.

10.8.13 readFile

The function `readFile` is called as

`readFile`(*file*, *lines*).

The second parameter *lines* is optional. It reads the specified file and returns a list of strings. Each string corresponds to one line of the file. This second parameter *lines* specifies the list of line numbers to read. For example, the statement

```
read("file.txt",[42] + [78..113])
```

will read line number 42 and the lines 78 up to and including line 113 of the given file. This feature can be used to read a file in chunks of 1000 lines as in the following example:

```
n := 1;
while (true) {
    content := readFile("file", [n .. n + 999]);
    if (content != []) {
        // process 1k lines
        ....
        n += 1000;
    } else {
        break;
    }
}
```

10.8.14 writeFile

The function `writeFile` is called as

`writeFile`(f, l).

Here, f is the name of the file and l is a list. The file f is created and the list l is written into the file f. The different elements of l are separated by newlines.

10.9 Plotting Functions

This section lists all plotting functions, specifies their parameters and provide small examples that demonstrate the use of these functions.

10.9.1 plot_createCanvas

The function `plot_createCanvas` returns a canvas object that can be used for plotting. All other plotting functions need a canvas object as their first argument. In general, it is possible to plot multiple graphs on a canvas object. However, it is not possible to plot both functions and charts onto a canvas. Hence, a canvas can either hold function graphs and scatter plots or it can hold statistical charts.

The function `plot_createCanvas` takes one parameter. This parameter is a string that adds a title to the canvas. This parameter is optional. The function returns a canvas object.

Example Code

```
c := plot_createCanvas();
c := plot_createCanvas("Canvas Name");
```

10.9.2 plot_addBullets

This function adds points into a canvas. Currently, these points are represented as squares. The function `plot_addBullets` is called as follows:

`plot_addBullets`(*canvas*, *coordinates*, *color*, *size*)

The parameters are interpreted as follows:

1. *canvas* is a canvas object that has been previously created using the function `plot_createCanvas`.
2. *coordinates* is a list of pairs of floating point numbers. Hence, these pairs have the form $[x, y]$. Here x specifies the x-coordinate of the point to be plotted, while y specifies its y-coordinate.
3. `color` is a list of three integers of the form $[r, g, b]$, where r, g, and b are elements of the set $\{1, \cdots, 255\}$ that together specify the color of the points to be plotted in the RGB color model.

   This parameter is optional and defaults to `[0,0,0]` which is the RGB code for the color black.
4. *size* is a floating point value that specifies the size of the bullet. This parameter is optional.

The function returns a reference to the points that have been created. This reference can be used to remove the points from the canvas via the function `plot_removeGraph`, which is discussed later.

Example Code

```
c  := plot_createCanvas();
b1 := plot_addBullets(c, [[0,0]]);                    // one bullet
b2 := plot_addBullets(c, [[1,1], [2,2], [3,3]]);      // three bullets
b3 := plot_addBullets(c, [[1,2]], [255, 0, 0]);       // one red bullet
b4 := plot_addBullets(c, [[1,2]], [0, 255, 0], 10.0); // one bigger bullet
```

10.9.3 plot_addGraph

The function `plot_addGraph` plots the graph of a function. The function `plot_addGraph` is called as follows:

`plot_addGraph`(*canvas*, *function*, *label*, *color*, *fill*)

The parameters are interpreted as follows:

1. *canvas* is a canvas object that has been previously created using the function `plot_createCanvas`.
2. *function* is a string that specifies the function that is to be plotted. This string uses the letter "`x`" to specify the independent variable. For example, in order to draw the function $x \mapsto x \cdot x$ we could use the string string "`x*x`"
3. *label* is a string that is used to label the function in the legend that is drawn at the bottom of the generated figure.
4. `color` is a list of three integers of the form $[r, g, b]$, where r, g, and b are elements of the set $\{1, \cdots, 255\}$ that together specify the color of the points to be plotted in the RGB color model.

   This parameter is optional and defaults to `[0,0,0]` which is the RGB code for the color black.

5. *fill* is a boolean parameter that specifies whether the area under the graph should be colored as well.

   This parameter is optional and defaults to `false`. Hence, if this parameter is not used then only the graph of the function is drawn.

The function returns a reference to the graph that has been created. This reference can be used to remove the graph from the canvas.

Example Code

```
c  := plot_createCanvas();
g1 := plot_addGraph(c, "x*x", "parabola");
g2 := plot_addGraph(c, "x**3", "cubic function in red", [255, 0, 0]);
g3 := plot_addGraph(c, "sin(x)", "area under the sine", [0, 0, 0], true);
```

10.9.4 plot_addListGraph

The function `plot_addListGraph` takes a list of points and connects these points via straight lines. This function is called as follows:

`plot_addListGraph`(*canvas*, *coordinates*, *label*, *color*, *fill*)

The parameters are interpreted as follows:

1. *canvas* is a canvas object that has been previously created using the function `plot_createCanvas`.

2. *coordinates* is a list of pairs of floating point numbers. Hence, these pairs have the form $[x, y]$. Here x specifies the x-coordinate of a point, while y specifies its y-coordinate. The points specified via *coordinates* are connected via straight lines.

3. *label* is a string that is used to label the graph in the legend that is drawn at the bottom of the generated figure.

4. `color` is a list of three integers of the form $[r, g, b]$, where r, g, and b are elements of the set $\{1, \cdots, 255\}$ that together specify the color of the points to be plotted in the RGB color model.

   This parameter is optional and defaults to `[0,0,0]` which is the RGB code for the color black.

5. *fill* is a boolean parameter that specifies whether the area under the lines should be colored as well.

   This parameter is optional and defaults to `false`. Hence, if this parameter is not used then only the lines connecting the points are drawn.

Example Code

```
c  := plot_createCanvas();
g1 := plot_addListGraph(c, [[10, 0],[-10,0]], "simple List graph");
g2 := plot_addListGraph(c, [[-10, -10], [-5, -5], [0, 1], [5,6]],
                        "listgraph with color", [255, 0, 0]);
g2 := plot_addListGraph(c, [[-10, 10], [-5, 5], [0, 0], [2.5,6], [5, -4]],
                        "listgraph with color and area", [255, 0, 255], true);
```

10.9.5 plot_addParamGraph

The function `plot_addParamGraph` plots a parametric curve. The function is called as follows:

`plot_addParamGraph`(*canvas*, *x-fct*, *y-fct*, *label*, *limits*, *color*, *fill*)

The parameters are interpreted as follows:

1. *canvas* is a canvas object that has been previously created using the function `plot_createCanvas`.
2. *x-fct* is a string that specifies how the x-coordinate is to be computed. This string uses the letter "x" to specify the parameter.
3. *y-fct* is a string that specifies how the y-coordinate is to be computed. Again, this string uses the letter "x" to specify the parameter.
4. *label* is a string that is used to label the function in the legend that is drawn at the bottom of the generated figure.
5. *limits* is a pair of two floating point numbers. This pair has the form

   [*xMin*, *xMax*] .

   Here, *xMin* is the smallest value of the parameter "x" that is used, while *xMax* specifies the biggest value.
6. `color` is a list of three integers of the form $[r, g, b]$, where r, g, and b are elements of the set $\{1, \cdots, 255\}$ that together specify the color of the points to be plotted in the RGB color model.

   This parameter is optional and defaults to `[0,0,0]` which is the RGB code for the color black.
7. *fill* is a boolean parameter that specifies whether the area between the graph and the x-axis should be colored as well.

   This parameter is optional and defaults to `false`. Hence, if this parameter is not used then only the graph of the function is drawn.

The function returns a reference to the graph that has been created. This reference can be used to remove the graph from the canvas.

Example Code

```
c  := plot_createCanvas();
g1 := plot_addParamGraph(c, "3*cos(x)", "3*sin(x)",
                         "circle with radius 3",
                         [-3.15, 3.15]);
g2 := plot_addParamGraph(c, "2*cos(x)", "2*sin(x)",
                         "colored circle with radius 2",
                         [-3.15, 3.15], [0, 255, 0]);
g3 := plot_addParamGraph(c, "cos(x)", "sin(x)",
                         "filled green circle",
                         [-3.15, 3.15], [0, 255, 0], true);
```

10.9.6 plot_addBarChart

The function `plot_addBarChart` creates a bar chart where the bars are drawn vertically. This function is called as follows:

`plot_addBarChart`(*canvas*, *values*, *labels*, *name*)

The parameters are interpreted as follows:

1. *canvas* is a canvas object that has been previously created using the function `plot_createCanvas`.
2. *values* is a list of floating point numbers. These numbers specify the height of the bars.
3. *labels* is a list of strings. This list has to have the same length as the list *values*. The strings in the list *labels* are uses as legends at the bottowm of the bar chart. For example, in Figure 8.12 on page 110 the list that is used as labels has the form

   ```
   ["Northern America", "Oceania", "Latin America", "Europe", "Africa",
   "Asia", "World"]
   ```

4. *name* is a string that attaches a common name to all values given in the list *value*. This parameter is used in the legend of the bar chart.
5. Note that there is no parameter to set the color of the individual bars. All bars resulting from the same invocation of a call of the function `plot_addBarChart` are filled with the same color, while SetlX uses different colors for different invocation of `plot_addBarChart`. For example, in order to generate Figure 8.12 on page 110 the function `plot_addBarChart` was called five times: The first invocation created the red bars corresponding to the year 1900, the second invocation created the blue bars showing the population in 1950 and so on.

The function returns a reference to the graph that has been created. This reference can be used to remove the graph from the canvas.

Example Code

```
c   := plot_createCanvas();
ch1 := plot_addBarChart(c, [1,2], ["a","b"], "data set number 1");
ch2 := plot_addBarChart(c, [2,1], ["a","b"], "data set number 2");
```

10.9.7 plot_addPieChart

The function `plot_addPieChart` creates a pie chart. This function is called as follows:

`plot_addPieChart`(*canvas*, *values*, *labels*)

The parameters are interpreted as follows:

1. *canvas* is a canvas object that has been previously created using the function `plot_createCanvas`.
2. *values* is a list of floating point numbers. These numbers specify the area of the pies associated with them.
3. *labels* is a list of strings. This list has to have the same length as the list *values*. The strings in the list *labels* are used as legends for the pie chart. For example, in the first pie chart shown in Figure 8.14 on page 112 the list that is used as labels has the form

   ```
   ["white", "black", "latino", "other"].
   ```

4. Note that there is no parameter to set the color of the individual pies. These colors are chosen automatically by SetlX.

The function returns a reference to the pie chart that has been created. This reference can be used to remove this chart from the canvas.

Example Code

```
c  := plot_createCanvas();
ch := plot_addPieChart(c, [1,2,3,4], ["a","b","c","d"]);
```

10.9.8 plot_addLabel

The function `plot_addLabel` takes a canvas and adds a label at a specified position. This function is called as

`plot_addLabel`(*canvas*, *coordinate*, *label*)

The parameters are interpreted as follows:

1. *canvas* specifies the canvas where the label needs to be added.
2. *coordinates* is a pair of floating point numbers. Hence, this pair has the form $[x, y]$. Here x specifies the x-coordinate of a point, while y specifies its y-coordinate. These coordinates specify the location of the *label*. Specifically, the pair $[x, y]$ specifies the position of the lower left corner of the label.
3. *label* is a string that is printed on the canvas at the position specified via the parameter *coordinates*.

The function `plot_addLabel` returns a reference to the label that has been created. This reference can be used to remove the label using the function `plot_removeGraph` discussed below.

Example Code

```
c := plot_createCanvas();
g := plot_addGraph(c, "x*x", "parabola");
l := plot_addLabel(c, [0,0], "origin");
```

10.9.9 plot_defineTitle

The function `plot_defineTitle` takes a canvas and adds a title to the graph. Note that the title is added below the title that is set with `plot_createCanvas`. The function `plot_createCanvas` is called as follows:

`plot_createCanvas`(*canvas*, *title*)

The parameters are interpreted as follows:

1. *canvas* specifies the canvas.
2. *title* is the string that is used as title.

This function does not return a value.

Example Code

```
c := plot_createCanvas();
plot_defineTitle(c, "Title of Canvas");
```

10.9.10 plot_exportCanvas

The function `plot_exportCanvas` stores a canvas on disk as a file in the PNG format. The function is called as follows:

`plot_exportCanvas`(*canvas*, *fileName*)

The parameters are interpreted as follows:

1. *canvas* is the canvas that is to be saved to disk.

2. *fileName* is a string that specifies the name of the file that is used to store the image of the canvas. Note that the file extension ".png" is automatically added to *fileName* when the file is stored on disk.

The function does not return a value.

Example Code

```
c := plot_createCanvas();
plot_addGraph(c, "cos(x)", "the cosine function");
plot_exportCanvas(c, "cosine");
```

10.9.11 plot_labelAxis

The function `plot_labelAxis` changes the labels of both x-axis and y-axis. The function is called as follows:

`plot_labelAxis`(*canvas*, *xLabel*, *yLabel*)

The parameters are interpreted as follows:

1. *canvas* specifies the canvas whose axis are to be modified.
2. *xLabel* is a string that is used as the new label of the x-axis. If this string is the empty string, then the label of the x-axis is removed.
3. *yLabel* is a string that is used as the new label of the y-axis. If this string is the empty string, then the label of the y-axis is removed.

This function does not return a value.

Example Code

```
c := plot_createCanvas();
plot_addGraph(c, "x*x", "parabola");
plot_labelAxis(c, "x-axis", "y-axis");
```

10.9.12 plot_legendVisible

The function `plot_legendVisible` shows or hides the legend of a graph. The function is called as follows:

`plot_legendVisible`(*canvas*, *flag*)

The parameters are interpreted as follows:

1. *canvas* specifies the canvas whose legends are to be shown or hidden.
2. *flag* is a Boolean value. If *flag* is `true`, the legend is shown. If *flag* is `false` the legend is hidden instead.

By default, the legend of a canvas is visible. In order to switch the legend off, the function `plot_legendVisible` has to be called with its second parameter set to `false`.

Example Code

```
c := plot_createCanvas();
plot_addGraph(c, "x*x-2", "parabola");
plot_legendVisible(c, false);
```

10.9.13 plot_modScale

The function plot_modScale changes the scale of both the x-axis and the y-axis. Of course, this function can only be called for those graphs that have an x-axis and a y-axis. Hence, this function cannot be called for pie charts or bar charts. The function is called as follows:

plot_modScale(*canvas*, *xLimits*, *yLimits*)

The parameters are interpreted as follows:

1. *canvas* specifies the canvas whose axis are to be changed.
2. *xLimits* is pair of floating point numbers, i.e. *xLimits* has the form

   [*xMin*, *xMax*].

   Here, *xMin* is the smallest value shown on the x-axis, while *xMax* is the largest value.
3. *yLimits* is pair of floating point numbers, i.e. *yLimits* has the form

   [*yMin*, *yMax*].

   Here, *yMin* is the smallest value shown on the y-axis, while *yMax* is the largest value.

This function does not return a value.

Example Code

```
c := plot_createCanvas();
plot_addGraph(c, "x**2-2*x", "parabola");
plot_modScale(c, [0, 5], [-2, 16]);
```

10.9.14 plot_modScaleType

The function plot_modScaleType changes the scaling type of an axis. An axis can either be scaled linearly or logarithmically. If an axis is scaled linearly, then the distance between any two numbers on the axis is proportionally to the difference of these numbers. On the other hand, if an axis is scaled logarithmically, then the distance between any two numbers on the axis is proportionally to the quotient of these numbers. The function plot_modScaleType is called as follows:

plot_modScaleType(*canvas*, *xType*, *yType*)

The parameters are interpreted as follows:

1. *canvas* specifies the canvas that is to be changed.
2. *xType* is a string that is a member of the set {"num","log"}. If *xType* is equal to the string "num", then the x-axis is scaled linearly. If *xType* is equal to the string "log", then the x-axis is scaled logarithmically.
3. *yType* is a string that is a member of the set {"num","log"}. If *yType* is equal to the string "num", then the y-axis is scaled linearly. If *yType* is equal to the string "log", then the y-axis is scaled logarithmically.

The function does not return a value.

Example Code

```
c := plot_createCanvas();
plot_addGraph(c, "x**2-2*x", "parabola");
plot_modScaleType(c, "num", "log");
```

10.9.15 plot_removeGraph

The function `plot_removeGraph` removes a graph or a chart from a canvas. The function is called as follows:

`plot_removeGraph`(*canvas*, *reference*)

The parameters are interpreted as follows:

1. *canvas* specifies the canvas from which a graph or chart is to be removed.
2. *reference* is a reference to the graph or canvas that is to be removed. This reference is the return value of any of the `plot_add*` functions.

This function does not return a value.

Example Code

```
c  := plot_createCanvas();
g1 := plot_addGraph(c, "sin(x)", "sine");
g2 := plot_addListGraph(c, [[-1, 1], [-2, 0], [1, 3], [2, 1]], "lines");
g3 := plot_addParamGraph(c, "cos(x)", "sin(x)", "circle", [-5,5]);
l1 := plot_addLabel(c, [1,1], "some text");
b1 := plot_addBullets(c, [[1, 2]]);
plot_removeGraph(c, g3);
plot_removeGraph(c, g2);
plot_removeGraph(c, g1);
plot_removeGraph(c, l1);
plot_removeGraph(c, b1);
```

10.9.16 plot_modSize

The function `plot_modSize` changes the size of the canvas. The function is called as follows:

`plot_modSize`(*canvas*, *size*)

The parameters are interpreted as follows:

1. *canvas* specifies the canvas whose size is to be changed.
2. *size* is a pair of natural numbers. This pair has the form

   [*width*, *height*].

   Here, *width* specifies the new width of the canvas area in pixles, while *height* specifies the height of the window.

The function does not return a value.

Example Code

```
c := plot_createCanvas();
plot_modSize(c, [1024,768]);
plot_addParamGraph(c, "sin(x)", "cos(x)", "circle", [-3.15, 3.15]);
```

10.10 Miscellaneous Functions

This final section lists some functions that did not fit into any of the other sections.

10.10.1 abort

This function aborts the execution of the current function. This is done by raising an exception. Usually, the function `abort` is called with one argument. This argument is then thrown as an exception. For example, the statement

```
try { abort(1); } catch (e) { print("e = $e$"); }
```

will print the result

```
e = Error:  abort:  1.
```

Note that an exception raised via `abort` can not be catched with the keyword `catchUsr`. The keyword `catchUsr` will only catch exceptions that are explicitly thrown by the user via invocation of `throw`.

10.10.2 cacheStats

The function `cacheStats` is called as

`cacheStats`(f).

Here, f is a cached procedure, i.e. a procedure that is declared using the keyword "`cachedProcedure`". Note that f has to be the procedure itself, not the name of the procedure! The returned result is a set of the form

```
{["cache hits", 996], ["cached items", 1281]}.
```

This set can be interpreted as a map.

10.10.3 clearCache

The function `clearCache` is called as

`clearCache`(f).

Here, f is a cached procedure, i.e. a procedure that is declared using the keyword "`cachedProcedure`". Note that f has to be the procedure itself, not the name of the procedure! The invocation of `clearCache`(f) frees the memory associated with the cache for the function f. It should be used if the previously computed values of f are not likely to be needed for the next computation.

10.10.4 compare

The function `compare` is called as

`compare`(x, y).

Here, x and y are two arbitrary values. This function returns -1 if x is less than y, $+1$ if x is bigger than y and 0 if x and y are equal. If x and y have a numerical type, then the result of `compare`(x, y) coincides with the result produced by the operator "<". If x and y are both lists, then the lists are compared lexicographically. The same remark holds if x and y are both sets. If x and y have different types, then the result of `compare`(x, y) is implementation defined. Therefore, the user should not rely on the results returned in these cases.

The function `compare` is needed internally in order to compare the elements of a set. In SetlX all sets are represented as ordered binary trees.

10.10.5 getOsID

The function `getOsID` returns an identifier for the operating system that setlX runs on.

10.10.6 getScope

The function getScope is called as

```
getScope().
```

It returns a term representing the *current scope*. Here, the current scope captures the binding of all variables. For example, suppose the user issues the following commands:

```
x := 1;
y := 2;
f := procedure(n) { return n * n; };
```

Then the current scope consists of the variables x, y, and f. Therefore, in this case the expression getScope() returns the following term:

```
^scope({["f", procedure(n) { return n * n; }],
        ["getScope", ^preDefinedProcedure("getScope")], ["params", []],
        ["x", 1],
        ["y", 2]
       })
```

10.10.7 logo

The function logo is called as

```
logo().
```

In order to find how this function works, try it yourself.

10.10.8 now

The function now returns the number of milliseconds that have elapsed since the beginning of the Unix epoch.

10.10.9 run

The function run executes a shell command, i.e. a command that would otherwise be invoked via a shell like bash or the windows command line. The function run(*cmd*) returns a pair of lists: The first list contains the strings that make up the output produced when running the command *cmd* in the command line. Every string in the list corresponds to one line of output produced by *cmd*. The second list is empty as long as *cmd* does not produce any error messages. If there are error messages, these are collected in the second list returned as the result of run(*cmd*). For example, the command

```
run('ls *.tex');
```

yields the output

```
[["backtracking.tex", "basics.tex", "closures.tex"], []]
```

in one of my directories containing TEXfiles. However, if this command is executed in a directory that does not contain any file with the extension ".tex", then the output is

```
[[], ["ls:  *.tex:  No such file or directory"]]
```

indicating that there are no files matching the pattern "*.tex".

10.10.10 sleep

The function sleep takes one argument that has to be a positive natural number. The expression

`sleep`(t)

pauses the execution of the process running SetlX for t milliseconds. When printing output, this can be used for visual effects.

Bibliography

[Ada80] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Harmony Books, New York, 1980.

[Bra90] Ivan Bratko. *PROLOG Programming for Artificial Intelligence*. Addison-Wesley, 2nd edition, 1990.

[Can95] Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Mathematische Annalen*, 46:481–512, 1895.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

[Fri06] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, 3rd edition, 2006.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[Kle09] Gerwin Klein. JFlex User's Manual: Version 1.4.3. Technical report, Technische Universität München, 2009. Available at: `http://jflex.de/jflex.pdf`.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal. British Computer Society.*, 6(4):308–320, 1964.

[Les75] Michael E. Lesk. Lex – A lexical analyzer generator. Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

[Mic68] Donald Michie. Memo functions and machine learning. *Nature*, 1968.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Sch70] Jacob T. Schwartz. Set theory as a language for program specification and programming. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[SDSD86] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming With Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[Sny90] W. Kirk Snyder. The Setl2 programming language: Update on current developments. Technical report, Courant Institute of Mathematical Sciences, New York University, 1990. Available at `ftp://cs.nyu.edu/pub/languages/setl2/doc/2.2/update.ps`.

[SS75] Gerald Jay Sussman and Guy L. Steele. Scheme: An interpreter for extended lambda calculus. Technical report, MIT AI LAB, 1975.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

[SS94] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed*. MIT Press, 1994.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms in Java*. Pearson, 4th edition, 2011.

[vR95] Guido van Rossum. Python tutorial. Technical report, Centrum Wiskunde & Informatica, Amsterdam, 1995.

[WS92] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly and Assoc., 1992.

Appendix A

Graphical Library Functions

The following appendix lists the name of the functions that support the animation of algorithms in SetlX. The functions are listed in alphabetical order. It should be noted that the graphical library is an experimental feature that might be subject to considerable changes in the future. It should also be noted that this library has been derived by Markus Jagiella from the file StdDraw.java that is part of the booksite accompanying the excellent book on algorithms written by Robert Sedgewick and Kevin Wayne [SW11].

1. `gfx_addPlayPauseButton`

   `gfx_addPlayPauseButton()`

   `gfx_addPlayPauseButton(boolean add)`

   Sets the Play/Pause Button on the animation frame visible (no parameter or true) or invisible (false). Default is invisible.

   Parameters:

   `add` – the flag to set the visibility

2. `gfx_addSpeedSlider`

   `gfx_addSpeedSlider()`

   `gfx_addSpeedSlider(boolean add)`

   Creates a speed slider on the animation frame if called with either no parameter or arguments `true`. Set the slider to invisible if the argument is `false`.

   Parameters:

   `add` – the flag to set the visibility

3. `gfx_arc`

   `gfx_arc(double x, double y, double r, double phi1, double phi2)`

   Draws an arc of radius `r`, centered on the point $\langle x, y \rangle$, from angle `phi1` to `phi2`. The angles are given in degrees.

   Parameters:

   `x` – the x coordinate of the center of the circle

   `y` – the y coordinate of the center of the circle

   `r` – the radius of the circle

   `phi1` – the starting angle. 0 would mean an arc beginning at 3 o'clock.

`phi2` – the angle at the end of the arc.

For example, in order to get a 90 degree arc, `phi2` should be `phi1` + 90.

4. `gfx_circle`

   `gfx_circle(double x, double y, double r)`

   Draws a circle of radius `r`, centered on the point $\langle x, y \rangle$. The circle degenerates to a pixel if the radius is too small.

   Parameters:

   `x` – the x-coordinate of the center of the circle

   `y` – the y-coordinate of the center of the circle

   `r` – the radius of the circle

5. `gfx_clear`

   `gfx_clear()`

   `gfx_clear(String color)`

   Clear the screen with the given color or the default color (white). Calls `gfx_show()`.

   Parameters:

   `color` – the color of the background

6. `gfx_ellipse`

   `gfx_ellipse(double x, double y, double semiMajorAxis, double semiMinorAxis)`

   Draws an ellipse with given semimajor and semiminor axis centered on $\langle x, y \rangle$.

   Parameters:

   `x` – the x-coordinate of the center of the ellipse

   `y` – the y coordinate of the center of the ellipse

   `semiMajorAxis` – the length of the semimajor axis of the ellipse

   `semiMinorAxis` – the length of the semiminor axis of the ellipse

7. `gfx_filledCircle`

   `gfx_filledCircle(double x, double y, double r)`

   Draws a filled circle of radius `r`, centered on $\langle x, y \rangle$. The circle degenerates to a pixel if the radius is too small.

   Parameters:

   `x` – the x-coordinate of the center of the circle

   `y` – the y-coordinate of the center of the circle

   `r` – the radius of the circle

8. `gfx_filledEllipse`

   `gfx_filledEllipse(double x, double y, double semiMajorAxis, double semiMinorAxis)`

   Draws an ellipse with given semimajor and semiminor axis centered on $\langle x, y \rangle$.

   Parameters:

   `x` – the x-coordinate of the center of the ellipse

   `y` – the y-coordinate of the center of the ellipse

   `semiMajorAxis` – the length of the semimajor axis of the ellipse

   `semiMinorAxis` – the length of the semiminor axis of the ellipse

9. `gfx_filledPolygon`

   `gfx_filledPolygon(double[] x, double[] y)`

   Draws a filled polygon with the given $\langle x[i], y[i] \rangle$ coordinates.

   Parameters:

   `x` – an array of all the x-coordinates of the polygon

   `y` – an array of all the y-coordinates of the polygon

10. `gfx_filledSquare`

    `gfx_filledSquare(double x, double y, double r)`

    Draws a filled square of side length $2 \cdot r$, centered on $\langle x, y \rangle$. This square degenerates to a pixel if r is too small.

    Parameters:

    `x` - the x-coordinate of the center of the square

    `y` - the y coordinate of the center of the square

    `r` - the radius r is half the length of any side of the square

11. `gfx_getFont`

    `String gfx_getFont()`

    Returns the name of the current font i.e. "`Arial`".

12. `gfx_getPenColor`

    `String gfx_getPenColor()`

    Returns the current color of the pen i.e.: "`BLACK`".

13. `gfx_getPenRadius`

    `double gfx_getPenRadius()`

    Returns the current pen radius.

14. `gfx_hasNextKeyTyped`

    `boolean gfx_hasNextKeyTyped()`

    Returns `true` if the user has typed a key, `false` otherwise.

15. `gfx_isKeyPressed`

    `gfx_isKeyPressed(int keyCode)`

    Returns `true` if the key with the given `keycode` is currently pressed and `false` otherwise.

    Parameters:

    `keyCode` – the physical key code of the Java programming language

16. `gfx_isPaused`

    `boolean gfx_isPaused()`

    Returns `true` if the animation is paused when this statement is executed and `false` otherwise.

17. `gfx_line`

    `gfx_line(double x0, double y0, double x1, double y1)`

    Draws a line from the point $\langle x_0, y_0 \rangle$ to the point $\langle x_1, y_1 \rangle$.

    Parameters:

`x0` – the x-coordinate of the starting point

`y0` – the y-coordinate of the starting point

`x1` – the x-coordinate of the destination point

`y1` – the y coordinate of the destination point

18. `gfx_mouseX`

    `double gfx_mouseX()`

    Returns the value of the x-coordinate of the mouse.

19. `gfx_mouseY`

    `double gfx_mouseY()`

    Returns the value of the y-coordinate of the mouse.

20. `gfx_nextKeyTyped`

    `String gfx_nextKeyTyped()`

    Returns the next unicode character that was typed by the user. This function cannot identify action keys such as "F1" or arrow keys.

21. `gfx_picture`

    `gfx_picture(double x, double y, String s)`

    `gfx_picture(double x, double y, String s, double w, double h)`

    Draws a picture in `gif`, `jpg`, or `png` format centered on the point $\langle x, y \rangle$. If the parameters `w` and `h` are given, the picture is rescaled to a width of `w` and a height of h. If `w` or `h` is too small, the picture degenerates into a pixel. This function calls `gfx_show()`.

    Parameters:

    `x` – the x-coordinate of the center of the image

    `y` – the y-coordinate of the center of the image

    `s` – the name of the file containing the image, i.e. "`flower.gif`"

    `w` – the width of the image

    `h` – the height of the image

22. `gfx_point`

    `gfx_point(double x, double y)`

    Draws a point at the coordinates $\langle x, y \rangle$.

    Parameters:

    `x` - the x-coordinate of the point to be drawn

    `y` - the y-coordinate of the point to be drawn

23. `gfx_polygon`

    `gfx_polygon(double[] x, double[] y)`

    Draws a polygon with the given $\langle x[i], y[i] \rangle$ coordinates.

    Parameters:

    `x` - an array of all the x-coordinates of the polygon

    `y` - an array of all the y-coordinates of the polygon

24. `gfx_rectangle`

    `gfx_rectangle(double x, double y, double halfWidth, double halfHeight)`

    Draws a rectangle of given half width and half height centered on $\langle x, y \rangle$.

    Parameters:

    `x` – the x-coordinate of the center of the rectangle

    `y` – the y-coordinate of the center of the rectangle

    `halfWidth` – the width of the rectangle is given as $2 \cdot$ `halfWidth`

    `halfHeight` – the height of the rectangle is given as $2 \cdot$ `halfHeight`

25. `gfx_setCanvasSize` `gfx_setCanvasSize(int w, int h)`

    Sets the window size to `w`×`h` pixels.

    Parameters:

    `w` – the width as a number of pixels

    `h` – the height as a number of pixels

26. `gfx_setPaused`

    `gfx_setPaused(boolean paused)`

    Sets the start value of the `Play/Pause` button and determines whether the animation is started in running or paused mode.

    Parameters:

    `paused` – if this is true the animation is paused

27. `gfx_setPenColor`

    `gfx_setPenColor()`

    `gfx_setPenColor(String color)`

    Sets the pen color to the given color or the default (`BLACK`) if no color is given. The available pen colors are `BLACK`, `BLUE`, `CYAN`, `DARKGRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, and `YELLOW`.

    Parameters:

    `color` – the color of the pen

28. `gfx_setPenColorRGB`

    `gfx_setPenColorRGB(double r, double g, double b)`

    Sets the pen color to the color specified by the values of `r`, `g`, and `b`.

    Parameters:

    `r` – the luminosity of the red component of the color

    `g` – the luminosity of the green component of the color

    `b` – the luminosity of the blue component of the color

29. `gfx_setPenRadius`

    `gfx_setPenRadius()`

    `gfx_setPenRadius(double r)`

    Sets the pen size to the given size or the default if no parameters are passed.

    Parameters:

    `r` – the radius of the pen

30. `gfx_setFont`

    `gfx_setFont(String fontName)`

    `gfx_setFont()`

    Changes the current font or restores the default font if no argument is passed.

    Parameters:

    `fontName` – the name of the font i.e. "Arial"

31. `gfx_setScale`

    `gfx_setScale(double min, double max)`

    Set the x and y scale to the given values or the default if no parameters are passed (a border is added to the values). Has the same effect as the calls `setXscale(min,max)` and `setYscale(min,max)` combined.

    Parameters:

    `min` – the minimum value of the x and y scale

    `max` – the maximum value of the X and Y scale

32. `gfx_setXscale`

    `gfx_setXscale()`

    `gfx_setXscale(double min, double max)`

    Set the x scale to the given values or the default if no parameters are passed (a border is added to the values).

    Parameters:

    `min` – the minimum value of the x scale

    `max` – the maximum value of the x scale

33. `gfx_setYscale`

    `gfx_setYscale()`

    `gfx_setYscale(double min, double max)`

    Set the y scale to the given values or the default if no parameters are passed (a border is added to the values).

    Parameters:

    `min` – the minimum value of the y scale

    `max` – the maximum value of the y scale

34. `gfx_show`

    `gfx_show()`

    Displays the animation on the screen. Calling this method means that the screen will be redrawn after each invocation of `line()`, `circle()`, or `square()`. This is the default.

35. `gfx_show(int t)`

    Displays the animation on screen and pause for `t` milliseconds. Calling this method means that the screen will not be redrawn after each invocation of `line()`, `circle()`, or `square()`. This is useful when there are many invocation of these function necessary to draw a complete picture.

    Parameters:

    `t` – number of milliseconds to pause

36. `gfx_square`

    `gfx_square(double x, double y, double r)`

    Draws a square of side length $2 \cdot r$, centered on $\langle x, y \rangle$. The square degenerates to a pixel if small r is too small.

    Parameters:

    `x` – the x-coordinate of the center of the square

    `y` – the y coordinate of the center of the square

    `r` – the side length of the square is $2 \cdot r$

37. `gfx_text`

    `gfx_text(double x, double y, String s)`

    Write the given text string in the current font, centered on $\langle x, y \rangle$. Calls `gfx_show()`.

    Parameters:

    `x` – the x coordinate of the center of the text

    `y` – the y coordinate of the center of the text

    `s` – the string

38. `gfx_textLeft`

    `gfx_textLeft(double x, double y, String s)`

    Write the given text string in the current font, right aligned at $\langle x, y \rangle$. Calls `gfx_show()`.

    Parameters:

    `x` – the x coordinate of the end of the text

    `y` – the y coordinate of the end of the text

    `s` – the string

39. `gfx_textRight`

    `gfx_textRight(double x, double y, String s)`

    Write the given text string in the current font, left aligned at $\langle x, y \rangle$. Calls `gfx_show()`.

    Parameters:

    `x` – the x coordinate of the beginning of the text

    `y` – the y coordinate of the beginning of the text

    `s` – the string