SetlX — A Tutorial

Version 1.4.1

Karl Stroetmann
Duale Hochschule Baden-Württemberg Stuttgart
stroetmann@dhbw-stuttgart.de

Tom Herrmann
Duale Hochschule Baden-Württemberg Stuttgart
setlx@herrmann-tom.de

November 13, 2012

Abstract

In the late sixties, Jack Schwartz, a renowned professor of mathematics at the Courant Institute for Mathematical Sciences in New York, developed a very high level programming language called SETL [Sch70, SDSD86]. The most distinguishing feature of this language is the support it offers for sets and lists. This feature permits convenient implementations of mathematical algorithms: As set theory is the language of mathematics, many mathematical algorithms that are formulated in terms of set theory have very straightforward implementations 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 those days, 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.

At the Baden-Württemberg Corporate State University, the first author has used SETL2 [Sny90] for some years in a number of his introductory computer science courses. He has noticed that the adoption of SETL has made the abstract concepts of set theory tangible to the students. Nevertheless, as the original version of SETL is more than forty years old, it has a number of shortcomings. One minor issue is the fact, that the syntax is quite dated and has proven difficult to master for students that are mainly acquainted with `C` and *Java*. Furthermore, SETL lacks any immediate support for first order terms. Therefore, 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*. In particular, SETLX supports matching.
- SETLX supports some ideas from functional programming. In particular, functions can be used as a primitive data type. Furthermore, SETLX supports *closures* and the *memoization* of functions.
- SETLX provides some means to support backtracking.
- Lastly, while SETL 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. Fortunately, for the time being he continues to maintain the language and even implements new features.

Contents

Chapter 1

Introduction

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 lanuages 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 wellfare of the entire universe is at stake and only 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 background of the average computer scientist does not permit them to follow advanced theological discussions. Therefore, we 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 as concise as 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. Indeed, the conciseness of SetlX programs was one of the reasons for the first author to adopt SetlX as a programming language in his courses on computer science: It is often feasible to write a complete SetlX program in a few lines onto the blackboard, since SetlX programs are as concise as mathematical formulae.

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 types are sets, lists, first-order terms, and functions. 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. The remainder of this tutorial is structured as follows:

1. In the second chapter, we discuss the available data types.
2. The third chapter shows the control structures available.
3. The fourth chapter deals with regular expressions.
4. The fifth chapter discusses the `try-catch` and `throw` mechanism and demonstrates how this mechanism can be used to simulate backtracking.

5. The final chapter explains the predefined functions.

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.

Downloading

The current distribution of SetlX can be downloaded from either

http://wwwlehre.dhbw-stuttgart.de/~stroetma/SetlX/setlX.php

or

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*, *Mac OS X*, *Microsoft Windows*, and *Android*.

The websites given above explain 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.

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

Karl Stroetmann: stroetmann@dhbw-stuttgart.de

Tom Herrmann: setlx@herrmann-tom.de

Acknowledgements

The autors 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

SetlX Types

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. However, 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 promt"`=>`". Commands are typed after the prompt. If the command is an assignment or an expression, then it is terminated by a semicolon. However, complex commands like, for example, loops are not terminated by a semicolon.

```
-====================================setlX============================v1.4.1=-

Welcome to the setlX interpreter!

Open Source Software from http://setlX.randoom.org/
(c) 2011-2012 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 SetlX interpreter can be used as a simple calculator: Typing

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

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

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

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. A rational number consists of a nominator and a denominator, both of which are integers. SetlX takes care to ensure that nominator and denominator are always in lowest terms. Furthermore, if the denominator has the value 1, only the nominator ist 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 >~.
```

If you prefer to calculate with floating point values, the easiest way is to add `0.0` at the end of the expression because if an arithmetic expression contains a floating point value, the result is 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 can also be achieved via the command

```
=> 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 2.2 on page 8 shows a program that computes $\sqrt{2}$ as the limit of the sequence $(b_n)_n$ that is defined as

$$b_0 := 2 \quad \text{and} \quad b_{n+1} := \frac{1}{2} \cdot \left(b_n + \frac{2}{b_n} \right).$$

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

[1] In SetlX, statements can extend over many lines. If the user intends to use multiline 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);`".

interpreter via the following command:

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

The output of the program is shown in Figure 2.3.

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

Figure 2.2: 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 2.3: The output produced by the program in Figure 2.2.

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

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

This will yield two lines of output:

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

The first lines show 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* 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;
```

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

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

Important: In SetlX, the names of variables have to start with a lower case letter. After the first letter, the name can contain upper or lower case letters, digits, or underscores. If the first letter is an upper case letter, then the identifier is interpreted as a *functor*. This notion will be explained when discussing *terms*.

2.2 Boolean Values

The Boolean values `true` and `false` represent truth and falsity. Boolean expressions can be constructed using the comparison operators "`==`", "`!=`", "`<`", "`>`", "`<=`", and "`>=`". SetlX provides the following propositional operators 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*, (also known as *disjunction*), so the expression

   ```
   a || b
   ```

   is true if either `a` or `b` is true. It is also true if both `a` and `b` is true.

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

   ```
   !a
   ```

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

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

   ```
   a => b
   ```

   is true if either `a` is false or `b` is true. Therefore, the expression

   ```
   a => b
   ```

   has the same 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 "`<!=>`".

Furthermore, 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 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`" *setExpr* "`|`" *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 is less than x^2.

Important: `forall` and `exists` statements 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.
```

This example shows that the values of the variables `x` and `y` is available outside of the existentially quantified expression. 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 sets the value of the variable `n` to 3, as this is the first integer in the list `[1..10]` where $n^2 > 2^n$ holds.

2.3 Sets

The most interesting data type provided by SetlX is *set* type. A *set* is a collection that contains all of its elements exactly once. To create a simple set containing the numbers 1, 2, and 3, we can write

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

Note that the comparison

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

yields the result

```
Result:  true
```

After all, the order of elements in a set does not matter. 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 ordering of elements in a set, it just remembers the elements and sorts them internally to maximize the efficiency of looking up elements from a set.

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 most straightforward of them 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 will give the result

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

Here, we have used the range operator "`..`". For given 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

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

If b is less than a, the expression

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

denotes the empty set $\{\}$.

While neighbouring 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 neighbouring elements. For example, the expression

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

yields

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

In general, when a set definition of the form

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

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

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

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

   $$s := a - b.$$

   Then, we have

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

   For example, we have

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

2.3.1 Operators on Sets

The most basic operators on sets are as follows:

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 written as $s_1 \backslash s_2$. It is defined as

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

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

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

5. "`%`" computes the *symmetric difference* of sets.

   In mathematics, the symmetric difference of two sets s_1 and s_2 is 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 >< 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 >< s2 = {[1, 2], [1, 3], [2, 2], [2, 3]}
s1 %  s2 = {1, 3}
```

Also note the use of *string interpolation* in the last example. This will be discussed in more detail in the section on strings.

For all of the above operators there is a variant of the assignment operator that includes the operator. For example, we can write

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

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

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

removes the element `x` from the set `s`.

In addition to the basic operators, 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 "`+/`". In order to compute the sum

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

we can use the command

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

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 usefull 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 always 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.

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 we 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.

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, $\cdots$, x_n `in` s_n | *cond* }.

Here, *expr* is some expression containing the variables x_1, $\cdots$, x_n, while s_1, $\cdots$, 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 $x_1 \in s_1$, $\cdots$, $x_n \in s_n$ 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 };
```

When used on sets, the operator "`-`" computes the difference set. 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 effcient 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.

There is a short form available for set comprehension: For a set s and a Boolean expression *cond*, the expression

{ x `in` s | *cond* };

will compute the subset of all those elements of the set s that satisfy the specified Boolean condition *cond*. For example, the expression

```
{ p in {1..n} | n % p == 0 }
```

computes the set of all those numbers p that divide n without leaving a remainder.

[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 "`*/`".

2.3.3 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 afterwards the set `s` will no longer contain the element `x`. 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 rational numbers, the ordering of these numbers is the usual ordering on numbers. If the sets contain sets or lists, these sets are themselves compared lexicographically. Things get more complicated if a set is heterogenous 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 natural number less or equal than 5. Essentially it is equal to `rnd({1..5});`

2.4 Lists

Besides sets, SetlX also supports lists. Syntactically, the main difference is that the braces "{" and "}" are substituted with the square brackets "`[`" and "`]`". Semantically, a list is an ordered collection of elements that can contain the same element multiple times. For example

```
[1,4,7,2,4,7]
```

is a typical list. The easiest way to construct a list is via the range operator "`..`": For example, the command

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

computes the result

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

The prefix operators "`+/`" and "`*/`" work for lists also. For example,

```
+/ [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 tuples

$$\langle x_1, \cdots, x_n \rangle \in s_1 \times \cdots \times s_n$$

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.

There are two infix operators for lists: The first is the operator "`+`" which concatenates its arguments. For example

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

yields

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

The second infix operator is "`*`". While the first argument of this operator is a list, the second needs to be an integer. 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].
```

The cardinality operator "`#`" works for lists the same way it works for sets: It returns the number of elements. For example

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

yields the result 3.

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 := [1..100];
```

the expression

```
l[5..10]
```

yields the sublist `[5, 6, 7, 8, 9, 10]`. There are two 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.

Lists can also be used on the left hand side of an assigment. The statement

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

sets the variable x to 1 and y to 2. This feature can be used to swap the values of variables: the assigment

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

swaps the values of 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 for all pairs

$[x, y_1] \in r$ and $[x, y_2] \in r$

we have that $y_1 = y_2$, 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. If a binary relation r is a map, SetlX permits us to use the relation 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 `function.stlx` shown below in figure 2.4 shows how maps can be used as functions in SetlX.

The program computes the map r that represents the function $x \mapsto x * x$ on the set $\{n \in \mathbb{N} \mid 1 \leq n \wedge n \leq 10\}$. In line 3, the relation is evaluated at $x = 3$. This is done using square brackets.

```
1  r := { [n, n*n] : n in {1..10} };
2  print( "r[3]      = $r[3]$"      );
3  print( "domain(r) = $domain(r)$" );
4  print( "range(r)  = $range(r)$"  );
```

Figure 2.4: Binary relations as functions.

Finally, *domain*(r) and *range*(r) are computed. We get the following result:

```
r[3]      = 9
domain(r) = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
range(r)  = {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

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 2.5 on page 18 computes the answer to 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 2.5: 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 5 of the program just 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, the program from Figure 2.5 yields the following results:

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

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 explicitely 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 2.6 defines a

procedure to compute all prime numbers up to a given 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 2.6: A procedure to compute the prime numbers.

In Figure 2.6, 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*.

2.7 Strings

Any sequence of characters enclosed in double 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`]`.

The slicing operators work 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"`.

SetlX provides *string interpolation*: If a string contains a substring enclosed in "`$`"-symbols, then SetlX parses this substring as an expression and substitutes the result 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.
```

String interpolation is turned off by the prefix operator "`@`": The command

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

prints

```
$n$!  = $n!$.
```

2.7.1 Literal Strings

Sometimes it is necessary to turn off any kind of processing when writing 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.

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 openeing and closing quotes to delimit the string, they are not part of the string.

Literal strings are very handy when writing *regular expressions*.

2.8 Terms

SetlX provides first order terms similar to the terms available in the language *Prolog*. Terms are built from *functors* and *arguments*. To distinguish functors from function symbols, a functor will start with a capital letter. For example, the expression

```
F(1,"x")
```

is a term with functor `F` and two arguments. Here, 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()).
   ```

Of course, this term can also be given directly as an expression. For example, 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 2.7: Inserting an element into a binary tree.

Figure 2.7 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 number 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. 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 number stored at the root, while the arguments `l` and `r` correspond to the left and the right subtree of `m` respectively.

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

Sometimes, the fact that a functor has to be written in uppercase is inconvenient. For these cases, there is an escape mechanism: If the functor of a term is preceded be the symbol "`@`", then it may start with a lowercase letter. For example, if we want to have terms representing algebraic expressions involving trancendental functions, we could use the same function names that are also used in SETLX to denote the corresponding trancendental functions. For example, we can write

```
t := @exp(@log(@x));
```

to represent the term

`exp(log(x))`.

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

Chapter 3

Writing Statements in SetlX

This section discusses the various possibilities to write statements and the features offered by SetlX to steer the control flow in a program. SetlX supports the follwing statements to control the flow of execution:

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

3.1 Assignment Statements

The most basic command is the assignment. In contrast to the programming languages `C` and *Java*, SetlX uses the operator "`:=`" for assignments. 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: 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 an anonymous variable. For example,

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

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 25 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 $p \,\%\, f = 0$. The second function `primes` takes a natural number n as an argument and computes the set of all those prime numbers that are less or equal to n.

The idea is that a number p is prime iff the set of all factors of p contains just 1 and p. Note that, as SetlX is a functional language, the functions that are defined by the keyword `procedure` are assigned to variables. These functions can be used like any other values.

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

Figure 3.1: A naive program to compute primes.

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. Note that in SetlX the name of a variable has to start with a lower case letter.

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

   paramList → (*paramSpec* ("`,`" *paramSpec*)*)?

   Therefore, a *paramList* is a possibly empty list of parameter specifications that are separated by a comma "`,`". A parameter specification is either just a variable or it is a variable preceded by the token "`rw`":

   paramSpec → (`ID` | "`rw`" `ID`).

   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. Parameters not specified as read-write parameters have a strict *call by value semantics*, and therefore 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 "`;`".

There is a variant syntax for defininig 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*. If the function takes more than one arguments, the argument list has to be enclosed in square brackets. For example, the function `hyp` that computes 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 → `ID` "`:=`" *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 3.2 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 3.2: An example of a lambda definition in use.

Of course, it is much easier to build the list of the first 10 squares using the statement

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

3.2.1 Memoization

The Fibonacci function $fib : \mathbb{N} \to \mathbb{N}$ is defined recursivly by the following equations:

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

These equations are readily implemented as shown in Figure 3.3. However, this implementation has a performance problem which can be easily seen when tracing the computation of $fib(4)$.

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

Figure 3.3: 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 3.4: Tracing the computation of the Fibonacci function.

In order to trace the computation, we change the program as shown in Figure 3.4. If we evaluate the expression `fibonacci(4)`, we get the output shown in Figure 3.5. 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

$$\texttt{fibonacci(4)} := \texttt{fibonacci(3)} + \texttt{fibonacci(2)}$$

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 3.5: 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. Figure 3.6 shows an implementation of the Fibonacci function as a cached function. If we compare the program in Figure 3.6 with our first attempt shown in Figure 3.3, 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 3.6: 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 reads global variables that might change.

   A variable `x` can be declared to be a global variable by writing

   ```
   var x;
   ```

   In that case, the variable is visible in the scope of all functions.

2. The function makes use of random numbers.

3. The function reads input either form 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 3.6 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 in order to compute the values of `fibonacci` for different arguments. The reason is that the values for the last three arguments 99, 100, and 101 have not yet been used for the computation of different values, but all other arguments have been used at least once for computing another value.

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 3.7 on page 29 shows a recursive function that computes the binary representation of a natural number.

```
1  toBin := procedure(n) {
2      if (n < 2) {
3          return str(n);
4      } else {
5          r := n % 2;
6          n := floor(n / 2);
7          return toBin(n) + toBin(r);
8      }
9  };
```

Figure 3.7: 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 3.8 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-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 3.8: A function to sort a list of three elements.

Figure 3.9 on page 30 shows how to sort a list of length 3 using a switch statement. Although this implementation is easier to understand, it is less efficient than the previous version. 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 are already covered.

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

The grammar rule for `switch` statements is as follows:

stmnt → "`switch`" "{" *caseList* "}"

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

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

Here, *boolExpr* is a boolean expression and *block* is a list 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 a case condition. There are two other important distinction between the `switch` statement in *Java* and the `switch` statement in SetlX:

1. The keyword switch is not followed by a value.
2. The conditions following the keyword `case` have to be Boolean values.

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

3.4 Matching

One of the most powerful branching construct is *matching*. Although the syntax for the matching statement is always the same, there are really four different types of matching for each of the types that matching has been implemented. Matching has been implemented for strings, list, 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 has to be split into its first character c and the remaning 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 have to make this kind of case distinction, SetlX provides the `match` statement. Consider the function[1] `reverse` shown in Figure 3.10. 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. If 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 3.10 can also be used to reverse a list.

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 function `reverse` has a default case in line 5 to deal with those cases where s is not a string.

```
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 3.10: A function that reverses a string.

The last 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 the corresponding components. Basically, the function `reverse` given above is interpreted as if it had been written in the way shown in Figure 3.11 on page 32. This example shows that the use of the `match` statement can make programs more compact while increasing their legibility.

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 3.12 on page 32. Notice that we can match the empty string with the pattern "`[]`" in line 3. The pattern "`[c]`" matches a string consisting of one 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 then 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

[1] There is also a predefined version of the function `reverse` which does exactly the same thing as the function in Figure 3.10. However, once we define the function `reverse` as in Figure 3.10, the predefined function gets overwritten and is no longer accessible.

```
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 3.11: A function to reverse a string that does not use matching.

```
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 3.12: A function to exchange pairs of characters.

```
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".

Therefore, list assignment can be seen as a lightweight alternative to matching.

3.4.2 List Matching

As strings can be seen as lists of characters, the matching of lists is very similar to the matching of strings. The function `reverse` shown in Figure 3.10 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 $z = 3$.

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 3.13 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 3.13: A function to sort a set of numbers.

Of course, sorting a set into a list is trivial, as a set in SETLX is represented by 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.
```

In general, for a list l and a set s the exression l `+` s creates a new list containing all elements of l. Then, the elements of s are appended to this list. The expression s `+` l creates a new set containing the elements of s. Then, the elements of l are inserted into this set.

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. Figure 3.14 shows an implementation of the function `insert` that uses term matching instead of the functions "`fct`" and "`args`" that had been used in the previous implementation shown in Figure 2.7 on page 22. In line 3 of Figure 3.14, the `case` statement checks whether m is the empty tree. This is more straightforward than testing that the functor of m is "`Nil`", as it is done in line 3 of Figure 2.7. However, the real benefit of matching shows in line 5 of Figure 3.14 since the case statement in this line does not only check whether the functor of m is "`Node`" but also assigns the variables `k2`, `l`, and `r` to the respective subterms of m. Compare this with line 5 and line 6 of Figure 2.7 where we had to use a separate statement in line 6 to extract the arguments of m.
Let us discuss a more complex example of matching. The function `diff` shown in Figure 3.15 on page 34 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 string that is interpreted as the name of a variable.

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

$$x \mapsto x^x$$

```
insert := procedure(m, k1) {
    match (m) {
        case Nil() :
             return Node(k1, Nil(), Nil());
        case Node(k2, l, r):
             if (k1 == k2) {
                 return Node(k1, l, r);
             } else if (compare(k1, k2) < 0) {
                 return Node(k2, insert(l, k1), r);
             } else {
                 return Node(k2, l, insert(r, k1));
             }
        default: abort("Error in insert($m$, $k1$, $v1$)");
    }
};
```

Figure 3.14: Inserting an element into a binary tree using matching.

```
diff := procedure(t, x) {
    match (t) {
        case t1 + t2 :
            return diff(t1, x) + diff(t2, x);
        case t1 - t2 :
            return diff(t1, x) - diff(t2, x);
        case t1 * t2 :
            return diff(t1, x) * t2 + t1 * diff(t2, x);
        case t1 / t2 :
            return ( diff(t1, x) * t2 - t1 * diff(t2, x) ) / t2 * t2;
        case f ** g :
            return diff( @exp(g * @ln(f)), x);
        case ln(a) :
            return diff(a, x) / a;
        case exp(a) :
            return diff(a, x) * @exp(a);
        case ^variable(x) : // x is defined above as second argument
            return 1;
        case ^variable(y) : // y is undefined, matches any other variable
            return 0;
        case n | isNumber(n):
            return 0;
    }
};
```

Figure 3.15: A function to perform symbolic differentiation.

we can invoke `diff` as follows:

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

Here the function `parse` transforms the string "`x ** x`" into a term. The form of this term will

be discussed in more detail later. For the moment, let us focus on the `match` statement in Figure 3.15. Consider line 3: 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 8. This line implements the product rule:

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

Note how the pattern

```
t1 * t2
```

in line 7 extracts the two factors from a term that is a product. Further, note that in line 12 and line 16 we had to prefix the function symbols "`exp`" and "`ln`" with the character "`@`" in order to convert these function symbols into functors.

In order to understand this example in Figure 3.15 in more detail, we have to discuss how the function `parse` converts a string into a term. The function `parse` needs to represent all operator symbols and it also needs to represent variables. A variable of the form `"x"` is parsed as a term of the form

```
^variable("x").
```

This should explain the patterns used in line 19 and line 21 of Figure 3.15. In order to inspect the internal representation of a term, we use the function "`canonical`". For example, the expression

```
canonical(parse("x ** x"))
```

yields the result

```
^power(^variable("x"), ^variable("x")).
```

This shows that the functor "`^power`" is the internal representation of the power operator "`**`". The internal representation of "`+`" is "`^sum`", "`-`" is represented as "`^difference`", "`*`" is represented as "`^product`", and "`/`" is represented as "`^quotient`".

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, the expression

```
parse("x") + 2
```

yields the term

```
^sum(^variable("x"), 2).
```

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 21 shows how a condition can be attached to a pattern: The pattern

```
case n:
```

would match anything. However, we want to match only numbers here. Therefore, we have used the pattern

```
case n | isNumber(n):
```

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

3.4.5 Term Decomposition via List Assignment

Similar to strings, terms can be decomposed via list assignment. For example, after the assignments

```
t := F(1, G(2), 2,3);
[x,y,z] := F(1, G(2), {2,3});
```

the variables x, y, and z have the values

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

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 two different kinds of loops: `for` loops and `while` loops. The `while` loops are the most general loops. Therefore, we discuss them first.

3.5.1 `while` Loops

The syntax and semantics of `while` loops in SetlX is really the same as in the programming language `C`. To demonstrate a `while` loop, let us implement a function testing the *Ulam conjecture*: Define the function

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

recursively as follows:

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

Then f is well-defined and, furthermore, $f(n) = 1$ for all $n \in \mathbb{N}$. Figure 3.16 shows an implementation of this function in SetlX.

```
1  f := procedure(n) {
2      if (n == 0) {
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 3.16: A program to test the Ulam conjecture.

The function f is implemented via a `while` loop. This loop runs as long as n is different from 1. 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. *block* is a list of statements. Note that 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 3.17 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) {
6              continue;
7          }
8          if (n == 4) {
9              break;
10         }
11         print(n);
12     }
13 };
```

Figure 3.17: This function demonstrates the semantics of `break` and `continue`.

3.5.2 `for` Loops

In order to perform a list of commands a predefined number of times, a `for` loop should be used. Figure 3.18 on page 38 shows some SetlX code that prints a multiplication table. The output of this program is shown in Figure 3.19 on page 38.

In the program in Figure 3.18, the printing is done in the two nested loops that start in line 8. 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*)* “`)`” “{” *block* “}”.

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

x “`in`” s.

Here 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); }
```

```
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 3.18: 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 3.19: Output of the program in Figure 3.18.

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 that contains lists of length n as elements. Figure 3.20 on page 39 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 following `if` statement selects those pairs of pairs of numbers such that the second component of the first pair is identical to the first component of the second pair.

Of course, in SetlX the relational product can be computed more easily via set comprehension. Figure 3.21 on page 39 shows an implementation that is based on set comprehension. It is a bit shorter as the test in line 4 of Figure 3.20 is essentially integrated in the definition of the set in line 2 of Figure 3.21. In general, most occurences of `for` loops can be replaced by equivalent set definitions. In most cases, the resulting code will be both shorter and easier to understand.

Iterators can be even more complex, since the tuples can be nested, so something like

```
for ([[x,y],z] in s) { ··· }
```

```
1  product := procedure(r1, r2) {
2      r := {};
3      for ([x, y1] in r1, [y2, z] in r2) {
4          if (y1 == y2) {
5              r += { [x, z] };
6          }
7      }
8      return r;
9  };
```

Figure 3.20: A program to compute the relational product of two binary relations.

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

Figure 3.21: Computing the relational product via set comprehension.

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

A `for` loop creates a local scope for the iteration variable. Therefore, the last line of the program shown in Figure 3.22 prints the message

```
x = 1.
```

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

Figure 3.22: 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 SetlX is based on *Java*, the syntax of the regular expressions supported by SetlX is the same as the syntax of regular expressions in *Java*. Therefore, it is not necessary to describe the syntax of regular expressions in this document. 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

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. Therefore, we will confine ourselves to show how regular expressions can be used in SetlX. In SetlX, there are 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 can takes a string 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 in line 5, since the regular expression

```
'\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`".

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 diffferent 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 remaning 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 the 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 instantiated with the country code. The groups of regular expressions that are not needed are

matched against the anonymous variable "_".

If there a regular expression contains nested groups of parentheses, then the order of the groups is determined by the left parenthesis of a 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 "//", while the regular expression

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

in line 6 matches comments that start with the string "/*" and end with the string "*/". This regular expression is quite difficult to read for two reasons:

1. We have to preceed the symbol "`*`" with a backslash in order to prevent it from being interpreted as a 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 follwing specification:

   (a) A part may consists of any character different from the character "`*`". This is specificed 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 "`*/`".

```
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 of length one. The first element of this list is the string "`file.stlx`". Therefore, the `for` loop in line 11 will call the function `printComments` with the string "`file.stlx`". 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 string "`|`". Figure 4.5 shows a program to search for palindroms in a given file. Line 6 shows how a condition can be attached to a `regex` clause. The regular

expression

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

matches any number of letters, but the string `c[1]` corresponding to the match is only added to the set of palindroms if the predicate `isPalindrom[c[1])` yields `true`.

```
1  findPalindrom := procedure(file) {
2      all := split(join(readFile(file), "\n"), '[^a-zA-Z]+');
3      palindroms := {};
4      for (s in all) {
5          match (s) {
6              regex '[a-zA-Z]+' as c | isPalindrom(c[1]):
7                    palindroms += { c[1] };
8              regex '.|\n':
9                    // skip rest
10         }
11     }
12     return palindroms;
13 };
14
15 isPalindrom := procedure(s) {
16     n := #s + 1;
17     return +/ [s[n - i] : i in [1 .. #s]] == s;
18 };
```

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

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 shows a function that instead extracts all comments form 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. The 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 inbetween 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 the scanned using the `scan` statement in line 3. The `scan` statement is explained in more detail below.

The general form of a scan statement is as follows:

```
scan (s) {
   regex r1 as l : b1
   ⋮
   regex rn as l : bn
}
```

Here, s is a string to be analyzed, the variables r_1, $\cdots$, r_n are 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.

```
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.

(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 are 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 a 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.

2. This is repeated as long as the string s is not empty. Therefore, a `scan` statement is like 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. The first line of this file shows that this is an exam about algorithms and the secomnd line gives the name of the group. The fourth 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 an exercise has not been worked on. In order to calculate marks, we just have to add up all the points. From this, the mark can easily be calculated.

Figure 4.8 shows a program that does 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 best mark. This number is a parameter that is needed to calculate the marks.

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"`. Initially, the variable `state` has this value.

```
Exam:  Algorithms
Group: TIT07AIX

Exercises:               1. 2. 3. 4. 5. 6.
Max Müller-Lüdenscheidt: 8  9  8  -  7  6
Daniel Dumpfbacke:       4  4  2  0  -  -
Susi Sorglos:            9 12 12  9  9  6
Jacky Jeckle:            9 12 12  -  9  6
```

Figure 4.7: Typical results from an exam.

```
evalExam := procedure(file, maxPoints) {
    all := join(readFile(file), "\n");
    state := "normal";
    scan(all) using map {
        regex '[a-zA-Z]+:.*\n': // skip header
        regex '[A-Za-zäöüÄÖÜß]+\s[A-Za-zäöüÄÖÜß\-]+:' as [ name ]:
              nPrint(name);
              state := "printBlanks";
              sumPoints := 0;
        regex '[ \t]+' as [ whiteSpace ] | state == "printBlanks":
              nPrint(whiteSpace);
              state := "normal";
        regex '[ \t]+' | state == "normal":
              // skip white space between points
        regex '0|[1-9][0-9]*' as [ number ]:
              sumPoints += int(number);
        regex '-':
              // skip exercises that have not been done
        regex '\n' | sumPoints != om:
              print(mark(sumPoints, maxPoints));
              sumPoints := om;
        regex '[ \t]*\n' | sumPoints == om:
              // skip empty lines
        regex '.|\n' as [ c ]:
              print("unrecognized character: $c$");
              print("line:   ", map["line"]);
              print("column: ", map["column"]);
    }
};
mark := procedure(p, m) {
    return 7.0 - 6.0 * p / m;
};
```

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

(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.

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

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

5. The regular expression inline 5 matches the name of a student. This name is printed and the number of points for this student is set to 0. Furthermore, when the name of a student is seen, the state is changed to the state `"printBlanks"`.

6. The regular expression in line 10 matches the white space following the name of a student. This white space is then printed and the state is switched back to `"normal"`.

7. Line 13 skips over white space that is encountered in state `"normal"`.

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

9. Line 17 skips hyphens as these correspond to an exercise that has not been attempted by the student.

10. Line 19 checks whether we are at the end of a line listingthe points of a student. 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.

11. Any empty lines are skipped in line 22.

12. Finally, if we encounter any remaing character, then there is a syntax error in our input file. In this case, line 24 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.

4.3 Additional Functions Using Regular Expressions

There are three 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 serveral *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".
   ```

Chapter 5

Functional Programming and Closures

We have already stated earlier 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 in the type of a function or, say, the type of a rational number, as both can be assigned to variables, converted to strings, parsed, etc. This works because functions are actually implemented as *closures* similar to the way this is done in languages like *Scheme* [SS75]. In order to present closures, we present some simple examples first. 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 idea of functional programming. After that, we discuss the notion of a closure.

5.1.1 Introducing Functional Programming

Using functions as arguments to other functions is something that is not often seen in conventional programming languages. Figure 5.1 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 function f mapping two arguments of l into a value.

The function `reduce` combines successive arguments of the list l 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`$(l,$ `add`$)$ sums up all the arguments of l. Therefore, in line 11 the sum $\sum_{i=1}^{n} i$ is assigned to the variable x. If instead the second arguments of the function `reduce` is the function `multiply` that returns the product of its arguments, then `reduce`$(l,$ `multiply`$)$ computes the product of all elements of the list 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 *second order function* because one of its arguments is itself a function. While the example given above might seem artificial, there are common applications of this scheme. One example is sorting. Figure 5.2 on page 50 shows a generic implementation of the merge sort algorithm. Here, the function `sort` takes two arguments:

```
1  reduce := procedure(l, f) {
2      match (l) {
3          case []     : return;
4          case [x]    : return x;
5          case [x,y|r]: return reduce([f(x,y) | r], f);
6      }
7  };
8  add      := procedure(a, b) { return a + b; };
9  multiply := procedure(a, b) { return a * b; };
10
11 l := [1 .. 36];
12 x := reduce(l, add,    );
13 y := reduce(l, multiply);
```

Figure 5.1: Implementing a second order function.

```
1  sort := procedure(l, cmp) {
2      if (#l < 2) { return l; }
3      m := #l \ 2;
4      [l1, l2] := [l[.. m], l[m+1 ..]];
5      [s1, s2] := [sort(l1, cmp), sort(l2, cmp)];
6      return merge(s1, s2, cmp);
7  };
8  merge := procedure(l1, l2, cmp) {
9      match ([l1, l2]) {
10         case [[], _] : return l2;
11         case [_, []] : return l1;
12         case [[x1|r1], [x2|r2]] :
13              if (cmp(x1, x2)) {
14                  return [x1 | merge(r1, l2, cmp)];
15              } else {
16                  return [x2 | merge(l1, r2, cmp)];
17              }
18     }
19 };
20 less    := procedure(x, y) { return x < y; };
21 greater := procedure(x, y) { return y < x; };
22 l  := [1,3,5,4,2];
23 s1 := sort(l, less);
24 s2 := sort(l, greater);
```

Figure 5.2: A generic sort function.

1. The first argument l is the list to be sorted.

2. The second argument `cmp` is a binary function that compares two list 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 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`.

So far, the function 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 denoted as Δf and 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 5.3: Computing the discrete derivative of a given function.

Figure 5.3 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 1 for ten times, while in the second case the the result is a list of the first ten powers of 2.

5.1.2 Implementing Counters via Closures

The concept of a closure is non-trivial, therefore we introduce the idea using some simple examples. Figure 5.4 shows the function `createCounter`. This function initializes the variable `count` to a given value and then returns a procedure that increments this value.

Notice that the variable `count` is defined outside the procedure `counter`. Still, the function `counter` has access to this variable: 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. Therefore, 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

```
1   createCounter := procedure(i) {
2       count   := i;
3       counter := procedure() {
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 5.4: Creating a counter as a closure.

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.

You should also 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 `u`, `v`, and `w` have the values 10, 11, and 12.

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 that converts this regular expression into a non-deterministic finite state machine using the *Thompson construction* [HMU06]. For the purpose of this section, *regular expressions* are defined inductively as follows.

1. Every character c 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 is a single character is represented by the string consisting of 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.

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.

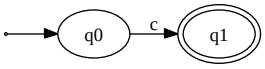

Figure 5.5: The finite state machine $A(c)$.

   Figure 5.6 shows the function `genCharNFA` that takes a character c and generates the fsm $A(c)$. The code corresponds closely 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 57. 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 singelton set $\{q_1\}$. Since this is the only transition of the fsm, in all other cases the set of next states is empty.

```
1  genCharNFA := procedure(c) {
2      q0 := getNewState();
3      q1 := getNewState();
4      delta := procedure(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.

2. In order to compute the fsm $A(r_1 \cdot r_2)$ we have to assume that the states of the fsms $A(r_1)$ and $A(r_2)$ are different. Let us assume that $A(r_1)$ 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 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 $A(r_1 \cdot r_2)$.

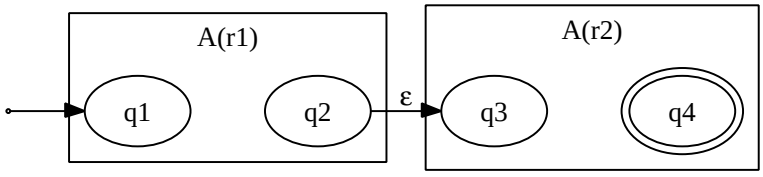

Figure 5.7: The finite state machine $A(r_1 \cdot r_2)$.

Figure 5.8 on page 55 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

```
1   catenate := procedure(f1, f2) {
2       [m1, delta1, q1, q2] := f1;
3       [m2, delta2, q3, q4] := f2;
4       delta := procedure(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 $A(r_1 \cdot r_2)$

$$A(r_1) = \langle Q_1, \delta_1, q_1, q_3 \rangle \quad \text{and} \quad A(r_2) = \langle Q_2, \delta_2, q_2, q_4 \rangle$$

the 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.

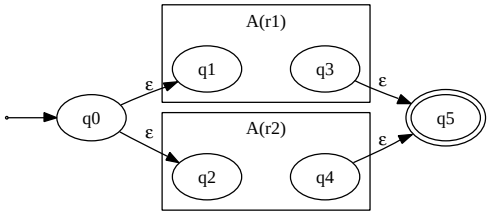

Figure 5.9: The finite state machine $A(r_1 + r_2)$.

Figure 5.10 on page 56 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

$$A(r_1 + r_2) = \texttt{catenate}\big(A(r_1), A(r_2)\big).$$

4. In order to define $A(r^*)$ we assume $A(r)$ is given as

```
1   disjunction := procedure(f1, f2) {
2       [m1, delta1, q1, q3] := f1;
3       [m2, delta2, q2, q4] := f2;
4       q0 := getNewState();
5       q5 := getNewState();
6       delta := procedure(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 $A(r_1 + r_2)$.

$$A(r) = \langle Q, \delta, q_1, q_2 \rangle.$$

Then $A(r^*)$ is defined as

$$\langle \{q_0, q_3\} \cup Q, \{\langle q_0, \varepsilon\rangle \mapsto q_1, \langle q_2, \varepsilon\rangle \mapsto q_1, \langle q_0, \varepsilon\rangle \mapsto q_3, \langle q_2, \varepsilon\rangle \mapsto q_3\} \cup \delta, q_0, q_3 \rangle.$$

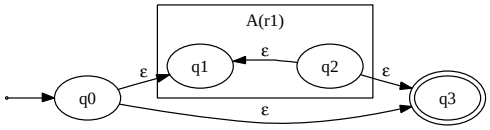

Figure 5.11: The finite state machine $A(r^*)$.

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.

Figure 5.12 on page 57 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

$$A(r^*) = \texttt{kleene}\big(A(r)\big).$$

```
kleene := procedure(f) {
    [m, delta0, q1, q2] := f;
    q0 := getNewState();
    q3 := getNewState();
    delta := procedure(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 $A(r^*)$.

```
var gStateCount;
gStateCount := -1;

getNewState := procedure() {
    gStateCount += 1;
    return "q" + gStateCount;
};
```

Figure 5.13: A function to generate unique states.

Finally, Figure 5.14 shows how to generate a finite state machine from a given regular expression. 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`$\big(A(r_0)\big)$.

The reader should note that we have made heavy use of closures to implement the functions `genCharNFA`, `catenate`, `disjunction`, and `kleene`.

```
1  regexp2NFA := procedure(r) {
2      match (r) {
3          case c | isString(c):
4               return genCharNFA(c);
5          case Cat(r1, r2):
6               return catenate(regexp2NFA(r1), regexp2NFA(r2));
7          case Or(r1, r2):
8               return disjunction(regexp2NFA(r1), regexp2NFA(r2));
9          case Star(r0):
10              return kleene(regexp2NFA(r0));
11     }
12 };
```

Figure 5.14: Generating a non-deterministic fsm from a regular expression.

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 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 as in *Java* and uses the keywords "`try`" and "`catch`". If we have a sequence of statements *stmntList* and we fear 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* does raise an exception, then the execution of *stmntList* is aborted and instead the statement *errorCode* is executed.

Typically, exception handling is necessary when processing user input. Consider the program shown in Figure 6.1. The function `findZero` implements the bisection method which can be used to find the zero of a function. The first argument of `findZero` is a function f, while the arguments a and b are the left and right boundary of the intervall 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 $[a, b]$.

The function `askUser` 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 to the `read` command 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, furthermore, the resulting string is parsed and 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.

```
1  askUser := procedure() {
2      try {
3          s := read("Please enter a function: ");
4          f := evalTerm(parse("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         askUser();
13     }
14 };
15 findZero := 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.

There are a couple of things that can go wrong with the function `askUser`. First, the string `s` input by the user might not be a proper function and then we would probably 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 either line 17 or line 21. Since we don't want to abort the program on the occurrence of an exception, the whole block of code in lines 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 `askUser` recursively so that the user of the program gets another chance.

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.

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 the 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 and is therefore not caught.

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, using unrestricted `catch` clauses counters this strategy and might lead to errors 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 *Prolog* programs that are shown in the text books describing *Prolog* [SS86, Bra90] it can be seen that very few programs actually make use of backtracking in its most general form. Also, the personal experience of the first author, who has programmed in *Prolog* for more than 10 years, suggests that *Prolog* programs that use backtracking in an unrestricted fashion tend to be very hard to maintain. In general, it is our believe that the use of backtracking should always follow the paradigm *generate and test*:

1. The set of possible values should be generated by a *generating function*.
2. These values should then be tested. If a test fails, the program backtracks to step 1 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 form

```
try {
   stmntList
} catch (e) {
   if (e != "fail") {
      throw(e);
   }
}
```

while the keyword "`backtrack`" is translated into "`throw("fail")`".

The program in Figure 6.2 on page 63 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. The procedure `solve` has two parameters.
   (a) The first parameter `l` 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 `l` do not attack each other.

[1] Technically, instead of the string `"fail"`, SetlX generates a unique exception, which can only be catched using `check`.

```
1  solve := procedure(l, n) {
2      if (#l == n) {
3          return l;
4      }
5      for (x in {1 .. n} - {i : i in l}) {
6          check {
7              testNext(l, x);
8              return solve(l + [x], n);
9          }
10     }
11     backtrack;
12 };
13 testNext := procedure(l, x) {
14     m := #l;
15     if (exists (i in {1 .. m} | i-l[i] == m+1-x || i+l[i] == m+1+x)) {
16         backtrack;
17     }
18 };
```

Figure 6.2: Solving the 8 queens puzzle using `check` and `backtrack`.

Technically, `l` is a list on integers. If `l[`i`]` $= k$, then row i contains a queen in column k. For example,

$$l = \texttt{[4, 8, 1, 3, 6, 2, 7, 5]}$$

is a solution of the 8 queens puzzle. This solution is depicted in Figure 6.3.

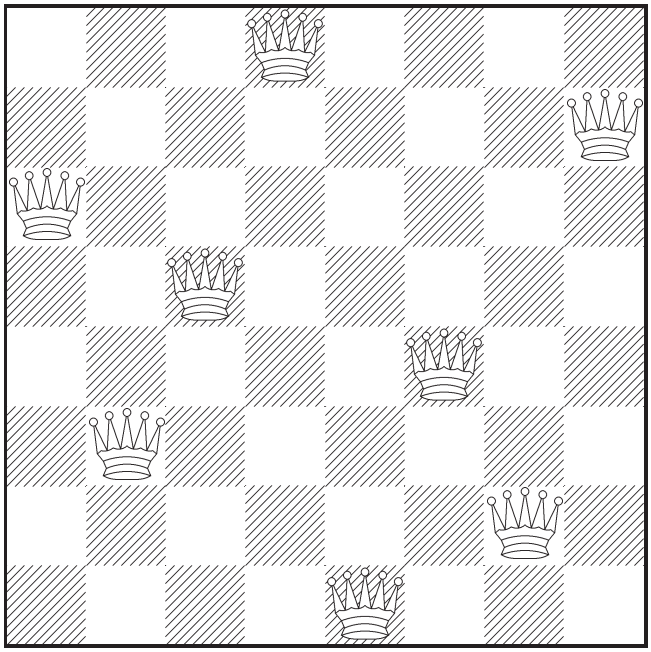

Figure 6.3: A solution of the 8 queens puzzle.

(b) The second parameter `n` is the size of the board.

In order to solve the 8 queens puzzle, the procedure `solve` is called in line 26 in the form

```
solve([], 8).
```

Taking the parameter `l` to be the empty list assumes that initially no queen has been set on the chess board. Of course, then the assumption that the queens already positioned in `l` do not attack each other is trivially satisfied.

2. In line 2 it is checked, whether the list `l` already specifies the positions of `n` queens. If this is the case, then because of the assumption that the queens specified in `l` do not attack each other, the problem is solved and therefore `l` is the solution and 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 row that has already been taken by one of the queens in the list `l`. Therefore, the number of positions available for the next queen is given by the set

   $\{$`1 .. n`$\}$ `-` $\{$`i : i in l`$\}$.

   Note that we had to convert the list `l` into the set $\{$`i : i in l`$\}$ in order to be able to subtract the positions specified in `l` from the set of all possible positions.

4. Once we have decided to position the next queen in row `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 `l` 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 next value of `x`. Now 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 check has failed. The answer is that the function `testNext` includes a call to `backtrack` if it is not possible to place a queen in position `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 `l + [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 statment 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 `l`.

   In order to understand the calculation in line 15 we have to realize that the cartesian coordinates of the queens in column `x` are

   $\langle$`#l` $+ 1,$ `x`$\rangle$.

   Now a diagonal is specified as the equation of a line with slope either $+1$ or -1. The i-th queen in `l` has the coordinates

$\langle$`i`, `l`[`i`]$\rangle$.

Therefore, it is on the same ascending diagonal as the queen specified by `x` if we have

`i-l[i]` = `#l+1-x`,

while it is one the same descending diagonal if we have.

`i+l[i]` = `#l+1+x`.

It is easy to change the program in Figure 6.2 such that all solutions are completed. Figure 6.4 on page 66 shows how this is done.

1. We have added a function `allSolutions`. This function gets on 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 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` is actually changed by the procedure `solve`.

3. The important change in the implementation of `solve` is that instead of returning a solution, a solution that is found is added to the set `all` in line 10. After that, the function `solve` backtracks to look for more solutions.

4. The implementation of the function `testNext` has not changed.

This program finds all 92 solutions to 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.

```
allSolutions := procedure(n) {
    all := {};
    check {
        solve([], n, all);
    }
    return all;
};
solve := procedure(l, n, rw all) {
    if (#l == n) {
        all += { l };
        backtrack;
    }
    for (x in {1 .. n} - {i : i in l}) {
        check {
            testNext(l, x);
            return solve(l + [x], n, all);
        }
    }
    backtrack;
};
testNext := procedure(l, x) {
    m := #l;
    if (exists (i in {1 .. m} | i-l[i] == m+1-x || i+l[i] == m+1+x)) {
        backtrack;
    }
};
```

Figure 6.4: Computing all solutions of the n queens puzzle.

Chapter 7

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.
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. The last sections lists all those procedures that did not fit in any of the previous sections.

7.1 Functions and Operators on Sets and Lists

Most of the operators and functions that are supported on sets and lists have already been discussed. However, for the convenience of the reader, this section describes all operators and functions. However, the discussion of those functions that have already been described previously will be quite short.

1. `+`: For sets, the binary operator "`+`" computes the union of its arguments. For lists, this operator appends its arguments.
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 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 only defined for sets.

   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 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.

6. `*/`: 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.

7. `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.

8. `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.

9. `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.

10. `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.

11. `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.

12. `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.

13. `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}`.

14. `max`: If s is a set or a list, the expression `max`(s) computes the biggest element of s. For example, the expression

    ```
    max({1,2,3})
    ```

    returns the number 3. In general, the elements in s are compared using the function `compare` discussed in the last section of this chapter. For example, strings are compared lexicographically, therefore

    ```
    max({"abc", "xy", "z" })
    ```

    yields the result `"z"`.

15. `min`: If s is a set or a list, the expression `min`(s) computes the smallest element of s. For example, the expression

    ```
    min({1,2,3})
    ```

    returns the number 1. In general, the elements in s are compared using the function `compare`. For example, strings are compared lexicographically, therefore

    ```
    min({"abc", "xy", "z" })
    ```

    yields the result `"abc"`.

16. `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}}.
    ```

17. `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}.
    ```

18. `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].
    ```

19. `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].
    ```

7.2 Functions for String Manipulation

SetlX provides the following functions that are related to strings.

1. `char`: For a natural number $n \in \{1, \cdots, 127\}$, the expression

   `char`(n)

   computes the character with the Ascii code n. For example, `char`(65) yields the string `"A"`.

2. `endsWith`: The function `endsWith` is called as

   `endsWith`(s, t).

   Here, 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.

3. `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.

4. `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; };
   ```

5. `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.

6. `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.

7. `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.

8. `replaceFirst`: The function `replaceFirst` is called as

   `replace`(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.

9. `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 CVS files.

10. `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; }".
    ```

11. `toLowerCase`: Given a string s, the expression `toLowerCase`(s) converts all characters of s to lower case.

12. `toUpperCase`: Given a string s, the expression `toUpperCase`(s) converts all characters of s to upper case.

13. `trim`: Given a string s, the expression `trim`(s) returns a string that is s without all leading or trailing white space characters. For example, the expression

    ```
    trim(" abc xyz\n")
    ```

    returns the string `"abc xyz"`.

7.3 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(^variable("n"))],
              ^block([^return(^product(^variable("n"), ^variable("n")))])
             )
   ```

   However, there is one twist: 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, 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` disussed 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` disussed 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 easier readability):

   ```
   ^block([^assignment(^variable("x"), 1),
           ^assignment(^variable("y"), 2),
           ^assignment(^variable("z"), ^sum(^variable("x"), ^variable("y")))
          ])
   ```

7.4 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 divsion 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 *flooring* function of x. Mathematically, `floor`(x) is defined as the largest integer that is less than or equal to x:

   `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 and Euler's number e are supported. Therefore, the expression

   ```
   mathConst("pi")
   ```

   yields the result 3.141592653589793, while

   ```
   mathConst("e")
   ```

   yields 2.718281828459045.

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 real numbers. For example, the expression

   ```
   rational(mathConst("e"))
   ```

   yields the result 543656365691809/200000000000000. The expression `rational`(q) returns q if q is already a rational number.

10. `real`(s) converts the string s into a floating point number. For example the expression

    ```
    rational("1.2")
    ```

    will return the floating point number 1.2. The function `real` can also be invoked on rational numbers. For example, the expression

    ```
    real(2/7)
    ```

    yields 0.2857142857142857. The expression `real`(r) returns r if r is already a floating point number.

    Warning: SetlX implements floating point numbers via the *Java* class `BigDecimal` instead of using the primitive data type `double`. Therefore, floating point operations are noticeably slower than in other programming languages.

11. Furthermore, the trigonometrical functions `sin`, `cos`, `tan` and the associated inverse trigonometrical functions `asin`, `acos`, and `atan` are all supported. For example, the expression

```
sin(mathConst("pi")/2);
```

yields 1.0.

12. `exp(`x`)` computes e^x where e is Euler's number. Therefore, the expression

    ```
    exp(1)
    ```

    yields 2.718281828459046.

13. `log(`x`)` computes the natural logarithm of x. This function is the inverse function of the exponential function `exp`. Therefore, we have the equation

    `log(exp(`x`))` $= x$.

    This equation is valid as long as there is no overflow in the computation of `exp(`x`)`.

14. `log10(`x`)` computes the base 10 logarithm of x.

15. `sqrt(`x`)` computes the square root $\sqrt{x}$. Therefore, as long as x is not too large, the equation

    `sqrt(`x`*`x`)`$= x$

    is valid.

16. `cbrt(`x`)` computes the cubic root $\sqrt[3]{x}$. Therefore, as long as x is not too large, the equation

    `cbrt(`x`*`x`*`x`)` $= x$

    is valid.

17. `round(`x`)` rounds the number x to the nearest integer.

18. `nDecimalPlaces(`q, n`)` takes a 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`".

19. `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*.

20. `signum(`x`)` computes the sign 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.

21. `sinh(`x`)` computes the hyperbolic sine of x. Mathematically, the hyperbolic sine is defined as

    `sinh(`x`)` $:= \frac{1}{2} \cdot \left(e^x - e^{-x}\right)$.

22. `cosh(`x`)` computes the hyperbolic cosine of x. Mathematically, the hyperbolic cosine is defined as

    `cosh(`x`)` $:= \frac{1}{2} \cdot \left(e^x + e^{-x}\right)$.

23. `tanh`(x) computes the hyperbolic tangent of x. Mathematically, the hyperbolic tangent is defined as

$$\texttt{tanh}(x) := \frac{\texttt{sinh}(x)}{\texttt{cosh}(x)}.$$

24. `isPrime` tests whether its argument is a prime number. Currently, this function has a naive implementation and is therefore not efficient.

25. `isProbablePrime` tests whether its argument is a prime number. The test used is probabilistic. If the function tested is indeed prime, the test will succeed. If the function is not prime, the predicate `isProbablePrime` neverthelsee might 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.

7.5 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

$$0 \leq x \quad \text{and} \quad x \leq 1$$

holds. Normally, this will be a floating point number with 64 bits. If SetlX is instead invoked with the either of the parameters "`--real32`", "`--real128`", or "`--real256`" then x will have 32 bits, 128 bits, or 256 bits respectively. 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 `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

   $n \leq k$ and $k \leq 0$.

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 possiblities, 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

   $a/100$ 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 immmediately 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.

7.5.1 `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]`.

7.5.2 `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 7.1 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 7.1: Printing all permutations of a given list or string.

7.5.3 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]}.

7.6 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. `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 an error. 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.

3. `isInfinite` tests whether its argument is a real number that represents either positive or negative infinity.

4. `isInteger` tests whether its argument is an integer number.

5. `isList` tests whether its argument is a list.

6. `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

   $$\texttt{[}x\texttt{,}y_1\texttt{]} \quad \text{and } \texttt{[}x\texttt{,}y_2\texttt{]}$$

   such that $y_1 \neq y_2$.

7. `isNumber` tests whether its argument is a number.

8. `isProcedure` tests whether its argument is a procedure.

9. `isRational` tests whether its argument is a rational number.

10. `isReal` tests whether its argument is a floating point 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.

7.7 Interactive Debugging

SetlX supports a command line interface for interactive debugging. The interface to the debugger consists of the following functions:

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. `setBreak`: The expression

   `setBreak`(f)

   sets a breakpoint for the function specified by the argument f. In the context of `setBreak`, the argument f is a string that is the name of a function.

3. `rmBreak`: The expression

   `rmBreak`(f)

   removes the breakpoint for the function f, where f is the name of a function.

4. `lsBreak`: The expression

   `lsBreak()`

   lists all breakpoints.

5. `step`: In general, the expression

   ```
   step()
   ```

   executes the next statement. However, if the next statement evaluates a complex function, the `step()` only enters this function so that the programmer can then step through the statements of the called function.

6. `uStep`: The expression

   ```
   uStep()
   ```

   evaluates the expression that is needed to evaluate the next statement. This is called a *micro step*.

7. `fStep`: The expression

   ```
   fStep()
   ```

   executes a *macro step*: this executes the next instruction. If the next instruction is a function call, then this function call is evaluated in one step, so this function steps over function calls.

8. `finish`: The expression

   ```
   finish()
   ```

   finishes execution of the current function. This function is handy if you accidentally step into a function.

9. `finishLoop`: The expression

   ```
   finishLoop()
   ```

   finishes the execution of the current loop without halting.

10. `resume`: The expression

    ```
    resume()
    ```

    resumes normal execution of the program.

11. `reset`: The expression

    ```
    reset()
    ```

    stops the execution of the current program and returns to the interactive prompt of the interpreter.

Finally the function `dbgHelp()` lists all commands that are available to support interactive debugging.

7.8 I/O Functions

This section lists all functions related to input and output.

7.8.1 `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.

7.8.2 `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.

7.8.3 `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 `": "`.

7.8.4 `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.

7.8.5 `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.

7.8.6 `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 serveral 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.

7.8.7 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.

7.8.8 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.

7.8.9 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.

7.8.10 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.

7.8.11 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.

7.8.12 `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;
    }
}
```

7.8.13 `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.

7.9 Miscellaneous Functions

This final section lists some functions that did not fit into any of the other sections.

7.9.1 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`.

7.9.2 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.

7.9.3 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.

7.9.4 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.

7.9.5 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]
})
```

7.9.6 logo

The function `logo` is called as

`logo`().

In order to find how this function works, try it yourself.

7.9.7 now

The function `now` returns the number of milliseconds that have elapsed since the beginning of the Unix epoch.

7.9.8 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.

[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, 2009. Available at: `http://jflex.de/jflex.pdf`.

[Les75] Michael E. Lesk. Lex – A lexical analyzer generator. Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

[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.

[vR95] Guido van Rossum. Python tutorial. Technical report, Centrum Wiskunde & Informatica, Amsterdam, 1995.