Programming Languages at a Glance

Programming Languages at a Glance

Published 2003

Table of Contents

List of Tables

Chapter 1. Introduction

When reading programs in today's popular programming languages, I often find myself wondering if all this code is really needed. Is is still so combersome to tell the computer what to do? Do we have to code thousands of lines just to explain the system how to get some data from there, transform it a little bit, and present it to user in a nice way? You may answer that modern development environments take a lot of this burden of the programmer by generating the code (MDA being the culmination of this idea), but doesn't this mean that we could express the instructions in a better way to begin with?

These thoughts spurred my interest in programming languages. It looks like there are plenty of ideas which could help to improve the way we talk to a computer. But most of this knowledge stays within the academic circles where these languages have been developed. If you want to learn multiple programming languages, you have to read a lot of books, since most of them focus on a single language. For popular languages such as Java and C#, you can actually buy dozens if not hundreds of books covering the different aspects of these languages.

This book tries to explain a number of programming languages, covering a wide range from currently popular ones such as Java, Perl, Python, and C# to less known languages such as ML, Haskell. There is one key requirement: the language has to be available for free (or otherwise I could not afford writing this book). We don't expect a full blown IDE; a good set of command line tools is just as fine. Fortunately, this requirement is fulfilled by most languages these days, since they don't stand a chance otherwise. In many cases, even a free IDE is available (e.g., Eclipse for Java or Smalltalk/X).

When describing the languages, I want to find out what they have in common, and what distinguished them from each over. The presentation shows my own programming language background: I started with Pascal and Modula 2 before switching to C/C++ and finally moving to Java web development. In parallel to C++, I started using Python for scripting and some small applications. If you have a similar background, you will probably find it very easy to follow the text. Otherwise, you might want to read some of the references in parallel.

As a programmer, I find small examples more instructive than lengthy explanations. As as result, you find many small code examples -- there may be more code than explaining text in this book -- each covering a small piece of the language. Most of the languages in this book offer an interactive environment where you can enter expressions and directly see the results. This way you can type the examples, compare the outcome, and experiment yourself.

Another choice I had to make was the kind of development environment for each language. All of them provide a set of command line tools, and most of them also offer an integrated development environment which nice features such as syntax highlighting, code/class browser, debugger, and so forth. I decided to stick to the command line (or a general purpose text editor such as Emacs), because it is a lot more efficient (in terms of time an space) to paste code examples into the text than to deal with a multitude of screenshots.

The book focuses on the languages themselves rather than their libraries, history, or different versions. For the curious, I will add a few notes about the origin of the language, but mostly refer to other resources for more detailed information.

After trying multiple ways to organize the chapters I decided to order the languages more or less chronologically using the year when a language first appeared "in public". However, I do not guarantee correctness, since this year is sometimes not easy to determine.

As a result of the chronological order, you will find completely different programming paradigms right next to each other. The 1970s, for example, saw the parallel rise of object-oriented (Smalltalk), logical (Prolog), functional (Scheme), data-driven (SQL) languages.

There are many ways to read this book. I believe that it is short enough to be read cover to cover, let’s say, one language per subway trip. Just reading the quick tours is probably enough to make intelligent remarks to your collegues ("If we were coding it in XXX, we could do it in two lines"). Most chapters can be read independently of each other. Even for languages that have a lot in common (in particular Lisp/Scheme and SML/Ocaml), it turned out to be easier to start from scratch than to describe one in terms of the other. The main exception is the C family which is best understood as a progression from C to C++ (and Objective C) to Java to C#.

1.1. Acknowledgements

Being an amateur for most of the languages covered in this book, I rely on the help of others to correct the most blatant mistakes. Fortunately, many of my collegues at AMS and Vodafone read individual chapters and provided me with valuable feedback. Our Perl master Alexander Brueck helped to get the Perl chapter in shape. Carten Bucker reviewed to Smalltalk chapter (neither of us is a Smalltalk expert, but hopefully two one-eyed can get around the worst pitfalls). Volker Garske carefully reviewed the C chapter.

Chapter 2. Fortran

Fortran is the oldest high level languages. It was developed at IBM between 1954 and 1957 by a team lead by John Backus (well known for the Backus normal form, BNF, for syntax definitions). As its name (FORmula TRANslator) indicates, the language is designed for numerical computations. The first ANS standard was defined in 1966 (Fortran66), a major update came out in 1977 (Fortran77), and the latest official standard is Fortran90.

This chapter is based on Fortran77 which is probably the version with the largest code base to date. In some place, I will hint at some of the improvements contained in the Fortran90 standard.

2.1. Software and Installation

I am using GNU's Fortran compiler `f77` on a Linux system for the examples. To test a program, call `f77` followed by the name of the source file and run the resulting executable `a.out` (as usual you can also store the executable in another file using the `-o` option).

2.2. Quick Tour

2.2.1. Hello World

If the "Hello World" program tests the simplicity of a programming language, Fortran scores quite high in this contest. In general, Fortran's syntax keeps programs relatively short. In our first example, just the asterisk as the first argument of the print statement is not immediately clear from the context (it is a wildcard for the print format).

```
      PROGRAM HELLO
      PRINT *, 'Hello World'
      END
```

Apart from string literals, Fortran is case insensitive, and identifiers are normally written with uppercase characters. In Fortran77, only the first 6 characters of an identifier are significant (Fortran90: 31 characters).

Of course, we are back to punch cards again. The program text is devided into lines of 72 characters (everything beyond the 72nd character is ignored). If a line starts with an asterisk or a `C` it is considered a comment.

```
C This is the Hello World program.
      PRINT *, 'Hello World'
      END
```

Otherwise, the first five columns are reserved for statement labels and the sixth column for the continuation markers. The actual statements start in the seventh column.

If a statement spans multiple lines, the continuation lines are marked by some character in the sixth column (often a `$` sign, but any character is fine).

Statement labels are integer values used targets for jumps and loops. Fortunately, we hardly need statement labels anymore when using the latest version of Fortran. The following example echos the user's input (must be numbers) in an infinite loop.

```
 100  READ *, X
      PRINT *, 'X=', X
      GO TO 100
      END
```

After this short excursion into the formatting of the program text, let's get back to the language itself. Aimed at numerical computations, Fortran fully supports arithmetical expressions with correct precedences, exponentiation, and many built-in mathematical functions. The last print statement demonstrates complex literals with the read and imaginary part written as a pair (2i squared is -4).

```
      PRINT *, 2 + 1.5 * 3 - 2 ** 3
      PRINT *, 2 * ASIN(1.0)
      PRINT *, (0, 2) ** 2
      END
```

```
result:
 -1.5
  3.14159274
 (-4.,0.)
```

2.2.2. Expressions and Variables

In Fortran, variables do not have to be declared in advance. However, they are strongly typed. How is this possible? If a variable is not declared explicitly, the type is derived from the first character of the variable's name. By default, all variables starting with with a character between "I" and "N" are integers, and all others are real numbers.

```
      I=55
      X=55
      PRINT *, I, X
      END
```

```
result:
 55  55.
```

Note the period in the output indicating the real number. The implicit typing rules can be changed using the `IMPLICIT` statement.

```
      IMPLICIT INTEGER(X-Z)
      X=55
      PRINT *, X
      END
```

```
result:
 55
```

In general, Fortran does a lot implicitly, which keeps programs short, but sometimes also leads to surprises. Assigning a real number to an integer, for example, automatically truncates the number.

```
      I=5.6
      PRINT *, I
```

```
result:
 5
```

To be on the safe side, it is better to declare variables explicitly. This is done by stating the type followed by the names of the variables.

```
      DOUBLE PRECISION X, Y
      X=5.6
      Y=1.5E-10
      PRINT *, X, Y
      END
```

```
result:
  5.5999999  1.49999999E-10
```

Table 2-1> lists the six types available in Fortran.

Table 2-1. Fortran types

Name	Description	Sample Declaration	Sample Assign
CHARACTER	single 8-bit character; strings are declared by "multiplying" the type with the length of the string	CHARACTER NAME*20	NAME=’Homer’
INTEGER	integer (corresponding to C’s `int`); typically 32 bit on 32 bit architectures	INTEGER I	I=1234
REAL	single precision (32 bit) floating point number (corresponding to C’s `float`)	REAL X	X=-1.23E-3

Name	Description	Sample Declaration	Sample Assign
DOUBLE PRECISION	double precision (64 bit) floating point number(corresponding to C's `double`)	DOUBLE PRECISION X	X=-1.23D-3
COMPLEX	single precision complex floating point number	COMPLEX Z	Z=(1.23, 2.34)
LOGICAL	boolean	LOGICAL B	B=.TRUE.

The two boolean values are called `.TRUE.` and `.FALSE.`. Fortran77 has a very limited character set which does not even contain the < or > symbol. Therefore, comparison operators are denoted by shortcuts for the english expressions. `.LT.`, for example, means "less than" and `.NE.` means "not equal". Fortran99 now also provides the usual operator such as <= and `/=`.

```
B=X .LE. 5.0 .AND. 5 > 4)
```

2.2.3. Arrays

Since vectors and matrices play a very important role in numerical computations, arrays are one of Fortran's strengths. To define an array, we add the dimensions to the variable declaration. Indexing starts at one.

2.2.4. Functions

Fortran uses the function name as a variable. This means that we define the return type just like we define a variable, and we can use the function name to manipulate the return value just like any other variable.

```
PROGRAM FUNCTION_TEST
PRINT *, '1.5 + 2.5=', ADD(1.5, 2.5)
END
FUNCTION ADD(X, Y)
REAL X, Y, ADD
ADD = X + Y
END
```

Bibliography

[PAGE95]> is nice, compact introduction to Fortran77 which is available for free on the Internet (the book is out of print).

Clive G. Page, http://www.glue.umd.edu/~nsw/fortran/tutorial/prof77.ps, 1995, *Professional Programmer's Guide to Fortran77*.

Chapter 3. Common Lisp

Lisp is one of the oldest programming languages which is still in use (only Fortran is older). It was developed by John McCarthy in the late 1950's leading to the first Lisp interpreters in 1959. During the next 20 years a number of dialects developed which motivated a standardization effort during the 1980's which culminated in the definition of ANSI standard in 1992.

Common Lisp is a big language. Whether it is list processing, string formatting, or objects and classes: Lisp has always something special to offer. I can only hope that this chapter gives you an idea what you can do with Lisp. There is a light weight Lisp dialect, Scheme, which is built on the same foundation, but is much smaller (and in some areas cleaner).

3.1. Software and Installation

There are several good open source implementations of Common Lisp. Two popular choices are

- Clisp (http://clisp.cons.org), another GNU Common Lisp implementation, compiling to bytecode and therefore slower than the alternatives.
- CMU Common Lisp (http://www.cons.org/cmucl/) (CMUCL), a native Common Lisp compiler originally developed at the Carnegie Mellon University.

Most examples in this chapter can be run with any of these Common Lisp implementations. For Linux, they are available as package of the standard distributions (e.g., Debian packages "clisp" and "cmucl"). On Windows, the easiest choice is Clisp, for which a Windows binary is available at the official Clisp site. To install it, unpack the zip archive and follow the instructions in the readme file. In the configuration file `config.lisp`, just adapt the location of the clisp installation in the definition of `*load-paths*` to the where you unpacked the zip file (e.g., `c:\\Program Files\\clisp-2.32\\...\\`).

I have used Clisp for the most part and later CMU Common Lisp (for example, to run the database examples).

3.2. Quick Tour

3.2.1. Lists and Expressions

To somebody (like me) used to "normal" procedural programming languages with a rich syntax, Lisp is just different. The syntax consists basically of nested sequences of literals enclosed in parentheses. No keywords, no indentation (like in Python), no statements separated by periods or semicolons, no curly braces. However, there is one intriguing advantage to this approach: code and data use the same presentation and can be treated and manipulated with the same means.

The examples follow Geoffrey Gordon's lisp tutorial and some examples from Paul Graham's "Common Lisp" book. Starting clisp leads us to a Lisp shell similar to the Python shell in the first chapter.

```
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ '+' /  I      8         8           8     8        8    8
   \  '-+-'  /       8         8           8      ooooo   8oooo
    '-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8ooooooo  ooo8ooo   ooooo   8

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2002

[1]>
```

It works similar to the Python shell in that expression are evaluated and the result is shown in the shell. This means, that our "Hello World" program is again the shortest possible version:

```
> "Hello World"
"Hello World"
>
```

Lisp is just echoing the string value "Hello World". The differences between Lisp and other languages become apparent once we try to start evaluating simple expressions. The only way to evaluate something, is to enclose a function (i.e., the symbol of a function) followed by its arguments in parentheses. For an expression such as "4 + 5", this means that we have to follow the symbol "+" with the arguments "4" and "5".

```
> (+ 4 5)
9
```

In other words: Lisp always uses prefix notation (remember Forth, which does the opposite?). For numerical expression this implies that you have to take care of the preference rules yourself using plenty of parentheses.

```
> (+ (* 2 3) (* 4 5))
26
```

There is also an advantage to this notation. In general, you can apply an arithmetic operator (and many other functions) to more then two arguments without repeating the operator, for example:

```
> (* 2 3 4 5)
120
```

As we see, Lisp treats the first element of a list a function which is then applied to the remaining elements in the list. The syntax is extremely simple: only parentheses and a few quotation symbols (as we will see below) are treated as special tokens. Otherwise, all kinds of characters can be combined to form symbols, and the symbols are separated with white space. We could call a variable or function `*+-x%z5/}$`. This can sometimes be surprising, because what looks like an arithmetic expression, e.g., `3*4+5` is just another symbol. Another remark: symbols are case insensitive, that is, the symbols `xY` and `Xy` are identical and will be shown in the default mode as `XY`.

The Lisp expressions we have seen are called *forms* and the first element is called the *form function*. Lisp uses this kind of syntax even if the first element is not a real function, but instructs Lisp to perform some special action. Such a form is called a *special form* and the first element is either a macro or a special built-in operator. In this short introduction we will not distinguish the different forms and always talk about forms and functions. Syntactically, all forms look the same.

As an example, consider a situation where we don't want the default behavior evaluating a form, but are instead interested in the list as such. In this case, we need to quote it. Quoting can't be a normal function since it changes the behavior of the Lisp system. Hence, there is a special `quote` form which turns the form evaluation off, and, since lists are such prominent data structures in Lisp, you can alternatively just put a single quote in front of the list.

```
> (quote (1 2 3))
(1 2 3)
> (list 1 2)
(1 2)
> '(1 2 3)
(1 2 3)
```

Not surprisingly, Lisp comes with a large library of functions manipulating lists. The most fundamental ones give you the head (first element) or the tail (the rest) of a list, or, as the opposite, create a new list from a given head and tail.

```
> (first '(1 2 3))
1
> (rest '(1 2 3))
(2 3)
> (cons 1 '(2 3))
(1 2 3)
> (list 1 2 3)
(1 2 3)
> (length '(1 2 3))
3
```

In older programs you will still find the original names `car` and `cdr` instead of `first` and `rest`. The old names date back to the first implementation of Lisp. There are many ways to access parts or elements of a list besides head and tail. Here are the most often used functions:

```
> (first '(1 2 3 4))
1
```

```
> (second '(1 2 3 4))
2
> (third '1 2 3 4))
3
> (nth 3 '(1 2 3 4))
4
> (nth 0 '(1 2 3 4))
1
> (elt '(1 2 3 4) 3)
4
> (nthcdr 2 '(1 2 3 4 5))
(3 4 5)
> (last '(1 2 3 4))
(4)
> (last '(1 2 3 4 5) 2)
(4 5)
```

Note the two functions `nth` and `elt` used to access individual elements in a list. The new function `elt` works for all kinds of sequences whereas the old `nth` works for lists only. Annoyingly, the two functions use a different argument order. Besides the core functions `cons` and `list`, there are also many functions creating new lists. Note, however, that these functions are different from the list manipulation functions we have seen in the more procedural oriented languages. The functions here always create new lists and leave the original ones unaltered.

```
> (append '(1 2 3) '(4 5 6) '(7 8 9))
(1 2 3 4 5 6 7 8 9)
> (remove 2 '(1 2 3 4))
(1 3 4)
```

There is a corresponding set of "destructive" list functions, but they are rarely used. First, moving away from the functional programming style makes it harder to follow the programs logic. Second, the non-destructive list functions are implemented very efficiently. The main overhead is the additional memory used.

Before we can define more interesting examples, we have to learn how to use variables. Variables are set using the `setf` function (which is actually a macro). If the variable does not exist yet, it is created automatically.

```
> (setf x "blah")
"blah"
> x
"blah"
> (setf x 1234)
1234
> x
1234
> (setf x '(1 2 3))
(1 2 3)
```

More general, you can use `setf` to assign a new value to any "settable" place (the alternative `setq` is restricted to setting variables). As an example, Lisp allows you to change a value in a list using `setf` combined with the list accessor function.

```
> (setf x '(1 2 3 4))
(1 2 3 4)
> (setf (elt x 2) 55)
55
> x
(1 2 55 4)
```

This is a first example of a destructive list function. Other useful examples are the stack functions `push` and `pop`.

```
> (setf x nil)
NIL
> (push 4 x)
(4)
> (push 5 x)
(5 4)
> (pop x)
5
> x
(4)
```

The variables defined with `setf` are global and therefore should be avoided. Local variables are defined using the let statement.

```
> (let ((x 5) (y 6))
    (+ x y))
11
```

This is a first example of a typical nested Lisp statement. Everything is expressed with nested lists. The `let` function takes two arguments: a list of local variable definitions and an expression. The local variable definitions are name-value pair. In this example we are defining two variables, `x` and `y`, set to 5 and 6, respectively. The scope of these variables is the expression given as the second argument to the `let` function. You may be able to guess now how control statements are written in Lisp.

```
> (if (> 1 2) 5 6)
6
> (setf x 50)
50
> (cond ((< x 10) "small")
        ((< x 100) "medium")
        (t "big"))
"medium"
> (case x
    (10 "ten")
    (20 "twenty")
    (otherwise "some other number"))
```

```
"some other number"
```

The if function, for example, has three arguments: the condition, the expression returned if the condition evaluates to true, and the expression returned if the condition is false. And since we are dealing with common Lisp, there is not just one conditional expression, but a whole set, the most useful ones besides `if` being `cond` and `case`.

```
[34]> (do ((i 0 (+ i 1)))
          ((> i 5))
        (format t "i=~A~%" i))
i=0
i=1
i=2
i=3
i=4
i=5
```

The `do` loop is similar to the `for` loop in C. Like the `if` function it has three arguments. The first argument is a list of loop variable definitions, the second argument contains the stop condition and optionally expression to be evaluated afterwards, and the third argument is the loop's body. A loop variable definition consists of three expressions: the name of the loop variable, its initial value, and the update expression to be evaluated at the end of the loop. In Java, the same loop would look like:

```
for (int i=0; i<=5; ++i) {
  System.out.println("i=" + i);
}
```

3.2.2. Functions

Next to lists, functions are the most prominent elements in Lisp (it's a functional language after all). A function is defined using another function called `defun`.

```
> (defun times2 (x) (* 2 x))
TIMES2
> (times2 55)
110
> (defun fac (n) (if (< n 2) 1 (* n (fac (- n 1)))))
FAC
> (fac 5)
120
```

The definition of the faculty function not only shows that you can define functions in a compact manner (including recursion), but also that the parentheses add up quickly. When writing Lisp programs, an editor showing matching parentheses helps a lot as does proper indentation:

```
> (defun fac (n)
    (if (< n 2)
```

```
        1
        (* n (fac (- n 1)))))
FAC
```

Optional and keyword arguments are possible (possibly with default) and so are variable argument lists. The optional arguments of a function are introduced by the keyword `&optional` followed by pairs containing the name of the argument and the default value.

```
> (defun times2 (x &optional (y 1)) (* x y 2))
TIMES2
> (times2 10)
20
> (times2 10 2)
40>
```

Similarly, keyword arguments follow `&key`. Each keyword argument is either just the name of the argument or a pair of name and default value like in the case of optional argument.

```
> (defun linfunc (x &key a (b 0)) (+ (* a x) b))
LINFUNC
> (linfunc 10 :a 2)
20
> (linfunc 10 :a 2 :b 5)
25
```

When calling a function with a keyword argument, the name of the argument is preceded with a colon. Finally, variable argument lists can be passed as lists to a function using the `&rest` parameter definition.

```
> (defun vararg (x &rest r) (print x) (print r))
VARARG
> (vararg 5)

5
NIL
NIL
> (vararg 1 2 3 4 5)

5
(2 3 4 5)
(2 3 4 5)
```

As seen above for the `car` and `cdr` functions, the result of a function can be used to set the corresponding value. To define such a function ourselves, `defun` is used together with `setf`.

```
> (defun primo (l) (car l))
PRIMO
> (defun (setf primo) (x l) (setf (car l) x))
(SETF PRIMO)
> (setf y '(1 2 3))
(1 2 3)
```

```
> (primo y)
1
> (setf (primo y) 55)
55
> y
(55 2 3)
```

So what is a function? Using `funcall`, we can apply a function explicitly, but when we try to apply our `times2` function, we receive an error message.

```
> (funcall times2 2)
     EVAL: Die Variable TIMES2 hat keinen Wert.
```

This means that the symbol `times2` does not have a value as a variable (but as a function). Functions and variables live in different worlds. More precisely, Common Lisp is a two cell Lisp dialect. What does this mean? Internally, a symbol is represented as a structure. In Common Lisp, this structure has two slots (or cells), one for the value of the symbol treated as a variable, one for the symbol treated as a function. Having seen languages such as Python and Smalltalk this strikes us as odd. Why not treat functions as any other value (or object)? It looks like this is one of the things Common Lisp has carried over from its beginnings. The light-weigth Lisp dialect Scheme uses the one cell approach. To put Common Lisp in perspective, don't forget that some modern languages (e.g., Java) do not treat functions as values at all.

Coming back to our example, we can apply our function using its symbol `'times2`.

```
> (funcall 'times2 4)
8
```

`funcall` knows how to get the function for a given symbol (from the symbol's function cell). This help us with `funcall`, but we still can't assign functions to other symbols or pass them as arguments. That is we can't treat functions as values (or objects) yet. To get the function object (called closure in the functional world), there is a special function called `function`, and, since it is used quite often, the prefix `#'` as a shortcut.

```
> (funcall (function times2) 4)
8
> (funcall #'times2 4)
8
> (print #'times2)
#<CLOSURE TIMES2 (X) ...>
```

Hence, `funcall` takes either a closure or a symbol. When given a symbol, `funcall` retrieves the function from the function cell of the symbol.

We have seen anonymous functions in the form of Python's lambda expressions. In Lisp, a lambda expression looks like a function definition with the `defun` replaced by `lambda`. A lambda expression returns a function object or closure directly.

```
> (lambda (x) (+ x 2))
#<CLOSURE :LAMBDA (X) (+ X 2)>
> (funcall (lambda (x) (+ x 2) 3))
5
> (setf x (lambda (x) (+ x 2)))
#<CLOSURE :LAMBDA (X) (+ X 2)>
> (funcall x 3)
5
```

Now we can go from functions to closure and define closures directly, but we can't turn these closures into new functions. In the example above, `x` is not a function.

```
> (x 3)
*** - EVAL: the function X is undefined
```

The missing link is `symbol-function` which can be used in combination with `setf` to set a function, that is, the function cell of a symbol.

```
> (setf (symbol-function 'add2) (lambda (x) (+ x 2)))
#<CLOSURE :LAMBDA (X) (+ X 2)>
> (add2 3)
5
```

Defining a function with `defun` is equivalent to setting the `symbol-function` of a symbol to a closure.

There are two ways to get the function object from a symbol, either by applying `function` to the function or by applying symbol-function to the symbol

```
> (function +)
#<SYSTEM-FUNCTION +>
> (function add2)
#<CLOSURE ADD2 (X) ... >
> (symbol-function '+)
#<SYSTEM-FUNCTION +>
> (symbol-function 'add2)
#<CLOSURE ADD2 (X) ... >
```

3.2.3. More Collections

Now that we know lists and functions we can tackle more complicated list processing. Let's start with the familiar map and reduce functions.

```
> (mapcar #'(lambda (x) (* 2 x)) '(1 2 3))
(2 4 6)
> (map 'list #'(lambda (x) (* 2 x)) '(1 2 3))
(2 4 6)
> (reduce #'+ '(1 2 3 4))
```

```
10
> (reduce #'+ '(1 2 3) :initial-value 10)
16
```

As you can imagine, the cryptic name `mapcar` denotes the original function. The `map` function belongs to a set of relatively new functions in Common Lisp which generalize the original versions to arbitrary sequences. The first argument is always the type of the result. The filter function is called `remove-if`, but otherwise works as expected.

```
> (remove-if #'oddp '(1 2 3 4 5))
(2 4)
```

The `dolist` function iterates through a list (just like Smalltalk's `do:`).

```
> (dolist (i '(1 2 3 4)) (print i))

1
2
3
4
NIL
```

Besides these straight-forward functions there is an almost overwhelming amount of functions which look for elements, check properties of the list (e.g., do all elements satisfy a certain condition), and so forth. In most cases, you do not need to iterate through the list yourself, but can apply one of these functions.

```
>
```

For now it looks like Lisp really is a pure list processing language. While this was true in the beginning, Common Lisp now supports a whole range of collection classes. Starting with hash tables, the Lisp syntax requires some getting used to, but we can do all the things we know other high level languages.

```
> (setf x (make-hash-table :test 'equal))
#S(HASH-TABLE EQUAL)
> (setf (gethash "blah" x) 55)
55
> (gethash "blah" x)
55 ;
T
> (setf (gethash "blub" x) 66)
66
> (maphash #'(lambda (key value)
  (format t "~&~A=~A" key value)) x)
blub=66
blah=55
NIL
```

The first expression creates a hash table which uses the `equal` operator for the equality test when looking for a key. The `gethash` function looks for a value for a given key. It is a settable function so that we can also use it in combination with `setf` to insert a key-value pair into the table.

Common Lisp also offers multidimensional arrays of fixed size. An array is constructed using the `make-array` function whose first argument is the list of sizes, one for each dimension (or just a single size in case of a one-dimensional array).

```
> (setf x (make-array '(2 3) :initial-element 0.0))
#2A((0.0 0.0 0.0) (0.0 0.0 0.0))
> (setf (aref x 1 2) 55)
55
> x
#2A((0.0 0.0 0.0) (0.0 0.0 55))
```

Elements can be get and set using the `aref` function which is the equivalent of `elt` for arrays. One-dimensional arrays are vectors and can alternatively be contructed with the `vector` function which works like the `list` function, but constructs a vector instead of a list.

```
> (setf x (vector 1 2 3))
#(1 2 3)
> (svref x 1)
2
```

An instead of `aref`, one can use the more efficient "simple vector reference" function `svref`.

3.2.4. Structures

Lisp's equivalent of a C structure is defined with `defstruct`. This special form creates the type and associated constructor, copy, and accessor functions.

```
> (defstruct point x y)
POINT
> (setf p (make-point :x 5 :y 6))
#S(POINT :X 5 :Y 6)
> (format t "p=~A" p)
p=#S(POINT :X 5 :Y 6)
> (setf q (copy-point p))
#S(POINT :X 5 :Y 6)
> (setf (point-y p) 10)
10
> p
#S(POINT :X 10 :Y 6)
> q
#S(POINT :X 5 :Y 6)>
```

Note that the constructor and copy function end with the structure's name, whereas the accessor functions use it as a prefix.

3.2.5. Input and Output

In many programming languages reading complex data structures is a serious task. As an example, most applications use some kind of configuration file which is interpreted when starting the application. One could even argue that technologies such as XML (used for the representation of hierarchical data structures, not for markup of text) with its corresponding parsers and interfaces were invented to remedy the deficiencies of these programming languages. In Lisp, reading an arbitrarily complex Lisp expression is a single call to `read`.

```
> (setf x (read))
(1 2 ("blah" "blub"))
(1 2 ("blah" "blub"))
> x
(1 2 ("blah" "blub"))
```

Read reads exactly one Lisp expression from the input stream. Without arguments, `read` reads from standard input. The second line shows my input, the nested list `(1 2 ("blah" "blub"))`. The second line is the return value of the `setf` function. Once we have read the Lisp expression, we can process it like any other Lisp expression. As an example, we can read a lambda expression and execute it.

```
> (setf x (read))
(lambda (x) (* 2 x)
(LAMBDA (X) (* 2 X)
> (funcall (eval x) 5)
10
```

A configuration file could just contain a Lisp expression.

Lisp has mainly two modes when printing objects: the output is either meant for a human being or for a program. As an example, a string is surrounded by double quotes when printed for a computer. This way, a reading program can reconstruct the type. The human reader just gets the string itself. I havn't found an explanation for the extremely mnemonic function names: `princ` is the people oriented print function, `prin1` the one for programs, and `print` is like `prin1`, but starts with an additional newline.

```
> (princ "bla")
bla
"bla"
> (prin1 "bla")
"bla"
"bla"
> (print "bla")

"bla"
"bla"
```

Playing with standard input and output is one thing, but in the real world, we would like to read from and write to files. Like many I/O systems, Lisp I/O is stream oriented. The standard input and output we have used so far are examples of streams. The proper starting point to accessing files in Lisp is to create a `pathname`.

```
> (setf path (make-pathname :name "test.dat"))
#P"test.dat"
```

Common Lisp's pathname library offers an abstraction of file names on the different operating systems. Once we have a path, we can open the file, write to it, and close it again.

```
> (setf out (open path :direction :output))
#<OUTPUT BUFFERED FILE-STREAM CHARACTER #P"test.dat" @1>
> (princ "Hello World" out)
"Hello World"
> (close out)
T
```

The problem with this code is that we have to make sure that the `close` function under any circumstances unless we want to risk a file descriptor leak. A better way to achieve the same thing is to use `with-open-file`.

```
> (with-open-file (out path :direction :output)
  (format out "~A~%" "Hello World")
  (format out "~A~%" "Hello Again"))
"Hello World"
```

The `with-open-file` function ensures that the specified stream is available for the block of expressions and that it is properly closed afterwards even in case of an error. You can view it as a predecessor of the `using` statement in C#. The same can be used to read the file.

```
> (with-open-file (in path :direction :input)
  (format t "line: ~A" (read-line in)))
line: Hello World
NIL
```

Here are some more interesting functions reading from a file.

```
> (defun process-lines (in task)
  (do ((line (read-line in nil 'eof) (read-line in nil 'eof)))
      ((eql line 'eof))
      (funcall task line)))
PROCESS-LINES
> (with-open-file (in path :direction :input)
  (process-lines in #'print)))

"Hello World"
"Hello Again"
NIL
```

The function `process-lines` reads a stream line by line applying a function to each line. We then apply this new function to the implicitly opened input stream `in` and simple print each line.

3.3. More Features

3.3.1. Object Oriented Programming

One of the reasons Lisp is still alive after so many years is its extensibility. As an example, Lisp can be easily extended with object oriented features, and Common Lisp defines a sophisticated package called the Common Lisp Object System or CLOS (pronounced "see-loss") for short, which offers everything you need to develop object oriented programs including some extensions not available in other OO languages.

Let’s develop an account class similar to the one we have used as a Smalltalk example.

```
> (use-package "CLOS")
T
> (defclass account ()
    ((balance :accessor balance :initform 0.0 :initarg :balance)))
#<STANDARD-CLASS ACCOUNT>
```

Once we have imported the CLOS package (this step is required only for Clisp), the `defclass` call defines a class called `account`. The name of the class is followed by the list of superclasses which in our case is empty. Next is a list of attribute definitions. For now, our account has a single attribute `balance`. Attribute definitions are lists starting with the name of the attribute followed by a number of keyword arguments. We can define a default value for the attribute (`:initform`), whether it should be part of a constructor call (`:initarg`), and the accessor methods (reader/getter, writer/setter, or both) we would like to have. In our example, the attribute `balance` has the default value zero, can be set during the constructor call using the `:balance` keyword, and we would like a reader and a writer method, both called `balance`. Lisp’s generic constructor function (comparable to `new` in other OO languages) is called `make-instance`.

```
> (setf a (make-instance ’account :balance 10.0))
#<ACCOUNT #x1A4EE1F1>
> (balance a)
10.0
> (setf (balance a) 55)
55
> (balance a)
55
```

Note the call of the writer method using `setf`. The next thing we would like to do with the account is spending money.

```
> (defmethod spend ((a account) amount)
  (setf (balance a) (- (balance a) amount)))
> (spend a 5)
50
> (balance a)
50
```

This definition looks more like a normal function definition rather than a method definition as we know it from the object oriented languages we have seen so far. The instance `a` does not play a prominent role. The argument list consists of simple arguments like `amount` and object arguments which are given as name and class. We could have as easily defined the method to take the amount as the first argument and the account object as the second.

```
> (defmethod spend2 (amount (a account))
  (setf (balance a) (- (balance a) amount)))
> (spend2 5 a)
45
```

We can also pass multiple objects.

```
> (defmethod sum ((a account) (b account))
  (+ (balance a) (balance b)))
> (sum a a)
90
```

To see the polymorphic behaviour of generic functions, we need a sub class of account.

```
(defclass checking-account (account)
  ((history :accessor history
    :initform nil)))
```

The `checking-account` class extends `account` with a `history` attribute which is initially set to `nil`. We will use this history attribute to keep track of the transactions on our account. To this end we need to change the implementation of the `spend` method.

```
(defmethod spend ((a checking-account) amount)
  (progn
    (setf (balance a) (- (balance a) amount))
    (setf (history a) (cons '(spend amount) (history a)))))
```

In addition to updating the balance, we add the type and amount of the transaction to the history. Let's see how the new class behaves.

```
> (setf b (make-instance 'checking-account :balance 100.0))
#<CHECKING-ACCOUNT #x2041E6ED>
> (balance b)
100.0
> (spend b 25.0)
((SPEND . 25.0))
```

```
> (spend b 10.0)
((SPEND . 10.0) (SPEND . 25.0))
> (balance b)
65.0
> (history b)
((SPEND . 10.0) (SPEND . 25.0))
```

As expected, the new implementation of the `spend` method applied to the checking account. Compared to other object-oriented languages, CLOS does not restrict this kind of polymorphism (dynamic dispatch) to a single argument. Generic functions can be specialized with respect to any of their arguments.

What we don't like about the implementation of the `spend` method for the checking account is the repetition of the code of the account's original implementation. Of course, Common Lisp has a way to achieve the same behavior without this code duplication. We can add behavior before or after an existing method by just placing the `:before` or `:after` keywords behind the method name. An equivalent better implementation is therefore

```
(defmethod spend :after ((a checking-account) amount)
  (setf (history a) (cons '(spend amount) (history a))))
```

3.3.2. Macros

The flexibility of Lisp comes from the possibility to extend Lisp using Lisp itself. Since programs and data use the same structure, we can use functions to create new programs. The key for doing this efficiently (at compile time) are macros. They are similar to C macros in that they generate code, but in the case of Common Lisp, macros have the full power of Lisp to construct this code. Here is a macro adding a `while` statements to Lisp.

```
> (defmacro while (test &rest body)
  `(do ()
       ((not ,test))
      ,@body))
WHILE
> (setf i 0)
0
> (while (< i 5) (print i) (incf i))

0
1
2
3
4
NIL
```

This example shows a number of typical macro features. First, the definition looks like the definition of a function apart from the use of `defmacro` instead of `defun`. The macro has a normal parameter list including a variable argument list `body` indicated by the `&rest` parameter. The body of the macro definition starts with a symbol we have not used before: the back quote. By iself, the back quote works just like the normal "forward" quote protecting its argument from evaluation.

```
> ’(a (+ 4 5) b)
(A (+ 4 5) B)
> ‘(a (+ 4 5) b)
(A (+ 4 5) B)
```

But in contrast to the normal quote, one can turn the evaluation on again for certain sub expressions by preceding those expression with a comma.

```
> ‘(a (+ 4 5) b)
(A 9 B)
```

Finally, the odd operator comma-at `,@` expects a list argument and copies the elements of this list into the output.

```
> (setf x ’(1 2 3))
(1 2 3)
> ‘(a x b)
(A X B)
> ‘(a ,x b)
(A (1 2 3) B)
> ‘(a ,@x b)
(A 1 2 3 B)
```

Combining all these features we can understand the `while` macro. A helpful tool for macro development is the `macroexpand-1` function which shows the generated Lisp code.

```
> (macroexpand-1 ’(while condition task))
(DO NIL ((NOT CONDIION)) TASK)
> (macroexpand-1 ’(while (< i 5) (print i) (incf i)))
(DO NIL ((NOT (< I 5))) (PRINT I) (INCF I))
```

The macro construct a `do` loop using the back quote syntax and inserting the test condition with comma and the body of the while loop with the comma-at operator.

3.3.3. Exceptions

Exceptions as a means to organize error handling belong to the standard toolset of a modern programming language. Lisp talks about conditions rather than exceptions, but the idea is the same. You can raise a condition and thus interrupt the normal program flow, the condition can be caught using a handler expression (`handler-case`), and you can perform actions whether an exception is raised or not using ...

3.3.4. Analysis Functions

Common Lisp comes with a standard set of utilities which help to understand Lisp programs. The `trace` function allows to switch on tracing for a function. This feature is especially useful for recursive functions.

```
> (defun fac (n) (if (< n 2) 1 (* n (fac (- n 1)))))
FAC
> (trace fac)
;; Tracing function FAC.
(FAC)
> (fac 5)

1. Trace: (FAC ’5)
2. Trace: (FAC ’4)
3. Trace: (FAC ’3)
4. Trace: (FAC ’2)
5. Trace: (FAC ’1)
5. Trace: FAC ==> 1
4. Trace: FAC ==> 2
3. Trace: FAC ==> 6
2. Trace: FAC ==> 24
1. Trace: FAC ==> 120
120
```

With the function `untrace` tracing is switched off again. The `time` function can be used for simple performance measurements (similar to the UNIX time command). This is particularly interesting when comparing the interpreted version of a function to its compiled form which is about eight times faster.

```
> (defun f (n) (dotimes (i n) nil))
F
> (time (f 1000000))

Real time: 2.624 sec.
Run time: 2.5837152 sec.
Space: 0 Bytes
NIL
> (compile ’f)
F ;
NIL ;
NIL
> (time (f 1000000))

Real time: 0.34 sec.
Run time: 0.3304752 sec.
Space: 0 Bytes
NIL
```

References

Paul Graham's book presents Lisp on 400 densely written pages. It can be used as an introduction as well as a reference. Sonja Keene's book [KEENE88]> is a very readable introduction to CLOS including the advanced features such as the meta object protocol. If you want to learn about what is considered the major domain for Lisp, Peter Norvig's book about artificial intelligence is for you. It contains plenty of AI programs implemented in Common Lisp.

Paul Graham, 0-13-370875-6, Prentice-Hall, 1996, *ANSI Common Lisp*.

Sonja E. Keene, 0-201-17589-4, Addison-Wesley, 1988, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*.

Peter Norvig, 1-55860-191-0, Morgan Kaufmann, 1992, *Artificial Intelligence Programming: Case Studies in Common Lisp*.

Chapter 4. Cobol

Together with Fortran and Lisp, Cobol (COmmon Business Oriented Language) is one of the oldest programming languages. It was defined in 1960 at the DoD sponsored Conference on Data System Languages (CODASYL). The first ANSI standard came out in 1968 with further updates in 1974 (Cobol74) and 1985 (Cobol85). A new version (with significant new features including Object-Cobol) is on its way.

Without doubt, Cobol remains the most important language for business applications. There are a few hundred billion lines of Cobol code in production and about five billion lines are still added every year. Many reasons to have a closer look.

4.1. Software and Installation

Since Cobol is all about "real business", free Cobol compilers are rare. Among those, Tiny Cobol (http://tiny-cobol.sourceforge.net/) seems to be the most mature. For this chapter, we use version 0.60 on a Linux system. To installation from the source uses the standard `configure`, `make`, `make install` sequence.

Tiny Cobol is a preprocessor which translates Cobol code to C. The installation comes with a large number of examples (in the `test.code` directory) which can be used as templates for the code samples below. Compiling and running the "Hello world" program program then looks like this.

```
ahohmann@kermit:~/programming/cobol/book/hello$ make
htcobol -c  -P -D -I/home/ahohmann/opt/share/htcobol/copybooks -I. example.cob
gcc -g -o example example.o  -L/usr/local/lib -L/usr/lib -L/home/ahohmann/opt/lib -lhtcobol
ahohmann@kermit:~/programming/cobol/book/hello$ example
Hello World!
```

4.2. Quick Tour

4.2.1. Hello World

Here is the Cobol version of "Hello World".

```
IDENTIFICATION DIVISION.
PROGRAM-ID. hello.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
    DISPLAY "Hello World!"
```

```
STOP RUN.
```

It can not beat the "scripting" languages in terms of conciseness, but at least the program looks well organized and readable. These are indeed two characteristics of Cobol programs. A Cobol program is organized in a hierarchical structure consisting of divisions, sections, paragraphs, and sentences. At the top of the hierarchy are the four divisions which we can already see in our first program. The identification section gives some general information about the program (in our case just its name). The next two division, environment and data, are dealing with files. Since our "Hello World" program just writes to the screen, these divisions are empty. The actual program is contained in the procedure division. In our example, it contains the two statements to print the message and stop the program. In the following examples we often omit the divisions which are not required or clear from the preceding examples.

The format of a Cobol program is line-oriented (showing Cobol's punchcard origins). A line is divided into five areas as described in Table 4-1>.

Table 4-1. Areas of a Cobol Line

Columns	Area	Contents
1-6	line	page and line number
7	indicator	*: comment, -: continuation, /: page skip
8-11	A	division, section, paragraph identifiers and some level numbers
12-72	B	program items not belonging to A
73-80	program identification	optional name of the program

The line and program identification area were useful in case two piles of punchcards got mixed up. When using the areas A and B it is decisive where the code starts. As we have seen in the "Hello World" program, the organizational identifiers start in area A, and the code statements in area B.

4.2.2. Variables and Arithmetic

Cobol is tailored for a particular kind of application: processing files consisting of character data organized in fixed length records. This approach leads to a different way of defining data structures when compared to "modern" languages. To describe a field in a record of characters we need to know how many characters are used and how these characters are interpreted (e.g., as integers, decimal numbers, or strings). In Cobol terms, we have to define the *picture* of a field.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    addition.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  A  PICTURE 99V999.
```

```
PROCEDURE DIVISION.
    MOVE 1.5 TO A
    ADD  1.5 TO A
    DISPLAY "A=", A
    STOP RUN.
```

Running the program results in the output `A=03.000`. The most interesting line is the definition of the variable `A` in the working storage section. It starts with the elementary level `77` (more on levels below) followed by the name of the variable, the `PICTURE` keyword (most often abbreviated `PIC`), and the picture format `99V999`. The format strings use a syntax which makes it easy to literally picture the field. Each `9` stands for a decimal digit and the `V` indicated the position of the decimal point. Hence, our variable `A` represents a decimal number which occupies five characters, each of which must be a decimal digit, and the last three digits are interpreted as the decimals. The string `12345` is interpreted as the decimal number 12.345 (which explains the way the result of our program is displayed). Next to the `9` representing a decimal digit, the letter `X` representing an arbitrary character is the most often used format symbol. We will see plenty of examples in the following sections.

Apart from the picture clause, the program demonstrates the rather wordy (but readable) way to perform calculations with Cobol. Assignment uses the `MOVE TO` statement, and similarly the `ADD TO` statement is used to add a value to a variable. There are equally expressive ways to subtract, multiply, and divide.

```
WORKING-STORAGE SECTION.
77  A  PICTURE 99V999.
77  B  PICTURE 99V999.
PROCEDURE DIVISION.
    MOVE 3.5 TO A
    MULTIPLY 2 BY A
    DISPLAY "A=", A
    MULTIPLY A BY 3 GIVING B
    DISPLAY "B=", B
    STOP RUN.
```

```
result:
A=07.000
B=21.000
```

Without the `GIVING` clause, the result of the multiplication is stored in the second operand (which therefore must be a variable). The division statement `DIVIDE` has even more variations using either `BY` (mathematical order of arguments, dividend first) or `INTO` (opposide order).

```
WORKING-STORAGE SECTION.
77  A  PICTURE 99V999.
77  B  PICTURE 99V999.
PROCEDURE DIVISION.
    MOVE 50 TO A
    DIVIDE 10 INTO A
    DISPLAY "A=", A
    DIVIDE A BY 2 GIVING B
    DISPLAY "B=", B
```

```
            STOP RUN.

result:
A=05.000
B=02.500
```

The `DIVIDE INTO` version stores the result by default (without a `GIVING` clause) in the second operand.

`ADD` and `SUBTRACT` can also take multiple arguments. Also note the special constant `ZEROS` used to reset the variable `A`.

```
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        77  A  PICTURE 99V999.
        PROCEDURE DIVISION.
            MOVE ZEROS TO A
            ADD  1.5 2.5 10 TO A
            DISPLAY "A=", A
            SUBTRACT 3 1 FROM A
            DISPLAY "A=", A
            STOP RUN.

result:
A=14.000
A=10.000
```

For more involved computations, Cobol offers the `COMPUTE` statement which allows us to use arithmetic formulas as in the following example. And, yes, Cobol uses the correct precedence rules.

```
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        77  A  PICTURE 99V999 VALUE 10.0 .
        PROCEDURE DIVISION.
            COMPUTE A = 4 + 1.5 * 3
            DISPLAY "A=", A
            STOP RUN.

result:
A=18.500
```

The example also demonstrates an initializer for the variable `A` using a `VALUE` clause.

All these computations were within the limits of our variable `A`. But what happens if the result does not fit into the assigned field?

```
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        77  A  PICTURE 99V999 VALUE 10.
        PROCEDURE DIVISION.
```

```
            MULTIPLY 10 BY A
            DISPLAY "A=", A
            STOP RUN.
```

```
result:
00.000
```

The result vanishes! We will see later on, how to detect overflows in the program and act accordingly. For now, we must assume that the fields were defined big enough.

4.2.3. Subroutines and Control Statements

After some basic expressions, we have usually tackled functions as the main means to structure a program. Cobol does not support functions with arguments and return values, but uses subroutines to organize a program into smaller units. Subroutines are simply paragraphs of the procedure division. Each paragraph is introduced with a paragraph name starting in area A and consists of a sequences of statements (all beginning in area B). Without arguments and return values, the communication between these units relies on the shared access to the objects defined in the data devision. The statements in a paragraph can be called (making it a subroutine) using the `PERFORM` statement.

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID. hello.
        ENVIRONMENT DIVISION.
        DATA DIVISION.
        PROCEDURE DIVISION.
        MAIN.
            PERFORM DISPLAY-HELLO
            PERFORM DISPLAY-HELLO
            PERFORM DISPLAY-BYE
            PERFORM DISPLAY-BYE.
            STOP RUN.
        DISPLAY-HELLO.
            DISPLAY "Hello World!".
        DISPLAY-BYE.
            DISPLAY "Bye!".
```

```
result:
Hello World!
Hello World!
Bye!
Bye!
```

The first paragraph is executed when starting the program (it does not have to be called `MAIN`. The `PERFORM` statement executes the named paragraph. At the end of the paragraph, control is returned to the calling paragraph. Note that the `STOP` command is required at the end of the main routine. Otherwise, the program will continue and execute the two subroutines (resulting in another "Hello World!" and "Bye!" message).

If we want to repeat an action multiple times, we can use add the `TIMES` clause to the `PERFORM` statement.

```
       PROCEDURE DIVISION.
       MAIN.
           PERFORM DISPLAY-HELLO 3 TIMES
           STOP RUN.
       DISPLAY-HELLO.
           DISPLAY "Hello World!".
```

```
result:
Hello World!
Hello World!
Hello World!
```

To repeat an action while a certain condition holds, we combine the `PERFORM` statement with the `UNTIL` clause followed by the condition.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. count.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  I  PICTURE 99 VALUE 0.
       PROCEDURE DIVISION.
       MAIN.
           PERFORM LOOP UNTIL I = 5
           STOP RUN.
       LOOP.
           DISPLAY "I=", I
           ADD 1 TO I.
```

```
result:
I=00
I=01
I=02
I=03
I=04
```

Cobol85 also has the equivalent of a `repeat-until` or `do-while` loop found in other languages by just adding `WITH TEST AFTER` to the `PERFORM` clause.

```
       WORKING-STORAGE SECTION.
       77  I  PICTURE 99 VALUE 0.
       PROCEDURE DIVISION.
       MAIN.
           PERFORM LOOP UNTIL I = 5
           STOP RUN.
       LOOP.
           DISPLAY "I=", I
           ADD 1 TO I.
```

For this example it does not make any difference whether we check the condition before or after the loop, but in some cases we want the loop to be executed at least once or the condition only makes sense at the end of the loop.

We can achieve the same thing without a subroutine using the second form of `PERFORM UNTIL` which takes a sequence of statements (the body of the loop) instead of the subroutine.

```
       WORKING-STORAGE SECTION.
       77  I  PICTURE 99 VALUE 0.
       PROCEDURE DIVISION.
       MAIN.
           PERFORM WITH TEST AFTER UNTIL I = 5
               DISPLAY "I=", I
               ADD 1 TO I
           END-PERFORM
           STOP RUN.
```

```
result:
I=00
I=01
I=02
I=03
I=04
```

In fact, Cobol also offers an integer loop using the `VARYING` clause of the `PERFORM` statement.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. hello.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  I  PICTURE 99 VALUE 0.
       PROCEDURE DIVISION.
       MAIN.
           PERFORM VARYING I FROM 1 BY 2 UNTIL I > 10
               DISPLAY "I=", I
           END-PERFORM
           STOP RUN.
```

```
result:
I=01
I=03
I=05
I=07
I=09
```

It is also possible to nest multiple integer loops, for example, when indexing a multidimensional array.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. hello.
       ENVIRONMENT DIVISION.
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  I  PICTURE 99.
77  J  PICTURE 99.
PROCEDURE DIVISION.
MAIN.
    PERFORM
        VARYING I FROM 1 BY 1 UNTIL I > 3
        AFTER   J FROM I BY 1 UNTIL J > 3
        DISPLAY "I=", I, " J=", J
    END-PERFORM
    STOP RUN.
```

```
result:
I=01 J=01
I=01 J=02
I=01 J=03
I=02 J=02
I=02 J=03
I=03 J=03
```

Continuing with control statements, Cobol of course supports the basic if-then-else.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. hello.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
MAIN.
    IF 10 IS LESS THAN 100
    THEN
        DISPLAY "Yes, that’s right"
    ELSE
        DISPLAY "No, that’s wrong"
    END-IF
    STOP RUN.
```

Note that lengthy (but readable) `IS LESS THAN` can be replaced by a < sign. Multiple if statement can be combined in a case statement which is called `EVALUATE` in Cobol.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. hello.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  INPUT-VALUE  PICTURE 99 VALUE 0.
PROCEDURE DIVISION.
MAIN.
    DISPLAY "value: "
    ACCEPT INPUT-VALUE
    EVALUATE INPUT-VALUE
        WHEN 1
```

```
            DISPLAY "ONE"
        WHEN 2
            DISPLAY "TWO"
        WHEN 3
            DISPLAY "THREE"
        WHEN OTHER
            DISPLAY "MORE"
    END-EVALUATE
    STOP RUN.
```

We use the opportunity to write our first interactive program which asks with the `ACCEPT` command for the value to be used in the case statement.

4.2.4. Data Structures

In the first section we have defined a single variable in the working storage section of the data devision. We have used the level number 77 to indicate that the variable is elementary field. The level number will become much clearer when looking at the following definition of a nested structure modelling a person.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    addition.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PERSON.
    05  FIRST-NAME          PIC X(20).
    05  LAST-NAME           PIC X(20).
    05  ADDR.
        10  POSTAL-CODE     PIC X(5).
        10  CITY            PIC X(20).
        10  STREET          PIC X(20).
        10  STREET-NO       PIC X(5).
PROCEDURE DIVISION.
    MOVE SPACES TO PERSON
    MOVE '40547DUESSELDORF' TO ADDR
    DISPLAY 'CITY=', CITY
    STOP RUN.
```

A person consists of a first name, a last name, and an address. An address is comprised of postal-code, city, street, and street number. The nested structure is defined in Cobol using level numbers. Fields with the same level number belong to the same level of the nested structure. In our example we start with level 01 which has to be in area A. Since this field does not have a picture definition is must be a structure. On the next level (we have chosen the level number 05) are first name, last name, and address. The first two field are elementary fields and therefore must have a picture clause defining their size and format. The address field is again a structure which needs to be further decomposed into elementary fields. Note the use of the size subscripts simplifying the field formats.

We can access the structure on all levels. The first statement of the procedure division resets the whole person structure. Next, we set the address. And finally, we display the city as an individual field. If the field names are unique (as in our example), the fields do not have to be qualified with their surrounding structure (in C we would need to write `person.addr.city` to access the city). If the same field name occurs in multiple places, it has to be qualified using the `OF` (or synonymously `IN`) clause.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   persons.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PERSON-1.
    05  FIRST-NAME          PIC X(20).
    05  LAST-NAME           PIC X(20).
01  PERSON-2.
    05  FIRST-NAME          PIC X(20).
    05  LAST-NAME           PIC X(20).
PROCEDURE DIVISION.
    MOVE 'Homer' TO FIRST-NAME OF PERSON-2
    STOP RUN.
```

Besides composing fields to structures, Cobol supports arrays. In the simplest case, we can define an array of fixed size of some elementary field by adding an `OCCURS` clause to the field definition.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    addition.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  MY-ARRAY.
    05 A OCCURS 10 TIMES    PIC 99.
77  I                       PIC 99 VALUE 0.
PROCEDURE DIVISION.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 10
        COMPUTE A (I) = 5 + 2 * I
    END-PERFORM
    DISPLAY "A(5)=", A (5)
    STOP RUN.
```

```
result:
A(5)=15
```

The individual elements of the array are accessed using the index in parentheses. Like Fortran and Smalltalk (and unlike Lisp and the C family), indexing starts at one. There should be white space before the open parenthesis (although Tiny Cobol does not complain if we omit the space), and we must not put any white space inside of the parentheses.

Similarly, we can define fixed length arrays of structures by adding the `OCCURS` to the structure level.

```
IDENTIFICATION DIVISION.
```

```
PROGRAM-ID.    addition.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PERSON OCCURS 10 TIMES.
    05  FIRST-NAME          PIC X(20).
    05  LAST-NAME           PIC X(20).
PROCEDURE DIVISION.
    MOVE 'Homer' TO FIRST-NAME OF PERSON(3)
    DISPLAY 'Third person:', FIRST-NAME OF PERSON(3)
    STOP RUN.
```

4.2.5. Files and Records

Since file (batch) processing is such a dominant area for Cobol programs, most Cobol books start with what we will cover next: the definition of files and records. Here is a small program which reads a file containing the items of an invoice and computes the total of the invoice.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. invoice.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL. SELECT INVOICE ASSIGN TO 'invoice.dat'
              ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD  INVOICE     LABEL RECORDS ARE STANDARD.
01  ITEM.
    05  NAME                PIC X(20).
    05  AMOUNT              PIC 9(3)V.
    05  PRICE               PIC 9999V99.
WORKING-STORAGE SECTION.
77  MORE-RECORDS            PIC XXX     VALUE 'YES'.
77  TOTAL                   PIC 9(5)V99  VALUE ZEROS.
PROCEDURE DIVISION.
MAIN.
    OPEN INPUT INVOICE
    PERFORM UNTIL MORE-RECORDS = 'NO '
        READ INVOICE
             AT END
                MOVE 'NO ' TO MORE-RECORDS
             NOT AT END
                COMPUTE TOTAL = TOTAL + AMOUNT * PRICE
    END-PERFORM
    DISPLAY 'TOTAL=', TOTAL
    CLOSE INVOICE
    STOP RUN.
```

The file contains records of fixed length. Each record has three fields: the name of the item, the amount of items bought, and the price per item. We recognize the structure definition in the file section which looks just like the structure definitions in the working storage section we have used before. The new part is the mapping to a file. Cobol separates the logical file from the physical implementation. The logical view is defined in the file section of the data division. Each logical file is defined by a file descriptor (`FD`) paragraph in the file section. The environment division contains the mappings of the logical files defined in the data division to the physical files controlled by the operating system. For each file, there is a `FILE-CONTROL` paragraph with a `SELECT` statement which assigns the name of the logical file to a physical file name. The environment division is the only part of the Cobol program which depends on the operating system and has to be adapted when migrating to a new environment.

Since we are running on a PC, we have added the `ORGANIZATION IS LINE SEQUENTIAL` clause which causes Cobol to interpret each line in the file as a record. Without this instruction, there is no separation (newline) between the records.

Once we have defined the record structure of our file, a simple `READ` statement reads a new record into the structure so that we can access the individual fields. The `READ` statement takes two blocks: one for the normal case when a new record has been read and one for reaching the end of the file. In our case we either update the total or set the `MORE-RECORDS` flag to `NO`.

4.3. More Features

4.3.1. Table Search

Since batch programs often deal with reference data kept in tables, e.g., a list of prices, Cobol makes it easy to look up data in tables. One solution is the use of indexed arrays in combination with the `SEARCH` statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    price-table.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ITEM-TABLE.
    05 ITEMS OCCURS 3 TIMES INDEXED BY ITEM-INDEX.
        10  ITEM-NAME       PIC X(20).
        10  ITEM-PRICE      PIC 999V99.
77  ITEM-INPUT  PIC X(20).
PROCEDURE DIVISION.
MAIN.
    PERFORM INIT-PRICE-TABLE.
    DISPLAY 'ENTER NAME OF ITEM:'
```

```
    ACCEPT ITEM-INPUT
    SET ITEM-INDEX TO 1
    SEARCH ITEMS
        AT END DISPLAY 'UNKNOWN ITEM'
        WHEN ITEM-INPUT = ITEM-NAME (ITEM-INDEX)
            DISPLAY 'PRICE=', ITEM-PRICE (ITEM-INDEX)
    STOP RUN.
INIT-PRICE-TABLE.
    MOVE 'APPLE'    TO ITEM-NAME  (1)
    MOVE 0.50       TO ITEM-PRICE (1)
    MOVE 'ORANGE'   TO ITEM-NAME  (2)
    MOVE 1.50       TO ITEM-PRICE (2)
    MOVE 'PEAR'     TO ITEM-NAME  (3)
    MOVE 0.75       TO ITEM-PRICE (4).
```

The working storage section defines a table which contains the prices for some items. To keep the example simple, we fill our price table with hard-coded values in the program itself. Normally, the table would be read from reference data stored in a file or database.

The first new construct is the index attached to the array of `ITEMS`. We can view it as a loop variable for the array. We can set the index using the `SET` statement, but we do not have to define the index variable explicitly in the working storage section as we have done earlier. The compiler takes care of the correct size of the variable.

The index is used implicitly in the `SEARCH` statement. The `SEARCH` statement is comparable to the `for-in` loops in languages like Python or Perl, only that the loop variable is not part of the loop statement, but defined as part of the array. The interesting part is the body of the `SEARCH` statement. It consists of the (optional) `AT END` clause telling what to do if the end of the array is reached and any number of `WHEN` clauses defining the search conditions and actions. For each item in the table, the search loop checks the conditions of the `WHEN` clauses. If the condition is true, the associated action is performed and the search loop left.

While we could easily achieve the same with a `PERFORM` statement, the `SEARCH` command turns the focus away from the (imperative) loop to what we are looking for. Because of this, it is easy to exchange the algorithm used to find the entry. Adding the keyword `ALL` to the `SEARCH` statement, we switch from linear to binary search.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    price-table.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ITEM-TABLE.
    05 ITEMS OCCURS 3 TIMES
            ASCENDING KEY IS ITEM-NAME INDEXED BY ITEM-INDEX.
        10  ITEM-NAME       PIC X(20).
        10  ITEM-PRICE      PIC 999V99.
77  ITEM-INPUT  PIC X(20).
```

```
PROCEDURE DIVISION.
MAIN.
    PERFORM INIT-PRICE-TABLE.
    DISPLAY 'ENTER NAME OF ITEM:'
    ACCEPT ITEM-INPUT
    SEARCH ALL ITEMS
        AT END DISPLAY 'UNKNOWN ITEM'
        WHEN ITEM-INPUT = ITEM-NAME (ITEM-INDEX)
            DISPLAY 'PRICE=', ITEM-PRICE (ITEM-INDEX)
    STOP RUN.
INIT-PRICE-TABLE.
    MOVE 'APPLE'    TO ITEM-NAME  (1)
    MOVE 0.50       TO ITEM-PRICE (1)
    MOVE 'ORANGE'   TO ITEM-NAME  (2)
    MOVE 1.50       TO ITEM-PRICE (2)
    MOVE 'PEAR'     TO ITEM-NAME  (3)
    MOVE 0.75       TO ITEM-PRICE (4).
```

Of course, the binary search requires the table to be sorted (which we ensured in the initialization routine), and we have to specify the sort order of the table in the working storage section.

4.3.2. Macros (Copy Books)

Cobol's macro mechanism uses copy books and the `COPY` statement. Suppose we need the same structure or subroutine over and over again in multiple Cobol programs. Instead of copying the source code and performing some modifications manually, we can store the reused code in a separate file (a copy book) and call the `COPY` statement to let the Cobol compiler copy the code for us including some adaptations we might need. Here is the "Hello World!" of copy books:

```
01 HELLO-CONSTANTS.
    05 HELLO-TEXT PIC X(20) VALUE 'Hello World!'.
```

The text is stored in a file called `hello.cpy` (this is for Tine Cobol; different environments might have different ways to handle copy books). And here is the main program using it.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    copy-sample.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY hello REPLACING
    HELLO-TEXT BY HELLO-MESSAGE
    ==World== BY ==You==.
77  A  PICTURE 99V999.
PROCEDURE DIVISION.
    DISPLAY HELLO-MESSAGE.
```

```
        STOP RUN.
```

We reference the copy by its base name (without the `.cpy` suffix) and perform two replacements. Identifiers such as `HELLO-TEXT` can be replaced directly. The textual replacement inside of the message string uses `==` to delimit the original string and its replacement.

Copy books can not only be used for the data division, but for the environment and procedure division as well. As you can imagine, copy books are a very powerful mechanism to avoid code duplication. They are used heavily in large Cobol applications.

4.3.3. Subprograms

We have said in the beginning that Cobol does not have the notion of functions with arguments and return values. However, a similar effect can be achived by calling programs.

To turn a program into a subprogram which can be called by other programs, we have to declare the arguments of the subprogram in the `LINKAGE` section of the data division (right before the procedure devision so that it looks like a signature for the program).

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUBPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01  X   PIC X(20).
PROCEDURE DIVISION USING X.
    DISPLAY 'X=', X
    MOVE 'We were here!' TO X
    EXIT PROGRAM.
```

The header of the procedure division contains the "parameter list" in the `USING` clause. Parameters to subprograms are always in-out parameters (they are passed by reference). The subprogram can read them and set them to new values. The calling program now calls the subprogram by name specifying which fields to pass as the arguments to the subprogram.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    call-sample.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A   PIC X(20) VALUE 'Hello World!'.
PROCEDURE DIVISION.
    CALL 'SUBPROG' USING A
    DISPLAY 'A=', A
    STOP RUN.
```

`result:`

```
X=Hello World!
A=We were here
```

First, the subprogram prints the original message as passed from the main program. After returning from the subprogram, the main program prints the message which has been changed by the subprogram.

4.3.4. Sort and Merge

A common task for a batch application is to sort a file consisting of fixed length records with respect to some key fields. The following example sorts a file with respect to the first two characters of each record (of 40 characters).

```
IDENTIFICATION DIVISION.
PROGRAM-ID. sort-sample.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT UNSORTED-FILE    ASSIGN TO 'unsorted.dat'
           ORGANIZATION IS LINE SEQUENTIAL.
    SELECT SORT-FILE        ASSIGN TO 'sort.dat'
           ORGANIZATION IS LINE SEQUENTIAL.
    SELECT SORTED-FILE      ASSIGN TO 'sorted.dat'
           ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD  UNSORTED-FILE LABEL RECORDS ARE STANDARD.
01  UNSORTED-REC           PIC X(40).
SD  SORT-FILE.
01  SORT-REC.
    05  SORT-NO            PIC XX.
    05  FILLER             PIC X(38).
FD  SORTED-FILE  LABEL RECORDS ARE STANDARD.
01  SORTED-REC             PIC X(40).
PROCEDURE DIVISION.
MAIN.
    SORT  SORT-FILE
        ASCENDING KEY SORT-NO
        USING UNSORTED-FILE
        GIVING SORTED-FILE
    STOP RUN.
```

Three files have to be provided. The original unsorted file, the file for the sorted result, and the sort file used for intermediate storage. The sort key (here: `SORT-NO`) has to be defined for the sort file only. For the unsorted input file and the sorted result, we only specify the length of the record.

In the `SORT` statement, the list of sort keys is defined by the `KEY` clauses specifying either `ASCENDING` or `DESCENDING` sort order. The correct syntax requires the keyword `ON` before the first key clause, but Tiny Cobol does not accept this. If the unsorted input file `unsorted.dat` contains the following four records

```
20xxxxxxxxxxxfirst recordxxxxxxxxxxxxxxx
10xxxxxxxxxxxsecond recordxxxxxxxxxxxxxx
15xxxxxxxxxxxthird recordxxxxxxxxxxxxxxx
05xxxxxxxxxxxforth recordxxxxxxxxxxxxxxx
```

the result `sorted.dat` will be a nicely sorted file containing the records sorted with respect to the first two characters.

```
05xxxxxxxxxxxforth recordxxxxxxxxxxxxxxx
10xxxxxxxxxxxsecond recordxxxxxxxxxxxxxx
15xxxxxxxxxxxthird recordxxxxxxxxxxxxxxx
20xxxxxxxxxxxfirst recordxxxxxxxxxxxxxxx
```

The sort command provides hooks for input and output. Instead of reading from a file, a subroutine can be called which fills the sort file. In the following example, the input routine filters out records whose amount field is zero.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. sort-sample.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE    ASSIGN TO 'input.dat'
           ORGANIZATION IS LINE SEQUENTIAL.
    SELECT SORT-FILE        ASSIGN TO 'sort.dat'
           ORGANIZATION IS LINE SEQUENTIAL.
    SELECT SORTED-FILE      ASSIGN TO 'sorted.dat'
           ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE LABEL RECORDS ARE STANDARD.
01  INPUT-REC.
    05  INPUT-TYPE       PIC XX.
    05  INPUT-AMOUNT     PIC 9(8).
    05  FILLER              PIC X(30).
SD  SORT-FILE.
01  SORT-REC.
    05  SORT-NO             PIC XX.
    05  FILLER              PIC X(38).
FD  SORTED-FILE  LABEL RECORDS ARE STANDARD.
01  SORTED-REC              PIC X(40).
WORKING-STORAGE SECTION.
77  MORE-RECORDS PIC X(3) VALUE 'YES'.
PROCEDURE DIVISION.
MAIN.
    SORT SORT-FILE
        ASCENDING KEY SORT-NO
        INPUT PROCEDURE READ-INPUT
        GIVING SORTED-FILE
    STOP RUN.
READ-INPUT.
```

```
    OPEN INPUT INPUT-FILE
    PERFORM UNTIL MORE-RECORDS = 'NO '
        READ INPUT-FILE
            AT END
                MOVE 'NO ' TO MORE-RECORDS
            NOT AT END
                PERFORM HANDLE-INPUT-RECORD
    END-PERFORM
    CLOSE INPUT-FILE.
HANDLE-INPUT-RECORD.
    IF INPUT-AMOUNT = ZEROS
    THEN
        CONTINUE
    ELSE
        MOVE INPUT-REC TO SORT-REC
        RELEASE SORT-REC
    END-IF.
```

We have replaced the `USING` clause specifying the input file by the input procedure `READ-INPUT`. This subroutine contains the standard loop reading the input file and calls `HANDLE-INPUT-RECORD` for each record in the input file. This subroutine only copies records with a non-empty amount field to the sort record and tell the sort input procedure to advance using the `RELEASE` statement. The `RELEASE` statement is equivalent to the `WRITE` statement used for writing to normal files.

Similarly, we can use the output procedure hook to use our own routine to handle the sorted data instead of the standard output to a file using the `GIVING` clause.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. sort-sample.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT UNSORTED-FILE    ASSIGN TO 'unsorted.dat'
           ORGANIZATION IS LINE SEQUENTIAL.
    SELECT SORT-FILE        ASSIGN TO 'sort.dat'
           ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD  UNSORTED-FILE LABEL RECORDS ARE STANDARD.
01  UNSORTED-REC           PIC X(40).
SD  SORT-FILE.
01  SORT-REC.
    05  SORT-NO            PIC XX.
    05  FILLER             PIC X(38).
WORKING-STORAGE SECTION.
77  MORE-RECORDS PIC X(3) VALUE 'YES'.
PROCEDURE DIVISION.
MAIN.
    SORT  SORT-FILE
        ASCENDING KEY SORT-NO
        USING UNSORTED-FILE
```

```
            OUTPUT PROCEDURE WRITE-OUTPUT
        STOP RUN.
    WRITE-OUTPUT.
        PERFORM UNTIL MORE-RECORDS = 'NO '
            RETURN SORT-FILE
                AT END
                    MOVE 'NO ' TO MORE-RECORDS
                NOT AT END
                    PERFORM HANDLE-OUTPUT-RECORD
        END-PERFORM.
    HANDLE-OUTPUT-RECORD.
        DISPLAY 'NO=', SORT-NO.
```

```
result:
NO=05
NO=10
NO=15
NO=20
```

Similar to the input routine which uses `RELEASE` instead of `WRITE` to write to the special sort file, the output routine reads a record from the sort file using the `RETURN` statement instead of the normal `READ`. Otherwise, the output routine contains the usual read loop. In the example, we just display the sort number of each record.

Often, we would like to merge two sorted files into one, for example, when applying some updates to a large master file. Similar to the sort command, Cobol provides a special statement for this task. It works just like the sort command only that it takes multiple files in the `USING` clause.

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. sort-sample.
    ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
        SELECT INPUT-FILE-1     ASSIGN TO 'input1.dat'
               ORGANIZATION IS LINE SEQUENTIAL.
        SELECT INPUT-FILE-2     ASSIGN TO 'input2.dat'
               ORGANIZATION IS LINE SEQUENTIAL.
        SELECT SORT-FILE        ASSIGN TO 'sort.dat'
               ORGANIZATION IS LINE SEQUENTIAL.
        SELECT MERGED-FILE      ASSIGN TO 'merged.dat'
               ORGANIZATION IS LINE SEQUENTIAL.
    DATA DIVISION.
    FILE SECTION.
    FD  INPUT-FILE-1 LABEL RECORDS ARE STANDARD.
    01  INPUT-REC            PIC X(40).
    FD  INPUT-FILE-2 LABEL RECORDS ARE STANDARD.
    01  INPUT-REC            PIC X(40).
    SD  SORT-FILE.
    01  SORT-REC.
        05  SORT-NO             PIC XX.
        05  FILLER              PIC X(38).
```

```
FD  MERGED-FILE  LABEL RECORDS ARE STANDARD.
01  MERGED-REC              PIC X(40).
PROCEDURE DIVISION.
MAIN.
    SORT  SORT-FILE
        ASCENDING KEY SORT-NO
        USING INPUT-FILE-1, INPUT-FILE-2
        GIVING MERGED-FILE
    STOP RUN.
```

Merging the "master" file `input1.dat` containing the four records

```
10xxxxxx input1 record 1 xxxxxxxxxxxxxxx
20xxxxxx input1 record 2 xxxxxxxxxxxxxxx
30xxxxxx input1 record 3 xxxxxxxxxxxxxxx
40xxxxxx input1 record 4xxxxxxxxxxxxxxx
```

with the "update" file `input2.dat` containing the two records

```
05xxxxxx input2 record 1 xxxxxxxxxxxxxxx
25xxxxxx input2 record 4 xxxxxxxxxxxxxxx
```

results in the new sorted file `merged.dat`.

```
05xxxxxx input2 record 1 xxxxxxxxxxxxxxx
10xxxxxx input1 record 1 xxxxxxxxxxxxxxx
20xxxxxx input1 record 2 xxxxxxxxxxxxxxx
25xxxxxx input2 record 4 xxxxxxxxxxxxxxx
30xxxxxx input1 record 3 xxxxxxxxxxxxxxx
40xxxxxx input1 record 4xxxxxxxxxxxxxxx
```

4.3.5. Screen Definitions

The interactive input and output we have used so far is very limited: we display some message with the `DISPLAY` statement and ask for a single variable with `ACCEPT`. These statements can also be used to define much more sophisticated dialogs (not quite what we consider graphical user interfaces these days, but the kind of terminal screen you typically see in banks). Instead of a single variable, you provide the name of a complete screen defined in the screen section of the data division. Here is the interactive demonstration of the case statement using a screen definition.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. hello.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  RESULT PICTURE X(10) VALUE SPACES.
77  INPUT-VALUE  PICTURE 99 VALUE 0.
```

```
SCREEN SECTION.
01  INPUT-SCREEN.
    05  BLANK SCREEN.
    05  LINE 1 COLUMN 1 VALUE 'RESULT:'.
    05         COLUMN 12 PIC X(10) FROM RESULT.
    05  LINE 2 COLUMN 1 VALUE 'INPUT:'.
    05         COLUMN 8 PIC 99 TO INPUT-VALUE.
PROCEDURE DIVISION.
MAIN.
    PERFORM UNTIL INPUT-VALUE = 99
        DISPLAY INPUT-SCREEN
        ACCEPT INPUT-SCREEN
        EVALUATE INPUT-VALUE
            WHEN 1
                MOVE 'ONE' TO RESULT
            WHEN 2
                MOVE 'TWO' TO RESULT
            WHEN 3
                MOVE 'THREE' TO RESULT
            WHEN OTHER
                MOVE 'MORE' TO RESULT
        END-EVALUATE
    END-PERFORM
    STOP RUN.
```

When running the program, you will see an empty black screen showing the previous result (blank after startup) and asking for the new input value. The whole "look-and-feel" is defined in the screen section.

4.4. Libraries and Common Examples

Bibliography

I'm not a Cobol expert, but considering the number of copies I found in American bookstores, Stern & Stern [STERN00]> seems to be *the* book used to teach Cobol. It is indeed a very readable introduction to Cobol.

Nancy B. Stern and Robert A. Stern, 0-471-31881-7, John Wiley and Sons, 2000, *Structured Cobol Programming: For the Year 2000 and Beyond*.

Chapter 5. C

C was created by Dennis Richie and Kenneth Thompson at Bell Labs during the early 1970's as the system programming language for the UNIX operating system. Like Lisp and Scheme, C went through a standardization process during the 1980's leading to the ANSI/ISO standard in 1989 known as ANSI-C. It contains major improvements over the original language, in particular function prototypes (adopted from C++). A new version of the standard known as C99 has been published, guess, 1999. It includes, for example, wide character support. The following description uses ANSI-C as of 1989.

This short review of the C languages focuses on the features we will need when discussing the other programming languages in the following chapters. Readers who are familiar with the C language may quickly browse this chapter or skip it entirely.

5.1. Software and Installation

The examples in this chapter use the GNU C compiler which is available on almost any operating system. To enforce ANSI compliance we set the `-ansi` switch.

5.2. Quick Tour

C is a compiled language without an interpreter which would allow us to explore the language interactively. Instead we have to go through the usual edit-compile-run loop. The code is placed into files with the suffix `.c` or `.h` (header files for declarations shared by multiple programs, see below).

5.2.1. Hello World

```
#include <stdio.h>

main() {
  printf("Hello World");
}
```

Compared to the simple `print "Hello World"` we have seen so far, we need to understand a number of language features to implement our favorite message in C: libraries, preprocessor instructions, statement blocks, functions, and the entry point `main`.

Let's start with C's module structure. The core language is rather small and contains just the bare minimum to define functions and structures. Everything else, from input and output to memory management and string processing is defined in libraries. We will use only a few standard libraries, the most important one being `stdio` which allows us to show the results of our programs.

The `stdio` library contains the function `printf` for formatted output. It is declared in the header file `stdio.h`. To use the `printf` function, we have to include this header file using the preprocessor directive `#include`. The translation of our little program happens in two steps. First, the C preprocessor executes all the preprocessor commands (starting with a hash character) and expands the macros (defined with `#define`, see Section 5.2.6>). The result is plain C code without any preprocessor instructions or macros. If you want to see the generated code, you can run the preprocessor `cpp` directly. In case of the `#include` directive, the included file is copied into the generated code. The process is recursive, that is, the included file is actually processed by the preprocessor before inserting it into the code.

The remaining part of the program defines the function `main`. The function definition consists of the name of the function followed by the empty argument list `()` and a block of statements enclosed in the characteristic curly braces. Each statement ends with a semicolon. The call of the `printf` function looks just like a mathematical function call. Like in many other languages, String constants are enclosed in double quotes.

The compiler takes the preprocessed C code and turns it into machine code. This machine code contains a table declaring the data and functions used and defined by the code (the symbol table which you can display on UNIX systems using the `nm` command). When executing this code, the operating system looks for a function called `main` and calls it.

The `main` function as shown above is actually a shortcut. The "correct" version would define the return type in front of the function name and actually return a value.

```
#include <stdio.h>

int main() {
  printf("Hello World");
  return 0;
}
```

Since this is the default behavior of a function in C, we can omit both, the return type `int` and the `return` statement. In the following examples, we will omit the scaffolding and just list the main function followed by the result printed on the screen.

Simple arithmetic uses the familiar mathematical infix notation.

```
main() {
  printf("result=%d\n", 3+4*5);
  printf("result=%f\n", 1.5 * 2 + 3.5);
}

result=23
result=6.500000
```

Here we use the formatting capabilities of the `printf` function which work just like Python’s formatting operator (or better the other way round, since Python’s formatting was derived from C’s). C support a

large number of arithmetic operators including bit operations and update shortcut operators such as `+=` and `++` (equivalent to `+=1`). A very useful operator is the conditional expression (the one feature we miss the most in Python - it is going to be added soon). It is a ternary expression which corresponds to the `if` expression in functional languages and uses a question mark after the condition followed by the two alternatives separated by a colon.

```
int abs(int i) { return i<0? -i : i; }

main() {
  printf("result=%d\n", abs(-55));
  printf("result=%d\n", abs(55));
}

result=55
result=55
```

5.2.2. Control Flow

C provides the conditional and loop statements known from procedural languages such as `if/else` and `do/while`.

```
main() {
  int i;

  if (1 < 2) {
    printf("yes\n");
  }
  else {
    printf("no\n");
  }

  i = 0;
  while (i<3) {
    printf("i=%d\n", i);
    i += 1;
  }

  do {
    i -= 1;
    printf("i=%d\n", i);
  } while (i>0);
}

yes
i=0
i=1
i=2
i=2
i=1
i=0
```

This example already introduces an integer variable `i` with a type declaration which we will cover in the next section in more detail. The most useful control statement is the `for` loop. It consists of four parts: initialization, test, update, and a statement. The initialization statement is executed before entering the loop, the test is performed before every iteration, and the update after every iteration. The statement, which is typically a block of statements is executed in between. If the test fails, the loop is left. The loop every C programmer has entered about a million times walks through the integers between zero and some upper bound:

```
main() {
  int i;
  for (i=0; i<5; i++) {
    printf("i=%d\n", i);
  }
}
```

5.2.3. Basic Types

To me, the key to understanding C is the type system in general and pointers and type declarations in particular. Once you can read a declaration such as the following one of the `signal` function without blinking twice, there is hardly anything that can stop you in C.

```
void (*signal(int sig, void(* func)(int)))(int);
```

Before dissecting this declaration, let's start with the basics. C is a statically typed language with explicit type declarations. In contrast to dynamically typed languages, the type information is attached to variables rather than values. In C, the programmer has to use type declarations to tell the compiler that a variable or function is of a certain type. As we will see in the chapters about ML and Haskell, there are languages which leave most of this burden to the compiler using implicit types and type inference.

The following example introduces a simple integer function and uses it in the main function.

```
int times2(int x) { return 2*x; }

main() {
  int n = 5;
  double result;
  result = times2(n);
  printf("result=%f\n", result);
}

result=10.000000
```

In the simple cases, a C type declaration is put in front of the associated variable or function (for the return type). In the `main` function we introduce two local variables, the integer variable `n` and the floating point variable `result`. In C, local variables must be declared at the beginning of a block (in the newer members of the C family such as C++, Java, and C#, this rule was dropped and you can declare variables

where they are used for the first time). In reality, most C programmers declare local variables at the beginning of a function. As you can see, variables can be initialized as part of the declaration. The example also shows that C converts between types automatically whenever it makes sense, for example, from integer to double, but not vice versa. In other words, type conversion if performed automatically if no information is lost. We can also force a type conversion by putting the required type in parentheses in front of the value, an operation known as casting.

```
main() {
  double x = 5.5;
  int n = (int)x;
  printf("n=%d", n);
}

n=5
```

The next example shows the first deviation from the rule that type declarations are always put in front.

```
static void squares(int a[], int n) {
  int i;
  for (i=0; i<n; i++) {
    a[i] = i*i;
  }
}

main() {
  int i;
  int a[5];

  squares(a, 5);
  for (i=0; i<5; i++) {
    printf("square(%d)=%d", i, a[i]);
  }
}
```

One of C's (few) built-in data structures are arrays. An array contains a fixed number of values of the same type. The type declaration uses the same syntax as array access, that is, the index operator (square brackets). The array part of the type declaration is put behind the variable in questions. When used to access the array, the square brackets contain the zero-based index (to be interpreted as an "offset"), and when used in a type declaration they contain the size of the array. If the size of the array is not known (as for the argument `a` to the `squares` function), the square brackets are left empty.

The example also give a first glimpse at the dangers of C. The array does not contain information about its size, and there is no boundary checking when accessing array elements. The application has to make sure that the arrays are used properly, or otherwise it will experience one of the infamous buffer overflow bugs. That's why we have to pass the length of the array explicitly to the `squares` function. We can make our example a little bit safer by defining a constant for the size of the array.

```
main() {
  const int n = 5;
```

```
  int i, a[n];

  squares(a, n);
  for (i=0; i<n; i++) {
    printf("square(%d)=%d\n", i, a[i]);
  }
}
```

This shows another type feature introduced with the ANSI standardization of C: constants which are declared by preceding the type with the `const` qualifier.

Since arrays may contain arbitrary types, we can define multidimensional arrays as arrays of arrays as in the next example.

```
main() {
  const int m = 2;
  const int n = 3;
  double a[m][n];

  a[1][2] = 55;
  printf("x=%d", a[1][2]);
}
```

The variable `a` is a matrix of two rows and three columns. The six elements are stored in memory as one block of six integers, and the array access is implemented by computing the offset as row index times column size plus column index. Obviously, this approach generalizes to arrays of dimension greater than two.

As you can tell we are moving closer and closer to the machine level. The next topic takes us directly to the computer memory. C's pointers are nothing but addresses in (nowadays typically virtual) memory, and C gives the developer a lot of freedom to manipulate pointers and use them in creative manners. What the index operator `[]` is for arrays, the asterisk `*` is for pointers. In type declarations, an asterisk preceding an expression indicates a pointer, and in statements the same operator retrieve the value the pointer is pointing to (that is, the contents of the memory at the address contained in the pointer). The ampersand `&` does the opposite by returning the address of a variable as a pointer.

```
main() {
  int n = 5;
  int *np = &n;

  *np = 10;
  printf("n=%d", n);
}

n=10
```

What makes things interesting is that you can calculate with pointers. As an example, we can access the elements of an array using pointer calculations.

```
main() {
  int a[5];
  int *ap = &a[0];

  a[2] = 1;
  printf("a[2]=%d\n", a[2]);
  *(ap + 2) = 2;
  printf("a[2]=%d\n", a[2]);
  ap[2] = 3;
  printf("a[2]=%d\n", a[2]);
}
```

```
a[2]=1
a[2]=2
a[2]=3
```

The example shows three different ways to access an element of an array. First, we let the pointer `ap` point to the first element of the array. The first assignment uses the standard array index operator. The second one, `*(ap + 2) = 2` uses pointer arithmetic. We add the offset 2 to the pointer resulting in a pointer to the third element of the array. Using the asterisk, we access the value of this element and set it to two. The last assignment, `a[2] = 3` demonstrated C's shortcut syntax for the same expression: For a pointer `p`, the expression `p[i]` is equivalent to `*(p+i)`. Note that the pointer arithmetic takes the size of the underlying type (here integer) into account. At least, we don't have to compute the addresses in bytes.

Pointers are essential when dealing with dynamic data structures, for examples, arrays whose size is not known at compile time. The memory for these structures has to be allocated at run time (on the heap) and given back to the operating system once the memory is not used any more. Using C, those tasks are the responsibility of the programmer. New memory is allocated with the `malloc` function (and its siblings) and deallocated with `free`.

```
#include <stdio.h>
#include <malloc.h>

main() {
  const int n = 5;
  int* a = malloc(n * sizeof(int));
  a[2] = 55;
  printf("a[2]=", a[2]);
  free(a);
}
```

```
a[2]=55
```

The `malloc` function takes the number of bytes to be allocated and returns a generic pointer (of type `void*`). We therefore have to compute the actual size of our integer array in bytes using the `sizeof` function. The resulting pointer to the memory block allocated by the `malloc` function is automatically converted to an integer pointer.

With all the knowledge about arrays and pointers we can revisit strings. The standard strings in C are null-terminated arrays of 8-bit characters.

5.2.4. Structures and Type Definitions

Besides the basics types, arrays, and pointers, C allows you to define you own structures. A structure combines a fixed number of named fields.

```
main() {
  struct { const char* name; int age; } person;

  person.name = "Homer";
  person.age = 55;

  printf("person=%s, %d", person.name, person.age);
}
person=Homer, 55
```

The fields of structures are declared just like local variables of a functions. Like arrays, structures are static in the sense that the size and memory layout is completely determined at compile time. The fields of a structure are accessed using the dot notation. We can give the structure definition a name for reuse. In the following example, we use the named structure to define a pointer to the `person`.

```
main() {
  struct PERSON { const char* name; int age; } person;
  struct PERSON* person_ptr = &person;

  person.name = "Homer";
  person.age = 55;

  printf("person=%s, %d", person.name, person.age);
  (*person_ptr).name = "Bart";
  person_ptr->age = 10;
  printf("person=%s, %d", person.name, person.age);
}

person=Homer, 55
person=Bart, 10
```

As you can see, C provides the arrow operator `->` as a shortcut for the dereference operator `*` combined with the field access.

When dealing with complex types such as structures, it is useful to give the type a new name using C's type definitions. A type definition looks like a variable definition preceded by the keyword `typedef`. The variable name become the name of the new type. This makes the previous example much more readable.

```
typedef struct {
  const char* name;
```

```
  int age;
} Person;

main() {
  Person person;
  Person *person_ptr = &person;

  ...
}
```

With this concept we can start to define a number of functions manipulating `Person` structures.

```
void Person_print(Person* person) {
  printf("name=%s, age=%d", person->name, person->age);
}
```

Note that just the pointer is passed to function instead of copying the whole structure.

5.2.5. Functions

We have already used the basic syntax of function definitions in the previous sections. Functions are defined with their return type followed by the function name, the argument list, and a block of statements. A function can have local variables which must be declared at the beginning of the block. In contrast to other precedural languages (e.g., the Pascal family), function definitions can not be nested. In other words, functions live in a single global scope. However, we can hide a definition of global variables and functions inside a file using the `static` keyword.

```
static n = 5;
static int timesN(int x) { return n*x; }
```

Preceding a definition with `static` prevents the compiler from adding the symbols to the symbol table so that the associated variables and functions can not be seen by another module. As a rule, define all objects static unless they are used by other modules. (This is a UNIX-centric presentation of the situation. On Windows, one has to explicitly export definitions in order to make them available in other modules.)

C supports recursive functions as well.

```
static int fac(int n) { return n<2? 1 : n*fac(n-1); }
main() {
  printf("fac(5)=%d", fac(5));
}
```

```
120
```

You can call a function before it is defined by using a function declaration (this is essential for defining mutually recursive functions). These declarations are basically function definitions without the statement block. Like other declarations they end with a semicolon.

```
static int fac(int n);

main() {
  printf("fac(5)=%d", fac(5));
}

static int fac(int n) { return n<2? 1 : n*fac(n-1); }
```

```
120
```

Without the forward declaration, this code would have lead to an error. The C compiler reads and translates the file sequentially (leading to fast compilers). Function declarations are also the main means to share information between multiple compilation units. The header files such as `stdio.h` contain just the declarations of the exported functions. The associated compiled function definitions of the standard I/O library are contained in the system library and linked to the executable during linking.

We have seen functions of elementary types and structures, but does C support higher order functions? Is it possible to define a function like `map` which applies another function to all elements, say, in an array? In other words, can we pass a function as an argument to another function? As you might have guessed by now, the typical answer in C involves pointers, in this case, function pointers. But how do we define a function pointer type? The syntax follows the symmetry between the declaration of a variable and the declaration of the corresponding type. In case of a function, we take the function declaration, e.g., `int f(int i)`, replace the function name by the type name, and make it a pointer by preceding the name with an asterisk. All that's missing for our function pointer type definition is the `typedef` keyword and parentheses around the asterisk and the name to prevent C's preference rules from binding the asterisk to the `int` return type.

```
typedef int (*IntMap)(int i);

void map(IntMap f, int a[], int b[], int n) {
  int i;
  for (i=0; i<n; ++i) {
    b[i] = f(a[i]);
  }
}

static int times2(int i) { return 2*i; }

main() {
  const int n = 5;
  int i, a[n], b[n];

  for (i=0; i<n; i++) a[i] = i;
  map(times2, a, b, n);
  for (i=0; i<n; i++) {
    printf("b[%d]=%d\n", i, b[i]);
```

```
  }
}

b[0]=0
b[1]=2
b[2]=4
b[3]=6
b[4]=8
```

Note that the function `times2` is passed to the `map` function just using the function's name. You may have thought that the name has to be preceded by an ampersand to turn the function into a function pointer, but the syntax makes sense since the parentheses required when calling a function clearly distinguish a function call from the function itself.

It is not surprising that C ows much of its flexibility to function pointers. They can be used in any other data structures such as arrays and structures. Functions can also return function pointers, which leads us to the explanation of the `signal` function.

```
 void (*signal(int sig, void(* func)(int)))(int);
```

The definition becomes are lot more readable by introducing a type definition for signal handlers which are pointers to functions taking a single integer argument (the signal number) and returning nothing.

```
typedef void (*SignalHandler)(int signal);
```

With this definition, the `signal` function looks like this:

```
SignalHandler signal(int signal, SignalHandler handler);
```

The `signal` function is used to bind a signal handler to a signal. It returns the old signal handler so that the application can keep it in mind and restore the old situation if necessary.

We finish this section with a slightly longer example showing one tiny step towards the implementation of object oriented features in C. It demonstrates how structures and function pointers can be used to implement polymorphism.

```
typedef void (*PrintFunction)(void* object);

typedef struct {
  PrintFunction print;
} Class;

typedef struct {
  Class *class;
} Object;

static void print(void* o) {
  Object* object = (Object*)o;
```

```
  object->class->print(object);
}

/* Cat */

typedef struct {
  Class* class;
  const char* name;
} Cat;

static void Cat_print(void* p) {
  Cat* cat = (Cat*)p;
  printf("Miouw, I'm a cat, my name is %s", cat->name);
}

static Class Cat_class = { Cat_print };

static Cat* Cat_new(const char* name) {
  Cat* cat = (Cat*)malloc(sizeof(Cat));
  cat->class = &Cat_class;
  cat->name = name;
  return cat;
}

/* Dog */

typedef struct {
  Class* class;
  const char* name;
  const char* breed;
} Dog;

static void Dog_print(void* p) {
  Dog* dog = (Dog*)p;
  printf("Wouff, I'm a %s, my name is %s", dog->breed, dog->name);
}

static Class Dog_class = { Dog_print };

static Dog* Dog_new(const char* name, const char* breed) {
  Dog* dog = (Dog*)malloc(sizeof(Dog));
  dog->class = &Dog_class;
  dog->name = name;
  dog->breed = breed;
  return dog;
}

main() {
  Cat* cat = Cat_new("Felix");
  Dog* dog = Dog_new("Alex", "Terrier");

  print(cat);
  print(dog);
```

```
  free(dog);
  free(cat);
}

Miouw, I'm a cat, my name is Felix.
Wouff, I'm a Terrier, my name is Alex.
```

How does it work? Objects as well as their classes are represented as structures. The object structures have a pointer to their associated class structure as the first field. The class structure contains a pointer to a print function. Note that there is a lot of casting going on which can have desastrous effects if the structure does no look like expected. This admittedly oversimplified example is not so far from the actual implementation of object oriented extensions of C such as C++ and Objective C which hide all the casting and additional pointers from the developer. C++ does not use a pointer to a class structure, but to an array of function pointers (the virtual method table).

5.2.6. Macros

Since a programming language is hardly ever complete, its success often depends on the ability to extend the language with means of the language itself (rather than writing a new compiler). For C, this ability is provided by the preprocessor. We have used it from the very beginning to include the declarations of the standard I/O library. Besides merging other files into the source code, the preprocessor has two more main tasks: macros and conditional compilation. Macros allow you to substitute arbitrary text for symbols used in the code.

```
#define HELLO printf("Hello World")

main() {
  HELLO;
}
```

Some symbols such as the current file name (`__FILE__`) and line number (`__LINE__`) are predefined.

```
#include <stdio.h>

main() {
  printf("file=%s, line=%d", __FILE__, __LINE__);
}

file=sample.c, line=4
```

They are very handy when defining diagnostic messages, and, since the macro definitions are resolved at compile time, come at no cost. Macros can also have arguments which are substituted verbatim into the macro expansion.

```
#include <stdio.h>
```

```
#define LOG(message) printf("%s:%d %s", __FILE__, __LINE__, message)

main() {
  LOG("here we are");
}

sample.c:6 here we are
```

The macro syntax uses white space to separate the macro declaration (here: `LOG(message)`) from the definition which is read until the end of the line. If you want to define macros spanning multiple lines, you need to end all but the last line with a backslash as C's continuation character.

Other typical examples are small generic functions. Since C's function symbols are unique (no overloading), macros are the only way to define a "function" for multiple types.

```
#define MAX(a, b) (((a)>(b))? (a) : (b))

main() {
  int n = 50;
  double x = 1.23;
  printf("MAX(n + 5, 10)=%d\n", MAX(n + 5, 10));
  printf("MAX(x, 10)=%f\n", MAX(x, 10));
}

MAX(n, 10)=50
MAX(x, 10)=10.000000
```

Note that the macro calls look like function calls, but they are not. Instead the macro expression is pasted into the code. In contrast to functions, the arguments are not computed once and passed by value. The argument expressions are substituted literally for the placeholders in the macro definition. That's why we put parentheses around the arguments in the macro definition. Hence, `MAX(n+5,10)` will be replaced by the expression `(((n+5)>(10))? (n+5) : (10))`. Another example is the SWAP macro which swaps the values of two variables.

```
#define SWAP(T, a, b) { T tmp=a; a=b; b=tmp; }

main() {
  int x=5, y=10;
  SWAP(int, x, y);
  printf("x=%d, y=%d", x, y);
}

x=10, y=5
```

Here, we need to pass the type so that the macro can declare the temporary variable `tmp`. Originally, macros were also used to define constants, but with the introduction of proper typed constants in ANSI C this is not necessary anymore. So, instead of macro

```
#define PI 3.1415926535897931
```

better use the constant:

```
static const double PI = 3.1415926535897931;
```

We won't cover conditional compilation in this short chapter apart from saying that it is mainly used to adapt the source code to the different environments, hardware, and operating systems.

References

The one and only book one really needs about C is the language description [KERNIGHAN88]>, the second edition describing the ANSI standard ANSI-C. Peter van der Linden's book explains the more subtle features of the language and provides hundreds of tips and tricks to avoid the most common pitfalls.

Brian W. Kernighan and Dennis M. Ritchie, 0-13-110330-X, Prentice Hall, 1988, *The C-programming-language*.

Peter van der Linden, 0-13-177429-8, Prentice Hall, 1994, *Expert C Programming: Deep C Secrets*.

Chapter 6. Smalltalk

Alan Key created Smalltalk ("programming should be a matter of smalltalk") in 1971 at the famous Xerox Palo Alto Research Center (PARC). The Smalltalk implementation went through several releases before being made available outside Xerox Parc in 1980 (Smalltalk-80). Several commercial Smalltalk systems were developed during the next two decades, the most successful ones by Parc Place (Xerox's Smalltalk spinoff) IBM. Although considered a very productive development environment by many, Smalltalk never caught on as expected. One reason (besides the price tag of the commercial implementations) is probably the lack of a standard which means that Smalltalk programs can not be easily ported from one Smalltalk implementation to another. The advent of Java (and later C#) further dimished Smalltalk's market share.

6.1. Software and Installation

We use GNU Smalltalk (gst) as the test implementation. For the UNIX people, it is just the configure/make procedure using the latest version (at the time of this writing 2.0.11) of GNU Smalltalk downloaded from ftp.gnu.org/smalltalk. On Windows I've used cygwin, but couldn't compile the new version successfully. For the examples, I'm now using the last 1.x version, namely 1.95.12. For a successful compile I had to remove the tcp and example targets from the Makefile. Starting the gst command line application without options will give you plenty of diagnostics for each executed statement. It get quieter with the "-q" option (or even "-Q").

```
$ ./gst.exe -q
GNU Smalltalk Ready

st>
```

6.2. Quick Tour

6.2.1. Message Passing

The key to understanding Smalltalk is not its syntax (which is very small), but its concept. Smalltalk is pure object-orientation. Everything is an object: integers, strings, other instances of classes, the classes themselves, and even blocks of code. The only way to accomplish something in Smalltalk is to let these objects send messages to each other. Messages are sent to objects by writing the name of the message (the message selector) behind the object the message is sent to.

```
st> 'Hello World' printNl !
'Hello World'
```

What does this example tell us? First, Smalltalk strings are enclosed in single quotes. Double quotes are used for comments. Second, strings seem to understand the printNl and print themselves including the quotes. And finally, as GNU Smalltalk specialty, statements are finished with an exclamation mark (most other implementations use a period). Something you will appreciate about Smalltalk is its consistency. The message printNl is understood by any object, and, as we said in the beginning, everything is an object, even "primitive" types.

```
st> 1234 printNl !
1234
st> 1.234 printNl !
1.234
```

For printing object the message passing might make sense, but how is this supposed to work for, say, an expression such as "4 + 5"?

```
st> (4 + 5) printNl !
9
st> (2 + 3 * 4) printNl !
20
st> (2 + (3 * 4)) printNl !
14
st> (5 negated) printNl !
-5
```

The expression looks like a normal arithmetical expression, but behind the scenes Smalltalk interprets the first expression as the message "add 5" sent to the object "4". The second expression first sends the message "add 3" to the number "2". The resulting object "5" is then told to "multiply with 4". Message passing is simple evaluated from left to right, and you have to take care of the preference rules yourself. As a prefix operator, the minus sign does not fit into the message passing concept. Instead you need to send the "negated" message to a number to achieve the same effect.

The arithmetical operators are special because messages with arguments normally have method selectors ending with a colon. For example, sending the message "to: 10" to an integer n creates the interval from n to 10 (including, no half open interval semantics like in Python).

```
st> (1 to: 10) printNl !
Interval(1 2 ... 10)
st> ((1 to: 10) at: 2) printNl !
2
```

And for any sequence object, the message "at:" followed by an index n gives us the n’th element (counting from 1 unlike the languages influenced by C).

To make things more interesting, we have to introduce a variable in the environment of the Smalltalk interpreter.

```
st> Smalltalk at: #x put: 0 !
st> x printNl !
```

```
0
st> x := 'Hello' !
st> x printNl !
'Hello'
```

Don't worry about the strange name "#x". It is just the name "x", but considered a unique symbol (in contract to strings which allow multiple objects to have the same value 'x'). One the variable exists, we can assign to it using the assignment operator ":=". Variables are not bound to a type. Like in Python, a variable can first contain an integer and later be changed to a string (or any other object).

At this point we still wonder if the wonderful world of message passing makes live easier or harder. Just consider control statements such as if clauses and loops. To see how those can expressed quite elegantly, we first have to introduce another important feature which has not made it in many languages (Ruby being one of the notable exceptions): code blocks. A code block in Smalltalk is just a sequence of statements in square brackets. Code block are objects (surprise) and the most important message they understand is "value" which evaluates the code.

```
st> x := ['Hello' printNl] !
st> x value !
'Hello'
```

Since code blocks are objects, we can assign them to variables, pass them as arguments of messages, store them in collections, and so forth. Now you might guess how message passing can be used to implement control statements: The control commands become messages and the actions are passed as code blocks.

```
st> (2 < 3) ifTrue: x !
'Hello'
st> (2 > 3) ifTrue: x ifFalse: [ 'Ok' printNl ] !
'Ok'
st> 5 timesRepeat: x !
'Hello'
'Hello'
'Hello'
'Hello'
'Hello'
```

For iterations we need blocks with arguments. They are listed as identifiers, each preceded by a colon, and separated from the block's statements by a vertical bar.

```
st> x := [ :i | i printNl ] !
st> x value: 15 !
15
st> y := [ :i :j | (i*j) printNl ] !
st> y value: 2 value: 3 !
6
```

Blocks with multiple arguments respond to the value message with multiple value arguments. Code blocks can be viewed as anonymous procedures. They can become arbitrarily complex with multiple statements being separated by periods. You can even use local variables which have to be declared within a pair of vertical bars.

```
st> x := [ :i |
st>   | k |
st>   k := 5 * i.
st>   'k=' display.
st>   k printNl.
st>   (i > 5) ifTrue: [ 'big number' printNl ]
st> ] !
st> x value: 10 !
k=50
'big number'
st> x value: 3 !
k=15
```

The definition of "x" can be interpreted as the definition of a procedure similar do the following Python code.

```
>>> def x(i):
        k = 5 * i
        print "k=%d" % k
        if i > 5: print "big number"
>>> x(10)
k=50
big number
>>> x(3)
k=15
```

Like Smalltalk, Python allows to treat functions as objects, but it does not allow arbitrary anonymous functions (only lambda expression). On the other hand, Smalltalk's code blocks can not return values.

6.2.2. Collections

Now we are all set for collections and iterations. Kent Beck [BECK97]> devotes them a whole chapter in his highly recommended Smalltalk patterns book. The available methods are powerful enough to prevent you from using any explicit loop. The first example is the equivalent for a simple "for" loop (like in Pascal, not C's generalized version):

```
st> 1 to: 5 do: x !
1
2
3
4
5
st> 1 to: 6 by: 2 do: [:x | x printNl ] !
1
```

```
3
5
st> 3 to: 1 by: -1 do: [:x | x printNl ] !
3
2
1
```

How is the first statement to be read? We already know that sending the "to: 5" message to the integer object "1" creates the interval from one to five. Intervals respond (like all collections) to the message "do:" with a code block as an argument. When receiving this message, the interval iterates through itself and calls the block with the current value of the iterator. The second and third statement extends this kind of loop by adding a step argument (2 and -1, respectively). Here is a more interesting example:

```
st> y := Set new !
st> y isEmpty printNl !
true
st> y add: 'a'; add: 'b'; add: 'c' !
st> y printNl !
Set ('a' 'b' 'c' )
st> y do: x !
'a'
'b'
'c'
st> (y includes: 'a') printNl !
true
st> (y includes: 'q') printNl !
false
st> y remove: 'b' !
st> (y includes: 'b') printNl !
false
```

We create a set by sending the "new" message to the Set class (more on this below) and add three strings to the set. Note the shortcut notation when sending messages to the same object. Instead of repeating the object we can chain the messages separated by semicolons. The loop (`do:`) works exactly as in the previous example. The remaining statements demonstrate some more messages of sets.

All Smalltalk environments contain a rich set of collection classes including sets, lists, bags, ordered lists, arrays, and dictionaries. They all adhere to the same basic protocol which makes it easy to keep their usage patterns in mind.

```
st> Smalltalk at: #l put: (OrderedCollection new) !
st> l add: 'Joe'; add: 'John'; add: 'Mary' !
st> l do: [:each | each displayNl ] !
Joe
John
Mary
st> (l at: 2) printNl !
'John'
```

The "map" operation creating a new collection by applying a function to every element in the original collection corresponds to Smalltalk's "collect" message.

```
st> (l collect: [ :each | 'my name is ', each]) printNl !
OrderedCollection ('my name is Joe'
                   'my name is John'
                   'my name is Mary' )
```

If the equivalent of the "map" function exists, we also expect "filter" and "reduce". Filtering comes in two flavors, selection and rejection.

```
st> ((1 to: 10) select: [:each | each > 5]) printNl !
(6 7 8 9 10 )
st> ((1 to: 10) reject: [:each | each > 5]) printNl !
(1 2 3 4 5 )
```

Of course, rejection can be easily expressed using selection and negation (and vice versa), but having both helps to convey the intention of the program. The equivalent of "reduce" has an unusual selector "inject:into:", but otherwise behaves as expected.

```
st> ((1 to: 4) inject: 5 into: [:sum :each | sum + each]) printNl !
15
```

A bag is an unordered collection which, in contrast to a set, allows for duplicates.

```
st> y := Bag new !
st> y add: 'a'; add: 'b'; add: 'b'; add: 'c' !
st> y size printNl !
4
st> y do: [:i | i printNl] !
'a'
'b'
'b'
'c'
st> (y includes: 'b') printNl !
true
st> (y occurrencesOf: 'a') printNl !
1
st> (y occurrencesOf: 'b') printNl !
2
st> y asSet printNl !
Set ('a' 'b' 'c' )
```

We have used the global dictionary "Smalltalk" already to introduce global variables. Here are a few more messages including the iteration over the key-value pairs (equivalent to Java's Map.Entry).

```
st> y := Dictionary new !
st> y at: 'a' put: 1; at: 'b' put: 2 !
st> y printNl !
Dictionary (
```

```
        'a'->1
        'b'->2
)
st> (y at: 'a') printNl !
1
st> (y includesKey: 'a') printNl !
true
st> y associationsDo: [ :each |
        'key=' display.
        each key display.
        ', value=' display.
        each value displayNl
] !
key=a, value=1
key=b, value=2
```

Smalltalk also has something like a tuple, the Array. It is a read-only, fixed length collection, just like Python's tuple, and even has a built-in syntax.

```
st> Smalltalk at: #x put: #(1 2 3)
st> x inspect !
An instance of Array
  contents: [
    [1]: 1
    [2]: 2
    [3]: 3
  ]
```

6.2.3. Objects and Classes

Having covered the basics of Smalltalk syntax, it is now time to move to the heart of Smalltalk: objects and classes. It should be no surprise anymore that Smalltalk does not have any special syntax for class definitions, but relies on talking to existing classes to create new ones.

```
st> Object subclass: #Person
        instanceVariableNames: 'name age'
        classVariableNames: ''
        poolDictionaries: '' !
st> Person new printNl !
a Person
```

Here we create a new class called "Person" derived from Object with two attributes (or instance variables) "name" and "age". In Smalltalk speak, we tell the class object "Object" to create a subclass with the symbol #Person, and as part of the message we give Object the lists of instance variables names, class variable names, and poolDictionaries (which we won't cover here). As you can see, we can already create instances of our new class, but that's about all. We have no access to the attributes, not any Person-specific methods to call. As a next step, we can give the new class a description.

```
st> Person comment: 'I am representing persons with name and age' !
st> Person comment printNl !
'I am representing persons with name and age'
```

The first statement set the comment using the "comment:" message, and the second one reads the comment using the "common" message and prints it. To do the same with the name of a person, we have to define our first own methods.

```
st> !Person methodsFor: 'setters'!
st> name: aName
st>   name := aName
st> !!
st> !Person methodsFor: 'getters'!
st> name
st>   ^name
st> !!
st> x := Person new !
st> x name: 'Homer' !
st> x name printNl !
'Homer'
```

The notation with the exclamation marks is GNU Smalltalk specific. Normally, you create methods in a class browser where you enter the name and code in separate text fields. The argument to methodsFor is the so-called protocol the new method should belong to. Protocols are just groups of related methods. Following the protocol we find the signature of the method: the message name (or selector), and optionally the list of arguments with their names. The rest of the method definition is a sequence of statements, the body of the method. Values are returned using the caret operator. Within methods, you have access to the instance variables and to the two special variables "self" (the instance itself) and super (which we will demonstrate later).

Going through the definition of the method, there is nothing that prevents us from adding a method to an existing class. That's how we defined the methods for the Person class one after the other. As an example, we can teach every object how to say hello to the world by adding such a method to the base class Object.

```
st> !Object methodsFor: 'fun'
st> !
st> hello
st> 'Hello World' printNl
st> !!
st> 5 hello !
'Hello World'
```

As you can imagine this opens the door for all kinds of uses (and abuses). An application can easily extend the base library. In Java applications you often find collections of utility classes containing static methods which provide additional functionality for existing objects. In Smalltalk, you would add this behaviour directly where it belongs.

Also notice that Smalltalk does not have private methods. Methods are always public and attributes are always private. If you want to mark methods as private you have to resort to some naming scheme (e.g., myDoSomething).

6.3. More Features

The quick tour covered most of Smalltalk's structure, but touched only the surface of object oriented programming in Smalltalk and the existing class libraries.

6.3.1. An Object-Oriented Example

After what we've seen, Smalltalk seems like an interesting alternative to other programming languages with message passing as a unique consistent approach. But is it really better for larger projects? What does is feel like to program more complex tasks in Smalltalk? The example presented in this section tries to give a glimpse at object oriented programming in Smalltalk. If you are interested in more, I recommend Total Telecommunication's billing system in Martin Fowler's "Analysis Patterns" [FOWLER97]>. The example is taken from the GNU Smalltalk tutorial and models bank accounts. We start with the class main Account class which has just the one attribute every account has: the balance.

```
Object subclass: #Account
        instanceVariableNames: 'balance'
        classVariableNames: ''
        poolDictionaries: '' !

!Account class methodsFor: 'obtaining instances'!
new
|result|
result := super new.
result initialize.
^result
!!

!Account methodsFor: 'initialization'!
initialize
balance := 0
!!
```

This sequence of definitions is typical for a Smalltalk class. The constructor (class method "new") creates a new account object and initializes it by sending the "initialize" message. This allows derived class to easily extend the initialization. Next we add a few simple methods.

```
!Account methodsFor: 'printing'!
printOn: stream
super printOn: stream.
stream nextPutAll: ' with balance: '.
```

```
balance printOn: stream.
!!

!Account methodsFor: 'moving money'!
spend: amount
balance := balance - amount
!
deposit: amount
balance := balance + amount
!!
```

This is enough functionality to run a few tests.

```
Smalltalk at: #a put: (Account new) !
a printNl !
a deposit: 125!
a deposit: 20!
a printNl!
a spend: 10!
a printNl!
```

In reality, there are different kinds of accounts. First, we consider a savings account. Its 'interest' attribute contains the total interest of this account (starting with zero).

```
Account subclass: #Savings
instanceVariableNames: 'interest'
classVariableNames: ''
poolDictionaries: ''
category: nil!

!Savings methodsFor: 'initialization'!
initialize
interest := 0.
^ super initialize
!!

!Savings methodsFor: 'printing'!
printOn: stream
super printOn: stream.
stream nextPutAll: ' and interest: '.
interest printOn: stream.
!!

!Savings methodsFor: 'interest'!
interest: amount
interest := interest + amount.
self deposit: amount
!

clearInterest
|oldInterest|
oldInterest := interest.
```

```
interest := 0.
^oldInterest
!!
```

In the initialization method we see the application of the special variable "super" which allows us to call the method of the parent class Account after doing our own initialization of the interest. The second account is a checkings account maintaining the number of used and remaining checks (they still use checks in some parts of the world - and it is not old Europe). When writing a check (using the ’writeCheck’ method), the amount is spend and the check numbers updated.

```
Account subclass: #Checking
instanceVariableNames: ’checkCount checksLeft history’
classVariableNames: ”
poolDictionaries: ”
category: nil !

!Checking methodsFor: ’initialization’!
init
checksLeft := 0.
history := Dictionary new.
^super init
! !

!Checking methodsFor: ’printing’!
printOn: stream
super printOn: stream.
stream nextPutAll: ’ and checkCount: ’.
checkCount printOn: stream.
stream nextPutAll: ’ and checksLeft: ’.
checksLeft printOn: stream.

"Print the history of checks"
history associationsDo: [ :each |
stream nextPutAll: ’\ncheck no ’.
(each key) printOn: stream.
stream nextPutAll: ’: ’.
(each value) printOn: stream.
]
! !

!Checking methodsFor: ’spending’!
newChecks: number count: checkcount
checkCount := number.
checksLeft := checkcount
!

writeCheck: amount
| num |

"Check that we have checks left"
(checksLeft < 1)
ifTrue: [ ^self error: ’Out of checks’ ].
```

```
"Make sure we've never used this check number before"
num := checkCount.
(history includesKey: num)
ifTrue: [ ^self error: 'Duplicate check number'].

"Record the check number and amount"
history at: num put: amount.

"Update check numbers and balance"
checkCount := checkCount + 1.
checksLeft := checksLeft - 1.
self spend: amount.
^num
!!

!Checking methodsFor: 'scanning'!
checksOver: amount do: aBlock
history associationsDo: [ :each |
((each value) > amount)
ifTrue: [aBlock value: each ]
]
!!

Smalltalk at: #c put: (Checking new) !
c printNl !
c deposit: 250 !
c printNl !
c newChecks: 100 count: 50 !
c printNl !
(c writeCheck: 32) printNl !
c printNl !



c checksOver: 250 do: [:x | x printNl ] !
```

References

Kent Beck, 0-13-476904-X, Prentice Hall, 1997, *Smalltalk Best Practice Patterns*.

Martin Fowler, 0-201-89542-0, Addison Wesley Longman, 1998, *Analysis Patterns: Reusable Object Models*.

Chapter 7. Prolog

By now, we have seen procedural, object-oriented, and functional languages, but there is yet another promising approach to programming: "programming in logic" as embodied in the Prolog programming language. As you can imagine, the main area of application for a "logical" programming language is the so-called "artificial intelligence". While the data structures and algorithms of AI can be implemented in any language, we will see that Prolog allows for very natural and elegant solutions for these tasks. One remark you will find when looking for Prolog is that Lisp is the assembly language and Prolog the high level language of AI.

Prolog was invented in 1972 by the Alan Colmeraurer at the University of Marseilles as a theorem prover implementing the ideas of Robert Kowalski (University of Edinburgh). The first efficient Prolog compilers were developed by David Warren (also at the University of Edinburgh). Prolog has been primarily used in research, most notably as part of the Japan's ICOT Fifth Generation Computer Systems Initiative. A successful implementation was Borland's Turbo Prolog in the 1980's. In 1995, Prolog became an ISO standard.

7.1. Software and Installation

For the examples in this chapter, I have used the GNU implementation of prolog (version 1.2.18) by Daniel Diaz. It is a Prolog compiler (based on the Warren Abstract Machine) and implements most of the ISO standard. Calling `gprolog` takes us the the interactive shell.

```
GNU Prolog 1.2.18
By Daniel Diaz
Copyright (C) 1999-2003 Daniel Diaz
| ?-
```

Another popular open source Prolog compiler is SWI-Prolog by Jan Wielemaker.

7.2. Quick Tour

7.2.1. Goals, Facts, and Rules

What does our "Hello World" program look like in a logical programming language? It is just a single line.

```
| ?- write('Hello World').
Hello World

yes
```

The program itself does not look too different from many we have seen. The statement looks like a function call finished with a period. The output, however, shows an additional message "yes" that gives us some hint that this program does something else behind the scenes.

Prolog is all about facts, rules, and how to satisfy goals. The `write` "function" is actually a built-in goal that always succeeds and that happens to write a Prolog expression (in our case just a string) to an output stream as a side effect.

The first example in any Prolog tutorial is a family tree. The facts define who is whose child, and the after defining a few rules you can ask all kinds of questions concerning the relationships of the family members. Here are some facts about a well-known family.

```
parent(elizabeth, elizabeth2).
parent(george6, elizabeth2).
parent(elizabeth2, charles).
parent(elizabeth2, andrew).
parent(elizabeth2, edward).
parent(philip, charles).
parent(philip, andrew).
parent(philip, edward).
parent(charles, william).
parent(charles, henry).
parent(diana, william).
parent(diana, henry).
```

Using a Prolog interpreter, we could enter these facts directly in the shell, but since `gprolog` is a compiler we have to read them from an file. Hence, we place the facts in a file called `family.pl` and read it into the Prolog system using the `consult` goal.

```
| ?- consult('family.pl').
compiling .../family.pl for byte code...
.../family.pl compiled, 16 lines read - 1781 bytes written, 15 ms

yes
```

A shorthand for `consult(file)` is the file in square brackets, `[file]`. Alternatively, we could also read the facts from standard input by specifying `user` as the file name.

Now that these important facts are loaded, we can ask questions, that is, we can ask the Prolog system to satisfy goals. We can, for example, check if the system has really learned the facts.

```
| ?- parent(philip, andrew).
```

```
true ?

yes
| ?- parent(philip, hugo).

no
```

Things get a little bit more interesting when we introduce variables. You may have noticed that up to now we have used lowercase identifiers only. Prolog uses the case of identifiers to distinguish their meaning. Identifiers starting with an uppercase letter denote variables. If we want to find out who are andrew's parents, we just replace the first argument by a variable.

```
| ?- parent(X, andrew).

X = elizabeth2 ?
```

After finding the first answer, Prolog displays it and gives us the choice to stop the search or ask for more. Hitting the enter key lets us return to the main prompt, the semicolon gives us the next answer, and the key `a` (for all) lets Prolog look for all possible answers.

```
| ?- parent(X, andrew).

X = elizabeth2 ? ;

X = philip ? ;

no
```

For, we have only dealt with facts, but Prolog's real power comes with rules. How can we define a grand parent, for example? A grand parent is a parent who has at least one child who is a parent. In Prolog this can be expressed by the following rule.

```
grandparent(C,G) :- parent(C,P), parent(P,G).
```

The symbol `:-` means "follows from", and the comma combines multiple goals with "and" (conjunction). The whole rule can be read as "C is a grand parent of G if there is a P such that C is parent of P and P is parent of G". This is first order predicate logic with an implicit "for all" on the left and "exists" on the right hand side.

To enter this rule, you have to again enter it in a file and load this file using the `consult` goal, or read it from standard input using `consult(user)`. To finish your input you can either use the end-of-file key (CTRL-D on UNIX), or the built-in `end_of_file` term.

```
| ?- consult(user).
compiling user for byte code...
grandparent(C,G) :- parent(C,P), parent(P,G).
end_of_file.
```

```
user compiled, 2 lines read - 418 bytes written, 10844 ms

yes
```

With this rule in the system, we can now ask for henry's grand parents.

```
| ?- grandparent(X, henry).

X = elizabeth2 ? a

X = philip

no
```

Here is another rule defining the sibling relationship.

```
sibling(A,B) :- parent(P,A), parent(P,B), A \= B.
```

The only new element is the `/=` operator checking if two terms are not identical (a child is not a sibling of itself).

```
| ?- sibling(X,charles).

X = andrew ? a

X = edward

X = andrew

X = edward

no
```

Unfortunately, we get multiple entries since the rule applies to both parents. We have to wait until we can handle lists properly to solve this problem.

7.2.2. Structures

Prolog structures combine related data into a single term. They syntactically indistinguishable from goals, that is, they look like function calls as well.

```
| ?- X = a(1, 2).
```

```
X = a(1,2)

yes
| ?- a(X, Y) = a(1, b(c, d)).

X = 1
Y = b(c,d)

yes
```

The "function" is called the "functor" of the structure. I like to think of Prolog structures as named tuples (just put the functor as a name in front of a tuple).

As you can see, structures don't have to be declared, and they can participate in matching operations. Structures are the building blocks of Prolog's other data structures such as lists and arithmetical expressions. These other forms are just syntactic sugar for the equivalent nested structures.

7.2.3. Collections

It seems like programming for artificial intelligence requires lots of list processing, since the both, Lisp and Prolog, have strong list support. A Prolog list literal is a sequence of comma separated terms enclosed in square brackets (a list syntax which looks familiar by now).

```
| ?- X = [a, b, c].

X = [a,b,c]

yes
| ?- X = [1, 'blah', [x, y]].

X = [1,blah,[x,y]]

yes
```

Internally, list are represented as trees of linked nodes with head and tail, just like in Lisp. We can create a list using the dot operator `.` combining an element and a list as head and tail of the new list.

```
| ?- X = .(a, [b, c]).

X = [a,b,c]

yes
| ?- .(X, Y) = [a, b, c].
```

```
X = a
Y = [b,c]

yes
```

An alternative syntax is the vertical bar inside of square brackets separating head and tail.

```
| ?- X = [a | [b, c]].

X = [a,b,c]

yes
| ?- [X | Y] = [a, b, c].

X = a
Y = [b,c]

yes
```

It is interesting to see the built-in list predicates in comparison to the list fuctions in functional languages. There is, for example, a `member` goal which checks if a term is contained in a list. Using pattern matching, we can also use this goal to walk through the member of a list.

```
| ?- member(b, [a, b, c]).

true ? a

no
| ?- member(X, [a, b, c]).

X = a ? a

X = b

X = c

no
```

We could define this predicate ourselves with the following two rules.

```
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
```

Similarly, the built-in concatenation goal `append` can be used to show all the combinations of lists whose concatenation is identical to a given result list.

```
| ?- append([1, 2], [3, 4], X).

X = [1,2,3,4]
```

```
yes
| ?- append(X, Y, [1, 2, 3]).

X = []
Y = [1,2,3] ? a

X = [1]
Y = [2,3]

X = [1,2]
Y = [3]

X = [1,2,3]
Y = []

no
```

7.2.4. Arithmetic

You may have wondered why we did not test arithmetical expressions right after the "Hello World" program. The reason is that arithmetic does not fit easily into the Prolog's logical programming paradigm. Prolog was not invented to solve numerical problems. However, it is possible to perform computations in Prolog. The key are special operators which cause the Prolog system to evaluate an arithmetical expression before continuing the resolution.

If you would like to use Prolog as a calculator, the `is` operator is all you need. It works like the equality operator, but evaluates its right hand side before matching it with the left hand side.

```
| ?- X is 3 + 4 * 5.

X = 23

yes
```

The comparison operators evaluate the expressions on both sides and compare the numerical results.

```
| ?- 2 + 3 > 1 + 2.

yes
| ?- 2 + 2 =< 1 + 3.

yes
| ?- 2 + 2 >= 1 + 3.
```

Note that the "less or equal" operator has the unusual symbol =< (the only argument I can think of is the symmetry between "greater or equal" and "less or equal").

There are special operators for numerical equality and inequality, `=:=` and `=\=` which also force both sides to be evaluated.

```
| ?- 1 + 3 =:= 2 + 2.

yes
| ?- 4 =\= 5.

yes
```

7.3. More Features

7.3.1. Controlling Backtracking

7.3.2. Operators

Bibliography

[BRATKO01]> .

Ivan Bratko, Pearson Education, 2001, 0-201-40375-7, *Prolog Programming for Artificial Intelligence: Third Edition*.

Chapter 8. Ada

"Ada, language for a complex world" is the title of a (german) tutorial (http://www.uni-kassel.de/~bretz/computer/ada-tutorial/vorwort.html). One could also say "a complex language for a complex world", since Ada is a big, complex language containing a wealth of features.

Ada is the only language in this book that is the result of a contest. This contest was initiated by the DoD in 1975 to find *the* new programming language for mission critical systems (in particular real-time and embedded systems). The winner, designed by Honeywell-Bull, was named Ada after the world's first programmer, Lady Ada Lovelace, the assistant of Charles Baggabe. The first standard came out in 1983 (Ada83), and a significant extension including object-oriented features in 1995 (Ada95).

8.1. Software and Installation

We will use the GNU Ada95 compiler *Gnat*, version 3.14p, to run the examples in this chapter. It is part of most Linux distributions. The compiler consists of a whole set of tools to compile, link, and organize Ada programs. For most of our sample programs it is sufficient to call the utility `gnatmake` on the source file. It will perform the required compile and link steps automatically.

8.2. Quick Tour

8.2.1. Hello World

Here is the message to the (complex) world as seen by Ada.

```
with Ada.Text_IO;
procedure Hello is
begin
  Ada.Text_IO.Put_Line("Hello World");
end;
```

To run the program, put the source code in a file called `hello.adb` and call `gnatmake`.

```
ahohmann@kermit> gnatmake hello.adb
gnatgcc -c hello.adb
gnatbind -x hello.ali
gnatlink hello.ali
ahohmann@kermit> hello
Hello World
```

What does this simple example tell us? Apparently, Ada has a package structure, and we have to tell the compiler which packages we would like to use. In this case, it is the standard library package `Ada.Text_IO`. Coming from a structural programming background, Ada does not insist on classes for everything. Procedure `Put_Line`, for example, is a direct member of the standard `Text_IO` package, and our own procedure `Hello` does not have any surrounding structure.

From this tiny example we can tell that Ada definitely does not belong to the C family, but in contrast to Eiffel we have to live with plenty of semicolons separating statements. As a matter of style, Ada uses the underscore character and capitalization of the first letter to separate words in identifiers. However, this is nothing but a style convention, since Ada is case insensitive (apart from string literals).

Since Ada is mainly aimed at technical applications, is should not be surprising that standard arithmetical expressions work as expected.

```
with Ada.Text_IO;
with Ada.Float_Text_IO;

procedure Test_Arithmetic1 is
   X, Y: Float;
begin
   Y := 10.0;
   X := Y + 3.5 * 4.0 + 2.0 ** 3;
   Ada.Text_IO.Put("X=");
   Ada.Float_Text_IO.Put(X);
   Ada.Text_IO.New_Line(1);
end;

result:
X= 3.20000E+01
```

What's new in this example? First, the Pascal-like declaration of the variables `X` and `Y`. All local variables of a procedure have to be declared in advance in the block between `is` and the `begin`. Second, Ada uses, again like Pascal, the proper assignment operator `:=`. Third, arithmetical expressions use standard infix operators and preferences.

We also use a new standard package for printing floating point number, `Ada.Float_Text_IO`. Since the standard library calls start getting tedious and hard to read, we should introduce the `use` directive which imports all the symbols of a package into the local namespace (just like C++ and C#).

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;

procedure Test_Arithmetic2 is
   X, Y: Float;
begin
   Y := 10.0;
   X := Y + 3.5 * 4.0 + 2.0 ** 3;
   Put("X=");
   Put(X);
```

```
   New_Line(1);
end;
```

As we see, Ada supports overloading and picks the correct procedure depending on the argument type. If Java, C#, and Eiffel are called strongly typed languages, I'm tempted to call Ada a "very strongly typed" language. The constants in the example above, for example, have to be floating point constants or the program will not compile. If we use integer constants, we need to explicitly cast them to floating point numbers like in the following assignment.

```
procedure Test_Arithmetic is
   X: Float;
begin
   X := Float(5);
end;
```

8.2.2. Enumerations

We get another glimpse of Ada's very strong typing when looking at our first own type, a simple enumeration. While C's enumerations are just integers in disguise and Java abandoned enumerations alltogether, Ada fully supports them with type safety and a complete set of operations.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Enum_Test is
   type Color is (Red, Blue, Green);
   package Color_IO is new Enumeration_IO(Color); use Color_IO;
begin
   Put("Color'Pos(Blue)=");
   Put(Color'Pos(Blue), 1);
   Put(" (");
   Put(Blue);
   Put_Line(")");
end;

output:
Color'Pos(Blue)=1 (BLUE)
```

This example introduces a number of new features. We start with the definition of the enumeration type `Color` itself. Remember that identifiers are case insensitive, so that `Blue`, `BLUE`, and `bLue` are all the same value.

The next line is a first example of the instantiation of a generic package. A strongly typed language can hardly live without parametrized types (if only for type-safe collections), and generic packages are Ada's implementation of this concept. We have already used generic packages without thinking about it. The

standard package `Float_Text_IO`, for example, is an instantiation of the generic package `Float_IO`. We could have defined it as

```
package Float_Text_IO is new Float_IO(Float);
```

`Float_IO` is a generic package defining input and output operations for any kind of floating point type. We will learn how to define our own generic types in Section 8.3.2>.

Similar to `Float_IO`, the generic `Enumeration_IO` package defines procedures for any kind of enumeration type. In the example, we instantiate it with our own enumeration type `Color`.

The next new feature is the expression `Color'Pos(Blue)`. Ada defines a number of attributes for the entities (such as types, packages, array) occuring in the language. The syntax for accessing such a predefined attribute uses an apostrophe. One attribute of an enumeration type is the `Pos` function which return the integer position (starting at zero) of an enumeration value. Observe that, unlike C, the position is completely separated from the actual enumeration value.

8.2.3. Functions and Procedures

We have already defined simple procedures without parameters in the examples above. Parameters are specified in Pascal manner with the parameter name followed by a colon and the parameter's type. A function returns a value whose type is is stated with the `return` keyword.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   function Times2(X: Integer) return Integer is
   begin
      return 2*X;
   end;
begin
   Put("Times2(55)=");
   Put(Times2(55));
   New_Line(1);
end;
```

The example also demonstrates that fuctions can be defined within other functions or procedures. In general, Ada allows to definitions inside other definitions.

Multiple parameter declarations are separated with semicolons, and multiple parameters of the same type use a comma.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
```

```
procedure Main is
   function Add(A: Integer; B, C: Integer) return Integer is
   begin
      return A + B + C;
   end;
begin
   Put("Add(4, 5, 6)=");
   Put(Add(4, 5, 6));
   New_Line(1);
end;
```

By default, a function or procedure can not modify the values passed to it. The arguments are passed by value (and not copied back). However, we may qualify a parameter using the `in` and `out` qualifiers to change this default behavior. Here is the `Times2` function realized as a procedure modifying its argument.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   procedure Times2(X: in out Integer) is
   begin
      X := 2*X;
   end;

   X: Integer := 55;
begin
   Times2(X);
   Put(X);
   New_Line(1);
end;
```

8.2.4. Control Structures

Ada has all the control structures we expect from a modern procedural language. All use the same block structure with the `end` keyword following by the name of the control structure. There are two conditional statements, `if` and `case`.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   function Sign(X: Integer) return Integer is
   begin
      if X < 0 then
         return -1;
      elsif X > 0 then
         return 1;
```

```
      else
         return 0;
      end if;
   end;
begin
   Put(Sign(123));
   Put(Sign(-123));
   New_Line(1);
end;
```

The `case` statement allows us to uses ranges and alternatives to select the alternative actions.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

   function WorkDay(D: Day) return Boolean is
   begin
      case D is
         when Mon .. Fri => return True;
         when Sat | Sun => return False;
      end case;
   end;
begin
   if WorkDay(Mon) then
      Put("Monday is a work day");
   end if;
   New_Line(1);
end;
```

Alternatively, we could have used the `others` keyword to define the action for the remaining cases.

```
function WorkDay(D: Day) return Boolean is
begin
   case D is
      when Mon .. Fri => return True;
      when others => return False;
   end case;
end;
```

Loops come in three different flavors: plain loops with exit statements, `while` loops, and `for` loops over ranges of enumerated types. Here are three different ways to count from zero to nine.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   N: constant Integer := 10;
   I: Integer;
begin
   I := 0;
```

```
   loop
      Put(I);
      I := I + 1;
      if I = N then exit; end if;
   end loop;

   I := 0;
   while I < N loop
      Put(I);
      I := I + 1;
   end loop;

   for I in 0 .. N - 1 loop
      Put(I);
   end loop;
end;
```

The example also demonstrates how to define constants in Ada: just put the `constant` keyword in front of the type.

The `for` loop lets us go backwards as well, but we can not vary the step size. Note that the range is always specified in ascending order.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   N: constant Integer := 10;
begin
   for I in reverse 1 .. N loop
      Put(I);
   end loop;
end;
```

Ada's `for` loops are not restricted to integers. We can use any range of an enumerated type.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
begin
   for I in Day loop
      Put(Day'Pos(I));
   end loop;

   for I in Mon .. Fri loop
      Put(Day'Pos(I));
   end loop;
end;
```

The `exit` statement can also be used with the `while` and `for` loops, and it may refer to a label if we would like to exit not just the innermost loop. Just like the `goto` statement, we will conciously skip this part, since there are typically better solutions available.

8.2.5. Subtypes

While other strongly types languages offer a number of predefined integer and floating point types, Ada takes strong typing one step further and allows us to define any range as a new type. The statement

```
subtype Die is Integer range 1 .. 6;
```

defines the type consisting of the integer numbers one to six. The following program uses this type for a simple electronic die.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Numerics.Discrete_Random;

procedure Test_Arithmetic5 is
   subtype Die is Integer range 1 .. 6;
   package Random_Die is new Ada.Numerics.Discrete_Random(Die);
   use Random_Die;

   D: Die;
   G: Generator;
begin
   loop
      D := Random(G);
      Put(D);
      delay Duration(1);
   end loop;
end;
```

Again, we make use of a generic standard package. We define the random number package for our new type by instantiating `Discrete_Random`. We then use the `Generator` type out of this package to declare the random number generator for our subtype. The `Random` function (also from the `Random_Die` package) is called in an infinite loop to generate new random numbers. The `delay` statement holds the program for one second.

Similarly, the floating point interval between zero and one can be defined as the subtype

```
subtype Chance is Float range 0.0 .. 1.0;
```

When using such a subtype, leaving its range will cause a constraint exception.

```
procedure Test_Arithmetic3 is
   subtype Chance is Float range 0.0 .. 1.0;
   X: Chance;
begin
   X := 1.5;
end;

result:
raised CONSTRAINT_ERROR : test_arithmetic4.adb:5
```

8.2.6. Arrays

Another example of Ada's careful treatment of types are arrays. An array can be seen as a map of a fixed number of indexes to values. Each index is taken from a finite, enumerated set. In other words, an array maps the cross product of some finite enumerated types to the value type. In the simplest case we can define a vector, that is, a one dimensional array whose indexes are taken from an integer range.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;

procedure Main is
   A: array (Integer range 0 .. 2) of Float;
begin
   A(2) := 55.0;
   Put(A(2));
   New_Line(1);
end;
```

Instead of the integer range, we can also use our own enumeration type.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   type Color is (Red, Blue, Green);
   A: array (Color) of Integer;
begin
   for I in Color loop
      A(I) := 0;
   end loop;

   A(Red) := 37;

   for I in A'Range(1) loop
      Put(A(I));
   end loop;

   New_Line(1);
end;
```

This example also demonstrates how we can extract the range information from the array itself. The `Range(1)` attribute gives us the range of the first dimension.

We can initialize the array while defining it using Ada's flexible array "aggregates" which can be used to set individual elements, alternatives, ranges, or all remaining elements all in one expression.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   A: array (Day) of Integer := (
      Tue => 1, Mon | Wed => 2, Thu .. Sat => 3, others => 4);
begin
   for I in A'Range(1) loop
      Put(A(I));
   end loop;
   New_Line(1);
end;
```

Multidimensional arrays work the same way with full support for aggregates.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   type Schedule is array (Day, 0 .. 23) of Boolean;
   Busy: Schedule := (Mon .. Fri => (8 .. 16 => True, others => False),
                      others => (others => False));

   package Day_IO is new Enumeration_IO(Day); use Day_IO;
begin
   for D in Busy'Range(1) loop
      Put(D); Put(":");
      for H in Busy'Range(2) loop
         Put(" ");
         if (Busy(D, H)) then Put("T"); else Put("F"); end if;
      end loop;
      New_Line(1);
   end loop;
end;
```

Here we define a `Schedule` type mapping the days of the week and the 24 hours to the boolean type. We then declare the `Busy` variable of this type and initialize it with `True` during the working hours of the work days.

8.2.7. Access Types

Ada's typesafe variant of pointers are *access types*. Like in many other languages, the `new` operator (also called "allocator") creates an object on the heap and returns a reference to it, that is, an access type value which allows us to access the allocated object.

As one of Ada's unique twists, we dereference a pointer with the special `all` attribute.

8.2.8. Records and Objects

A record consists of named fields called "components" in Ada. Like in most languages we cover in this book, the components are referenced using a period between the record variable and the component name.

```
with Ada.Float_Text_IO; use Ada.Float_Text_IO;

procedure Main is
   type Point is record
      X: Float;
      Y: Float;
   end record;

   P: Point := (0.0, 0.0);
begin
   P.Y := 1.5;

   Put(P.X); Put(P.Y);
end;
```

Record initializer can become as complex as the array aggregates in the last section. In the previous example we have used positional values, but we can as easily use the component names `(X => 0.0, Y => 0.0)`, multiple alternatives `(X | Y => 0.0)`, the `others` keyword `(others => 0.0)`, or even a combination of positional and named initializer `(0.0, Y => 0.0)` to achieve the same effect.

Here is the Ada version of the simplistic `Person` structure.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type Ref_String is access String;
   type Person is
      record
         Name: Ref_String;
         Age: Integer := 0;
      end record;

   P: Person;
```

```
begin
   P.Name := new String'("Homer");
   P.Age := 55;

   Put_Line("Name=" & P.Name.all);
end;
```

The first step towards object orientation is a the ability to extends record types. In order to do so the base type has to be "tagged". For now, this only means adding the `tagged` keyword to the type definition. Under the hood, it implicitly adds type information to the record which can be used by Ada at runtime to determine the type of a record.

8.2.9. Packages

The main organizational unit of Ada is a package. A package combines types and functions which belong together just like attributes and methods of a class in a "pure" object-oriented language. Similar to C++ and Objective C, Ada splits the package definition into two parts: package declaration and package body. The package declaration is used by the compiler to support separate compilation units with well-defined interfaces (and enough information to link the separately compiled units to an executable).

The package version of "Hello World" consists of three files (this example is taken from gnat's user guide): The first file, `greetings.ads` ("ads" like "Ada Specification") contains the package specification declaring the package `Greetings` with two procedures `Hello` and `Goodbye`

```
package Greetings is
   procedure Hello;
   procedure Goodbye;
end Greetings;
```

The package body contained in `greetings.adb` ("adb" like "Ada Body") implements these procedures.

```
with Text_IO; use Text_IO;
package body Greetings is
   procedure Hello is
   begin
      Put_Line ("Hello WORLD!");
   end Hello;
   procedure Goodbye is
   begin
      Put_Line ("Goodbye WORLD!");
   end Goodbye;
end Greetings;
```

Finally, we need a main program using our new package. The filename `helloworld.adb` corresponds to the name of the main procedure.

```
with Greetings;
procedure HelloWorld is
begin
   Greetings.Hello;
   Greetings.Goodbye;
end HelloWorld;
```

When compiling the main program using `gnatmake helloworld.adb`, the compiler sees the import statement `with Greetings` and automatically looks for this package. It finds the specification and body in the local directory, compiles it, and links all the components together to obtain the `helloworld` application.

8.2.10. Objects

8.3. More Features

8.3.1. Function Pointers

In many situations, passing functions to other functions as parameters is the most suitable way to create flexible programs. Therefore, Ada95 supports type-safe function pointers or "access types for functions" in Ada parlance. Here is an implementation of the trapezoidal rule to compute the integral of a function.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;

procedure Main is
   type Float_Function is access function (X: Float) return Float;

   function Integrate(F: Float_Function; A, B: Float) return Float is
      X: Float := A;
      Y: Float := 0.0;
      DeltaX: Float := 1.0e-5;
```

```
   begin
      loop
         Y := Y + DeltaX * (F(A) + F(B)) / 2.0;
         X := X + deltaX;
         if X >= B then exit; end if;
      end loop;
      return Y;
   end;

   function Square(X: Float) return Float is
   begin
      return X*X;
   end;
begin
   Put(Integrate(Square'Access, 0.0, 2.0));
   New_Line(1);
end;
```

As with other values, we obtain the pointer (or access) to a function using the `Access` attribute. We can call the access function as if it were a regular function (as long as it has parameters - otherwise we need to user `F.all`).

8.3.2. Generic Packages

Strong typing necessitates some mechanism to define types depending on other types. Ada lets us parametrize functions and packages with type, functions, values, and even other packages.

The following first example shows a generic `Swap` procedure using the `Item` type as a type parameter.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   generic
      type Item is private;
   procedure Swap(X, Y: in out Item);

   procedure Swap(X, Y: in out Item) is
      T: Item;
   begin
      T := X; X := Y; Y := T;
   end;

   procedure Integer_Swap is new Swap(Integer);

   A: Integer := 55;
```

```
   B: Integer := 66;
begin
   Put(A); Put(B); New_Line(1);
   Integer_Swap(A, B);
   Put(A); Put(B); New_Line(1);
end;
```

The procedure becomes generic by placing the declaration of the generic parameters between the `generic` and `procedure` keywords. When using generic constructs, we always have to separate the declaration of the generic entity (here the procedure `Swap`) from its definition. In contrast to C++, we do not have to repeat the generic part in the definition.

To use the generic procedure, we have to instantiate it with a generic parameter. In our example, we instantiate the generic `Swap` procedure with the `Integer` type. The generic parameters are passed just like any other function or procedure parameters.

As mentioned above, generics can be used with almost every construct in Ada. Here is a slightly more complex example defining a generic package implementing a stack.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   generic
      Max: Positive;
      type Item is private;
   package Stack is
      procedure Push(X: in Item);
      function Pop return Item;
      function Is_Empty return Boolean;
   end Stack;

   package body Stack is
      Data: array(0 .. Max) of Item;
      Top: Integer range 0 .. Max := 0;

      procedure Push(X: in Item) is
      begin
         Data(Top) := X;
         Top := Top + 1;
      end Push;

      function Pop return Item is
      begin
         Top := Top - 1;
         return Data(Top);
      end Pop;

      function Is_Empty return Boolean is
      begin
```

```
         return Top = 0;
      end Is_Empty;
   end Stack;

   package My_Stack is new Stack(10, Integer);
   use My_Stack;

begin
   Push(55);
   Push(66);

   while not Is_Empty loop
      Put(Pop); New_Line(1);
   end loop;
end;
```

The approach is the same: we have to split declaration and definition and put the generic part ahead of the declaration. The example also demonstrates the use of two different kinds of generic parameters: the positive integer `Max` and the (arbitrary) type `Item`.

The example becomes more useful if we define a stack type inside a generic package. This way we can use multiple stacks.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   generic
      type Item is private;
      Max: Positive;
      Default: Item;
   package Stack_Of is
      type Stack is private;

      procedure Push(S: in out Stack; X: in Item);
      procedure Pop(S: in out Stack; X: out Item);
      function Is_Empty(S: in Stack) return Boolean;
   private
      type Stack_Data is array(0 .. Max) of Item;
      type Stack is
         record
            Top: Integer range 0 .. Max := 0;
            Data: Stack_Data := (others => Default);
         end record;
   end Stack_Of;

   package body Stack_Of is
      procedure Push(S: in out Stack; X: in Item) is
      begin
         S.Data(S.Top) := X;
         S.Top := S.Top + 1;
```

```
      end Push;

      procedure Pop(S: in out Stack; X: out Item) is
      begin
         S.Top := S.Top - 1;
         X := S.Data(S.Top);
      end Pop;

      function Is_Empty(S: in Stack) return Boolean is
      begin
         return S.Top = 0;
      end Is_Empty;

   end Stack_Of;

   package My_Stack is new Stack_Of(Integer, 10, 0);

   S: My_Stack.Stack;
   X: Integer;
begin
   My_Stack.Push(S, 55);
   My_Stack.Push(S, 66);

   while not My_Stack.Is_Empty(S) loop
      My_Stack.Pop(S, X);
      Put(X); New_Line(1);
   end loop;
end;
```

Once we have understood Ada's generic machinery, its application to more complex situations is straight forward.

8.3.3. Overflow

Ada was clearly designed with safety as a first priority (that is, for "mission critical" applications in the sense of "space mission"). As an example, all numerical computations are checked for overflows. The following program would run happily forever using C, but stops rather quickly in Ada once the limit of the integer range is exceeded.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Int_Overflow is
   X: Integer := 1;
begin
   loop
```

```
      X := 2 * X;
      Put("X=");
      Put(X);
      New_Line(1);
   end loop;
end;

output:
X=          2
X=          4
X=          8
...
X=  268435456
X=  536870912
X= 1073741824

raised CONSTRAINT_ERROR : int_overflow.adb:8
```

Using the `gnat` compiler, we have to specify the `-gnato` option to experience the desired behavior, since `gnat` switches the overflow checking off by default.

8.3.4. Modular Types

Here is another example of Ada's nifty little features solving everyday programming problems: modular types. Who has not dealt with some cyclic integer type throughout his or her programming career coding the module arithmetic by hand? In Ada, we simply define the type as `mod N` where `N` is some positive integer.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   N: constant Integer := 3;
   type Ring is mod N;
   X: Ring := 0;

   package Ring_IO is new Modular_IO(Ring); use Ring_IO;
begin
   for I in 0 .. 10 loop
      Put(X);
      X := X + 1;
   end loop;
end;
```

The modular type `Ring` consists of the numbers zero to two. All computations, such as adding on in the loop, are performed modulo three. Like enumerations, modular types have their own generic input/output package `Modular_IO` which we instantiate here as `Ring_IO` in order to be able to print the value of the modular variable `X`.

The standard package `Interfaces` (called this way because it is used to interface with other languages such as C) contains a number of modular types where the modulus `N` is a power of two, for example `Unsigned_8` with modulus 256. Together with these types, the package also provides shift and rotate functions.

```
with Ada.Text_IO; use Ada.Text_IO;
with Interfaces; use Interfaces;

procedure Main is
   X: Unsigned_8 := 1;

   package Unsigned_8_IO is new Modular_IO(Unsigned_8); use Unsigned_8_IO;
begin
   for I in 0 .. 10 loop
      Put(X);
      X := Rotate_Right(X, 1);
   end loop;
end;

output:    1 128  64  32  16   8   4   2   1 128  64
```

8.3.5. Parallelism

Most modern programming languages support parallel programming with some multithreading API and sometimes additional primitives for synchronization such as Java's `synchronized` keyword. Ada uses a different approach and models typical patterns of defining and controlling parallel activities directly in the language.

The first tool is the `task` construct which allows us to define an activity which is run in parallel to the instructions of a procedure.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   task Sub;
   task body Sub is
   begin
      for I in 1 .. 5 loop
         Put("Sub:  "); Put(I); New_Line(1);
         delay 0.2;
      end loop;
   end Sub;
begin
   for I in 1 .. 5 loop
      Put("Main: "); Put(I); New_Line(1);
      delay 0.1;
```

```
   end loop;
end;
```

The main procedure counts from one to five with a delay of 0.1 seconds. The task `Sub` also counts from one to five, but with a delay of 0.2 seconds. The resulting output shows how the two activities are executed in parallel. The procedure stops after all threads are done, that is, the main thread is waiting until the sub task is finished.

```
output:
Main:           1
Sub:            1
Main:           2
Sub:            2
Main:           3
Main:           4
Sub:            3
Main:           5
Sub:            4
Sub:            5
```

A task allows us to run multiple activities in parallel, but to become useful they have to be able to communicate with each other. There are two way to accomplish this kind of communication in Ada: using shared data or sending messages.

Beginning with messages, we can define entry points which cause a task to wait until another task calls (sends a message to) the entry point. This mechanism is called a rendezvous. Apart from the different keywords (`entry` for the declaration and `accept` for the definition), an entry looks just like a procedure.

The following example defines the entry `Wake_Up` in the task `Sub` with a single integer argument. The task consists of a loop which processes `Wake_Up` messages until the argument is equal to zero.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   task Sub is
      entry Wake_Up(I: Integer);
   end Sub;

   task body Sub is
      Stop: Boolean := False;
   begin
      while not Stop loop
         Put("Sub:  Wait"); New_Line(1);
         accept Wake_Up(I: Integer) do
            Put("Sub:  "); Put(I); New_Line(1);
            if I = 0 then
               Stop := True;
            end if;
```

```
         end Wake_Up;
      end loop;
      Put("Sub:  Stop"); New_Line(1);
   end Sub;
begin
   for I in reverse 0 .. 3 loop
      delay 0.1;
      Put("Main: Send"); New_Line(1);
      Sub.Wake_Up(I);
   end loop;
   Put("Main: Stop"); New_Line(1);
end;
```

The main procedure sends `Wake_Up` messages counting from three down to zero. The statement `Sub.Wake_Up(I)` submitting the message looks like a method call. Here is corresponding output:

```
Sub:  Wait
Main: Send
Sub:            3
Sub:  Wait
Main: Send
Sub:            2
Sub:  Wait
Main: Send
Sub:            1
Sub:  Wait
Main: Send
Sub:            0
Main: Stop
Sub:  Stop
```

Of course, we are not retricted to a single entry. Here is an example of a buffer containing a single value. It is implemented as a task with the two entries `Set` and `Get`.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   task Buffer is
      entry Set(X: in Integer);
      entry Get(X: out Integer);
   end;

   task body Buffer is
      Value: Integer;
   begin
      loop
         accept Set(X: in Integer) do
            Put("Buffer.Set: "); Put(X); New_Line(1);
            Value := X;
         end Set;
```

```
         accept Get(X: out Integer) do
            Put("Buffer.Get: "); Put(Value); New_Line(1);
            X := Value;
         end Get;
      end loop;
   end Buffer;

   task Consumer;
   task body Consumer is
      I: Integer;
   begin
      loop
         Buffer.Get(I);
         Put("Consumer:   "); Put(I); New_Line(1);
      end loop;
   end Consumer;

begin
   for I in 0 .. 2 loop
      Buffer.Set(I);
      delay 1.0;
   end loop;
end;
```

The `Consumer` task continuously asks the buffer for the latest value and "consumes" it. The main procedure inserts the values zero to two into the buffer with a one second delay. Here is the resulting output:

```
Buffer.Set:           0
Buffer.Get:           0
Consumer:             0
Buffer.Set:           1
Buffer.Get:           1
Consumer:             1
Buffer.Set:           2
Buffer.Get:           2
Consumer:             2
```

Note how the buffer task and its entries make sure that the actions are carried out in the correct order.

In the previous example the consumer was fast and the producer (the main task) slow. The opposite case is often handled with a queue. In this situation, the task may receive both events (provided the queue is neither empty nor completely full). Ada supports this with the `select` statement and guarding conditions which can be associated with entries.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   task Queue is
```

```
      entry Push(X: in Integer);
      entry Pull(X: out Integer);
   end;

   task body Queue is
      N: Integer := 10;
      In_Ptr, Out_Ptr, Count : Integer := 0;
      Values: array (Integer range 0 .. N - 1) of Integer;
   begin
      loop
         select
            when Count < N =>
               accept Push(X: in Integer) do
                  Values(In_Ptr) := X;
                  In_Ptr := (In_Ptr + 1) mod N; Count := Count + 1;
               end;
         or
            when Count > 0 =>
               accept Pull(X: out Integer) do
                  X := Values(Out_Ptr);
                  Out_Ptr := (Out_Ptr + 1) mod N; Count := Count - 1;
               end;
         end select;
      end loop;
   end Queue;

   task Consumer;
   task body Consumer is
      I: Integer;
   begin
      loop
         delay 1.0;
         Queue.Pull(I);
         Put("Consumer: "); Put(I); New_Line(1);
      end loop;
   end Consumer;

   task Producer;
   task body Producer is
   begin
      for I in 10 .. 12 loop
         delay 3.0;
         Put("Producer: "); Put(I); New_Line(1);
         Queue.Push(I);
      end loop;
   end Producer;

begin
   for I in 0 .. 2 loop
      Put("Main:     "); Put(I); New_Line(1);
      Queue.Push(I);
   end loop;
end;
```

To make things more interesting we added a second (slow) producer. This way we see some of the scenarios in the output:

```
Main:               0
Main:               1
Main:               2
Consumer:           0
Consumer:           1
Producer:          10
Consumer:           2
Consumer:          10
Producer:          11
Consumer:          11
Producer:          12
Consumer:          12
```

First, the fast main producer pushes the values zero to two on the queue. They are consumed by the consumer, while the slow producer starts pushing his values. In the end the consumer is faster than the producers.

For shared data access as the second way of communication between parallel tasks, Ada has so-called *protected objects*. Apart from the `protected` keyword they look like packages, but the data (which must be defined in the private section of the protected object's declaration) is protected from simultaneous access. The following example wraps a simple integer value in a protected object `Shared_Data`.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   protected Shared_Data is
      function Get return Integer;
      procedure Set(New_Value: Integer);
   private
      Value: Integer := 0;
   end Shared_Data;

   protected body Shared_Data is
      function Get return Integer is
      begin
         return Value;
      end Get;

      procedure Set(New_Value: Integer) is
      begin
         Value := New_Value;
      end Set;
   end Shared_Data;
begin
   Shared_Data.Set(55);
   Put(Shared_Data.Get); New_Line(1);
```

```
end;
```

In many cases we would like to define not just a single protected object, but a type which we can use to create as man protected objects as we want to. This is achieved by adding the `type` keyword to the declaration of the protected object. Here is a variation of the previous example using such a protected type.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   protected type Shared_Data is
      function Get return Integer;
      procedure Set(New_Value: Integer);
   private
      Value: Integer := 0;
   end Shared_Data;

   protected body Shared_Data is
      function Get return Integer is
      begin
         return Value;
      end Get;

      procedure Set(New_Value: Integer) is
      begin
         Value := New_Value;
      end Set;
   end Shared_Data;

   X: Shared_Data;
begin
   X.Set(55);
   Put(X.Get); New_Line(1);
end;
```

As you can imagine, protected types are useful for the typical semaphore patterns.

8.4. Discussion

In this short chapter (which is already longer than most of the other ones), we could cover only a fraction of Ada's features. Ada is a complex language with dozens of special syntactic constructs and rules how they can applied. However, this also means that Ada has a direct answer for many computing tasks as we

have see, for example, when discussing parallelism. It just takes a little bit more time and experience to find this answer.

Chapter 9. SQL

SQL (Structured Query Language) in a book about general purpose programming languages? Our main goal is to find the most expressive programs, and SQL is definitely able to describe the solution of some complex tasks in a concise and expressive way. These tasks do not only include queries and updates, but also the related transaction handling which is not as easily handled by any of other language in this book.

The relational database model was developed by Ted Codd at IBM in the late 1960's and first published in 1970. SQL dates back to IBM's implementation of a relational database system during the 1970's. This system had a query language called SEQUEL (Structured English Query Language) whose name was later shortened to SQL. The language was also adopted by another (at that time) small company that created a relational database management system called Oracle.

The first ANSI standard for SQL, SQL86, was adopted in 1986. Since then there have been three updates, SQL89, SQL92, and finally SQL99 (now standardizing everything from complex data types (arrays, etc.) to a call level interface similar to ODBC). However, there are significant differences between the between the SQL implementations of the different relational database systems, especially when it comes to the procedural extensions of SQL.

9.1. Software and Installation

SQL is always implemented as part of a relational database system (RDBMS) which means that we have to pick a suitable RDBMS for our examples. Sticking to open source, PostgreSQL (http://www.postgresql.org) offers very good SQL support and also has a extension similar to Oracle's psql which shows how procedural elements can be added to SQL to make it a fully-fledged general purpose programming language.

Since we are dealing with a database system based on the client-server model, getting started takes a little bit more effort than just starting an interactive shell. We need to install the software, start the database server, create a user and database, and finally start the interactive client. Fortunately, PostgreSQL is fairly easy to install, and most of this burden is handled automatically. When installing the PostgreSQL package on a Linux system (Debian in my case), the system creates the UNIX user `postgres`under which the server runs and adds the server to the boot sequence (`/etc/init.d/postgresql`) so that it starts automatically. The only thing left to do is to create a database user for our personal UNIX user (in my case `ahohmann`) by calling the `create_user` command under the `postgres` user.

Once this is accomplished, we can start PostgreSQL's interactive client `psql` and start our experiments.

```
ahohmann@kermit:~$ psql sample
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
```

```
\h for help with SQL commands
\? for help on internal slash commands
\g or terminate with semicolon to execute query
\q to quit
```

This interactive shell allows us to enter any SQL statement. Shell commands start with a backslash (e.g., `\q` to leave the shell). We can use the help commands `\?` and `\?` to get an overview of the available commands and SQL statements.

9.2. Quick Tour

9.2.1. Expressions

Typically, SQL tutorials start with a small sample table and some simple queries, but since we are looking at SQL as a general purpose programming language, we stick to our "Hello World" program.

```
sample=> select 'Hello World';
  ?column?
-------------
 Hello World
(1 row)
```

Nonetheless, we end up with a `select` statement. Whenever we want to get some response from SQL, we have to define a query using the `select` keyword. In the simplest case, the `select` keyword is followed only by an expression. The expression is evaluated and the resulting value is the result of the `select` statement.

The result of a query is always a table (as the central concept of the relational model). Hence, we get our single message string in form of a table with the single column named `?column?` and a single row containing the actual data. We can help the server to choose a better name for the column using an `as` clause.

```
sample=> select 'Hello World' as message;
   message
-------------
 Hello World
(1 row)
```

If we can an individual string, what about arithmetical expressions? We can indeed abuse our PostgreSQL client as a calculator.

```
sample=> select 3+4*5;
```

```
 ?column?
----------
       23
(1 row)

sample=> select 1.5e-2 * 5;
 ?column?
----------
    0.075
(1 row)

sample=> select 2 ^ 10;
 ?column?
----------
     1024
(1 row)
```

We can also call functions as part of the `select` expression. The following examples use the built-in square root function `sqrt`, the substring function `substr`, and the function `now` returning the current date and time.

```
sample=> select sqrt(2.0);
      sqrt
-----------------
 1.4142135623731
(1 row)

sample=> select substr('Hello World', 2, 6);
 substr
--------
 ello W
(1 row)

sample=> select now();
              now
--------------------------------
 2003-12-22 12:01:20.401652+01
(1 row)
```

There are hundreds of built-in functions for mathematics, strings, dates, and so forth. To see a complete list, use the `\df` (describe functions) command. To find out more about a particular function, call `\df` followed by the name of the function.

```
sample=> \df sqrt
               List of functions
 Result data type | Name | Argument data types
------------------+------+---------------------
 double precision | sqrt | double precision
 numeric          | sqrt | numeric
(2 rows)
```

9.2.2. Tables and Queries

It is about time to introduce our own tables and data so that we can approach SQL's main features. SQL consists of (at least) two languages: a language to define relational data models (DDL - Data Definition Language) and a language to query and manipulate the actual data contained in these tables (DML - Data Manipulation Language).

The relational model is simple and powerful. All data is organized in tables. A table has a fixed number of columns. The data is contained in the rows (or tuples) of the table. A row is uniquely identified by its values (in the table's columns). In more mathematical terms, a table is a subset of the cross product of the column domains, where a column's domain is the set of allowed values in the column. All operations selecting and combining tables can be interpreted as operations on these sets (the "relations").

As our first relation, let's define a table containing first name, last name, and birthday of our friends.

```
sample=> create table friend (
sample(>   firstname char(20),
sample(>   lastname char(20),
sample(>   birthday date);
CREATE
```

A table is defined with the `create table` command followed by the name of the table and the list of column definitions as a comma separated list in parentheses. Each column definition consists of the column's name, its type, and optionally additional flags controlling the behavior of the column. In our example we use two types: `char(20)` for strings of up to 20 characters and `date` for dates (just the date, no time).

We can now insert tuples into this table using SQL's `insert` command.

```
sample=> insert into friend values ('Homer', 'Simpson', '15/05/1950');
INSERT 16576 1
sample=> insert into friend values ('Bart', 'Simpson', '20/07/1990');
INSERT 16577 1
```

The first number in PostgreSQL's response if the object id supplied by PostgreSQL. This id is a PostgreSQL specific features we can forget for now. The second number is the number of rows inserted into the table.

Now that we have some data available, we can

Bibliography

Written by one of the PostgreSQL developers, [MOMJIAN01]> is a gentle introduction to SQL using the PostgreSQL database system.

Bruce Momjian, Addison-Wesley, 2001, 0-201-70331-9, *PostgreSQL*.

Chapter 10. Scheme

Besides Common Lisp, Scheme is the second major Lisp dialect. It was developed by Gerald Jay Sussman and Guy Steele in 1975. Like Common Lisp, Scheme went through a standardization process which lead to an IEEE and ANSI standard in 1991. Scheme can be seen as a lightweight version of Common Lisp (although this is a little bit misleading, since Common Lisp was defined later and adopted some of Scheme's features). Scheme carries less of the historical burden which lead to some of Common Lisp's less obvious features (or quirks). The main advantages when compared to Common Lisp are a more consistent syntax and the uniform treatment of functions and other values (that is, Scheme is a one cell Lisp implementation). On the other hand, Scheme is lacking the comprehensive standard library of Common Lisp including the object oriented extension CLOS.

The current standard document is the fifth revision of the report on Scheme, or R5RS for short.

10.1. Software and Installation

For this chapter we are using MzScheme (http://www.plt-scheme.org/software/mzscheme/) on Linux, one of the PLT () implementations of Scheme. If you prefer a graphical development environment, you can equivalently use DrScheme (http://www.drscheme.org), whose Windows version comes with a convenient installer and graphical user interface.

Starting `mzscheme` takes us to the interactive shell, which we will use to explore the Scheme language.

```
ahohmann@kermit:~$ mzscheme
Welcome to MzScheme version 204, Copyright (c) 1995-2003 PLT
>
```

10.2. Quick Tour

10.2.1. Expressions

Like Lisp, Scheme is a functional language with some procedural elements. Therefore, we start our investigation with a number of expressions entered at the interactive MzScheme prompt.

```
> "Hello World"
"Hello World"
> (display "Hello World\n")
Hello World
> (+ 4 5 6)
15
```

The interactive shell evaluates the expression and displays the result (in case there is any). The first expression is a string constant which evaluates to itself. The result is shown with the surrounding double quotes indicating the string constant.

The second expression is a function call that writes our message to standard output. Scheme inherits its syntax from Lisp. An expression is either a literal or a list of expressions enclosed in parentheses. And as in Lisp, these S-expressions are evaluated by interpreting the first expression (the head of the list) as a prefix operator to be applied to the remaining expressions of the list (its tail). In the example, the built-in `display` function is applied to our message string. In contrast to the first example, the output is the result of the side effect of the function call. The result of the expression is empty and therefore not displayed.

The third example exemplifies one of the advantages of the simple but powerful S-expression syntax: many functions (such as the arithmetic operators) can be applied to an arbitrary number of arguments.

Scheme has a well designed set of numerical types including arbitary long integers, rationals, floating point numbers. We can also construct complex numbers on top of these types.

```
> (exp 1)
2.718281828459045
> (expt 2 4)
16
> (expt 2 200)
1606938044258990275541962092341162602522202993782792835301376
> (sqrt -1)
0+1i
> (+ 1/3 1/2)
5/6
> (/ 5 2)
5/2
> (quotient 5 2)
2
> (remainder 5 2)
1
> (* 1/2 -3+2i)
-3/2+1i
> (* 1.5 -3+2i)
-4.5+3.0i
```

Note how the use of the operators `-`, `+`, and `/` as part of the number literals lets us express all these different types in a natural way.

Scheme always tries to find the "best" type for the result of an expression. Dividing two integers, for example, results in a rational. Multiplying a complex integer with a rational, we obtain a complex rational number. Only if an operand is a floating point number or the expression results in an irrational number such as `exp 1`, the result is a floating point (and therefore inexact) number.

Boolean expression work as expected (once we get used to the prefix notation) with `#t` and `#f` denoting the boolean literals true and false.

```
> (and (> 2 1) (< 1.5 2.0))
#t
> (not #t)
#f
```

The Scheme standard also guarantees a number of useful string and character functions. Character literals start with `#\` followed by the character itself.

```
> (string-length "blah")
4
> (string-ref "blah" 2)
#\a
> (string=? "blah" "blub")
#f
> (string<? "blah" "Blub")
#f
> (string-ci<? "blah" "Blub")
#t
> (string->number "123")
123
> (string->number "100" 16)
256
> (string-append "blah" "blub")
"blahblub"
> (substring "blah" 1 3)
"la"
```

Two things are worth pointing out. First, the use of the question mark and the arrow `->` as part of the function name to indicate the meaning of the function: Predicates such as the string comparisons always end with a question mark, and conversion functions use the arrow to denote what is being converted. Second, the indexing of strings (and all other sequences): indexes start with zero and use the half-open interval semantics just like Python (ok, Scheme was first), that is, `(substring "blah" 1 3)` is equivalent to Python's `"blah"[1:3]`.

In case you have not guessed it already: the string comparison suffix `-ci` stands for "case insensitive".

We can bind a value to a symbol in the global environment with the `define` command.

```
> (define x 1.5)
> (* 2 x)
3.0
```

`define` is an example of a *structure* (also called "special form" like in Common Lisp) which looks like a function, but is actually implemented directly by the Scheme interpreter, because the functionality can not be expressed by a normal Scheme function.

Scheme has three different kinds of conditional expressions: `if`, `cond`, and `case`. The `if` expression chooses one of two alternatives depending on a condition, `cond` generalized this to multiple conditions and alternatives, and the `case` selects an alternative by checking if the discrimator is contained in the associated list.

```
> (define x 3)
> (if (< x 10) "small" "big")
"small"
> (cond
    ((< x 0) "negative")
    ((= x 0) "zero")
    (else "positive"))
"positive"
> (case x
    ((1 2) "small")
    ((3 4) "medium")
    (else "big"))
"medium"
```

10.2.2. Functions

Naturally, functions play a central role in a functional language, and Scheme really treats them as first class citizens. In contrast to Common Lisp, Scheme is a single-cell Lisp dialect, that is, a symbol is bound to a value which may be an expression or a function.

One way to define a function is to bind a symbol to a lambda expression (an anonymous function).

```
> (define times2 (lambda (x) (* 2 x)))
> (times2 55)
110
```

`lambda` is a special form which takes a formal parameter list and an expression and returns the associated function object.

As a shortcut, Scheme allows us to define the function directly using `define` followed by the signature (name and arguments) and expression of the function.

```
> (define (times2 x) (* 2 x))
> (times2 55)
110
```

Of course, function definitions can be recursive.

```
> (define (fac n) (if (< n 2) 1 (* n (fac (- n 1)))))
> (fac 5)
120
```

We can also define functions with variable argument lists. Normally, the arguments passed to a function are bound to the formal parameters defined in the function definition one by one. We can accept an arbitrary number of arguments by separating the last formal parameter with a period. When the function is called, the arguments which can not be bound to the "normal" formal parameters (before the period) will be bound as a list to this additional formal parameter.

```
> (define (show-args first-arg . more-args)
    (display first-arg) (newline)
    (display more-args) (newline))
> (show-args 1 2 3 4)
1
(2 3 4)
```

The `show-args` function also demonstrates that a function definition can consist of multiple expressions which will be evaluated in sequence and the result of the last expression will be returned as the result of the function.

```
> (define (blah) 1 2 3)
> (blah)
3
```

Since functions are treated just like any other value, defining higher order functions (also called "meta procedures" in Scheme) is straight forward.

```
> (define (compose f g) (lambda args (f (apply g args))))
> ((compose times2 times2) 4)
16
```

Here we use the built-in `apply` function which applies (surprise) a function to a list of arguments.

10.2.3. Collections

There are three built-in collection types: lists, vectors, and pairs. Lists we have used already throughout this chapter since they are not only a collection type, but also the base of Scheme's expression syntax. Vectors are similar to lists, but optimized for random access. Pairs combine two values and are also the building block of lists.

Pairs follow the Lisp tradition: they are constructed with the `cons` function, and can be taken apart with the `car` (head), and `cdr` (tail) function.

```
> (define p (cons "a" "b"))
> (car p)
"a"
> (cdr p)
"b"
```

As we have seen, lists literals are sequences of expressions separated by white space and enclosed in parentheses. Since they are by default evaluated as expressions, we have to quote them to obtain the lists as such.

```
> '(1 2 3)
(1 2 3)
> '("a" \#b 37)
("a" |#b| 37)
```

Internally, lists are nested pairs, or more precisely, a list is either empty (`()`) or a pair whose second element is a list. Therefore, the list syntax is just a shortcut for a nested pair structure and we can apply the pair functions to obtain the head and tail of a list.

```
> (cons 1 ())
(1)
> (cons 1 (cons 2 ()))
(1 2)
> (cons "a" '("b" "c"))
("a" "b" "c")
> (car '(1 2 3))
1
> (cdr '(1 2 3))
(2 3)
```

As a generalization, we can combine the letters `a` and `d` between `c` and `r` to retrieve the n-th element at the beginning or end of the list.

```
> (cadr '(1 2 3 4 5))
2
> (caddr '(1 2 3 4 5))
3
> (caar '((1 2 3) 4 5))
1
> (cdar '((1 2 3) 4 5))
(2 3)
> (cadar '((1 2 3) 4 5))
2
```

Besides these primitive list functions, we are a number of useful list functions in the standard.

```
> (length '(1 2 3))
3
> (reverse '(1 2 3))
(3 2 1)
```

```
> (append '(1 2) '(3 4))
(1 2 3 4)
> (list-ref '(1 2 3 4) 2)
3
> (member "a" '(1 "a" 5.5))
("a" 5.5)
> (member "b" '(1 "a" 5.5))
#f
```

Note that the `member` function returns the sublist starting with the element we look for or false if it is not contained in the list.

As an example of a higher order function, we have a look at the `map` function which applies a function to each element in a list and returns the list of the results.

```
> (map times2 '(1 2 3))
(2 4 6)
> (map (lambda (x) (* 2 x)) '(1 2 3))
(2 4 6)
```

Here we appreciate again the simple handling of functions as values. The standard does not define many higher order list functions, but most Scheme implementation come with a library containing the usual functions we have met with in the previous chapters. To use these functions with MzScheme, we first have to load the list library `list.ss`.

```
> (require (lib "list.ss"))
> (filter odd? '(1 2 3 4 5))
(1 3 5)
> (foldl + 10 '(1 2 3 4))
20
> (quicksort '(5 3 7 2 4 1 8) <)
(1 2 3 4 5 7 8)
```

Vector literals look like lists following a # sign. Their main characteristic is the efficient access to individual elements by index using the `vector-ref` and `vector-set!` functions.

```
> #(1 2 3 4)
#4(1 2 3 4)
> (define v '#(1 2 3 4))
> (vector-length v)
4
> (vector-ref v 2)
3
> (vector-set! v 2 55)
> v
#4(1 2 55 4)
```

10.3. More Features

10.3.1. Objects

Standard Scheme has no concept of data structures (or even classes) as we know it from "traditional" programming languages. The only built-in data structures are the collection types pair, list (being a special case of pair), and vector. Nonetheless, there are surprisingly simple ways to implement abstract data types in Scheme.

The starting point is the observation that each function has its own frame which contains the local variables defined in the function. We have also seen, that functions can return other functions which keep references to the frame of the original function. Combining these two elements, we can use the frame of a function as a structure holding the state of an object. The "trick" is to return a method dispatcher function.

```
> (define (make-account balance)
    (define (withdraw amount)
      (set! balance (- balance amount))
      balance)
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (dispatch method)
      (cond ((eq? method 'withdraw) withdraw)
            ((eq? method 'deposit) deposit)))
    dispatch)
> (define account (make-account 100))
> ((account 'deposit) 25)
125
> ((account 'withdraw) 50)
75
```

The initial balance is stored in the local frame of the constructor function `make-account`. Within this function we define two functions, the "methods" `withdraw` and `deposit`, which manipulate the balance. Next we define the method dispatcher function `dispatch` which just return the method function associated with the given method symbol. This method dispatcher is return as our account "object". The state of the object is contained in the frame continues to exist, since it is used by the returned function.

We can improve the method call syntax by not returning the dispatch function itself, but a function applying it to the given arguments.

```
> (define (make-account balance)
    (define (withdraw amount)
      (set! balance (- balance amount))
      balance)
    (define (deposit amount)
      (set! balance (+ balance amount))
```

```
      balance)
    (define (dispatch method)
      (cond ((eq? method 'withdraw) withdraw)
            ((eq? method 'deposit) deposit)))
    (lambda (method . args) (apply (dispatch method) args)))
> (define account (make-account 100))
> (account 'deposit 25)
125
> (account 'withdraw 50)
75
```

References

The best Scheme book [ABELSON96]> is not a book about Scheme, but an introduction to computer science using Scheme as the language of implementation. In this sense, it is the equivalent to Norvig's AI book [NORVIG92]> which is based on Common Lisp. The language reference is the R5RS (the number 5 standing for the fifth revision) which describes the complete language on less than 50 pages.

Harold Abelson, Gerald Jay Sussman, and Julie Sussman, 0-262-51087-1, MIT press, 1996, *Structure and Interpretation of Computer Programs: Second Edition*.

Jon Pearce, 0-387-98320-1, Springer-Verlag, 1998, *Programming and Meta-Programming in Scheme*.

1998, *Revised 5 Report on the Algorithmic Language Scheme*.

Chapter 11. Objective C

Objective C is a relatively small object oriented extension of C. The model for the object oriented features is Smalltalk (in contrast to C++ which combines C with Simula's classes). It was developed by Brad Cox and the StepStone corporation in the early 1980's. The biggest applications written in Objective C are probably NeXTstep/OpenStep (which now turned into Apple's Mac OS X) and WebObjects, an early and much acclaimed web application server also developed by NEXT and now owned by Apple. The developers I know who used WebObjects and Objective C keep raving about the much more elegant and powerful language when compared to C++. Enough reasons to have a closer look at Objective C.

11.1. Software and Installation

The GNU compiler `gcc` supports Objective C. It is also the one used by the Apple system. The compiler automatically recognizes the `.m` suffix of an Objective C source file. If you use the emacs editor, it will also automatically use the Object C major mode for editing `.m` files. For the examples, I'm using `gcc` version 3.0 on a Linux (Debian) system. To compile the test programs, we simply apply `gcc` to the source file and obtain the standard UNIX executable `a.out` (of course, you can also choose another name for the executable using the `-o` option). On some systems, you may need to add the libraries for Objective C and threads manually using the linker flags `-lobjc -lpthread`.

11.2. Quick Tour

11.2.1. Objects and Classes

Let's start with a simple class demonstrating the syntax added to the C language. Like C++, Objective C separates the interface definition of a class from the implementation. Here is the interface for our familiar `Person` class. Ideally, this interface is put into a separate file, e.g., `Person.h`, so that other classes can include the declaration just like any other C header file. But Objective C does not enforce any file policy and we can keep multiple interfaces and implementations in a single file.

```
#include <Object.h>

@interface Person : Object {
  const char* name;
  int age;
}
- init;
- (void)display;
- (const char*)name;
- name: (const char*)aName;
@end
```

The interface definition starts with `@interface` and ends with `@end`. All Objective C directives use keywords starting with an "at" sign `@`. Next we see the name of the class and the name of the base class separated by a colon. Here, we use the base class `Object` which is part of GNU's Objective C environment. The class names are followed by a block of variables which looks like the body of a C structure. In fact, these fields are the instance variables of the class. The remaining lines contain the decisive new features. The minus sign indicates an instance method (as opposed to a class method which starts with a plus sign). The method declarations use a syntax which combines Smalltalk's message declaration with C types. The first method is has no argument and returns the object itself (the default like in Smalltalk). The `display` method does not return anything. The next two methods are the accessor methods for the attribute `name`. The getter `(const char*)name` has no argument and returns a C string. The setter takes a C string `aName` as an argument and like `init` returns the object itself. As usual for C, the declarations are determined with a semicolon. Next, let's look at the implementation of the class.

```
@implementation Person
- init {
  name = "Homer";
  age = 55;
}
- (void)display {
  printf("name=%s, age=%d\n", name, age);
}
- (const char*)name { return name; }
- name: (const char*)aName { name = aName; }
@end
```

In analogy to the interface, the implementation is enclosed in a pair of `@implementation` and `@end` directives. All the information defined in the interface does not have to be repeated anymore (but it can, e.g., to define initial values for the instance variables). The main part of the implementation defines the methods. Again, the syntax combines Smalltalk with C. For each method, the signature copied from the interface is followed by a C block of statements. The statements can access the instance variables of the class directly.

Now that we have defined our `Person` class, let's try and use it.

```
main() {
  Person* person = [[Person alloc] init];
  [person display];
  printf("name=%s\n", [person name]);
  [person name: "Frank"];
  printf("name=%s\n", [person name]);
  [person free];
}

name=Homer, age=55
name=Homer
name=Frank
```

With our Smalltalk background the new syntax is not hard to follow. [1] Objective C allows us to use Smalltalk's message passing enclosed in square brackets anywhere in the C code. The first statement

sends the built-in `alloc` message to the `Person` class object which return a new empty instance of `Person`. We then pass the `init` message to this new object which causes our initialization code to be executed. The result is the initialized `Person` instance which can be assigned to a `Person` pointer. The next line sends the `display` message to this new instance printing name and age of the person. Next, we call the getter for the attribute `name` as part of the `printf` statement. As an example of argument passing, the new name `Frank` is passed with the message selector `name` to the person object. We then print the name again to see if it has really been changed. Finally, the memory allocated for the person object is returned to the operating system using the `free` message defined in the `Object` class.

11.2.2. Message Passing and Inheritance

The similarities to Smalltalk are not only syntactical. The semantics are very similar as well. When a message is passed to an object, the system decides at run-time which method to call. It does not need to be known at compile-time which messages an object understands. As we will see, this is in sharp contrast to the other object oriented members of the C family which rely on compile-time method dispatching.

The messaging becomes clearer if we replace the pointer type `Person*` by the generic object identifier `id`.

```
main() {
  id person = [[Person alloc] init];
  [person display];
  printf("name=%s\n", [person name]);
  [person name: "Frank"];
  printf("name=%s\n", [person name]);
  [id free];
}
```

Now, the program does not know the type of `person` at compile-time anymore, but the program behaves exactly the same. The main difference is that the first version using the explicit type `Person*` will be checked at compile-time. If we try to call a message a `Person` does not respond to, we will get a compile-time error. Using the generic `id`, the problems shows only at run-time.

Just like Smalltalk, a method can send a message to itself or its superclass using `self` and `super`, respectively. The latter is particularly useful for the initialization methods.

```
@interface Employee : Person {
  int number;
}
- (int)number;
}
@end

@implementation Employee
- init {
  [super init];
  number = 100;
```

```
}
- (void)display {
  [super display];
  printf(", number=%d", [self number]);
}
- (int)number { return number; }
@end

main() {
  id employee = [[Employee alloc] init];
  [employee display];
  [employee free];
}

name=Homer, age=55, number=100
```

11.2.3. Categories and Protocols

In Smalltalk we were able to extend an existing class by simply adding a new method to it. In Objective C, an extension of an existing class is defined using a new interface and implementation with the same class name and marking it with a so-called category. The category name follows the class name in parentheses. The next example adds a getter for the age attribute.

```
@interface Person (Accessor)
- (int)age;
@end

@implementation Person (Accessor)
- (int)age { return age; }
@end

main() {
  id employee = [[Employee alloc] init];
  printf("age=%d", [employee age]);
}
```

This category adds the getter method to the `Person` class and all its subclasses. As in Smalltalk, this feature avoids the utility classes found in less dynamic object oriented languages. Note that the new methods have access to the attributes just like the original methods, but is not possible to add new attributes to a class.

Objective C's interfaces correspond to classes in the other object oriented languages of the C family. But there is also the option to define a pure interface, that is, a collection of method signatures which other classes can implement. In Objective C this is called a protocol. As an example, we encapsulate the ability to display oneself as a `Displayable` protocol.

```
@protocol
- (void)display;
```

```
}
```

A class can list the protocols it adopts in angle brackets after the superclass.

```
@interface Person : Object <Displayable>
...
@end

main() {
  id person = [[Person alloc] init];
  printf("displayable=%d", [person conformsTo: @protocol(Displayable)]);
  [person free];
}

1
```

At run-time we can check if a given object conforms a protocol. The `@protocol` directive converts the protocol name to a protocol object which is passed with the `conformsTo` message. Protocols are not as needed in a language such as Objective C or Smalltalk, since any message can be sent to any object. However, protocols allow the Objective C compiler to do some additional type checking and thus avoid run-time errors. To do so, the protocols an object is supposed to conform to can be explicitly mentioned as part of the identifier type.

```
main() {
  id <Displayable> person = [[Person alloc] init];
  [person display];
  [person free];
}
```

In this case, the compiler makes sure that the class assigned to the `person` variable complies with the protocol (or protocols) listed in angle brackets behing the id type. It also checks if the messages passed to `person` are part of the protocol. Trying to pass the another message (such as `age`) will cause a compile-time error although the underlying `Person` class is able to respond to the message. The use of protocols thus allows for the same compile-time type checking as the use of interfaces in Java or C#.

It is not unusal in object oriented systems that an existing class has to be adapted to a new protocol defined somewhere else, for example, when interfacing with a third party library. If the original class can’t be changed, the only solution is an adapter class which delegates the new methods to the old class. In Objective C, a protocol can be added to an existing class by defining it in a category.

```
@protocol Printable
- (void)print: (FILE*)stream;
@end

@interface Person (Print) <Printable>
@end

@implementation Person (Print)
```

```
- (void)print: (FILE*)stream {
  fprintf(stream, "name=%s, age=%d", name, age);
}
@end

main() {
  id <Printable> person = [[Person alloc] init];
  [person print: stdout];
  [(Object*)person free];
}
```

In this example, we define a new protocol `Printable` which allows for printing an object on a stream. The `Person` is adopted to this protocol by adding the required method with the category `Print`. Note that we have to cast the id to an `Object` pointer in order to sent it the `free` message, since this message is not part of the `Printable` protocol.

11.3. More Features

11.3.1. Visibility

By default, Objective C adopts Smalltalk's visibility rule: attributes are private and method are public. However, it also allows to override this rule using the directives `@private`, `@protected`, and `@public`. All attributes or method following one of these directives aquire the associated visibility. Private access is retricted to the class itself (and its categories), protected access to the class and its subclasses and public access to everybody.

```
@interface Account : Object {
@private
int number;
@public
double balance;
}
- init: (int)aNumber;
@end

@implementation Account
- init: (int)aNumber {
  number = aNumber;
  balance = 0;
}
@end

main() {
  Account* account = [[Account alloc] init: 123];
  printf("balance=%f", account->balance);
}
```

A public instance variable can be accessed from outside the class just like an element of a C structure, but the object needs to be statically typed. In practice, public instance variables are hardly used, and in most cases the default visibility rule works just fine.

11.3.2. The Object Class

Every Objective C environment is shipped with a base class which provides common behavior for all objects. The GNU compiler offers the the `Object`, and the OpenStep/Mac OS X system has the `NSObject` class. Both provide a large number of methods covering memory management (alloc, free, copy), comparison (isEqual, compare, hash), and access to the underlying class and messaging. `NSObject` adds amoung other things a reference counting interface which provides a simple memory management scheme.

As an example of the meta information which is available at run-time, you can ask an object for its class or check if it belongs to a class or one of its subclasses.

```
main() {
  id employee = [[Employee alloc] init];
  printf("class=%s\n", [[employee class] name]);
  printf("is kind of Person=%d\n",
         [employee isKindOfClassNamed: "Person"]);
  printf("is member of Person=%d\n",
         [employee isMemberOfClassNamed: "Person"]);
}
```

```
class=Employee
is kind of Person=1
is member of Person=0
```

Since `Employee` is derived from `Person`, the "is kind of" relationship is satisfied, but the `employee` is not a member of the `Person` class.

Since the class information is available at run-time, it is also possible to vary the message itself at run-time, that is, to determine the message selector dynamically.

```
main() {
  id person = [[Person alloc] init];
  SEL selector = @selector(display);
  [person perform: selector];
  [person free];
}
```

The `@selector` directive gives us the selector object for a selector name, and the `perform:` method calls the messaging mechanism directly with this selector.

11.3.3. Arrays

Object oriented systems thrive on their libraries, and Objective C is no exception. The combination of static types on the one hand and dynamic features on the other (including extending existing classes) provide a good foundation for powerful libraries. The NeXTStep/OpenStep/OS-X system as *the* example for an Objective C framework provides classes covering everything from basic collections and operating system function to graphical user interfaces. We can only give a first glimpse here using the GNU's GNUStep implementation.

Since GNUStep is not part of the standard `gcc` installation, we have to make sure that the libraries are known to the compiler and linker. The easiest way to accomplish this is to use GNUStep's make system. Here is the `GNUmakefile` for the example below.

```
GNUSTEP_INSTALLATION_DIR = $(GNUSTEP_SYSTEM_ROOT)
GNUSTEP_MAKEFILES = $(GNUSTEP_SYSTEM_ROOT)/Makefiles

include $(GNUSTEP_MAKEFILES)/common.make

# The tools to be compiled
TEST_TOOL_NAME = sample

# The Objective-C source files to be compiled
sample_OBJC_FILES = sample.m

SRCS = $(TEST_TOOL_NAME:=.m)
HDRS =
DIST_FILES = $(SRCS) $(HDRS) Makefile
include $(GNUSTEP_MAKEFILES)/test-tool.make
```

Besides demonstrating GNUStep's array objects, the following example also introduces a number of important GNUStep features such as memory management with "auto release pools" and string constants.

```
#include <Foundation/Foundation.h>
#include <Foundation/NSString.h>
#include <Foundation/NSArray.h>

static int compare(id a, id b, void* context) {
  return [a compare: b];
}

int main() {
  CREATE_AUTORELEASE_POOL(pool);
  NSString* name = @"Homer";
  NSArray* a = [NSArray arrayWithObjects: @"one", @"two", nil];

  NSLog(@"a=%@\n", a);

  NSObject* obj;
  NSEnumerator* enumerator = [a objectEnumerator];
```

```
  while ((obj=[enumerator nextObject])) {
    NSLog (@"Next Object is: %@", obj);
  }

  NSMutableArray* ma = [NSMutableArray array];
  [ma addObjectsFromArray: a];
  [ma addObject:  @"three"];
  NSLog(@"ma=%@\n", ma);
  [ma sortUsingFunction: compare context: nil];
  NSLog(@"ma=%@\n", ma);

  DESTROY(pool);
}
```

11.4. Discussion

Objective C is a surprisingly elegant combination of the two very different languages C and Smalltalk. It does not take away any of the low level complexities of C (pointers, memory management, etc.), but sets the foundation for large object oriented systems without sacrificing the benefits such as speed and simplicity. Some unique features such as the compile time type checking combined with the ability to add protocols to existing classes (bearing some resemblance to Haskell's type classes) would definitely help the other C-based object-oriented languages.

Notes

1. But the deviation from the typical C function call syntax is probably one of the reasons that Objective C did not get as popular as C++.

Chapter 12. ML

ML is a functional programming language developed in the late 1970's at Edinburgh University. It is a strongly typed (in contrast to Lisp) functional language with type inference as one of its main features. Its first standard incarnation was published in 1990 (ML90), and a new version in 1997 (ML97). The major implementations now all support ML97 which we will use for the presentation.

12.1. Software and Installation

For the introduction we are using the Moscow ML (http://www.dina.dk/~sestoft/mosml.html) implementation of Standard ML (SML) including the little shell program mml from CarrotSoft (http://www.carrotsoft.com/win/ml.html). To install the program, unzip both, start the mml program, and drag the installation directory of Moscow ML from a Windows Explorer into the mml window. Starting the mml shell, you get the following welcome message.

```
MosML Windows Interface v1.1 by Andrew Pontzen
Bug reports to andrew@carrotsoft.com or app26@cam.ac.uk
Latest version from www.carrotsoft.com

Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
-
```

You are now ready to enter the wonderful world of Standard ML on the command line. The dash "-" is the prompt, and the response of the interpreter will be prefixed with a ">" symbol.

12.2. Quick Tour

12.2.1. Expressions

Yes, we will start with our favorite message again by entering "Hello World" and finishing the expression with a semicolon.

```
- "Hello World";
> val it = "Hello World" : string
```

This time, we get a much more sophisticated answer. The line tells us that the symbol `it` has been bound to the value "Hello World", and that this value is of type `string`. The symbol `it` always represents the result of the last expression evaluated in the interactive shell. As usual, let's try simple arithmetic next.

```
- 4 + 5*6;
> val it = 34 : int
```

```
- 5.0 / 2.5 + 3.0;
> val it = 5.0 : real
```

At least we have found a functional language which can handle arithmetic expressions with infix operators including the right preferences. The next example shows a less convenient aspect of ML's type system.

```
- 2.5 + 1;
! Toplevel input:
! 2.5 + 1;
!       ^
! Type clash: expression of type
!   int
! cannot have type
!   real
```

It looks like ML is very strict about types, since it does not even coerce an integer into a real number. Instead, we have to tell the explicitly what we want using one of the built-in conversion functions between integers and floating point numbers.

```
- real(1)
> val it = 1.0 : real
- floor(2.5)
> val it = 2 : int
- 2.5 + real(1);
> val it = 3.5 : real
```

Another unusual syntactical element is the tilde ~ as the unary minus operator instead of the simple minus.

```
- ~1.5 - 2.0;
> val it = ~3.5 : real
```

12.2.2. Functions

Since ML is all about functions, defining them is literally "fun".

```
- fun f(x) = 2*x;
> val f = fn : int -> int
```

The function is defined by writing down the mathematical definition after the `fun` keyword. As a result, ML tells us that we have defined a function taking a single integer argument and returning another integer. But how does ML determine that `x` has to be an integer without us declaring any type in the function definition? This is of the characteristic features of ML. It tries to deduct types from the context. Since the literal "2" is an integer and the multiplication takes two arguments of the same numeric type, the argument `x` has to be an integer and so does the result of the function. Note that the function is just

another value, just of type `fn : int -> int`. We could have defined the function as well by assigning an anonymous function to the symbol `f` as follows:

```
- val f = fn(x) => 2*x;
> val f = fn : int -> int
```

The anonymous function starts with the keyword `fn` (which plays the role of `lambda` in other languages) followed by the parameter list, and arrow `=>` and the expression returned by the function.

We expect a functional language to be able to deal with functions as easily as with any other value. As an example, it should be easy to define the composition of two functions.

```
- fun compose(f, g) = fn x => f(g(x));
> val ('a, 'b, 'c) compose = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

This looks tricky. First, how did we define the function `compose`? The result of applying `compose` to the functions `f` and `g` is an anonymous function which maps an argument `x` to `f(g(x))` as required. The more difficult part is understanding ML's response. The first thing ML can infer is that `f` and `g` have to be functions. Otherwise, `g` could not be applied to `x`, and `g` could not be applied to `g(x)`. Moreover, the function `f` must be applicable to the return type of `g`. But that's all. We know neither the argument type of `g` nor the return type of `f`. How does ML display this information? A symbol preceded by a quote `'` denotes a type variable. The answer to our definition tells us that compose is a function depending on three types `'a`, `'b`, and `'c`. [1] The function can be applied to two function arguments with the constraint that the return type `'a` of the second function is identical to the argument type of the first function. The multiplication of types on the left hand side can be interpreted as the mathematical cross product of the two sets of functions. Alternatively, we could have defined the compose function using the basic value syntax:

```
- val compose = fn (f, g) => fn x => f(g(x));
> val ('a, 'b, 'c) compose = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

And finally, ML also supports the more readible derived form:

```
- fun compose(f, g) x = f(g(x));
> val ('a, 'b, 'c) compose = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Before moving on, we should test if the compose function does what it is supposed to do.

```
- fun g(x) = x + 5;
> val g = fn : int -> int
- fun f(x) = 2 * x;
> val f = fn : int -> int
- val h = compose(f, g);
> val h = fn : int -> int
- h(3);
> val it = 16 : int
```

And, by the way, we could have saved all this work, since the composition is already predefined as the compose operator "o":

```
- (f o g)(3);
> val it = 16 : int
```

As another typical feature for a functional language, let's try recursion. This is also a good opportunity to introduce another ML specialty: pattern matching. ML functions can be defined in a piecemeal fashion with the alternatives separated using a vertical bar "|".

```
- fun fac(0) = 1
    | fac(n) = n * fac(n-1);
> val fac = fn : int -> int
- fac(5);
> val it = 120 : int
```

ML will try to match a given argument with the pattern on the left hand side of each definition in the order in which they appear and evaluate the right hand side of the matching rule. Using the basic syntax, we have to tell the ML system explicitly with the keyword "rec" that we are about to define a recursive function.

```
- val rec fac = fn 0 => 1
                 | n => n * fac(n-1);
> val fac = fn : int -> int
```

12.2.3. Collections

For now we have seen that it is easy to define complex functions in ML and that ML has a strong type system which gives us compile time type checking and probably also good performance. In fact, the ML dialect Ocaml which we will tackle in the next chapter is fighting with plain old C for the performance crown in the great computer language shootout (http://www.bagley.org/~doug/shootout/). What we have not seen yet are more complex data types. Because of the math look and feel, we expect at least built-in tuples, and indeed here they are.

```
- val t = (1, "bla");
> val t = (1, "bla") : int * string
- #1(t);
> val it = 1 : int
- #2(t);
> val it = "bla" : string
```

Knowing Python, we possibly would have preferred a simple index notation to access the element of a tuple (starting with index zero), but ML instead defines the special functions #n which fetch the n'th element of the tuple. This will become more natural as we see that tuple are just a convenient shortcut notation for a record.

```
- val joe = {name="joe", age=33};
```

```
> val joe = {age = 33, name = "joe"} : {age : int, name : string}
- #name(joe);
> val it = "joe" : string
- #age(joe);
> val it = 33 : int
- val t = {1=1, 2="bla"};
> val t = (1, "bla") : int * string
```

Besides tuples and records, we also expect sequences to be part of the standard types in a functional language. ML supports two kinds of built-in sequence types: *lists* are optimized for sequential access (similar to Lisp's lists or Java's LinkedList), and *Vectors* allow for fast random access. We will first treat lists, and then have a quick look at vectors. List literals look like Python lists.

```
- val l = [1, 2, 3];
> val l = [1, 2, 3] : int list
- val l = ["Joe", "John", "Mary"];
> val l = ["Joe", "John", "Mary"] : string list
- val l = [(1, "Joe"), (2, "John")];
> val l = [(1, "Joe"), (2, "John")] : (int * string) list
```

As you can tell from the interpreters type notifications, ML's strong typing requires the elements in a list to be of the same type.

```
- val l = [1, "Joe"];
! Toplevel input:
! val l = [1, "Joe"];
!             ^^^^^
! Type clash: expression of type
!   string
! cannot have type
!   int
```

Since ML's lists are implemented as linked structures, they share many of Lisp's list functions, although with a more expressive syntax. You get the head element and the tail of the list using the functions "hd" and "tl" (corresponding to car and cdr in Lisp).

```
- val l = [1, 2, 3];
> val l = [1, 2, 3] : int list
- hd(l);
> val it = 1 : int
- tl(l);
> val it = [2, 3] : int list
```

You can also concatenate lists using the "@" operator or prepend an element in front of a list using a double colon (corresponding to Lisp's cons function) which complements the head and tail functions.

```
- [1, 2, 3] @ [4, 5, 6];
> val it = [1, 2, 3, 4, 5, 6] : int list
- 1 :: [2, 3, 4];
> val it = [1, 2, 3, 4] : int list
```

The head and tail functions are hardly needed in ML, since we can exploit ML's pattern matching and match a list with the pattern "x::xs", that is, it head and tail. The variable names in the pattern "x::xs" are a typical naming convention and should be read "ex" and "exes". To demonstrate that these apparently simple operations turn out to be powerful when combined with recursion, we first define a function which reverses a list.

```
- fun reverse([]) = []
    | reverse(x::xs) = reverse(xs) @ [x];
> val 'a reverse = fn : 'a list -> 'a list
- reverse l;
> val it = [3, 2, 1] : int list
```

The first match takes care of the empty list. A non-empty list we can always split into head and tail, and reversing is equivalent to appending the head at the reversed tail. Inserting an element into a sorted list is not much more difficult than this:

```
- fun insert(x, [])   = [x]
    | insert(x, h::t) = if x <= h
                        then x::h::t
                        else h::insert(x, t);
> val insert = fn : int * int list -> int list
```

Here, the non-trivial second case compares the element to be inserted with the head of the list. If the new element is smaller or equal, we can put it in front of the list. Otherwise, we need to insert the element into the tail. Once we can insert, we can sort a list (although not very efficiently).

```
- fun sort []    = []
    | sort(h::t) = insert(h, sort t);
> val sort = fn : int list -> int list
- sort([2, 5, 1, 3, 7, 2]);
> val it = [1, 2, 2, 3, 5, 7] : int list
```

ML also has the typical higher order list processiong functions map and reduce.

```
- map (fn x => 2*x) [1, 2, 3];
> val it = [2, 4, 6] : int list
```

Note that in contrast to Python, map takes only one argument (the function applied to all the elements in the list) and returns the function which modifies any list. You can see this more clearly, when checking the types of the partial expressions.

```
- map;
> val ('a, 'b) it = fn : ('a -> 'b) -> 'a list -> 'b list
- map (fn x => 2*x);
> val it = fn : int list -> int list
```

In functional jargon this is called a curried function (after Haskell B. Curry who also gave the purely functional language Haskell its name). When defining a function, we can eiter pass the arguments as a

tuple (uncurried form) or pass just the first argument and return a function taking the second argument and so forth (curried form).

ML's equivalent of Python's reduce function is called foldr (for "fold right").

```
- foldr (fn (x, y) => x + y) 5 [1, 2, 3];
> val it = 11 : int
```

The first argument is the two argument function used to combine the values in the list, the second (curried) argument the initial value, and the last argument the list to reduce. Formally, foldr's type is:

```
- foldr;
> val ('a, 'b) it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Let's turn to ML's random access sequence, the vector. Vectors literals are similar to lists, but start with a hash sign "#".

```
- val v = #[1, 2, 3];
> val v = #[1, 2, 3] : int vector
```

Again, we see that the collection is strongly typed. Access to vector elements or parts of vectors is index oriented.

```
- open Vector;
- sub(v, 0);
> val it = 1 : int
- extract(v, 1, SOME 2);
> val it = #[2, 3] : int vector
```

The function `sub` retrieves a single element, the function `extract` a slice of the vector. Indexes start at zero and the optional upper bound of the slice is inclusive.

The concat function concatenates a list of vectors into one big vector and the map and reduce (foldr) functions work like their list counterparts.

```
- concat([#[1, 2], #[3, 4]]);
> val it = #[1, 2, 3, 4] : int vector
- map (fn x => 2*x) #[1, 2, 3];
> val it = #[2, 4, 6] : int vector
- foldr (fn (x, y) => x+y) 5 #[1, 2, 3];
> val it = 11 : int
- app (fn x => print("x=" ^ Int.toString(x) ^ "\n")) #[1, 2, 3];
x=1
x=2
x=3
> val it = () : unit
```

You can also apply these functions to slices of a vector. In this case, the function has access to the iteration index as well.

```
- fun printElem(i, x) = (
    print(Int.toString(i)); print(": ");
    print(Int.toString(x)); print("\n"));
> val printElem = fn : int * int -> unit
- appi printElem (#[1, 2, 3, 4, 5], 1, SOME 3);
1: 2
2: 3
3: 4
> val it = () : unit
```

12.2.4. Data Types

Up to now we have defined a lot of values including functions. Even the records were defined directly as values. The type of these value was always inferred by the language without our declaring it. But how can we define types ourselves? The simplest way is to define a shortcut for an existing type using the "type" command. Here is the definition of the type associated with the simple record structure introduced above.

```
- type person = {name: string, age: int};
> type person = {age : int, name : string}
```

The type definition just binds a symbol to a type expression. We have seen the type expressions all along in the responses of the ML system. How can we use this type? Sometimes, we have to help ML to determine the type of a function. We do this by appending a type declaration (Pascal style) to a function argument.

```
- fun name(p : person) = #name(p);
> val name = fn : {age : int, name : string} -> string
- name(joe);
> val it = "joe" : string
```

In this example, ML is not able to derive the type of the function "name". We have to give ML a hint using an explicit type declarations. The definition is equivalent to using the explicit type expression; it only saves us some typing when using the type more than once.

```
- fun name(p: {name: string, age: int}) = #name(p);
> val name = fn : {age : int, name : string} -> string
```

Combining built-in types in type expressions is useful, but it does not allow us to create new types. As an example, think about an enumeration type representing colors or the type of a tree structure. These types require alternatives. A color is either red or green or blue (or some other color from a finite set of colors we want to deal with). A tree is either empty or a node whose branches are again tree structures. We can’t express those types by combining existing types in type expressions. That’s where ML’s so-called data types come in. They generalize the concepts of enumerations and variants (like in Pascal or C’s

unions). An data type is defined as a sequence of alternatives, each alternative being some type. A value belonging to the new data type belongs to one of the sub types. To tell the alternatives apart, one has to put a so-called constructor in front of the value of the sub type. Let's look at a few examples. The first one defines a new data type called "color" consisting of three alternatives.

```
- datatype color = Red | Green | Blue ;
> New type names: =color
  datatype color =
  (color,{con Blue : color, con Green : color, con Red : color})
  con Blue = Blue : color
  con Green = Green : color
  con Red = Red : color
- val c = Red;
> val c = Red : color
```

This is the one extreme of a data type corresponding to an enumeration. The sub types are empty or, better, they only contain the single value "nothing". Still, there are three different kinds of nothing (red, green, blue) and we can define color values by using the associated three constructors Red, Green, and Blue. The constructors can also be used in patterns:

```
- val colorName = fn
      Red   => "red"
    | Green => "green"
    | Blue  => "blue" ;
> val colorName = fn : color -> string
- colorName c ;
> val it = "red" : string
```

The next example demonstrates another extreme: a data type with just a single sub type (which we can't call alternative anymore).

```
- datatype person = Person of string * int ;
> New type names: =person
  datatype person = (person,{con Person : string * int -> person})
  con Person = fn : string * int -> person
- val joe = Person("Joe", 25);
> val joe = Person("Joe", 25) : person
- fun name(Person(name, age)) = name;
> val name = fn : person -> string
- name(joe);
> val it = "Joe" : string
```

The new data type "person" has a single constructor "Person" taking two arguments, a string and an integer (representing the person's name and age, but this is not clear from the definition of the data type). The sub type (here the tuples consisting of a string and an integer) follows the constructor and the keyword "of". Defining new values of type person now really looks like a constructor call. The name function uses pattern matching again, albeit in a simple form since we have to alternative constructors.

The next example demonstrates a data type used in the sense of a variant. Suppose we would like to store integers and strings in a list. We can't do this directly like in the dynamically typed languages. But we can define a data type which contains strings and integers as sub types.

```
- datatype stringOrInt = I of int | S of string;
> New type names: =stringOrInt
  datatype stringOrInt =
  (stringOrInt, { con I : int -> stringOrInt,
                  con S : string -> stringOrInt})
  con I = fn : int -> stringOrInt
  con S = fn : string -> stringOrInt
- val l = [I(1), S("blah"), I(55)];
> val l = [I 1, S "blah", I 55] : stringOrInt list
- map (fn I(i) => Int.toString(i) | S(s) => s) l;
> val it = ["1", "blah", "55"] : string list
```

Data types can also be parametrized using type variables. We can generalize the previous example to construct lists containing one of two possible types.

```
- datatype ('a, 'b) alt = A of 'a | B of 'b;
> New type names: =alt
  datatype ('a, 'b) alt =
  (('a, 'b) alt,
   {con ('a, 'b) A : 'a -> ('a, 'b) alt,
    con ('a, 'b) B : 'b -> ('a, 'b) alt})
  con ('a, 'b) A = fn : 'a -> ('a, 'b) alt
  con ('a, 'b) B = fn : 'b -> ('a, 'b) alt
- [A(1.2), B(4), A(2.34), B(5)];
> val it = [A 1.2, B 4, A 2.34, B 5] : (real, int) alt list
```

If there is one element of ML which convinced me immediately, it is the handling of optional results of functions combined with pattern matching. Think about a lookup function which either returns the value you look for or indicates in some way that it could not find any value. How can we implement this result in the mainstream languages? Either we return a null pointer (or reference) in case the value was not found, or we throw an exception. Both solutions have their pros and cons. Null pointers are too often not checked leading to nasty crashes (or not much better runtime exceptions). Exceptions should be "exceptional" and cause clumsy code when used for normal program logic. ML has a built-in data type called "option" which is a perfect fit for a return value of a lookup function. We define it (but we don't have to since it is a standard data type) as

```
- datatype 'a option = NONE | SOME of 'a;
> New type names: =option
  datatype 'a option =
  ('a option,{con 'a NONE : 'a option,
               con 'a SOME : 'a -> 'a option})
  con 'a NONE = NONE : 'a option
  con 'a SOME = fn : 'a -> 'a option
```

A value of type "int option", for example, is either the value NONE or the value SOME x, where x is some integer. The List.find function return values of type "'a option" where 'a is the type of the list elements.

```
- List.find;
> val 'a it = fn : ('a -> bool) -> 'a list -> 'a option
- List.find (fn x => x > 10) [1, 3, 15, 3, 17];
> val it = SOME 15 : int option
- List.find (fn x => x > 10) [1, 3];
> val it = NONE : int option
```

To handle this type, we have to use pattern matching. There is no way to silently ignore a null pointer or exception.

```
- fun g(NONE)   = print("nothing found")
    | g(SOME x) = print("found " ^ Int.toString(x));
> val g = fn : int option -> unit
- g(SOME 5);
found 5
> val it = () : unit
- g(NONE);
nothing found
> val it = () : unit
- g(List.find (fn x => x > 10) [1, 3, 15, 3, 17]);
found 15
> val it = () : unit
```

Defining this kind of function is as easy as handling the option return value. The pattern matching in the calling function correponds to an if expression in the called function.

```
- fun f(x) = if x<0 then NONE else SOME x;
> val f = fn : int -> int option
- g(f(10));
found 10
> val it = () : unit
- g(f(0));
found 0
> val it = () : unit
```

The last example of a data type has to occur in any description of ML: a binary tree fined as a recursive, parametrized data type. Sounds complicated? But it fits in one line (for clarity we use three).

```
- datatype 'a btree =
      Empty
    | Node of 'a * 'a btree * 'a btree;
> New type names: =btree
  datatype 'a btree =
  ('a btree,
   {con 'a Empty : 'a btree,
    con 'a Node : 'a * 'a btree * 'a btree -> 'a btree})
  con 'a Empty = Empty : 'a btree
```

```
  con 'a Node = fn : 'a * 'a btree * 'a btree -> 'a btree
```

A binary tree is either empty, or it is a node consisting of a value of type `'a`, a left branch, and a right branch. Both branches are binary trees of type `'a` again. Here are some examples:

```
- val p = Node(100, Empty, Empty);
> val p = Node(100, Empty, Empty) : int btree
- val q = Node(200, Empty, Empty);
> val q = Node(200, Empty, Empty) : int btree
- val r = Node(150, p, q);
> val r = Node(150, Node(100, Empty, Empty), Node(200, Empty, Empty)) :
  int btree
```

To make our binary tree more useful, let us define lookup and insertion (the slightly more complicated deletion you find in ML book such as [ULLMAN98]>).

```
- val rec lookup = fn
      (x, Empty) => false
    | (x, Node(y, left, right)) =>
         if      x < y then lookup(x, left)
         else if y < x then lookup(x, right)
         else true;
> val lookup = fn : int * int btree -> bool
- lookup(200, r);
> val it = true : bool
- lookup(5, r);
> val it = false : bool
```

Note that the algorithm works only for sorted trees, and in fact only for integer trees, since we use the "less than" directly (we'll fix this later when talking about structures).

```
- val rec insert = fn
      (Empty, x) => Node(x, Empty, Empty)
    | (T as Node(y, left, right), x) =>
         if      x < y then Node(y, insert(left, x), right)
         else if y < x then Node(y, left, insert(right, x))
         else T;
> val insert = fn : int btree * int -> int btree
- insert(r, 400);
> val it =
    Node(150, Node(100, Empty, Empty),
         Node(200, Empty, Node(400, Empty, Empty))) : int btree
- insert(r, 120);
> val it =
    Node(150, Node(100, Empty, Node(120, Empty, Empty)),
         Node(200, Empty, Empty)) : int btree
```

12.2.5. Module System

When discussing Python, we complained about the lack of well-defined interfaces caused by the dynamic typing. With ML, we have a very strong type system. Where are the interfaces? ML includes a so-called module system consisting of structures, signatures, and functors. A structure is a combination of types and functions, a signature is a "type" of a structure, and functors are functions mapping structures to structures. In other words, we go from the ordinary level of values, types, and function to the meta level of structures, signatures, and functors.

Since the examples become a little bit longer in this section, you are better off entering them in a file which you can then load with the `use` command as if you had typed all the code on the command line. If the file containing the code is `c:\sml\sample.sml`, type:

```
- use "c:\sml\sample.sml";
[opening file "c:\sml\sample.sml"]
...
[closing file "c:\sml\sample.sml"]
```

At first sight, structures provide a namespace for types and values (including functions). We have already used some standard structures such as `Int` and `List`. As an example of our own, let us wrap the `color` type together with the `colorName` function into a structure.

```
- structure Color = struct
    datatype color = Red | Green | Blue;
    val name = fn
        Red   => "red"
      | Green => "green"
      | Blue  => "blue";
  end;
> ...
- Color.name(Color.Red);
> val it = "red" : string
```

Since all identifiers defined in the structure have to be qualified with the name of the structure, we can keep the names short and crisp. One of the most popular examples is a stack.

```
structure Stack = struct
  exception Empty;
  type 'a stack = 'a list;
  val create = [];
  fun push(s, x) = x::s;
  fun pop([])   = raise Empty
    | pop(x::s) = (s, x);
end;
```

We implement the stack using a list where `push` is equivalent the basic list construction operator, and `pop` takes the head of the list (unless it is empty). Since the implementation is functional, the `pop` function returns the popped element as well as the new stack (the tail of the origianal one).

```
- val s = Stack.create;
> val 'a s = [] : 'a list
- val s = Stack.push(s, 55);
> val s = [55] : int list
- val s = Stack.push(s, 66);
> val s = [66, 55] : int list
- Stack.pop(s);
> val it = ([55], 66) : int list * int
```

When using the stack, we can see the internal data structure. We can even directly pass a list to the structure's function although the list was created by the other functions of the structure.

```
- Stack.pop(["blah", "blub"]);
> val it = (["blub"], "blah") : string list * string
```

Besides the namespace, a structure does not offer us any benefit such as encapsulation or information hiding. How shall we define an interface for the structure which hides the internals of the implementation? All a client needs to use the structure in a strongly typed environment such as ML is the type information of the components in our structure. In ML, this is the provided by a signature.

```
signature STACK = sig
  exception Empty;
  type 'a stack;
  val create : 'a stack;
  val push : 'a stack * 'a -> 'a stack;
  val pop : 'a stack -> 'a stack * 'a;
end;
```

A signature is a structure lifted to the type level. The two keywords `signature` and `sig` correspond to the ones used to define a structure, `structure` and `struct`. The shorter name is used to define the object, the longer one to bind a symbol to this object.

We can tell the compiler that a structure implements a given signature by adding a so-called signature constraint to the structure. To this end, we place the signature behind the structure's name using a colon (like a superclass in C++).

```
structure Stack : STACK = struct
  exception Empty;
  type 'a stack = 'a list;
  val create = [];
  fun push(s, x) = x::s;
  fun pop([])   = raise Empty
    | pop(x::s) = (s, x);
end;
```

The relationship between structures and signatures is comparable to the relationship between classes and interface in objected-oriented languages. The signature describes the interface of all the structures which

implement the signature. Now the compiler makes sure that the structure is compliant with the signature. Trying to compile the following definition of our stack causes a signature mismatch

```
structure Stack : STACK = struct
  exception Empty;
  type 'a stack = 'a list;
  val create = [];
  fun push(s, x) = x::s;
end;
...
! Signature mismatch: the module does not match the signature ...
! Missing declaration: value pop
! is specified in the signature as
!   val 'a pop : 'a list -> 'a list * 'a
! but not declared in the module
```

Because this kind of signature constraint does not hide the implementation of the stack, it is called a transparent signature constraint.

```
> val it = () : unit
- val s = Stack.create;
> val 'a s = [] : 'a list
- Stack.pop([1, 2, 3]);
> val it = ([2, 3], 1) : int list * int
```

Fortunately, ML97 introduces an opaque signature constraint which hides the data types of the structure. To use this kind of constraint we only have to replace the colon by the symbol `:>`.

```
structure Stack :> STACK = struct
  exception Empty;
  type 'a stack = 'a list;
  val create = [];
  fun push(s, x) = x::s;
  fun pop([])   = raise Empty
    | pop(x::s) = (s, x);
end;

- val s = Stack.create;
> val 'a s = <stack> : 'a stack/2
- val s = Stack.push(Stack.push(s, 55), 66);
> val s = <stack> : int stack/2
- val (s, x) = Stack.pop(s);
> val s = <stack> : int stack/2
  val x = 66 : int
- Stack.pop([1, 2, 3]);
! Toplevel input:
! Stack.pop([1, 2, 3]);
!            ^^^^^^^^
! Type clash: expression of type
!   'a list
! cannot have type
```

```
!   'b stack/2
```

ML's first means of encapsulation is the concept of an abstract data type. It combines a datatype with functions while hiding the actual definition of the datatype itself.

```
abstype 'a stack = S of 'a list with
  exception Empty;
  val create = S [];
  fun push(S s, x) = S (x::s);
  fun pop(S [])   = raise Empty
    | pop(S(x::s)) = (S s, x);
end;
```

The first part of the definition until the `with` keyword is a datatype definition with the `datatype` keyword replaced by `abstype`. The essential declaration follows between the `with` and `end` keywords. From the outside, the type can only be used through the components defined in this section. Note that we had to use a datatype `S of 'a list` although `'a list` is already a well defined type.

```
- val s = create;
> val 'a s = <stack> : 'a stack
- val s = push(s, 55);
> val s = <stack> : int stack
- pop(s);
> val it = (<stack>, 55) : int stack * int
- val s = push(s, 66);
> val s = <stack> : int stack
- pop(s);
> val it = (<stack>, 66) : int stack * int
- val (s, x) = pop(s);
> val s = <stack> : int stack
  val x = 66 : int
- val (s, x) = pop(s);
> val s = <stack> : int stack
  val x = 55 : int
- val (s, x) = pop(s);
! Uncaught exception:
! Empty
```

The datatype constructor `S` can only be used within the declaration part of the abstract data type. It is not visible from the outside.

```
- val s = S [];
! Toplevel input:
! val s = S [];
!         ^
! Unbound value identifier: S
```

Now we have approached encapsulation and data hiding from two angles, structures and abstract data types, the former providing namespaces and the latter hiding a datatype.

To hide the internals of an implementation, we have to start with a signature. The relationship between structures and signatures is comparable to the relationship between classes and interface in objected-oriented languages. The signature describes the interface of all the structures which implement the signature. The interface contains all the information a client needs to use the structure and hides the rest. In a strongly typed language such as ML, a client needs to know the names and types of all the values defined in the structure (values again including functions). It also may need to know the names of types and exceptions required for the values. As an example, consider a functional interface (i.e., a signature) of a stack. To keep it simple, we start with a stack of integers.

```
signature IntStack = sig
  exception Empty;
  type stack;
  val create : stack;
  val push : stack * int -> stack;
  val pop : stack -> stack * int;
end;
```

A signature starts with the keyword `sig` and ends with the keyword `end`. It can be bound to a symbol using the keyword `signature`. Exceptions and types are just declared with their names. For values we also give their types in the notation we know from ML's answers. Let's turn to the implementation of this interface, a structure providing all the types and values declared in the signature.

```
structure ListIntStack : IntStack = struct
  exception Empty;
  type stack = int list;
  val create = [];
  fun push(s, x) = x::s;
  fun pop([])   = raise Empty
    | pop(x::s) = (s, x);
end;
```

The structure `ListIntStack` implements the `IntStack` with a simple list. The compiler makes sure that the structure really implements all the required types and values. If not, we get a signature mismatch error. We can use the stack operations with any structure implementing the signature.

```
- val s = ListIntStack.create;
> val s = [] : int list
- val s = ListIntStack.push(s, 55);
> val s = [55] : int list
- val s = ListIntStack.push(s, 66);
> val s = [66, 55] : int list
- val (s, x) = ListIntStack.pop(s);
> val s = [55] : int list
  val x = 66 : int
- val (s, x) = ListIntStack.pop([2, 3]);
> val s = [3] : int list
  val x = 2 : int
```

However, this is not exactly what we wanted. We can still see the internal structure of the stack, and, even worse, we can apply the functions to values defined outside like in the last `pop` call.

With ML97, opaque signatures were introduced which hide all the internals. A structure implementing a signature in this manner uses the operator `:>` instead of the simple colon.

```
structure IntStack :> IntStack = struct
  exception Empty;
  type stack = int list;
  val create = [];
  fun push(s, x) = x::s;
  fun pop([])   = raise Empty
    | pop(x::s) = (s, x);
end;
```

We also change the name of the structure to the name of the signature (which is not required, but uncommon either). When now using the structure, we don't see anymore that it is implemented with a list.

```
- val s = IntStack.create;
> val s = <stack> : stack/3
- val s = IntStack.push(s, 55);
> val s = <stack> : stack/3
- val (s, x) = IntStack.pop(s);
> val s = <stack> : stack/3
  val x = 55 : int
- val (s, x) = IntStack.pop([2, 3]);
! Toplevel input:
! val (s, x) = IntStack.pop([2, 3]);
!                            ^^^^^
! Type clash: expression of type
!   'a list
! cannot have type
!   stack/3
```

And consequently, we can not apply the `IntStack` functions to integer lists directly.

Next we would like to overcome the restriction of our stack to integers. As a first step, we generalize the signature by introducing a type for the elements contained in the stack.

```
signature Stack = sig
  exception Empty;
  type elem;
  type stack;
  val create : stack;
  val push : stack * elem -> stack;
  val pop : stack -> stack * elem;
end;
```

Now we need some kind of parametrization similar to C++ templates. ML's solution is the third element of the module system: functors. As mentioned at the beginning of this section, a functor maps a structure to another structure. In other words, is allows us to parametrize a structure with another one.

```
signature Type = sig type elem end;

functor MakeStack(T: Type) : Stack =
  struct
    exception Empty;
    type elem = T.elem;
    type stack = T.elem list;
    val create = [];
    fun push(s : stack, x) = x::s;
    fun pop([])   = raise Empty
      | pop(x::s) = (s, x);
  end;
```

Now we can apply the functor to a structure just like we would apply a function to its arguments. The result is a new structure, in our case a stack for elements of a given type.

```
- structure RealStack = MakeStack(struct type elem = real end);
> ...
- val s = RealStack.create;
> val s = [] : real list
- val s = RealStack.push(s, 1.5);
> val s = [1.5] : real list
```

You may wonder why we have not used an opaque signature here to hide the list implementation. The problem is that attaching `Stack` as an opaque signature also hide the `elem` type with the result that we can't push any element on the stack. To get out of this dilemma, we need to declare the signature as part of the functor definition.

```
functor MakeStack(T: Type) :>
  sig
    exception Empty;
    type stack;
    val create : stack;
    val push : stack * T.elem -> stack;
    val pop : stack -> stack * T.elem;
  end
=
  struct
    exception Empty;
    type stack = T.elem list;
    val create = [];
    fun push(s : stack, x) = x::s;
    fun pop([])   = raise Empty
      | pop(x::s) = (s, x);
  end;
```

This way we an create stacks for arbitrary types with hidden implementation.

```
- structure RealStack = MakeStack(struct type elem = real end);
> ...
- val s = RealStack.create;
```

```
> val s = <stack> : stack/5
- val s = RealStack.push(s, 1.5);
> val s = <stack> : stack/5
- RealStack.pop(s);
> val it = (<stack>, 1.5) : stack/5 * real
```

12.2.6. Procedural Features

I hope you agree by now that ML is a powerful language for functions and expressions, but can we also do some "normal" processing such as file I/O? Yes, we can. Like Lisp, ML is not a pure functional language. There are objects with state, mutable arrays, and even an equivalent to conventional variables which can change their value.

Let’s start with references. Up to now we have worked without variables. The `var` statements bind symbols to value, but the value can’t be changed. Using the same symbol again only means that we create a new binding with a new scope. Introducing variables so late in the game clearly indicates that they are overrated, but there are situations where they make sense. In ML, variables are implemented by binding a symbol to a reference with some additional syntax to read and set the value of the reference.

```
- val i = ref 0;
> val i = ref 0 : int ref
- !i;
> val it = 0 : int
- i := 5;
> val it = () : unit
- !i;
> val it = 5 : int
```

We create a reference pointing to some initial value with the `ref` function. The exclamation mark gives us the current value of the reference, and the assignment operator `:=` (also used by the Pascal family) allows us to change the value of the reference. Setting the value is a pure non-functional operation. It returns nothing, but does all its work as a side-effect.

While a reference contains just a single value, an array is a mutable version of a vector. Applications for this kind of data structure are obviously numerical computations on vectors and matrices, because they call for an efficient in-place implementation.

```
- open Array;
> ...
- val a = array(5, 0.0);
> val a = <array> : real array
- length(a);
> val it = 5 : int
- sub(a, 3);
> val it = 0.0 : real
- update(a, 0, 1.5);
```

```
> val it = () : unit
- sub(a, 0);
> val it = 1.5 : real
```

All array functions are contained in the `Array` module which we open first. We create an array with the `array` function supplying the array's length and an initial value. This value determines the type of the array. The functions `sub` and `update` fetch and set an element in the array, respectively.

There are multiple functions in the `Array` module which apply a function to all elements in an array. We can, for example, print the array using a combination of `app` and `print`.

```
- app (fn x => print (Real.toString(x) ^ " ")) a;
1.5 0.0 0.0 0.0 0.0 > val it = () : unit
```

We can also modify the array (or parts of it) with the `modify` function.

```
- modify (fn x => 2.0 * x) a;
> val it = () : unit
- app (fn x => print (Real.toString(x) ^ " ")) a;
3.0 0.0 0.0 0.0 0.0 > val it = () : unit
```

Again, the `update` and `modify` functions are completely non-functional and therefore should be handled with care.

Now that we have references at our disposal, other typical elements of procedural languages start making sense. The `while` loop executes an expression while a condition is satisfied. Such a loop make sense only if the result of the condition can be changed through some side-effect. Here is a classical procedural loop using a reference for the loop variable.

```
- val i = ref 0;
> val i = ref 0 : int ref
- while !i < 10 do (
    print(Int.toString(!i) ^ " ");
    i := !i + 1);
0 1 2 3 4 5 6 7 8 9 > val it = () : unit
```

This really looks like procedural code. While ML focuses on functional programming, it allows for procedural code as well in order to solve real-life problems.

The following example reads just like procedural code.

```
- load "TextIO";
> val it = () : unit
- val out = TextIO.openOut("test.txt");
> val out = <outstream> : outstream
- TextIO.output(out, "this is Joe\n");
> val it = () : unit
```

```
- TextIO.output(out, "this is John\n");
> val it = () : unit
- TextIO.closeOut(out);
> val it = () : unit
- val f = TextIO.openIn("test.txt");
> val f = <instream> : instream
- while not(TextIO.endOfStream(f)) do
    print(TextIO.inputLine(f));
this is Joe
this is John
> val it = () : unit
```

This is also the first time we use a so-called structure. Structures in ML correspond to modules in Python (or Modula). To use a structure we have to load it (unless we define it ourselves, but this is a different story covered later) and then precede the function calls with the module qualifier just like in Python. In the example, we use the standard structure TextIO which offers text-oriented input and output streams. These streams have state and can be used like streams in a procedural language. We also notice the procedural control statement, the while loop. Another way to implement the same iteration through the lines in a file uses a loop function.

```
- fun loop(hasNext, next, f) =
    while (hasNext(f)) do next(f);
> val ('a, 'b) loop = fn : ('a -> bool) * ('a -> 'b) * 'a -> unit
- val f = TextIO.openIn("test.txt");
> val f = <instream> : instream
- loop(not o TextIO.endOfStream, print o TextIO.inputLine, f);
this is Joe
this is John
> val it = () : unit
```

12.3. More Features

12.3.1. Collections

A more interesting example (copied from [ULLMAN98]>) is the following implementation of merge sort. The algorithm is implemented with three functions: split, merge, and mergeSort. The split function just splits a list into two halfs returned as a pair of lists.

```
- fun split(nil) = (nil, nil)
    | split([a]) = ([a], nil)
    | split(a::b::cs) =
        let val (M,N) = split(cs) in (a::M, b::N) end;
> val 'a split = fn : 'a list -> 'a list * 'a list
```

The first two lines take care of the trivial cases of an empty list and a list containing a single element. If the list has at least two elements, we split what's left behind the first two elements and prepend these element to the two resulting lists. As you can tell, this is harder to describe in plain english (at least for me) than in the two lines of ML code. Next we need the merge function which merges two ordered lists so that the result is again an ordered list.

```
- fun merge(nil, M) = M
    | merge(L, nil) = L
    | merge(L as x::xs, M as y::ys) =
        if x<y then x::merge(xs, M) else y::merge(L, ys);
> val merge = fn : int list * int list -> int list
```

The new feature here is the "as" expression in the last pattern. It lets us refer to the matching values in two different ways, as lists and split into head and tail, just as needed for the merge function.

```
- fun mergeSort(nil) = nil
    | mergeSort([a]) = [a]
    | mergeSort(L) =
        let val (M, N) = split(L) in
          merge(mergeSort(M), mergeSort(N))
        end;
> val mergeSort = fn : int list -> int list
- val l = [3, 4, 2, 7, 1];
> val l = [3, 4, 2, 7, 1] : int list
- mergeSort(l);
> val it = [1, 2, 3, 4, 7] : int list
```

After this preparation, the mergeSort function is not surprising anymore. ML also has the typical higher order list processiong functions map and reduce.

```
- map (fn x => 2*x) [1, 2, 3];
> val it = [2, 4, 6] : int list
```

12.3.2. Exceptions

References

Jeffrey D. Ullman, 0-13-790387-1, Prentice-Hall, 1998, *Elements of ML Programming, ML97 Edition*.

Stephen Gilmore, The University of Edinburgh, 2003, *Programming in Standard ML '97: A Tutorial Introduction*.

Andreas Rossberg and Jens Olsson, Universitaet des Saarlandes, , *Standard ML vs. Objective Caml: http://www.ps.unie-sb.de/~rossberg/SMLvsOcaml.html*.

Notes

1. According to [GILMORE03]>, the prime is supposed to indicate a greek letter, that is, `'a` stands for the greek alpha.

Chapter 13. C++

C++ was created by Bjarne Stroustrup in 1983 as an extension of C with the object oriented concepts of Simula-67. The first commercial compiler (a preprocessor generating C code) was released in 1985. During the the following years, many features were added or improved. The stardardization started in 1989 and the first official ISO/ANSI C++ standard was published in 1998.

In contrast to Objective C, C++ is a big extension of C. It almost doubles the number of keywords with the implied increase in complexity (probably growing exponentially with the number of keywords). C++ not only adds object oriented features to C, but also a number of unrelated extensions such as references, type parameters (templates), operator overloading, default arguments, and so forth. To further complicate our presentation, even a small program uses almost all these features together (at least as soon as you use the standard library). This makes it hard to explain our examples without getting lost in the details. This tendency is not restricted to our presentation, but is something every C++ developers has to be aware of.

13.1. Software and Installation

As for C and Objective C, we use the GNU compiler `gcc`, version 3.3, for the examples in this chapter.

13.2. Quick Tour

13.2.1. Hello World

C++ started as a C preprocessor (the `cfront` compile), but by now has become an independent language. And although it is still possible to use plain C code and C's standard libraries, C++ now offers its own libraries for the most important extensions such as input/output, strings, and collections (see Section 13.2.5>). We therefore start all over again with some simple examples.

```
#include <iostream>

using namespace std;

int main() {
  cout << "Hello World" << endl;
  return 0;
}
```

The code still resembles the original in C, but also shows many differences. First, the standard C++ header files don't have a suffix (since the standard commitee could not agree on one). Second, the namespace directive tells the compiler that we want to use the standard library without explicit qualifiers.

C++ solves (as a late addition) C's problem of name clashes between different libraries by introducing namespaces. Without the `using` directive, we would need to put a `std::` qualifier in front of all symbols defined in the `std` namespace, in our case the symbols `cout` and `endl` from the input/output library.

```
#include <iostream>

int main() {
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

As another observation, C++ seems to be stricter about the return type of the `main` function. We have to properly define `main` as an integer function and must not forget the return statement. However, the most striking line is the print statement which does not even remotely resemble the print statements we have seen so far.

```
cout << "Hello World" << endl;
```

It reminds more of a UNIX shell command, although the arrows point in the other direction. The analogy is not so far fetched, since the shift operator `<<` indicates that objects are pushed into the standard output stream `cout`. The last object `endl` is the newline. C++'s relies heavily on operator overloading to implement type-safe and efficient input and output streams. In its standard incarnation, streams are even template types depending on the underlying character implementation and its stream related properties (or traits). In other words, to understand the simple print statement to its full extend, we have to understand most of the complex C++ features in the first place.

13.2.2. Some Differences between C and C++

Before diving into the heart of C++, let us mention a few minor differences between C and C++. Besides the C-style comment `/* ... */`, C++ also ignores everything starting with two slashes `//` until the end of the line. This is the preferred style of comments in C++. Moreover, variables can be declared anywhere in the code, not just at the beginning of a block. As a useful applications of this rule, a loop variable can be declared as part of a `for` statement.

```
for (int i=0; i<n; i++) {
  ...
}
```

The scope of the variable is just the `for` statement, it is not visible outside (like a number of other things, this has changed during the evolution of C++).

Handling strings in an efficient and safe manner is a non-trivial task in C. You have to consider pointers, buffer lengths, and memory management (who is responsible for the deletion of a string allocated on the heap?). Therefore, the (late) addition of a standard string implementation to C++ was a most welcome and, compared to other languages, long overdue improvement. Since we will use them in the examples

below, we introduce strings here, although, just like the standard input/output streams, their implementation can't be fully understood without most of the other C++ features.

```
int main() {
  string name = "Homer";
  cout << "Hello, " << name << endl;
  cout << "How are you, " + name << endl;
  string ho = name.substr(0, 2).append(", ");
  cout << ho << ho << ho << endl;
  return 0;
}
```

```
Hello, Homer
How are you, Homer
Ho, Ho, Ho,
```

The most interesting line

```
string ho = name.substr(0, 2).append(", ");
```

takes the first two characters as a substring and appends the separator string `", "`. Besides these simple string operations, the standard implementation contains everything from access to individual characters to complex search methods.

In C, arguments are always passed by value. If we want a function to change a variable which is defined outside of the function, we have to pass the pointer (i.e., the address) of the variable to the function. The pointer is again passed by value.

```
static void count(int* counter) {
  *counter += 1;
}

void main() {
  int counter = 0;
  count(&counter);
}
```

C++ introduces the option to pass argument by reference. Semantically, this is equivalent to passing the pointer, but the syntax differs.

```
static void count(int& counter) {
  counter += 1;
}

int main() {
  int counter = 0;
  count(counter);
}
```

The actual advantage over the pointer notation is debatable, but references are used consistently throughout the C++ standard library. To avoid expensive copying of argument objects, most read-only arguments are passed as a `const` reference. In this case, the reference notation is more readable since it replaces the pass-by-value for performance reasons only.

```
static void show(const string& s) {
  cout << s;
}
```

13.2.3. Classes

Objects and classes are C++'s first and formost addition to C. A C++ class is basically a C structure with methods. C++ supports inheritance (including full multiple inheritance) and fine-grained visibility control. Like Objective C, C++ separates the declaration of a class from its implementation. Here is the declaration of the `Person` class (typically to be found in a header file called `Person.h`).

```
class Person {
  string _name;
  int _age;
public:
  Person(const string& name, int age);

  const string& name() const;
  void name(const string& name);

  void printOn(ostream& out) const;
};
```

Apart from the many `const` keywords, this declaration does not contain any surprises. First, we define the two attributes `_name` and `age`. The default visibility in a class is private, so that these attributes will be visible inside of the class only. All the methods are defined with public visibility as indicated by the `public:` directive. Similar to Objective C, a visibility directive is valid until overridden by a new one. The first method is a constructor. It is named after the class itself and has no return value. As we will see, constructors in C++ are initialization methods which are called automatically once the memory for an object has been reserved.

The next two methods are the accessor methods for the `name` attribute. There are about as many naming conventions for C++ as there are developers. The one we follow here uses the same name for the getter and setter. This demonstrates C++ ability to use a method name multiple times in a class as long as the method signatures differ. However, we can not use the same name for the attribute itself. Hence, we follow the convention to begin attribute names with an underscore character. The last method is supposed to print a `Person` object on an output stream.

Now it is time to explain the many `const` keywords. As explained above, most C++ APIs use constant references as an efficient replacement of passing by value. The `const` keyword behind the `name()` getter and the print method indicates that calling these methods does not change the person object. Both

these meanings of `const` play together. The compiler ensures that only constant methods are called for a constant object reference. As an example, the name setter obtains a constant reference to a string object (the new name). Inside of this method, we can call constant string methods of this string. Calling a destructive (non-const) method such as `append` will cause compile error.

After this lengthy explanation let's look at the implementation of the class. C++, each method is implemented individually by taking the method signature, qualifying method name with the class name, and following it with the method body. [1]

```
Person::Person(const string& name, int age) :
  _name(name), _age(age) {}

const string& Person::name() const { return _name; }

void Person::name(const string& name) {
  _name = name;
}

void Person::printOn(ostream& out) const {
  out << "name=" << _name << ", age=" << _age;
}
```

The only striking definition is the constructor which uses initializers for the two attributes. These initializers are put between the signature and the constructor body. When a person object is constructed, the attributes are directly initialized with the associated constructor calls. The alternative assignment in the constructor body

```
Person::Person(const string& name, int age) {
  _name = name;
  _age = age;
}
```

first initialized the name string with the default constructor and then assigns the actual name. Besides being more efficient, initializers may be the only option in scenarios where the attribute's class does not support default constructor or assignment operator.

It looks like we are eventually ready to use the new class.

```
int main() {
  Person person("Homer", 55);
  person.printOn(cout);
  return 0;
}

name=Homer, age=55
```

C++ leaves the developer many choices. One of these choices is the memory management. Like C's structures, instances of C++ classes can be allocated on the stack or the head. We have already used stack

based strings in the examples above. Here, we create a `Person` object on the stack by passing the constructor arguments to the `person` variable as if it were a function. Behind the scenes, C++ reserves a block of memory for the object on the stack and calls the constructor whose signature matches the passed arguments. The string initialization

```
string s = "blah";
```

is actually equivalent to

```
string s("blah");
```

and therefore calls the string constructor which takes a plain C-string (`const char*`).

The heap-based version of the problem looks more familiar in front of the background of the previous chapters.

```
int main() {
  Person* person = new Person("Homer", 55);
  person->printOn(cout);
  delete person;
  return 0;
}
```

The allocation is indicated by the new operator and the initialization by the constructor call. This is complemented by the `delete` operator which first calls the destructor (which we have not covered yet) before returning the allocated memory to the operating system. We will not get into the details here, but the new and delete operators can be changed to implement different allocation policies for particular classes or in general.

13.2.4. Templates

13.2.5. Collections

For a long time, C++ had no standard collection library. Instead, one had to rely on either the compiler's collection classes (e.g., as contained in Microsoft's MFC library) or third party libraries. There are two mainly two ways to implement collections in C++. On the one hand, there is the Smalltalk model based on a common base class for all elements in a collection and a hierarchy of classes modelling the different kinds of collections. On the other hand is the template model based on parametrized functions and classes. Both approaches have their virtues. The Smalltalk model provides clean interfaces, a simpler implementation, and smaller executables. As a trade-off, all objects in a collection must be heap based

and the application code requires a lot of casting which takes away some of the compile-time type safety of C++. The template approach is truely type safe, but requires a lot more effort on the implementation side. Also, the code bloat associated with the excessive use of templates causes long compile times and large executables. In the end, the template model became part of the ANSI C++ standard with the standard template library, or STL for short.

The STL applies templates in a radical way. Everything is a template. The collections themselves, their iterators, and the functions acting on them. Although going against object oriented design, these three elements are treated as separate entities. To start with, let's see what a standard iteration through a container looks like.

```
#include <iostream>
#include <vector>
#include <iterator>

using namespace std;

int main() {
  vector<int> v;

  for (int i=0; i<5; i++) v.push_back(i);

  for (vector<int>::iterator i=v.begin(); i!=v.end(); ++i) {
    cout << *i << ' ';
  }
}
```

We construct a vector, fill it with the five integers from zero to four, and print it by iterating through the container using STL's standard iterator syntax. To understand the STL iterator model we need to recollect the pointer-based loop through an array.

```
int main() {
  const int n = 5;
  int v[n];
  for (int i=0; i<n; i++) v[i] = i;

  int* begin = &v[0];
  int* end = begin + n;

  for (int* i=begin; i!=end; ++i) {
    cout << *i << ' ';
  }
}
```

The similarities are intentional. The standard template library was defined to accomodate the most efficient implementation, that is, pointer arithmetic. We can easily design take the last example and cast it into a class definition that complies with the STL algorithms.

```
class IntArray {
  int _n;
```

```
  int* _v;
public:
  IntArray(int n) : _n(n), _v(new int[n]) {}
  ~IntArray() { delete[] _v; }

  typedef int* iterator;

  iterator begin() { return _v; }
  iterator end() { return _v + _n; }

  int& operator[](int i) { return _v[i]; }
};
```

The standard STL loop looks exactly the same as for the built-in vector collection.

```
IntArray v(5);
for (int i=0; i<5; ++i) v[i] = i;
for (IntArray::iterator i=v.begin(); i!=v.end(); ++i) {
  cout << *i << ' ';
}
```

13.3. More Features

13.3.1. Smart Pointers

In contrast to most "modern" languages covered in this book (including good old Lisp), C++ leaves memory management to the developer. However, it is easier to hide the complexity of memory management in C++. We have already seen the standard implementations of strings and collections that manage the memory of the underlying data structures transparently.

13.3.2. Metaprogramming

When discussing C we have seen how to extend the C language using textual substitution with macros. In principle, C++ templates provide the same thing in a more structured, type-safe fashion. But as it turns out, the C++ template mechanism can be used to extend the C++ language in surprising ways.

The basic idea behind template metaprogramming is to use templates with non-type parameters as functions which are evaluated at compile time. Here is a simple example (see Gurtovoy>) computing the factorial function at compile time.

```
#include <iostream>
```

```
template<unsigned n>
struct factorial {
  static const unsigned value = n * factorial<n-1>::value;
};

template<>
struct factorial<0> {
  static const unsigned value = 1;
};

int main() {
  factorial<5> fac5;
  std::cout << fac5.value << std::endl;
  return 0;
}

--> 120
```

The "template function" or "metafunction" `factorial` is defined recursively using the `unsigned` parameter `n`. The constant class attribute `value` plays the role of the return value. The important part to notice is that the recursion is evaluated at compile time when instantiating the template `factorial<5>`. After this example, it is not surprising that C++ templates can theoretically solve any computable problem (that is, they are turing-complete).

In contrast to metaprogramming in the Lisp-like languages, template metaprogramming in C++ is totally different from the normal (non-template, procedural or object-oriented) programming in C++. In fact, it looks more like functional programming in ML using pattern matching and recursion.

13.4. Discussion

C++ is a complex beast and takes years to master, but when used properly, it offers all the power to create object-oriented systems with ultimate performance. As a example, most of today's desktop applications (including, e.g., Microsoft's office suite) are written in C++.

Bibliography

The definite guide to C++ is not surprisingly [STROUSTRUP00]> written by the inventor of the language. Besides the documentation of the language features it reveals the reasoning behind it. Scott Meyers' provides a lot of insights which a C++ programmer otherwise has to learn through painful experience. The documentation of the boost (www.boost.org) library contains a good introduction to template metaprogramming.

Bjarne Stroustrup, 0201700735, Addison-Wesley, 2000, *The C++ Programming Language, Special Edition*.

[Meyers97] Scott Meyers, 0201924889, Addison-Wesley, 1997, *Effective C++, 2nd edition: 50 Specific Ways to Improve Your Programs and Designs*.

[Meyers95] Scott Meyers, 020163371X, Addison-Wesley, 1995, *More Effective C++: 35 New Ways To Improve Your Programs and Designs*.

[Eckel03] Bruce Eckel, 0139798099, Prentice Hall, 2002, *Thinking in C++, 2nd edition*.

[Gurtovoy] Aleksey Gurtovoy and David Abrahams, http://www.boost.org/libs/mpl/doc/paper/html/index.html , *The Boost C++ Metaprogramming Library*.

Notes

1. As you can see, this individual implementation of each method requires more typing and reduces the readability when compared to treating the implementation as a block (like Objective C's `@implementation/@end`).

Chapter 14. Eiffel

Eiffel, [1] the clean, strongly typed, truely object-oriented programming language - so its supporters say. Eiffel was designed by Bertrand Meyer in 1985 and first sold as a commercial product by Eiffel Software (http://www.eiffel.com) (part of his company ISE - Interactive Software Engineering) in 1986. Its unique characteristic is "Design by Contract" (DBC), the ability to define semantics conditions of methods in the language itself, but the language contains many other interesting ideas.

14.1. Software and Installation

The Eiffel Software (http://www.eiffel.com) offers the development environment EiffelStudio for free for non-commercial use. However, preferring open source tools, I've used SmartEiffel (http://smarteiffel.loria.fr) (formerly known as SmallEiffel) on Linux. This is sufficient for our small examples which do not require a full blown IDE, but you should be able to run the examples in EiffelStudio as well.

SmartEiffel is part of most Linux distributions (Debian in my case), and the installation does not pose any problem. SmartEiffel first precompiles the Eiffel code to C, and then calls the C compiler to create the executable. This is done transparently for the Eiffel developer. All you have to do is set the environment variable `SmallEiffel` to the location of the SmallEiffel library (in my case `/usr/lib/smalleiffel`) and call `se-compile`.

14.2. Quick Tour

14.2.1. Hello World

Eiffel is designed to build large, reliable, object-oriented systems so that even our small greeting requires the scaffolding of a fully fledged class. To run the program, enter the code in a file called `hello.e` (same base name as the class, but lowercase), compile it with `co-compile -o hello hello.e`, and start the resulting executable `hello`.

```
class HELLO
create make
feature
   make is
      do
         print("Hello World%N")
      end
end
```

We have to grasp a number of concepts before understanding this program. First, Eiffel, like most object-oriented languages, talks about a *system* rather than a program. A system is a collection of classes (similar to a Smalltalk image although the latter is more a collection of objects with classes being special objects). To tell Eiffel where to start, we normally have to define a root class. Since our system contains only the `HELLO` class, this is not necessary.

Eiffel calls members of a class (attributes and methods) *features*. Classes mainly consist of feature definitions introduced with the keyword `feature`. In the example, we define a single method called `make` which prints the message.

This does not explain yet, how the method gets executed. When starting a system, Eiffel creates an instance of the root class, and that's where the `create` (or synonymously `creation`) statement comes in. It tells the compiler that the `make` method is a creation procedure (in other languages called "constructor") with no arguments which must be called when instantiating an instance of the class. Hence, when starting our "hello" application, Eiffel creates an instance of the root class `HELLO` and calls the constructor method `make` which prints the message.

Calling the constructor method `make` is just a convention. Any other name is syntactically just as fine. Note that, as another style convention, Eiffel always uses underscore characters to separate the parts of multi-word identifiers. Features and variables are always lowercase, classes uppercase, and constants start with an uppercase letter.

14.2.2. Variables, Arithmetic, and Control Statements

Local variables of a method are declared in advance in the optional `local` section of a method. Eiffel being an explicitly typed language lets us specify the type of each variable using a colon and the name of the type.

```
class ARITHMETIC
creation make
feature
   make is
      local
         i: INTEGER
         x: DOUBLE
      do
         i := 50
         x := 1.5 + 3 * 2.0^3 + i
         print("x=" + x.to_string + "%N")
      end
end
```

In the example, we use the two build-in types `INTEGER` and `DOUBLE` which correspond to C's `int` and `double`, respectively. The variables are all initialized automatically to a default value corresponding to their type. For numerical types, this is zero.

As for the variable declaration, Eiffel follows the Pascal syntax for the assignment operator (I still remember how unintuitive C's use of equal operator for assignment appeared to me when moving from Pascal/Modula to C). Arithmetical expression work as expected including the correct preferences, automatic conversion from integer to double, and the power operator `^`.

There are a few details in the print statement which we have not seen before. First, the statement prints three strings which are concatenated with the `+` operator demonstrating Eiffel's ability to use operators not just for numerical types. Second, we convert the floating point number `x` to a string using the `to_string` feature. Eiffel uses, like many other object-oriented languages, the dot notation to refer to features of objects. For methods without parameters, we can omit the empty parameter list so that the call looks just like the access to an attribute. Using parentheses in this case will result in a compiler warning.

The statements are not ended or separated by any special character. You only need to use a semicolon if you try and put multiple statements in a single line. Here is the packed version of the program above.

```
class ARITHMETIC_PACKED creation make
feature
   make is
      local i: INTEGER; x: DOUBLE
      do
         i := 50; x := 1.5 + 3 * 2.0^3 + i
         print("x=" + x.to_string + "%N")
      end
end
```

Eiffel also has a boolean type and supports the usual boolean operators (using their proper names, not C's symbols).

```
class ARITHMETIC
creation make
feature
   make is
      local
         x: DOUBLE
         b: BOOLEAN
      do
         x := 100
         b := x > 10 or x /= 50 and not ("blub" <= "blah")
         print("b=" + b.to_string + "%N")
      end
end
```

Here, `/=` is obviously not the devide-and-update operator used in the C family, but the unequal sign. Also note that the parentheses around the string comparison are required, because the `not` binds stronger than the comparison operators.

Next to arithmetic, we have usually covered functions, which, in a purely object-oriented language such as Eiffel, means a method returning a value.

```
class FUNCTION_EXAMPLE
creation make
feature
   make is
      do
         print("result=" + times_square(2, 3).to_string + "%N")
      end

   times_square(x: DOUBLE; i: INTEGER): DOUBLE is
      do
         Result := i * x
         Result := Result * Result
      end
end
```

Parameter and return types are specified like the types of local variables. The semicolon separating the two arguments is only needed if they are defined on the same line. The return value is defined using the implicit variable `Result`. As you can see, it can be used like any other variable. When the function is left, the value of this variable is returned to the calling routine.

Do not try to assign a value to a formal parameter of a method. In contrast to the C family, Eiffel does not allow this (mostly confusing) practice.

Eiffel restricts itself to a relatively small set of control statements, the usual if-then-else, a case statement, and a loop instruction that corresponds semantically to C's `for` loop. In all three cases, the syntax is straight forward.

```
class IF_EXAMPLE
creation
   make
feature
   make is
      do
         compare(4, 5)
      end

   compare(x: INTEGER; y: INTEGER) is
      do
         if x < y then
            print("less")
         elseif x = y then
            print("equal")
         else
            print("greater")
         end
      end
end
```

The case statement is called `inspect` in Eiffel, but otherwise works as expected. The inspected expression (and thus all the expressions it is checked agains) must be an integer or a character.

```
class INSPECT_EXAMPLE
creation
   make
feature
   make is
      do
         print("2=" + to_string(2))
      end

   to_string(x: INTEGER) : STRING is
      do
         inspect x
         when 1 then Result := "one"
         when 2 then Result := "two"
         when 3 then Result := "three"
         else Result := "another"
         end
      end
end
```

The `loop` instruction can be viewed as a readable version of C's `for` statement. You define initialization instructions, an exit condition, and the body of the loop which is executed until the exit condition becomes true. In Section 14.2.4> we will cover the possibility to add invariants and variants to loops as part of the Design by Contract.

```
class LOOP_EXAMPLE
creation
   make
feature
   make is
      local
         i: INTEGER
      do
         from
            i := 1
         until
            i > 3
         loop
            print("i=" + i.to_string + "%N")
            i := i + 1
         end
      end
end
```

14.2.3. Classes and Features

We had to define classes from the very beginning (even for our "Hello World" program), but for now we have used them merely to set the context for functions doing the rest. Let us now define our standard class example, a person, in Eiffel.

```
class PERSON
creation
   make
feature
   make(a_name: STRING, an_age: INTEGER) is
      do
         name := a_name
         age := an_age
      end

   to_string: STRING is
      do
         Result := "name=" + name + ", age=" + age.to_string
      end;
feature
   name: STRING
   age: INTEGER
end
```

The only new element are the two attributes `name` and `age`. They can be used inside the class like any other variable. Since we have not restricted the access to these features, they are also readable from any other class. However, you can not set an attribute from the outside, since this could cause an inconsistent state of the object. Eiffel does not know the concept of public read-write attributes (which is not good style in the languages supporting it). If you want other classes to be able to change an attribute, you have to define a setter method for it.

```
   set_name(a_name: STRING) is
      do
         name := a_name
      end
```

Also note that Eiffel does not allow us to use the same name for two different features of a class even if they have different signatures. In Eiffel, a feature name is always unique. If we would like two different constructor methods, we have to give them different names.

Having defined the `PERSON` class, we would like to create person objects. Using Eiffel, objects spring into life with a bang (actually two).

```
class TEST creation make
feature
   make is
      local
         person: PERSON;
```

```
      do
         !!person.make("Homer", 55)
         print("person=" + person.to_string + "%N")
         print("name=" + person.name + "%N")
      end
end
```

If you prefer a more readable syntax, you can also use the keyword `create` instead.

```
         create person.make("Homer", 55)
```

The variable `person` is a reference to an object of the class `PERSON`. At the beginning of the method, this reference is set to `Void`. We can easily verify this in the code using an assertion:

```
         check person = Void end
```

The predefined constant `Void` corresponds to a null pointer just like Pascal's `nil` or Java's `null`. In contrast to most other object-oriented languages, we do not create an object (for example, using some factory method such a `new`) and assign it to a variable. Instead, Eiffel performs both in one step with the `create` or "bang bang" instruction applied to the variable.

Next, let's again derive an employee class that adds an employee number to a person.

```
class EMPLOYEE inherit
   PERSON
      rename make as person_make
      redefine to_string
      end
creation
   make
feature
   make(a_name: STRING; an_age, a_number: INTEGER) is
      do
         person_make(a_name, an_age)
         number := a_number
      end

   to_string: STRING is
      do
         Result := Precursor + ", number=" + number.to_string
      end;
feature
   number: INTEGER
end
```

The first striking element is the extensive declaration of the inheritance in the `inherit` clause. Since we would like to define a new constructor taking the employee number as an addition argument, we have to hide the original `make` feature of the `PERSON` class by renaming it to `person_make` (remember that feature names must be unique).

We would also like to provide a new implementation of the `to_string` method so that the employee number gets printed as well. To avoid simple mistakes such as an incorrect spelling of the redefined method, we must state our intention explicitly using the `redefine` instruction.

Once we have prepared the class in this manner, the implementation is straight-forward. In the new constructor `make` we can call the old one using its new name `person_make`. Similarly, the special name `Precursor` refers to the implementation of the current method in the parent class. This is used in the redefinition of the `to_string` method to add the employee number to the string provided by the `PERSON` class.

All strongly typed object-oriented languages let us declare abstract methods, that is, methods which rely on subclasses to provide an implementation. In Eiffel, we do not talk about abstract methods, but *deferred* features. Here is an example defining the interface for an account with the minimal balance, deposit, withdraw functionality.

```
deferred class ACCOUNT feature
   balance: DOUBLE is
      deferred
      end
   deposit(amount: DOUBLE) is
      deferred
      end
   withdraw(amount: DOUBLE) is
      deferred
      end
end
```

Replacing the `do` block by the keyword `deferred` makes the features deferred. A class with at least one deferred feature is a deferred class and has to be marked as such.

Features without arguments can be implemented as methods or as attributes (that’s why the more general term "deferred feature" makes sense). Here is probably the simplest implementation of the account interface.

```
class SIMPLE_ACCOUNT inherit
   ACCOUNT
      redefine balance, deposit, withdraw end
feature
   balance: DOUBLE

   deposit(amount: DOUBLE) is
      do
         balance := balance + amount
      end

   withdraw(amount: DOUBLE) is
      do
         balance := balance - amount
      end
```

```
end
```

Implementing a deferred class works just like inheriting from any other class. In the `redefine` clause, we tell the compiler which features we are going to implement. In this implementation of the account, the `balance` feature is implemented as an attribute.

Of course, deferred classes can not be instantiated. But now that we have an implementation, we can use the class in a test program.

```
class TEST creation make
feature
   make is
      local
         account: ACCOUNT
      do
         !SIMPLE_ACCOUNT!account

         account.deposit(10.0)
         account.withdraw(5.0)
         print("balance=" + account.balance.to_string + "%N")
      end
end
```

If you did not like the "bang bang" syntax for object creation, you won't like special case for derived classes either. The concrete class to be instantiated is put between the two quotation marks of the object creation instruction.

14.2.4. Design by Contract

As mentioned in the introduction, Design by Contract sets Eiffel apart from other languages. When we look at a library, we want to know how to call a function and what a function does. The first question is answered by the function's signature. It tells us which parameters the function expects, and, in strongly typed languages, which type the supplied arguments must have. The semantics of the function, however, are normally described in comments only.

Eiffel goes one step further by giving us the means to specify some semantic information in the code. We can define semantic conditions on the input (preconditions), output (postconditions), and state of the object (invariants). Here is an example:

```
class ACCOUNT
create make
feature
   make(a_minimal_balance: DOUBLE; initial_balance: DOUBLE) is
      require
         consistent_balance: a_minimal_balance <= initial_balance
      do
```

```
         minimal_balance := a_minimal_balance
         balance := initial_balance
      ensure
         balance_set: balance = initial_balance
         minimal_balance_set: minimal_balance = a_minimal_balance
      end

   deposit(amount: DOUBLE) is
      require
         positive_amount: amount > 0
      do
         balance := balance + amount
      ensure
         balance_updated: balance = old balance + amount
      end

   withdraw(amount: DOUBLE) is
      require
         positive_amount: amount > 0
         enough_money: balance - amount >= minimal_balance
      do
         balance := balance - amount
      ensure
         balance_updated: balance = old balance - amount
      end

feature -- attributes
   minimal_balance: DOUBLE
   balance: DOUBLE

invariant
   balance_ok: balance >= minimal_balance
end
```

The example defines an account with a constructor and the two methods `deposit` and `withdraw`. Here is a test program using the account class.

```
class TEST
create make
feature
   make is
      local
         account: ACCOUNT
      do
         !!account.make(-1000, 0)
         account.deposit(50)
         account.withdraw(150)
         print("balance=" + account.balance.to_string + "%N")
      end
end
```

The interesting part is obviously not the minimal implementation of the method, but way the class and its method are adorned with conditions which ensure that the class works as expected. The constructor takes two arguments, a minimal and an initial balance. These arguments only make sense if the initial balance is not less than the minimal balance. Hence, we define a *precondition* in the `require` section of the method which checks exactly that. The purpose of the constructor is to set the attributes to the given values. This result expected by the client calling the method is verified using a *postcondition* in the `ensure` section of the method. Similarly, we define pre- and postconditions for the two other methods. The precondition makes sure that we never get below the minimal balance, and the postcondition check that the balance has been updated correctly. The nice syntactical `old` feature lets us refer to the value of the balance before the method is executed.

Finally, there is the `invariant` section of the class which lets us define conditions which have to be fulfilled by an object of the class at any time. In our case, we make sure that the balance never gets below the minimal balance.

These three elements, preconditions, postconditions, and invariants are at the heart of Eiffel's *design by contract*. I hope that even this simple example gives you an idea how much semantic information can be captured with these language constructs.

Eiffel also lets us add additional checks to the program flow. The simplest one is the `check` instruction which can be placed anywhere to check a condition (like C's `assert`).

```
class CHECK_EXAMPLE
creation make
feature
   make is
      local
         n: INTEGER
      do
         n := 5
         check
            is_five: n = 5
            is_positive: n > 0
         end
         print("checks succeeded")
      end
end
```

As already mentioned in Section 14.2.2>, it is also possible to add special checks to loops which help to prevent common errors such as infinite loops. Here is a program computing the greatest common divisor of two integers using Euclid's algorithm.

```
class GCD
creation make
feature
   make is
      do
         print("gcd(25, 35)=" + gcd(25, 35).to_string + "%N")
```

```
      end

   gcd(a, b: INTEGER): INTEGER is
      require
         a > 0
 b > 0
      local
         x, y: INTEGER
      do
         from
            x := a
            y := b
         invariant
            x > 0
            y > 0
         variant
            x.max(y)
         until
            x = y
         loop
            if x > y then
               x := x - y
            else
               y := y - x
            end
         end
         Result := x
      end
end
```

A loop *invariant* is a condition which must be true during the whole iteration. In the example, the two variables `x` and `y` must stay positive. The *variant* of a loop is an integer expression which is always positive and becomes smaller from iteration to iteration. This way we can guarantee that the loop will end. In the Euclidian algorithm, we know that the maximum of `x` and `y` is a good candidate for a loop variant.

You may raise at least two questions at this point: What happens if one of the conditions is violated and what is the performance impact of all these checks? To answer the first question, let's try to create an account with an invalid balance by calling `!!account.make(100, 10)`.

```
*** Error at Run Time ***: Require Assertion Violated.
*** Error at Run Time ***: consistent_balance
3 frames in current stack.
=====  Bottom of run-time stack  =====
System root.
Current = TEST#0x8061a60
line 4 column 2 file ./test.e
=====================================
make TEST
Current = TEST#0x8061a60
account = Void
```

```
line 8 column 4 file ./test.e
=====================================
make ACCOUNT
Current = ACCOUNT#0x8061a88
        [ minimal_balance = 0.000000
          balance = 0.000000
        ]
a_minimal_balance = 100.000000
initial_balance = 10.000000
line 7 column 42 file ./account.e
=====   Top of run-time stack    =====
*** Error at Run Time ***: Require Assertion Violated.
*** Error at Run Time ***: consistent_balance
```

That's what I call a comprehensive error description. Not only do we get the name of the violated condition and the values of the parameters passed to the constructor method, but also the state of the account object in question. Here is an example for the violation of a loop variant. Assume that we forget the update of the loop variable.

```
class LOOP_EXAMPLE
creation make
feature
   make is
      local
         i, n: INTEGER
      do
         n := 3
         from
            i := 1
         invariant
            i > 0
         variant
            n - i
         until
            i > n
         loop
            print("i=" + i.to_string + "%N")
         end
      end
end
```

Running this program results in the following error message.

```
i=1
*** Error at Run Time ***: Bad loop variant.
Loop body counter = 1 (done)
Previous Variant = 2
New Variant = 2

2 frames in current stack.
=====  Bottom of run-time stack  =====
```

```
System root.
Current = LOOP_EXAMPLE#0x8061ab8
line 4 column 4 file ./loop_example.e
=====================================
make LOOP_EXAMPLE
Current = LOOP_EXAMPLE#0x8061ab8
i = 1
n = 3
line 14 column 15 file ./loop_example.e
=====   Top of run-time stack    =====
*** Error at Run Time ***: Bad loop variant.
Loop body counter = 1 (done)
Previous Variant = 2
New Variant = 2
```

Again, the error message points precisely at the problem.

How do all these assertion impact the performance? Eiffel allows to switch the checks on or off without changing the source code. This way, we can decide on a case by case basis whether the performance hit for the evaluation of the conditions is justified or not.

14.2.5. Visibility

All features we have defined for now are public, that is, they can be accessed by any other class. However, Eiffel allows us to restrict the visibility of features. Eiffel does not use fixed visibility modifiers (e.g., private, protected, public). Instead, we can specify which classes are allowed to see a group of features. Together with the special classes `ANY` and `NONE`, we can express private, protected and public visibility, but have more freedom to give other classes access as well. Here is a simple example.

```
class COUNTER
feature {ANY}
   increment: INTEGER is
      do
         count := count + 1
         Result := count
      end
feature {COUNTER}
   reset is
      do
         count := 0
      end
feature {NONE}
   count: INTEGER
end
```

To restrict the visibility of a feature block, we add the names of the classes which are allowed to see the features in braces. The visibility always includes all subclasses of the specified classes. Since all

application classes derive from `ANY`, the first feature `increment` is public. The `reset` feature is visible by the class `COUNTER` itself and all its children (protected feature in other languages). Finally the last feature `count` is restricted to the class `NONE`. As the name suggests, no other class can be derived from this special class, which makes the feature private.

By default, features are public. The feature definitions in the previous sections are just shortcuts for `feature {ANY}`. Besides the basic visibility rules demontrates above, we can also selectively give other classes access to certain features (similar to the `friend` mechanism in C++).

14.3. More Features

14.3.1. Expanded Types

Ideally, an object-oriented language treats all values as instances of some class. We have encountered this uniform treatment in Python and Smalltalk, for example. However, the approach taken by these languages comes at a high price. Even elementary value such as integers and doubles are always wrapped into an object with the associated memory and performance overhead. Moreover, we sometimes expect different semantics for different kinds of values. Integers, for example, should have value semantics: When we assign an integer variable to another, we expect the integer value to be copied rather than just a reference to an integer object.

Objective C and C++ keep the types inherited from C as they are. This way, they don't have to pay object overhead, but also loose the uniform treatment of values. The semantics of assignment and comparison depend on whether we use pointers to objects or the objects themselves. In C++, we have complete control over where the object lives (stack or heap) and how to access it (pointer or value).

From what we have seen for now, Eiffel seems to be doing the right thing. All values are objects, that is, instances of classes. An integer is an instance of the `INTEGER` class defined in the standard library. We can access features (e.g., the `to_string` method) of elementary types just like of any other class.

The semantics, on the other hand, change according to the type. Elementary types such as integers and double expose value semantics whereas the class we defined ourselved showed reference semantics. Eiffel accomplishes this using the notion of an *expanded* type. Using an expanded type implies value semantics for assignment. Physically, the objects are allocated effiently on the stack. We can either define a whole class as an expanded type or single variable. The elementary type such as `INTEGER` and `DOUBLE` are all defined as expanded classes. Here is an example of an expanded class of our own modeling pairs of doubles.

```
expanded class DOUBLE_PAIR
feature
   make(a_first, a_second: DOUBLE) is
      do
```

```
         first := a_first
         second := a_second
      end

   to_string: STRING is
      do
         Result := "(" + first.to_string + ", " + second.to_string + ")"
      end

   first, second: DOUBLE
end
```

In a client program, we can now use the expanded class just like a built-in expanded class.

```
class DOUBLE_PAIR_TEST creation make
feature
   make is
      local
         a: DOUBLE_PAIR
      do
         a.make(2, 3)
         print("a=" + a.to_string + "%N")
      end
end
```

The variable `a` is not a reference to an object, but refers to the pair directly. Like an integer variable, the pair is automatically initialized to the default value (a pair of zeros).

14.3.2. Exceptions

Exception handling is another area where Eiffel adds an interesting twist. In other object-oriented languages we are free to raise and catch exceptions almost anywhere in the code. We can ignore an exception with an empty catch clause, use exceptions to implement conditional logic (in which case they become go-to statements in disguise).

As it turns out, exceptions are most properly used for the truely exceptional; events which are unexpected and interrupt the normal flow. Eiffel takes this position and supports only two ways to handle an exception: Either we can retry the affected operation or it fails.

Syntactically, this implies that a method has at most one exception handling block, which Eiffel calls the `rescue` clause. It is the last clause of an operation (after the postconditions which could raise exceptions as well). The following example merely demonstrates the syntax and should not be taking as a good example for the use of exceptions, since incorrect input is not unexpected and the same logic can be achieved with a simple loop.

```
class TEST inherit EXCEPTIONS creation make
feature
```

```
   make is
      local
         i: INTEGER
      do
         print("enter positive integer: ")

         std_input.read_integer
         i := std_input.last_integer

         check i>0 end

         print("i=" + i.to_string + "%N")
      rescue
         print("exception=" + exception.to_string)
         if exception = Check_instruction then
            std_input.skip_remainder_of_line
            retry
         end
      end
end
```

In the "main success scenario", we ask the user to enter a positive number, he or she does so, and we print the entered number. We use an assertion to check that the entered number is positive. If this assertion fails, it raises an exception. The exception is caught in the `rescue` clause which first print the exception number.

Eiffel's exceptions bear more resemblance with good old error codes than the exception objects found in newer languages. By inheriting from `EXCEPTIONS`, we have access to the error code in form of the `exception` feature. The `EXCEPTIONS` class also contains constants for the codes of the core exceptions. If the exception was raised by our `check` instruction, we skip the remaining input and try again using the `retry` command. Otherwise, we don't do anything which means that the operation fails (the exception is rethrown). We can test this behavior by interrupting the program (Ctrl-C on UNIX).

14.3.3. Operator Overloading

The only difference between an ordinary method and an operator is the name which consists of one of the two keywords `infix` or `prefix` followed by the operator string. Here is an example defining the plus operator for pairs of doubles.

```
   infix "+" (other: DOUBLE_PAIR): DOUBLE_PAIR is
      do
         Result.make(first + other.first, second + other.second)
      end
```

Once defined, we can use the operator just like a built-in one.

```
class DOUBLE_PAIR_TEST creation make
```

```
feature
   make is
      local
         a, b: DOUBLE_PAIR
      do
         a.make(2, 3)
         b.make(3, 4)
         print("a+b=" + (a+b).to_string + "%N")
      end
end
```

14.3.4. Generic Types

Having seen the complex template syntax of C++, generic types are surprisingly simple in Eiffel. Here is an example using the built-in array type.

```
class ARRAY_TEST creation make
feature
   make is
      local
         v: ARRAY[DOUBLE]
         i: ITERATOR[DOUBLE]
      do
         create v.make(0, 3)
         v.put(1.5, 1)

 print("v[1]=" + v.item(1).to_string + "%N")

         i := v.get_new_iterator
         from
            i.start
         until
            i.is_off
         loop
            print("value=" + i.item.to_string + "%N")
            i.next
         end
      end
end
```

What the angle brackets are for generic types in the C family, square brackets are in Eiffel. First, we declare an array and an iterator of doubles. We create an array with four elements indexed from zero to three. The `put` method lets us set individual elements in the array, and the `item` method is used to read them. Alternatively we can call the `@` operator.

```
         print("v @ 1=" + (v @ 1).to_string + "%N")
```

Like any collection (that is, class derived from `COLLECTION`), arrays provide an iterator with the `get_new_iterator` method. Eiffel's iterators use a more conventional API than the standard template library of C++.

As an example of our own generic type, let's generalize our pair class to arbitrary element types.

```
class PAIR[G] creation make
feature
   make(a_first, a_second: G) is
      do
         first := a_first
         second := a_second
      end

   to_string: STRING is
      do
         Result := "(" + first.to_string + ", " + second.to_string + ")"
      end

   first, second: G
end
```

We just add the type parameter `[G]` to the class name and replace all occurences of `DOUBLE` by the type parameter `G`. With the new generic type, the test program looks as follows:

```
class PAIR_TEST creation make
feature
   make is
      local
         a: PAIR[DOUBLE]
      do
         create a.make(1.5, 2.5)
         print("a=" + a.to_string + "%N")
      end
end
```

14.3.5. Agents

In the preceding chapters we saw many examples of higher order functions, that is, situations where a function or even just a block of code is treated as an object which is passed to another function. Strongly-typed object-oriented languages seem to have some difficulty with this concept. In Eiffel, a (bound) method is turned into an object using the `agent` instruction. The resulting object is either a `FUNCTION` or a `PROCEDURE`, and these two generic classes offer method to execute the method contained in the agent. As usual, it is best to see a small example first.

```
class TEST creation make
feature
   make is
```

```
      do
         apply(agent add, 44, 55)
      end

   add(x: DOUBLE; y: DOUBLE): DOUBLE is
      do
         Result := x + y
      end

   apply(f: FUNCTION[ANY, TUPLE[DOUBLE, DOUBLE], DOUBLE]
         x: DOUBLE; y: DOUBLE) is
      local
         z: DOUBLE
      do
         z := f.item([x, y])
         print("f(" + x.to_string + ", " + y.to_string + ")=")
         print(z.to_string + "%N");
      end
end
```

The most complicated part is the signature of the `apply` method. The first argument `f` is declared as a function (in Eiffel a method returning a value) which belongs to any class (any class derived from `ANY`), takes two doubles as arguments and returns a double.

In other words, `FUNCTION` is a generic type with three type parameters. The first type parameter if the class the method to be wrapped by the `FUNCTION` belongs to. The other two type parameters describe the signature of the method. The second type parameter is the tuple of argument types, and the third type parameter the return type.

This also explains how the function argument is applied to the other two arguments `x` and `y`.

```
z := f.item([x, y])
```

The `FUNCTION` object has a method `item` which takes the arguments as a tuple, applies the wrapped function, and returns the result. Together with the `agent` instruction which turns a method magically into a function object, we can implement higher order functions.

14.4. Discussion

Eiffel really leaves the impression of the carefully designed language. It obviously helps that Eiffel does not carry the burden of backward compatibility to another language, but could be developed from scratch. It offers a lot of interesting answers to critical questions such as inheritance, exception handling, generic types, not to mention the unique concept of Design by Contract.

The syntax is for the most part consistent and therefore easy to learn. The only syntax element which does not seem to fit is the object creation. To me, creation of objects and assignment are two different things, and the "bang" looks like an ad hoc notation. The agent concept clearly shows Eiffel's limits. Classes and methods are just not considered first class objects.

Eiffel is a strongly typed language with explicit type declarations and as such requires more typing than its dynamically typed counterparts such as Smalltalk. However, the elegant implementation of generic types makes the strong typing a lot less painful than in other strongly typed object-oriented languages.

Notes

1. Yes, Eiffel is named after Gustave Eiffel, the engineer who created the famous tower for the 1889 world fair.

Chapter 15. Objective Caml

Caml, the Categorial Abstract Machine Language, was developed at the french research institute INRIA starting in the mid 1980's. Besides Standard ML, it is the second main ML dialect in use. Objective Caml (Ocaml for short) add object-oriented features to this ML dialect.

This chapter does not assume that you know Standard ML or have read the previous chapter. The differences between the two ML dialects, although mostly syntactical, are significant enough to justify starting from scratch. We will, however, point out the differences between SML and Ocaml as we encounter them.

15.1. Software and Installation

The main Ocaml site (http://www.ocaml.org) gives access to the various tools and documentation. For the examples in this chapter I am using the interactive interpreter `ocaml` and the byte code compiler `ocamlc` version 3.04 on Linux. Starting the `ocaml` interpreter, we receive a short welcome message and enter Ocaml's command line.

```
ahohmann@kermit:~$ ocaml
        Objective Caml version 3.04

#
```

We will begin our examples using the interpreter which gives us immediate feedback and then turn to the compiler as soon as the programs become to large to be entered at the command line.

15.2. Quick Tour

The quick tour follows our standard route starting with simple expressions and slowly stepping up to functions, collections, and user defined types before covering high level program structures such as modules and classes.

15.2.1. Expressions

Since printing is an "advanced concept" in functional languages, we start with the simple "Hello World" expression entered at the prompt of the interactive Ocaml interpreter.

```
# "Hello World";;
- : string = "Hello World"
```

As you can see, string literals are enclosed in double quotes, and the double semicolon finished an expression. A single semicolon (which ends an SML expression) is used in Caml as a separator (see below). The Ocaml interpreter shows us the type and the value of the entered expression. The dash tells us that the value was not bound to any symbol.

There is one peculiarity about arithmetical expressions: Ocaml's operators are not overloaded for integers and floating point numbers. The standard operators act on integers, and the floating point operators have a period as a suffix. At least the minus sign works as expected (we don't have to use SML's tilde ~ for negative numbers).

```
# -3+4*5;;
- : int = 17
# -3.5 +. 4.0 *. 5.0;;
- : float = 16.5
```

If we try and use an integer operator with a float (or vice versa), we get a type error. We can explicitly convert between the two standard number types using the functions `int_of_float` and `float_of_int`.

```
# 1.5 + 1;;
This expression has type float but is here used with type int
# int_of_float 1.5 + 1;;
- : int = 2
```

Similarly, we need yet another operator to add (i.e., concatenate) strings, the hat `^`.

```
# "Hello " ^ "World";;
- : string = "Hello World"
```

Concerning boolean expressions, all but the `not` operator are borrowed from C.

```
# true || false ;;
- : bool = true
# true && false ;;
- : bool = false
# not true ;;
- : bool = false
```

The `let` command binds a symbol to an expression. Like in other functional languages, this is not be confused with the assignment to a variable (like, e.g., in Python). The symbol is bound once and its value does not change.

```
# let i = 1234;;
val i : int = 1234
# i;;
- : int = 1234
```

In a functional language, almost everything is an expression. We will nonetheless treat the more complicated ones such as functions and collections in the following sections, but the conditional expression fits into this section quite well. It is basically a readable form of C's question mark operator.

```
# let n = 50;;
val n : int = 50
# if n < 100 then "small" else "big";;
- : string = "small"
```

Another expression in this context is pattern matching. In its simplest form it can be viewed as the functional form of a case statement (`switch` in C).

```
# let i = 2;;
val i : int = 2
# match i with
    1 -> "one"
  | 2 -> "two"
  | 3 -> "three"
  | n -> "more";;
- : string = "two"
```

The matching rules are separated with vertical bars, and each rule consists of a pattern, the arrow, and the expression defining the result for this pattern. The patterns are evaluated one after the other until a match is found. The result of the match expression is then the expression of the matching pattern. The "default" clause `n -> "more"` gives a first glimpse at the real power of Ocaml's pattern matching. The pattern `n` matches any integer.

15.2.2. Functions

There are two ways to define a function. The classical way binds a symbol to a function expression. The function expression consists of the keyword `function` followed by the arguments, an arrow (`->`), and the expression defining the result of the function when applied to the arguments.

```
# let times2 = function x -> 2*x;;
val times2 : int -> int = <fun>
# times2 55;;
- : int = 110
```

The second way to define a function uses a shortcut notation which lets us write down the function expression directly.

```
# let times2 x = 2*x;;
val times2 : int -> int = <fun>
# times2 55;;
- : int = 110
```

This syntax is especially convenient when defining functions with multiple arguments. If we check the types of built-in functions such as the arithmetical operators, we will notice that most of them are curried functions, that is, multiple arguments are interpreted one by one leading to a series of functions.

```
# (+);;
- : int -> int -> int = <fun>
```

If we want to define such a curried function with the `function` syntax, we end up with a rather lengthy definition like in the following example.

```
# let add = function x -> function y -> x + y;;
val add : int -> int -> int = <fun>
# add 4 5;;
- : int = 9
```

Using the shortcut notation, the same definition reduces to

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
```

The uncurried version takes the pair of parameters as a single argument. This looks like a function with two arguments in the traditional sense, but it is actually one tuple argument as we can tell from the function type where the asterisk denotes the product or tuple type.

Recursive functions require the `rec` keyword.

```
# let rec fac n = if n<2 then 1 else n*fac(n-1);;
val fac : int -> int = <fun>
# fac 5;;
- : int = 120
```

Alternatively, we could have defined the same function using pattern matching.

```
# let rec fac n = match n with
    0 -> 1
  | n -> n*fac(n-1);;
val fac : int -> int = <fun>
```

It does not make a big difference in this example, but it turns out to be extremely useful for functions acting on recursively defined data structures.

Naturally, function are first class objects (or values) in a functional language such as Ocaml. We can pass them around like any other value and define higher order functions which have functions as arguments. A good example is again the composition of functions which can be defined in one short line.

```
# let compose f g x = f(g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
# let square x = x * x;;
val square : int -> int = <fun>
# let f = compose times2 square;;
val f : int -> int = <fun>
# f 5;;
- : int = 50
```

Alternatively, we can use the longer form which clearly shows the character of the higher order function.

```
# let compose f g = function x -> f(g(x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Note how Ocaml chooses the most generic type for the function `compose`. The specialization takes place only when composing two concrete functions.

15.2.3. Collections

Ocaml has the usual set of collection types we expect from a function language. We have already met the tuple in the last section when defining functions of multiple arguments.

The most important collection is of course the (linked) list whose node structure is organized like Lisp's lists leading to efficient functions acting on the head and tail of the list. The list literal uses the semicolon to separate the elements in the list. Most of the list functions are contained in the `List` module (more on modules below).

```
# let l = [1; 2; 3];;
val l : int list = [1; 2; 3]
# List.hd l;;
- : int = 1
# List.tl l;;
- : int list = [2; 3]
# 0 :: l;;
- : int list = [0; 1; 2; 3]
# List.concat;;
- : 'a list list -> 'a list = <fun>
# List.concat [l; l];;
- : int list = [1; 2; 3; 1; 2; 3]
# List.append l l;;
- : int list = [1; 2; 3; 1; 2; 3]
# l @ l;;
- : int list = [1; 2; 3; 1; 2; 3]
# List.nth l 2;;
- : int = 3
```

Note that the list index used in the `nth` functions starts at zero. The `concat` function is equivalent to the `@` operator.

On the next higher level we find the iterator functions such as map and reduce (in Ocaml `fold_left` and `fold_right`).

```
# List.map times2 l;;
- : int list = [2; 4; 6]
# List.fold_left (+) 10 l;;
- : int = 16
```

There are also similar functions acting on two lists in parallel as well as functions looking for particular elements in a list.

```
# List.map2 (+) l l;;
- : int list = [2; 4; 6]
# List.exists (function x -> x > 2) l;;
- : bool = true
# List.mem 2 l;;
- : bool = true
# List.mem 4 l;;
- : bool = false
# List.find (function x -> x > 2) l;;
- : int = 3
# List.filter (function x -> x > 2) l;;
- : int list = [3]
```

The `sort` function implements merge sort. It sorts a list with respect to a comparison function (which must return -1, zero, or +1 just like C's compare functions).

```
# List.sort;;
- : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
# compare;;
- : 'a -> 'a -> int = <fun>
# (compare 1 2, compare 1 1, compare 2 1);;
- : int * int * int = -1, 0, 1
# List.sort compare [2; 1; 4; 3];;
- : int list = [1; 2; 3; 4]
```

So, whatever we look for in terms of list processing, we will probably find it in the standard library. Caml's reference manual contains concise descriptions of all the available functions. You can also used the module browser (`ocamlbrowser`) to get a quick overview of the system's (and your own) libraries.

While we talking about lists and sorting, we can show another instructive example from the Ocaml tutorial. It uses two mutally recursive functions (defined at once using `and`) and pattern matching to implement insertion sort.

```
# let rec sort lst =
   match lst with
     [] -> []
   | head :: tail -> insert head (sort tail)
 and insert elt lst =
   match lst with
```

```
     [] -> [elt]
   | head :: tail ->
       if elt <= head then elt :: lst else head :: insert elt tail;;
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>
```

If your main concern is fast access to an element in a collection, Ocaml's array is the data structure of choice. The array literal looks almost like a list, but decorates the delimiting brackets with vertical bars. The index operator is `.()`.

```
# let a = [| 1; 2; 3 |];;
val a : int array = [|1; 2; 3|]
# let a = [| 1; 2; 3 |];;
val a : int array = [|1; 2; 3|]
# a.(1);;
- : int = 2
```

Arrays offer most of the collection functions we have seen for lists plus some additional functions which exploit the indexing. As an example, we can iterate over the index and the elements in an array at the same time.

```
# Array.iteri;;
- : (int -> 'a -> unit) -> 'a array -> unit = <fun>
# let f i x = Printf.printf "%d: %d\n" i x;;
val f : int -> int -> unit = <fun>
# Array.iteri f [|10; 20; 30|];;
0: 10
1: 20
2: 30
- : unit = ()
# Array.mapi (+) [|10; 20; 30|];;
- : int array = [|10; 21; 32|]
```

The first example also demonstrates the built-in functions for formatted output which are modeled after C's standard I/O library.

With arrays, we can also leave the pure functional world, since Ocaml's arrays are mutable collections. The assignment operator is the left arrow `<-`.

```
# a.(1) <- 55;;
- : unit = ()
# a;;
- : int array = [|1; 55; 3|]
```

Once we are changing objects, we can also talk about Ocaml's dictionary type defined in the `Hashtbl` (hash table) module. Using it already leaves some object-based impression. Hash tables are created with the `create` function (taking the expected size of the dictionary as an argument). The actual type is

undetermined until the we add the first key-value pair to the dictionary. The typical put and get operations are given as `replace` and `delete`.

```
# let h = Hashtbl.create 100;;
val h : ('_a, '_b) Hashtbl.t = <abstr>
# Hashtbl.replace h "Homer" 55;;
- : unit = ()
# h;;
- : (string, int) Hashtbl.t = <abstr>
# Hashtbl.replace h "Bart" 11;;
- : unit = ()
# Hashtbl.find h "Bart";;
- : int = 11
```

There is also an `add`function which overrides an already existing entry without actually deleting the old value. Removing the entry will reveal the old value again.

```
# let h = Hashtbl.create 100;;
val h : ('_a, '_b) Hashtbl.t = <abstr>
# Hashtbl.add h "Homer" 55;;
- : unit = ()
# Hashtbl.add h "Homer" 66;;
- : unit = ()
# Hashtbl.find h "Homer";;
- : int = 66
# Hashtbl.find_all h "Homer";;
- : int list = [66; 55]
# Hashtbl.remove h "Homer";;
- : unit = ()
# Hashtbl.find h "Homer";;
- : int = 55
```

This behavior models bindings with nested scopes (probably reflecting the Ocaml implementation).

Similar to the higher order functions for sequences, there are functions which operate on all elements in the hash table. The iterator function `iter`, for example, allows us to apply a function (as "visitor") to all name-value pairs in the hash table.

```
# let h = Hashtbl.create 100;;
val h : ('_a, '_b) Hashtbl.t = <abstr>
# Hashtbl.add h "Homer" 55;;
- : unit = ()
# Hashtbl.add h "Bart" 11;;
- : unit = ()
# let f key value = Printf.printf "%s: %d\n" key value;;
val f : string -> int -> unit = <fun>
# Hashtbl.iter f h;;
Homer: 55
Bart: 11
- : unit = ()
```

15.2.4. Data Types

For now, we have worked with a number of Ocaml's built-in types. Ocaml has two main tools to create new types: records and variants. A record is a collection of named fields just like a record in C or other procedural languages.

```
# type point = {x: int; y: int};;
type point = { x : int; y : int; }
# let add p1 p2 = {x = p1.x + p2.x; y = p1.y + p2.y};;
val add : point -> point -> point = <fun>
# let p1 = {x=10; y=20};;
val p1 : point = {x = 10; y = 20}
# p1.x;;
- : int = 10
# let p2 = {x=(-5); y=(-10)};;
val p2 : point = {x = -5; y = -10}
# add p1 p2;;
- : point = {x = 5; y = 10}
```

In contrast to SML, fields are accessed with the "usual" dot notation known from the descendents of C and Pascal.

By default, records are immutable. We can not change a field of a record, but only construct completely new ones. However, adding the `mutable` keyword to a field allows us to change its value. Like the mutable arrays, this feature leaves the clean functional world and lets us work with dangerous (but often efficient) side effects.

```
# type person = {mutable name: string; age: int};;
type person = { mutable name : string; age : int; }
# let p = {name="Homer"; age=55};;
val p : person = {name = "Homer"; age = 55}
# p.age <- 66;;
The record field label age is not mutable
# p.name <- "Bart";;
- : unit = ()
# p;;
- : person = {name = "Bart"; age = 55}
```

The second tool for type creation are variant types. Variant types model alternatives.

```
# type color = Blue | Red | Green;;
type color = Blue | Red | Green
# Red;;
- : color = Red
```

Note that Ocaml uses the `type` keyword for records and variants alike. We can define the equivalent of SML's `option` using a parameterized variant type.

```
# type 'a option = None | Some of 'a;;
```

```
type 'a option = None | Some of 'a
# None
  ;;
- : 'a option = None
# Some 55;;
- : int option = Some 55
```

But as in SML, we don't have to define the `option` type ourselves since it is already part of Ocaml's standard library.

The classic example for a recursive variant type, a binary tree, looks almost like the SML equivalent.

```
# type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

15.2.5. Module System

Ocaml's structures bundle type and functions to consistent units. Apart from minor syntactical differences, Ocaml follow the ML mode as defined in SML. The `module` command (replacing SML's `structure`) gives a structure a name. Here is the Ocaml version of the `Color` structure.

```
# module Color = struct
    type color = Red | Green | Blue;;
    let name c = match c with
        Red   -> "red"
      | Green -> "green"
      | Blue  -> "blue";;
  end;;
module Color :
  sig type color = Red | Green | Blue val name : color -> string end
# Color.name Color.Red;;
- : string = "red"
```

Signatures are to structures what types are to values. It is therefore only consistent that we bind a signature to a name using the `module type` command. Here is the interface for our extremely useful `Color` structure.

```
# module type COLOR = sig
    type color;;
    val name : color -> string;;
    val make : string -> color;;
  end;;
module type COLOR = sig type color val name : color -> string end
```

Since the actual type is hidden, we have added a constructor function which is the opposite of the `name` function. We can now restrict the visibility of our structure by declaring it as an implementation of the signature.

```
# module Color : COLOR = struct
    type color = Red | Green | Blue;;
    let name c = match c with
        Red   -> "red"
      | Green -> "green"
      | Blue  -> "blue";;
    let make s = match s with
        "red"   -> Red
      | "green" -> Green
      | "blue"  -> Blue;;
  end;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
""
module Color :
  sig
    type color = Red | Green | Blue
    val name : color -> string
    val make : string -> color
  end
```

This interpreter is nice enough to warn us about the incomplete definition of the `make`. We will ignore this warning until we know how to indicate errors with exception.

The actual colors are now not accessible anymore. Instead, we have to construct them from the color strings.

```
# Color.Red;;
Unbound constructor Color.Red
# let c = Color.make "red";;
val c : Color.color = <abstr>
# Color.name c;;
- : string = "red"
```

In contrast to SML, we see only what has been declared in the signature (similar to SML's opaque signature constraint `:>`).

15.2.6. Objects and Classes

We have finally caught up with SML and now tackle the object-oriented ML extensions implemented in Ocaml. Here is the first Ocaml version of our person class.

```
# class person = object
    val name = "Homer"
    val age = 55
    method get_name = name
    method get_age = age
    method hello = "Hello, I'm " ^ name
  end;;
class person :
  object
    method get_age : int
    method get_name : string
    method hello : string
    val age : int
    val name : string
  end
# let p = new person;;
val p : person = <obj>
# p.name;;
Unbound record field label name
# p#get_name;;
- : string = "Homer"
# p#hello;;
- : string = "Hello, I'm Homer"
```

Of course, the class is not very satisfactory yet, because we hard-coded the values of the attributes an have no way to change them. But at least we see the basic syntax of a class, object creation, and method call.

Attributes are just values and methods are functions which have direct access to the objects attributes. The attributes are not visible from the outside (unless exposed by a method), and the methods are called with a hash # replacing the dot we are used to from other OO languages.

As it stands, we can not change the values of the attributes, not even by a method, since they are immutable by default. So, the first option to turn the class into something useful is to make the attributes mutable and introduce setter methods. To keep the example short, we only consider the first attribute.

```
# class person = object
    val mutable name = ""
    method get_name = name
    method set_name a_name = name <- a_name
    method hello = "Hello I'm " ^ name
  end;;
class person :
  object
```

```
    method get_name : string
    method hello : string
    method set_name : string -> unit
    val mutable name : string
  end
# let p = new person;;
val p : person = <obj>
# p#get_name;;
- : string = ""
# p#set_name "Homer";;
- : unit = ()
# p#hello;;
- : string = "Hello I'm Homer"
```

Although this approach works, it is following neither functional nor object-oriented best practices. What we really would like to do is to pass the values to a constructor. In Ocaml this is accomplished by writing the class definition as a function of the initial values. Here is the according version (again reduced to the bare minimum).

```
# class person = fun a_name ->
    object
      val name : string = a_name
      method get_name = name
    end;;
class person :
  string -> object method get_name : string val name : string end
# let p = new person "Homer";;
val p : person = <obj>
# p#get_name;;
- : string = "Homer"
```

Here we have to specify the type of the attribute, since Ocaml can't derive it otherwise. Alternatively, we could have declared the type of the initialization argument.

```
# class person = fun (a_name: string) ->
    object
      val name = a_name
      method get_name = name
    end;;
class person :
  string -> object method get_name : string val name : string end
```

Like for function definitions, there is a shortcut syntax combining the function and class definition.

```
# class person (a_name: string) =
    object
      val name = a_name
      method get_name = name
    end;;
class person :
  string -> object method get_name : string val name : string end
```

Next, we would like to extend the person class using inheritance. To this end, we need to add an `inherit` clause at the beginning of the class definition.

```
# class employee a_name a_no =
    object
      inherit person a_name
      val no: int = a_no
      method hello = "Hello, I'm number " ^ string_of_int(no)
    end;;
class employee :
  string ->
  int -> object method hello : string val name : string val no : int
end
# let e = new employee "Homer" 1234;;
val e : employee = <obj>
# e#hello;;
- : string = "Hello, I'm number 1234"
```

If we need to refer to the original class, we can bind it to a name in the inherit clause. This name, typically `super`, can then be used to call the implementation of a method in the parent class.

```
# class person (a_name: string) =
    object
      val name = a_name
      method hello = "Hello, I'm " ^ name
    end;;
class person : string -> object method hello : string val name : string end
# class employee (a_name: string) (a_no: int) =
    object
      inherit person a_name as super
      val no: int = a_no
      method hello = super#hello ^ ", " ^ string_of_int(no)
    end;;
class employee :
  string ->
  int -> object method hello : string val name : string val no : int
end
# let e = new employee "Homer" 1234;;
val e : employee = <obj>
# e#hello;;
- : string = "Hello, I'm Homer, 1234"
```

To call a method on itself, an object has to define a name for itself (by convention `self`) following the `object` keyword.

```
# class person name =
    object (self)
      val name = name
      method hello = "Hello, I'm " ^ name
      method print = print_string self#hello
    end;;
```

```
class person :
  string ->
  object method hello : string method print : unit val name : string
end
# let p = new person "Joe";;
val p : person = <obj>
# p#print;;
Hello, I'm Joe- : unit = ()
```

In this example, I have also exploited (lazy as I am) the possibility to use the same name for the initialization parameter, the attribute, and the getter method.

Generic classes (classes with type parameters) are about as simple as in Eiffel. The type variable is placed in parentheses in front of the class name.

```
# class ['a] point (x_init: 'a) (y_init: 'a) =
    object
      val x = x_init
      val y = y_init
      method get_x = x
      method get_y = y
    end;;
class ['a] point :
  'a ->
  'a -> object method get_x : 'a method get_y : 'a val x : 'a val y : 'a end
# let p = new point 1.5 2.5;;
val p : float point = <obj>
# p#get_x;;
- : float = 1.5
```

Once we know how inheritance works, we would like solid OO design and define interfaces and their implementations. Some languages called them "abstract" (C++, Java, C#), some "deferred" (Eiffel), and Ocaml uses the notion "virtual": methods without an implementation. The interface for our simplistic account could look like this:

```
# class virtual account =
    object
      method virtual balance : float
      method virtual deposit : float -> unit
      method virtual withdraw : float -> unit
    end;;
class virtual account :
  object
    method virtual balance : float
    method virtual deposit : float -> unit
    method virtual withdraw : float -> unit
  end
```

To indicate that an account implementation actually realizes this interface, we inherit from the virtual class.

```
# class simple_account =
    object
      inherit account
      val mutable balance = 0.0
      method balance = balance
      method deposit amount = balance <- balance +. amount
      method withdraw amount = balance <- balance -. amount
    end;;
class simple_account :
  object
    method balance : float
    method deposit : float -> unit
    method withdraw : float -> unit
    val mutable balance : float
  end
```

Clients should only see the interface, not the implementation. We therefore need another operator which changes the type of an object to a base class (up-casting). In Ocaml, this is the `:>` operator.

```
# let a = (new simple_account :> account);;
val a : account = <obj>
# a#deposit 100.0;;
- : unit = ()
# a#withdraw 25.0;;
- : unit = ()
# a#balance;;
- : float = 75
```

As you see, the type of `a` is `account`, and therefore only the interface is visible to us.

The object-oriented paradigm is well suited for functions which can be attached to one of its arguments (which becomes the owner of the method). Symmetric functions such as comparisons are not as good a fit and often call for unconvenient solutions. Java's `equals` method, for example, takes an arbitrary object as an argument although this object should be of the same class as the object on which the method is called. Ocaml has an interesting solution for these binary methods. A virtual method can use the type of the owning object which is made available through a type variable in the `self` clause.

```
# class virtual comparable =
    object (_: 'a)
      method virtual compare: 'a -> int
    end;;
class virtual comparable : object ('a) method virtual compare : 'a -> int end
# class account balance =
      object
        inherit comparable
        val mutable balance : float = balance
        method balance = balance
```

```
        method compare other =
          if      balance < other#balance then -1
          else if balance > other#balance then 1
          else 0
      end;;
class account :
  float ->
  object ('a)
    method balance : float
    method compare : 'a -> int
    val mutable balance : float
  end
# (new account 50.0)#compare (new account 100.0);;
- : int = -1
```

Object-oriented programing with Ocaml offers another interesting feature which can help us to enjoy the advantages of objects without giving up functional reasoning: functional objects. To stay in the functional world, we have to refrain from mutable attributes. On the other hand, we often need to change some part of an object. Functional objects are immutable objects with methods that return chnaged copies of the original object. Instead of altering the state of the object itself, we create a copy that contains the change. This operation of "copying with change" is accomplished with the `{< ... >}` operator. Here is an example of a "functional account".

```
# class account =
    object
      val balance = 0.0
      method balance = balance
      method deposit amount = {< balance = balance +. amount >}
      method withdraw amount = {< balance = balance -. amount >}
    end;;
class account :
  object ('a)
    method balance : float
    method deposit : float -> 'a
    method withdraw : float -> 'a
    val balance : float
  end
# let a = new account;;
val a : account = <obj>
# a#balance;;
- : float = 0
# let a1 = a#deposit 50.0;;
val a1 : account = <obj>
# a1#balance;;
- : float = 50
# a#balance;;
- : float = 0
```

The two methods `deposit` and `withdraw` do not change the account object, but create a new one with the updated balance.

15.3. More Features

15.3.1. Exceptions

Like most modern languages, Ocaml handles error conditions using exceptions.

#

#

#

#

#

#

#

Chapter 16. Perl

Perl was created by Larry Wall in 1987 as a text processing language combining the power of the UNIX tools awk, sed, shell scripts, and some C. The name stands for "Practical Extraction and Report Language" and is also known as the "Pathologically Eclectic Rubbish Lister". It established itself quickly as the scripting language of choice for UNIX systems. The typical applications are system administration tools and batch jobs such as loading data from a file into a database. Due to its diverse background, Perl had a lot of limitations and quirks in the beginning, but became a general purpose programming language with its version 5 in 1994.

Originally I was not planning to include Perl in this book at all. Grown out of the two UNIX tools `sed` and `awk`, which were never supposed to be used for large programs, Perl is a collection of many individual features which are hard to describe on a few pages. But how can you ignore a language which is used by tens if not hundreds of thousands of programmers to solve their every day problems? I include this chapter mainly for two reasons. To give the background for the less obvious features of the scripting language Ruby covered in the next chapter, and to enable programmers to migrate the many legacy applications written in Perl during the last few years.

16.1. Software and Installation

For the examples, I'm using Perl version 5.6.1. Perl does not have a built-in interactive environment which we can use to explore the language on the command line. Instead, Perl compiles a script into some intermediate format and runs it as a whole. You can, however, run most examples with Perl's debug mode using the command `perl -d -e 1`.

16.2. Quick Tour

16.2.1. Expressions and Context

The hello world example is identical to Python's version apart from the semicolon ending the print statement.

```
print "Hello World";
-> Hello World
```

Strings can be delimited in multiple ways, each way causing the string to be interpreted differently. Using double quotes, for example, Perl interpretes C's escape sequences (and also interpolates variables as we will see later on). Using single quotes, the string is interpreted literally.

```
print "Hello\tWorld\n";
print 'Hello\tWorld\n';

-> Hello   World
Hello\tWorld\n
```

In contrast to Python, `print` is a function. Function calls look like in most procedural languages. However, the parentheses around the arguments can be omitted. The following statements are equivalent.

```
print 123, "blah", 5.6;
print(123, "blah", 5.6);

-> 123blah5.6123blah5.6
```

By default, the print function does not separate the printed fields and records (that is, multiple calls to the `print` function). But we can change this behavior by setting the special field and record separator variables. To keep our examples short, we emulate Python's print statement by setting the field separator to a space and the record separator to a newline.

```
$, = " ";
$\ = "\n";
print 123, "blah", 5.6;
print(123, "blah", 5.6);

-> 123 blah 5.6
123 blah 5.6
```

Besides the plain `print` function, Perl also supports C's formatted printing to streams and strings.

Basic arithmetic works as expected as well. The meaning of the operators is derived from C (and awk).

```
print 3 + 4*5 + 2**3;
-> 31
```

The next examples give us a first idea of Perl's context sensitivity.

```
print 10 + "10";
print 10 . "10";
print 10/3;
print 5.5 % 3;
print "ab" x 3;
print 10 x 5.5;
-> 20
1010
3.33333333333333
2
ababab
1010101010
```

Perl interpretes every statement in its context and freely converts between types as required. The first example adds the String "10" to the number 10. Since addition is defined for numbers only, the string is implicitly converted to a number. For string concatenation (the dot operator), the opposite happens, and the number is converted to a string. The third example shows that the division operator is using floating point arithmetic (unless you switch to integer mode using `use integer;`). On the other hand, the remainder operator `%` is an integer operator which causes the floating point number 5.5 to be converted to an integer before computing the remainder (in Python the same expression would return 1.5). The letter `x` denotes the string replication operator which takes the string on the left hand side and the number of times the string should be repeated on the right hand side. Because of this, the number 10 is actually interpreted as a string and the string `"5.5"` as the integer 5.

As a consequence of all these implicit type conversions, more operators are required to avoid ambiguities. For example, Perl can not overload the plus operator to also mean string concatentation, since the expression `10+"10"` could be either 20 or "1010". Similarly, the string replication must use a new operator and insist on the correct order of arguments in order to make sense of expressions like the one given above. The same is true for the comparison operators. Perl uses the normal mathematical comparison symbols <, <=, ==, >=, > for numbers and the Fortran-like operators `lt`, `le`, `eq`, `ge`, `gt`, `lt` for strings.

```
print "le", ("-10" le "+10")? "true" : "false";
print "<=", ("-10" <= "+10")? "true" : "false";

-> le false
   <= true
```

In the first statement, the "less or equal" comparison is carried out for strings using the `le` operator. Since the plus sign (43) comes before the minus sign (45) in the ASCII character set, the string "-10" is bigger than the string "+10". Using the numerical `<=` operator, the strings are implicitly converted to integers and we get the correct result.

Perl became famous for its string processing in general and regular expressions in particular. Strings enclosed in forward slashes are interpreted as regular expressions. Together with the match operator `=~`, we arrive at a very compact syntax for checking string with respect to regular expressions.

```
print "x1=y1" =~ /^\w+=\w+$/ ? "true" : "false";
-> true
```

Here, we test if the string `"x1=y1"` consists of two alphanumberic words separated by an equality sign (more on regular expressions later). The backslashes are now interpreted in the context of a regular expression. Followed by the letter `w` they represent the set of alphanumeric characters.

16.2.2. Variables

Perl knows three kinds of variables, each with different scoping rules: global, local, and lexical. Global variables live in the global namespace of the current package (see below; for now we have only used the

main package). They are visible to all programs. Local variables are dynamically scoped. They override the value of a global variable for the duration of a block. Finally, lexical variables are lexically scoped, that is, they are only visible within the block they are defined in. Perl's lexical variables correspond to the local variables we are used to from Scheme and the C family.

The default are global variables. Local variables have to be declared with the `local` keyword, lexical ones with the `my` keyword. In practice, most programs use almost only lexical variables. For most applications it is safe to assume that variables have to be declared with `my`. In this chapter, we will restrict ourselves to lexical variables (as we will this, this gets complicated enough in Perl). To ensure that we do not define global variables by accident, we follow the common Perl practice and switch into the `strict` mode and turn on the warnings at the beginning of the script.

```
use strict;
use warnings;
```

As one of Perl's curiosities, variables use the symbols dollar ("$"), at ("@"), and percent ("%") as prefixes depending on the context. Let's start with the simple case of scalar variables which start with a dollar sign.

```
my $x = "Hello World";
print $x;

-> Hello World
```

With the conventions we introduced above, the whole program look like this:

```
use strict;
use warnings;

$\ = "\n";
$, = " ";

my $x = "Hello World";
print $x;
```

We introduce the new lexical variable with the `my` keyword and initialize it with the string "Hello World". Alternatively, we could have declared the variable separately and assigned the string afterwards.

```
my $x;
$x = "Hello World";
print $x;
```

We can also declare and initialize multiple variables at the same time using `my` with a list of variables.

```
my ($x, $y) = ("John", "Joe");
print $x, $y;

-> John Joe
```

When trying to use a variable outside of its scope, we get a compile error.

```
{
    my $x = "Hello";
}
print $x;

-> Global symbol "$x" requires explicit package name
```

The compiler does not tell us that we try to use an undefined variable. Instead the compiler assumes that we want to access a global variable without the package name enforced by the `strict` mode. Without the `strict` mode, the `print` statement would have referred to the global variable `$x` in the `main` package. The program would have been compiled without errors, but the output would have been empty, since the global variable `$x` was not initialized.

Knowing variables, we can now return to the string interpolation. We often would like to insert the value a variable into a string. In Perl (like in UNIX shell languages), this is done automatically when using double quotes to delimit the string.

```
my $name = "Frank";
print "My name is $name";

-> My name is Frank
```

This process of substituting variables in strings is called interpolation.

Next to regular expressions, Perl’s most prominent feature are the built-in collection types, arrays and hashes. Both names refer to the underlying implementation. Arrays are dynamically resizing vectors containing scalar values. Hashes are maps (implemented as hash tables) with scalar keys and values.

With these collections, the context dependent variable symbols start to get funny. When referring to an array as a whole, a variable starts with an "at" sign, and a hash is indicated by the percent.

```
my @a = (1, 2, 3);
my %h = ("John", 55, "Joe", 33, "Mary", 110);

print @a;
print %h;

-> 1 2 3
Mary 110 Joe 33 John 55
```

Both, vectors and hashes are initialized with list literals which are just lists of comma separated values. In a hash context (e.g., assignment to a hash variable), the list is viewed as the alterating list of keys and values. In Perl programs you will often find a special shortcut for lists containing strings. It starts with the keyword `qw` (quoted words) followed by a list of unquoted string enclosed in some delimiter (often a slash or a parenthesis). The following expressions are all equivalent.

```
@a = ("John", "Joe", "Mary");
@a = qw/John Joe Mary/;
@a = qw(John Joe Mary);
```

Using this special list quote you save the double quotes and the commas (some people may not consider this worth a new syntax).

Another more useful syntactical element for lists is the arrow `=>`. It can be used whereever the comma is used in a list and makes the definition of hashes more readable.

```
my %h = ("John" => 55, "Joe" => 33, "Mary" => 110);
```

Besides the pure readability, it also automatically double quotes string so that the expression above can be shortened to the following version.

```
my %h = (John => 55, Joe => 33, Mary => 110);
```

You can also confuse the reader (which might be yourself), by using the arrow arbitrarily.

```
my @a = (1, 2 => 3 => 4, 5);
my %h = ("John", 55 => "Joe", 33 => "Mary" => 110);
```

The result is always the same.

To construct lists of consecutive values of some enumerated type, Perl also supports a range operator `..`

```
print (0 .. 10);
print ('a' .. 'g');

-> 0 1 2 3 4 5 6 7 8 9 10
a b c d e f g
```

The ranges include both limits (closed interval).

The funny thing about Perl's type prefixes is that they change depending on the usage of the variable (for the same variable!). Whenever we access an individual element of an array or a hash, the prefix becomes a dollar sign.

```
print $a[0], $h{"John"}, $h{John};
```

One could argue that the indexing extracts a scalar, but seriously, there is no convincing argument for this rule. Also note that a hash uses curly braces as the subscript operator. The hash subscript operator also automatically adds double quotes to unquoted string (just like `qw` and the arrow `=>`).

Perl's arrays also support negative indexes (counting from the end of the array) and slices.

```
my @a = (1 .. 5);
print $a[-2];
print @a[1..3];

-> 4
2 3 4
```

Since a slice is an array, the "at" sign is used as a prefix.

Here are a few more curiosities. What do you think is the value of an array in a scalar context, for example, when assigning it to a to scalar variable?

```
my @a = ("a", "b", "c");
my $n = @a;
print "n=$n";

-> n=3
```

Yes, it is the length of the array (the only reasonable scalar value of an array). What about an array literal in a scalar context?

```
my $x = ("John", "Joe", "Mary");
print "x=$x";
-> x=Mary
```

It is not interpreted as an array at all, but as an expression (in parentheses) using the comma operator. The comma operator evaluates both sides and returns the value of the right hand side. If applied multiple times, the value of the rightmost expression, here "Mary", is returned.

Since the different types of variables live in different name spaces, you can use the same name for a scalar variable, an array, and a hash.

```
my $a = "blah";
my @a = ("a", "b", "c");
my %a = (x => 1, y => 2, z => 3);

print '$a:', $a;
print '@a:', @a;
print '$a[1]:', $a[1];
print '%a:', %a;
print '$a{x}:', $a{x};

-> $a: blah
@a: a b c
$a[1]: b
%a: x 1 y 2 z 3
$a{x}: 1
```

Internally, Perl uses a "multi-cell" approach. Every symbol such as "a" has an associated structure with one cell for scalars, one cell for arrays, one cell for hashes, and so forth. Recall that the Common Lisp implementation uses two cells (one for "ordinary" values, one for functions) whereas Scheme is a one-cell implementation. As we have already seen when comparing Lisp to Scheme, more cells mean that objects can not be handled homogeneously which has to be accomodated by a more complex syntax.

16.2.3. Control Statements

Perl offers the whole set of control statements of procedural languages with a C-like syntax and adds a few less common variations. For example, there is not only the typical if/else statement, but also the negated variant unless/else.

```
my $x = 55;
if    ($x < 10)  { print "small"; }
elsif ($x < 100) { print "medium"; }
else             { print "big"; }

unless (1 < 2) { print "false"; } else { print "true"; }
```

The `while` and `for` loops work like their C counterparts. You can jump to the next iteration using `next` (equivalent to C's `continue`) and leave the loop using `last` (equivalent to C's `break`). It is even possible to repeat an iteration using the `redo` command. If you need some statements to be executed between two iterations, they can be placed in an optional `continue` block just after the `while`. Just like `unless` is a shortcut for if-not (in Perl: `if (!...)`, `until` is the negated version of `while`.

You can also iterate through an array using the `foreach` (you can also use just `for`) loop. Apart from the different syntax (and less general applicability), it works like Python's for-in loop.

```
my @a = (1 .. 3);
foreach $i (@a) {
    print "i=$i";
}
foreach (@a) {
    print;
}

-> i=1
i=2
i=3
1
2
3
```

If no loop variable is specified, the current element is assigned to the special variable `$_`. Since the `print` function prints this special variable by default, we arrive at the short form shown in the second loop.

Iterating through the key-value pairs of a hash map is best accomplished with the `each` function.

```
my %h = ("John" => 55, "Joe" => 66);
print each %h;
print each %h;
print each %h;
print each %h;
```

With each call, the `each` function returns the next key-value pair in the map until there are no entries left. It then returns an empty list. Calling `each` again, starts the process all over again (Somehow it must store the state of the iterator in the hash map itself). Combined with a while loop, we get all the key-value pairs of a hash.

```
my %h = ("John" => 55, "Joe" => 66);
while (my ($key, $value) = each %h) {
    print $key, $value;
}

-> Joe 66
John 55
```

The conditional statements `if` and `unless` and the loop statements `while` and `until` can also be put behind a statement as a so-called modifier.

```
print "too big" if ($x > 100);

@a = ("x", "y", "z");
print "elem: $elem" while $elem = shift @a;
```

16.2.4. References

Here is another quiz: What happens if we put an array into an array? More concrete, what is the length of the following array?

```
my @a = (1, 2, (3, 4, 5));
my $n = @a;
print "n=$n";

-> n=5
```

The resulting array contains the numbers one to five! The array `(3, 4, 5)` is not inserted into the array as a whole, but unpacked and inserted element-wise. Perl's collections contain scalar values only. As another little surprise, trying to store an array in a hash map stores the first element array (not the length and neither the last element).

```
my %h = ("John", (3, 4, 5));
print $h{"John"};
```

```
-> 3
```

At this point it is time to introduce another scalar type, the reference. A reference is a safe version of a C pointer, and like C's pointers, Perl's references add a lot of power to the language. To obtain a reference to an object, we apply the backslash operator. Going from a reference to the object the reference points to can be accomplished in multiple ways. In the simplest case, we use the reference variable as a variable name, that is, whereever you would normally use the name of the variable (without the prefix symbol), you now use the name of the reference variable (including the dollar sign). Here is a scalar example.

```
my $x = "Hello";
my $rx = \$x;
print $rx, $$rx
$x = "World";
print $rx, $$rx;

-> SCALAR(0x0173f4) Hello
   SCALAR(0x0173f4) World
```

And here are some examples using references to arrays and hashes.

```
my @a = ("John", "Joe", "Mary");
my $ra = \@a;
my $n = @$ra;
print "n=$n ", $$ra[1], $ra->[2];

-> n=3 Joe Mary
```

For array indexing there is an alternative syntax using an arrow suffix `ra->` instead of the dollar prefix `$ra` as the dereferencing operator.

```
my %h = ("John" => 55, "Joe" => 66, "Mary" => 77);
my $rh = \%h;
print $$rh{"John"}, $rh->{"John"};

-> 55 55
```

Reference do not need to point to variables but can also reference values ("anonymous data" in Perl parlance) directly.

```
my $rx = \"Hello";
print $$rx;
-> Hello
```

Because of the context sensitivity of the comma operator, it is not possible to create a reference to an anonymous array by just putting a backslash in front of it.

```
my $ra = \("John", "Joe", "Mary");
print $$ra;
```

```
-> Mary
```

Instead, there is a special syntax for references to anonymous arrays and hashes which makes them almost look like Python lists and maps.

```
my $ra = ["John", "Joe", "Mary"];
my $rh = { "John" => 55, "Joe" => 55 };
print $ra->[0], print $rh->{"John"};

-> John 55
```

Now, using references, we can put non-scalar values in collections.

```
my @a = (1, 2, [3, 4, 5]);
my $n = @a;
print "n=$n ", $a[2]->[1], $a[2][1]

-> n=3 4 4
```

Note that the dereferencing arrow between array and hash subscripts can be omitted, that is, `$a[1]{"a"}[2]` is equivalent to `$a[1]->{"a"}->[2]`.

Using the reference variable as a variable name, we always have to first assign the reference to a variable before we can use it. A typical situation is a function returning a reference.

```
sub names { return ["John", "Joe", "Mary"]; }
my $x = names();
print @$x;

-> John Joe Mary
```

To avoid the temporary variable, Perl also allows to use an arbitrary block (returning a reference) as a variable name.

```
sub names { return ["John", "Joe", "Mary"]; }
print @{names()};
```

Recalling the interpolation property of strings in double quotes, we can now put arbitrary expressions into strings.

```
print "3 + 4 = ${\(3+4)}";
-> 3 + 4 = 7
```

Here we compute the result of "3+4", take the reference of it using the backslash operator, and use it as the "variable name" for the interpolated variable. As usual, Perl’s syntax needs some getting use to, but it works.

16.2.5. Functions

Let's see how we can organize a Perl program using functions, which are called sub-routines in Perl.

```
sub times2 {
  my $x = shift(@_);
  return 2 * $x;
}
print times2(20), times2 55;

-> 20 110
```

Now, this is different from what we've seen in the previous chapters. Perl does not support formal function parameters even though one of its ancestors, GNU's `nawk`, does. Instead, Perl adopts a shell-like approach and passes the arguments as an array to the function. Inside of the function you can access the argument array with the implicit variable `@_`. In other words, Perl makes variable argument list, which are the exception in most languages, the rule. Without a formal parameter list, a function definition consists of just the keyword `sub`, the function name, and the function body which is a block of statements enclosed in curly braces. The first statement introduces a local variable `$x` and assigns the first element of the argument array to it. Since `@_` is the default array, we can just write `shift`.

```
sub times2 {
  my $x = shift;
  return 2 * $x;
}
```

We could also access this element using indexing `$_[0]`, but the shift operation is the more idiomatic way to access function arguments. With two argument, we get two identically looking local variables statements.

```
sub add {
    my $x = shift;
    my $y = shift;
    return $x + $y;
}
```

Even shorter, you can assign the argument array to list of local variables representing the arguments. This has the advantage that the arguments array `@_` is not changed.

```
sub add {
    my ($x, $y) = @_;
    return $x + $y;
}
```

So, in reality the missing formal argument list is always simulated with some idiomatic use of local variables.

How are arguments passed to a function? Since the arguments are passed as an array, the semantics are identical to the construction of arrays. If we pass an array, it will be unpacked and supplied to the function as individual elements in the argument array.

```
sub showArgs { print "args:", @_; }

showArgs(1, 2, 3);
showArgs(1, ("a", "b", "c"), 2);

-> args: 1 2 3
args: 1 a b c 2
```

Another surprising property of Perl's argument passing is that scalars are always passed by reference. This means that you can change the value of variables passed to a function.

```
sub change {
    for my $i (@_) {
$i *= 2;
    }
}

my @a = (1, 2, 3);
change @a;
print @a;

my $a = 55;
change $a;
print $a;

-> 2 4 6
110
```

Using references, functions can be passed around as variables and arguments to functions.

```
my $rs = \&times2;
print $rs, &$rs(5)

-> CODE(0xa01f6c0) 10
```

Using a function reference we can implement the `reduce` function.

```
sub reduce {
    my $f = shift;
    my $initial = shift;
    my @list = @_;
    my $result = $initial;
    foreach $i (@list) {
        $result = &$f($result, $i);
    }
    return $result;
}
```

```
sub add { return $_[0] + $_[1]; }

print "add", reduce(\&add, 10, (1, 2, 3));

$mult = sub { $_[0] * $_[1] };
print "mult", reduce($mult, 1, (1 .. 5));

-> add 16
mult 120
```

The second application of the `reduce` uses an anonymous function reference for the multiplication. Such a function reference works like a lambda expression. Its syntax is extremely simple: just omit the name of the function (I consider this as one of Perl's highlights).

Using anonymous functions, it is even possible to return functions from functions.

```
sub adder {
    my $x = shift;
    return sub { my $y = shift; return $x + $y; }
}

$add100 = adder 100;
$add200 = adder 200;
print $add100, &$add100(5);
print $add200, &$add200(5);

-> CODE(0xa0200c4) 105
   CODE(0xa020100) 205
```

Note that the different calls to `adder` really create different version of the function which carry the context in which they were created (here the value of the local variable `$x` which was passed as an argument to `adder`). In other words, the returned function references are closures.

With this knowledge, defining the `compose` functional is not a big step anymore.

```
sub compose {
    my ($f, $g) = @_;
    return sub { &$f(&$g(@_)); };
}

my $h = compose(\&times2, \&add);
print "compose", &$h(3, 4);
print "compose", &{compose(\&times2, \&add)}(3, 4);

-> 14
```

We pass the two function references as the arguments `$f` and `$g` to the `compose` function. It return a reference to an anonymous function which first applies `$g` to the arguments of the anonymous function

and then `$f` to the result of `$g`. To apply the functions, we have to dereference the function references using the plain dollar dereference operator. Similary, when applying the composed function, we have to dereference the anonymous function reference `$h` returned by the `compose` function. We could have printed the result directly using a dereference block.

```
print "compose", &{compose(\&times2, \&add)}(3, 4);
```

16.3. More Features

16.3.1. Input and Output

Perl uses a different scalar data type for input and output: file handles. Besides the three standard file handles STDIN, STDOUT, and STDERR, you can create your own by opening a file.

```
open(TEST, ">test.dat");
print TEST "Hello World";
close(TEST);
```

The `open` function takes the symbol of the file handle as the first argument and creates the associated new file handle as a side effect. The second argument describes which file to open and how to open it. The syntax is derived from the UNIX shell. A greater-than symbol opens a file for output, the shift symbol >> opens a file in append mode. Perl's print functions can write to different output streams, the default one being STDOUT. If the stream is specified, it is placed between the name of the `print` function and the argument list. The stream is not an additional argument (no comma between the stream and the following arguments). It is really a special syntax just for specifying an output stream.

The return value of `open` indicates the success or failure of the operation. The idiom you'll find in many Perl programs is the combination of the opening of a file and the `die` function which exits the program if the operation failes.

```
open(TEST, ">test.dat") || die "Could not create test.dat";
print TEST "Hello World";
close(TEST);
```

Once a file handle has been created, it can be used like any other scalar value. It can be assigned to variables and passed to functions.

```
sub hello {
    my $out = shift;
    print $out "Hello again";
}
```

```
open(TEST, ">test.dat") || die "Could not create test.dat";
hello(TEST);
close(TEST);
```

16.3.2. Special Variables

A Perl program can use a large number of predefined variables. You need the environment variables of the operating system context in which the script is running? They are readily available in the special hash variable `%ENV`. The command line arguments of your script are contained in the `@ARGV` array. The process id of the Perl process running your script or the name of your script file? They are found in the variables with the fancy names `$$` and `$0`, respectively. All in all there are a few dozens of special variables defined in Perl. Some of them customize the behavior of Perl functions, some of them are set as side effects of functions. We have already used the special variables `$,` and `$\` which control how the `print` function separates output fields and records.

16.3.3. Packages and Modules

Packages are Perl's namespace mechanism. When defining a symbol (variable, function), the symbol is always placed in the current package. You can refer explicitly to a symbol in a package by preceding the symbol's name by the package qualifier consisting of the package name and two colons. Up to now, we have only used the default package `main`. We could therefore refer to this package using the `main::` prefix.

```
$x = "Hello World";
print $main::x;
```

Perl's `package` statement switches to a different package. It can occur anywhere in the code and causes the current package to be changed to the new specified package until the end of the block or a new `package` statement.

```
sub hello {"hello-main"; }
{
    package A;
    print hello();
    sub hello {"hello-A"; }
    package B;
    print hello();
    sub hello {"hello-B"; }
}
print hello(), main::hello(), A::hello(), B::hello();
```

```
-> hello-A
hello-B
hello-main hello-main hello-A hello-B
```

Within a package, the call of the `hello` function without package qualifier refers to the function defined in the package, even if the function is defined after the call. The same could be demonstrated with global variables, but since the `strict` mode enforces fully qualified global variable names, this does not make sense anymore.

A Perl module is package defined in a file whose name is the package name with the `.pm`. Here is a tiny module called `Sample` and stored in a file `Sample.pm`.

```
package Sample;

BEGIN { print "Loading module Sample"; }
END { print "Unloading module Sample"; }

sub hello { print "Hello", $_[0]; }

1;
```

After the package declaration, we define a constructor and a destructor for the module. These functions have the special names `BEGIN` and `END` (an awk heritage) and are called before loading the module and right before unloading it, respectively. Note that the `sub` keyword of these functions can be omitted. Besides these special functions, we only define one more function `hello` to be used by clients of the module. The expression `1;` at the end of the module makes sure that the module returns a true value when loading it. To use the module (e.g., in our sample script `sample.pl`), we have to load it using the `require` command.

```
require Sample;
Sample::hello("World");

-> Hello World
```

We can then use the module's package as if it was defined in the script.

16.3.4. Object Oriented Perl

Perl uses its packages and modules to add object oriented features to the language. Object are references which have been "blessed" with a package. The package plays the role of the class definition, its functions are the methods. Class methods are functions which obtain the package (i.e., class) as the first argument, instance methods are functions whose first argument is the instance. Here is a first example.

```
package Person;

sub new {
    my $class = shift;
    my ($name, $age) = @_;
    my $self = { name => $name, age => $age};
    return bless $self, $class;
```

```
}

sub name {
    my $self = shift;
    $self->{name};
}

package main;

my $person = Person->new("Homer", 55);
print ref($person), $person->name();

-> Person Homer
```

We create a new package called `Person` (it does not have to be in a new module) and define two methods. The constructor `new` is a class method creating new instances of `Person` class. We use a hash reference (the most common case) for the object `$self` and set the name and age entries as supplied to the constructor. The next step is the magic blessing of this hash reference. It links the reference to the class so that Perl can find its method (i.e., the functions of the attached package). Next, we define a simple getter method for the name. It demonstrates how the instance is passed to the instance method as the first argument. By convention this argument is called `$self` like in Smalltalk and Python.

The only new syntax is the call of the class and instance methods using the arrow `->`. Doing so lets Perl pass the class and instance implicitly as the first argument to the respective functions of the package implementing the class. Alternatively, we could have done this ourselves.

```
$person = Person::new("Person", "Homer", 55);
print Person::name($person);
```

The method call supported by the new syntax implicitly uses the package attached to the reference `$person` during the "blessing". It then tries to find the method in this package (and its base class packages as we will see below) and calls the function with the instance as the first argument.

With the knowledge about the internal structure of our `Person` object, we could have accessed the `name` directly as `$person->{name}`, but this would have violated the encapsulation of the class. Encapsulation is not enforced by Perl, but merely a convention. Use the provided methods only and do not exploit knowledge about the structure of a class.

Inheritance is realized by putting the names of the base class packages into a special global array called `ISA`.

```
package Employee;

@Employee::ISA = qw(Person);

sub new {
    my $self = Person::new(@_);
    $self->{no} = $_[3];
```

```
    return $self;
}

sub no {
    my $self = shift;
    $self->{no};
}

package main;

my $employee = Employee->new("Homer", 55, 12345);
print ref($employee), $employee->name(), $employee->no();

Employee Homer 12345
```

In the `Employee` constructor `new`, we first call the `Person` constructor and then add the additional attribute to the blessed hash table.

16.3.5. Function Prototypes

16.4. Discussion

Perl marks an extreme of all the languages covered in this book. While some languages started with a clear set of powerful ideas (Lisp, C) or were designed carefully during years of research (ML, Haskell), Perl looks like a collection of solutions for a large number of small concrete problems. Perl's main characteristic is the context sensitivity which is related to Perl's use of funny symbols and which is always good for surprises. In this sense, it also marks the opposite of a pure functional language where everything has a well defined meaning independent of the context. Perl applications often are extremely stable software: Once you have created a large Perl application, it will be hard to find somebody willing to change it (including yourself).

References

The book on Perl is [WALL96]> by Larry Wall and two of his co-workers (the first edition edited by Tim O'Reilly himself). It presents the language in a rather casual way and gives a lot of background information (why Perl looks the way it does). The nutshell book [SIEVER99]> is a good desktop reference covering the most important modules including the interface to the GUI library Tk.

Larry Wall, Tom Christiansen, and Randal L. Schwartz, 1-56592-149-6, O'Reilly, 1996, *Programming Perl, Second Edition*.

Ellen Siever, Stephen Spainhour, and Nathan Patwardhan, 1-56592-286-7, O'Reilly, 1999, *Perl in a Nutshell*.

Tom Christiansen and Nathan Torkington, 1-56592-243-3, O'Reilly, 1998, *Perl Cookbook*.

Sriram Srinivasan, 1-56592-220-4, O'Reilly, 1997, *Advanced Perl Programming*.

Randal Schwartz, Tom Christiansen, and Larry Wall, 1-56592-284-0, O'Reilly, 1997, *Learning Perl*.

Chapter 17. Haskell

Haskell was designed by a committee as a standard pure functional language during the late 1980's. It is based on ML and Miranda (from which it inherits its lazy evaluation semantics). The first definition was defined in 1990, and the current standard Haskell98 is considered a stable foundation of the language. By now, Haskell has become the most popular pure functional language. Looking at it after the ML chapters, it appears as a "purer" version of ML.

17.1. Software and Installation

We use the Hugs (http://www.haskell.org/hugs/) implementation of Haskell98 on Windows and Linux for our experiments. On Windows, it comes with an installer and a small graphical development environment `winhugs.exe` which can be used to enter code interactively as we have done before. The window also contains a browser for the classes and modules (see below). Starting Hugs loads the standard module `Prelude` and then waits for our commands at the prompt.

```
ahohmann@kermit:~$ hugs
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: November 2003  _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Prelude>
```

As an alternative, all examples can be run using the Glasgow Haskell Compiler (http://www.haskell.org/ghc/) system, or GHC for short. It also comes with an interpreter called `ghci` which works similar to Hugs. In face, GHC is the more complete Haskell98 implementation including some standard library functions missing in Hugs (which probably have been added to Hugs by the time you are reading this).

17.2. Quick Tour

17.2.1. Expressions

Entering our first examples, we are happy not to encounter any surprises. We can print our favorite message and evaluate arithmetic expressions as if we were using Python.

```
Prelude> "Hello World"
"Hello World"
Prelude> print "Hello World"
"Hello World"
```

If we switch on the type printing in the options menu (or using the `+t` option with the command line interpreter), Hugs responds like the ML systems with the value and the type of an expression.

```
Prelude> "Hello World"
"Hello World" :: String
Prelude> 1.5
1.5 :: Double
```

How does Hugs perform as a calculator? Since Haskell belongs to the "syntactically rich" function languages, it supports arithmetic operators that behave like they are supposed to.

```
Prelude> 3 + 4 * 5
23 :: Integer
Prelude> 1.5 - 2
-0.5 :: Double
Prelude> mod 5 3
2 :: Integer
Prelude> mod 3 3 + 1
1 :: Integer
```

Note that, in contrast to ML (and Ocaml with its "dotted" operators such as `+.`), we can mix integers and floating point numbers in one expression. Haskell allows operators and functions to be overloaded for different argument types.

Functions such as `mod` are applied by just putting them in front of their arguments without any special syntax for the argument list. In fact, Haskell does not know the concept of multiple arguments. Every Haskell function is called with exactly one argument. The expression `mod 5 3` actually consists of two function calls. First the `mod` function is applied to the single argument `5` resulting in a new function which is then applied to the single argument `3`. The expression is therefore equivalent to `(mod 5) 3`.

```
Prelude> (mod 5) 2
1 :: Integer
```

The infix notation for operators is just syntactic sugar, and we can switch back and forth between operator and function syntax by enclosing the operator in parentheses or the function in back quotes.

```
Prelude> (+) 4 5
9 :: Integer
Prelude> 5 ‘mod‘ 2
1 :: Integer
```

Being able to treat operators as functions makes it easy to use them as arguments to higher order functions as we will see below.

If we are interested in just the type of an expression (and not its value), we can discover it with the `:type` command. This is of particular interest for functions, since they do not know how to display themselves.

```
Prelude> :type "Hello"
"Hello" :: String
Prelude> :type mod
mod :: Integral a => a -> a -> a
```

The first part `Integral a =>` of the function type tells us that `a` is a type which behaves like an integer. In other words, it must belong to the type class `Integer`. We will explain these concepts later in detail.

The right hand side `a -> a -> a` confirms that functions have one argument only. The type expression has to be read as `a -> (a -> a)` and tells us that `mod` is a function with an argument of type `a` returning a function which again takes an argument of type `a` and returns a value of type `a`. If we apply `mod` to the single argument 5, we obtain a new function of type `Integral a => a -> a`.

```
Prelude> :type mod 5
(5 ‘mod‘) :: Integral a => a -> a
Prelude> (mod 5) 2
1
```

Note that the function application is left associative with `f x y` being equivalent to `(f x) y`, whereas the arrow of the function types is right associative with `a -> b -> c` being equivalent to `a -> (b -> c)`).

As a functional language, Haskell has conditional expressions (rather than the conditional statements we are used to from imperative languages). Here is the if-then-else expression:

```
Prelude> if 1 < 2 then "ok" else "not ok"
"ok"
```

Since the result of the expression has to be well defined (and typed), the `else` part is required and both, `then` and `else` part have to be of the same type.

The other conditional expression if the `case` contruction. We can not show this directly on the Hugs command line, because Hugs does not allow multi-line commands. Instead, we have to put the definitions in a file and load this file using the `:load` command. We create a file called `test.hs` and enter the following code:

```
x = 5
y = case (x) of
  1 -> "one"
  2 -> "two"
  _ -> "more"
```

Each Haskell file consists of a list of definitions, in our case just two. We first bind the integer 5 to the symbol `x`. The next definition contains the `case` expression we were interested in. Depending on the value of `x`, we bind `y` to an appropriate string.

The `case` expression consists of the expression to be checked between the two keywords `case` and `of`, followed by the list of alternatives, each mapping a pattern on the left hand side to an expression on the right hand side of the array operator. A pattern can be a single value such as `1` or `2`, or a variable such as the dummy variable _ which is typically used for the default case. As we will see later, a pattern can also include a constructor such as a tuple or the list constructor `:`.

How does Haskell know that the four lines of the `case` construction belong together? Similar to Python, Haskell uses the layout to decide where a definition ends. Haskell's *offside rule* states that a new definition starts at the same indentation level or to the left of the start of the definition.

To use these definitions, we have to load the file from the Hugs interpreter using the `:load` command (all Hugs commands start with a colon).

```
Prelude> :load test.hs
Reading file "test.hs":

Hugs session for:
C:\Program Files\Hugs98\lib\Prelude.hs
test.hs
Main> y
"more"
```

Note that the prompt has now changed to `Main>` showing the name of the default module (since we have not specified any module in our file).

We can also separate the alternatives of the `case` with semicolons in a single line, but this style is much less readable and therefore hardly ever used.

```
Prelude> case (3) of 1 -> "one"; 2 -> "two"; _ -> "more"
"more" :: [Char]
```

17.2.2. Functions

Before we dive into types and type classes, we will define a few simple functions of our own. Again, we can not do this right away in the Hugs shell, but have to put the definitions our test file `test.hs`.

```
times2 x = 2*x

fac 0 = 1
fac n = n * fac(n-1)
```

We then reload the file and test the functions:

```
Main> :reload
...
Main> times2 55
110 :: Integer
Main> fac 5
120 :: Integer
```

Looking at the code, this is probably the shortest form possible for defining the `times2` function. It also explains why we can't define the function on the command line. There is no special keyword such as `def` or `val` introducing the definition, just the plain equation.

The `fac` functions shows Haskell's minimal syntax for pattern matching and recursive functions. We just write down the defining equations for the different patterns. A pattern can be a single value (such as `0` in our example), a variable (such as `n`), or some constructor expressions for a tuple, list, or user defined data type (as we will see further down).

Often patterns are not enough to distinguish the different cases of a function definition. In this case, we can define the function using so-called guards which allow for arbitrary conditions depending on the arguments. As an example, we define the `sign` function in our test module.

```
sign x | x < 0   = -1
       | x == 0  = 0
       | x > 0   = 1
```

After changing the module file, we can reload the module from the shell (without leaving the interpreter) using the `reload` command.

```
Main> :reload
...
Main> sign -55
-1 :: Integer
Test: sign 1.5
1 :: Integer
```

Of course, we could have also used the `if` expression to achieve the same effect in a less readable manner.

```
sign1 x = if x < 0 then -1 else if x > 0 then 1 else 0
```

We can't define named functions on the command line, but Haskell does understand lambda expressions:

```
Prelude> (\x -> 2*x) 4
8 :: Integer
```

Here, the backslash is supposed to look like a lambda, and the definition uses an arrow instead of the equal sign in the normal function definition above.

Higher order functions (functions taking function arguments and possibly returning new functions) following Haskell's mathematical style: just write down the defining equation. Here is the definition of the composition of two function (one function applied to the result of the other):

```
compose f g x = f (g x)
```

After entering this definition in our test module (`test.hs`), we can reload the module and apply the new higher order function.

```
Main> :reload
...
Main> :type compose
compose :: (a -> b) -> (c -> a) -> c -> b
Main> compose (5+) (10+) 5
20 :: Integer
```

Note that Haskell automatically derives the most general type for the `compose` function. Alternatively, we could have used the lambda syntax to clearly indicate that we are returning a new function.

```
compose f g = \x -> f (g x)
```

It probably does not come as a surprise that the composition is part of the standard library in form of the `.` operator.

```
Main> :type (.)
(.) :: (a -> b) -> (c -> a) -> c -> b
Main> ((5+) . (10+)) 5
20 :: Integer
```

17.2.3. Collections

You may have got the impression already that Haskell is a "mathematical" programming language. It shouldn't be surprising that tuples are a natural and common type (or better type constructor) in Haskell.

```
Main> (1, 1.5)
(1,1.5) :: (Integer,Double)
Main> fst (4, 5.5)
4 :: Integer
Main> snd (4, 5.5)
5.5 :: Double
```

We construct a tuple following the mathematical notation (like in Python). The two functions `fst` and `snd` give us the first and second element of the tuple, respectively. Note that these function calls look as

if we apply a function to multiple arguments, but we are in fact applying the functions to single arguments of type tuple.

Like infix operators, the tuple notation is just syntactic sugar for the application of the "tuple functions" `(,)`, `(,,)`, and so forth.

```
Prelude> :type (,)
(,) :: a -> b -> (a,b)
Prelude> :type (,,)
(,,) :: a -> b -> c -> (a,b,c)
Prelude> (,) 1.5 2
(1.5,2)
Prelude> (,,) 1.5 2 "a"
(1.5,2,"a")
```

List literals look like in Python, but must contain a single type of elements.

```
Main> [1, 2, 3]
[1,2,3] :: [Integer]
Main> [1, "bla"]
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : [1,"bla"]
*** Type       : Num [Char] => [[Char]]
```

Note that the notation for list types reflects list literals by enclosing the element type in square brackets and that strings are just lists of characters, that is, have type `[Char]`.

Haskell uses a single colon as the list constructor (`cons` in Lisp, double colon in ML) and the double plus `++` to concatenate two lists. We can access individual elements with the `!!` operator.

```
Prelude> 1:2:3:[]
[1,2,3] :: [Integer]
Prelude> [1, 2, 3] ++ [4, 5, 6]
[1,2,3,4,5,6] :: [Integer]
Prelude> "Hello " ++ "World"
"Hello World" :: [Char]
Prelude> [1,2] !! 1
2
Prelude> [1,2] !! 0
1
```

As a generalization of the familiar head and tail functions, Haskell offers a symmetric set of functions for accessing the beginning or end of a list: `head` and `last` give the first and last element, respectively, `tail` and `init` all but the first or last element, and `take` and `drop` give you the first so many elements or the rest of the list.

```
Prelude> head [1, 2, 3]
1
Prelude> last [1, 2, 3]
```

```
3
Prelude> init [1, 2, 3]
[1,2]
Prelude> tail [1, 2, 3]
[2,3]
Prelude> take 2 [1, 2, 3, 4]
[1,2]
Prelude> drop 2 [1, 2, 3, 4]
[3,4]
```

We can also check if a value is contained in a list (or not), reverse it, compute the sum or product, combine two lists into a list of pairs (zip) or vice versa (unzip). However you would like to process a list, Haskell has most likely a function in the Prelude which does the right thing.

```
Prelude> elem 5 [1, 2, 3]
False :: Bool
Prelude> elem 2 [1, 2, 3]
True :: Bool
Prelude> reverse [1, 2, 3]
[3,2,1] :: [Integer]
Prelude> sum [1, 2, 3]
6 :: Integer
Prelude> product [1, 2, 3, 4]
24 :: Integer
Prelude> zip [1, 2, 3] ["a", "b"]
[(1,"a"),(2,"b")] :: [(Integer,[Char])]
Prelude> unzip [(1,"a"),(2,"b")]
([1,2],["a","b"]) :: ([Integer],[[Char]])
```

Besides these "normal" list functions, a functional language such as Haskell has of course a number of higher order functions operating on the list as a whole.

```
Prelude> map (\x -> 2*x) [1, 2, 3]
[2,4,6] :: [Integer]
Prelude> foldr (+) 10 [1, 2, 3]
16 :: Integer
Prelude> filter odd [1, 2, 3, 4, 5]
[1,3,5] :: [Integer]
```

And guess where Python’s list comprehensions come from? In Haskell, they look just like the mathematical definition. The "element of" sign is `<-` (somewhat looking like an epsilon).

```
Prelude> [2*x | x <- [1,2,3] ]
[2,4,6] :: [Integer]
Prelude> [2*x | x <- [1,2,3], odd x ]
[2,6] :: [Integer]
Prelude> [2*x + y | x <- [1,2,3], odd x, y <- [10,20] ]
[12,22,16,26] :: [Integer]
```

On the left hand side of the bar we have an expression whose results are put into the generated list. On the right hand side of the bar we can put any combination of generators (such as `x <- [1,2,3]`) and tests (boolean expressions such as `odd x`). Haskell iterates through the generators, checks the tests, and, in case the tests are true, places the result of the expression in the list.

Haskell also offers a number of means to generate lists. In the simplest case, we can use ellipses to generate a sequence of consecutive integers.

```
Prelude> [1..5]
[1,2,3,4,5] :: [Integer]
Prelude> [100..110]
[100,101,102,103,104,105,106,107,108,109,110] :: [Integer]
```

We can also vary the step size and generate decreasing numbers.

```
Prelude> [1,4..20]
[1,4,7,10,13,16,19] :: [Integer]
Prelude> [10,8..0]
[10,8,6,4,2,0] :: [Integer]
```

We can even generate infinite sequences. Since Haskell's `Integer` type is unlimit (ok, limited only by the memory constraints), these sequences go on forever unless we interrupt the program by pressing the stop button or Ctrl-C.

```
Prelude> [0,10..]
[0,10,20,30,40,50,60,70,80,90,100,110,120,130,{Interrupted!}
```

Studying the Prelude, we discover more functions generating sequences such as replicate (repeating the same value a specified number of times) and iterate (applying a function over and over again). The `iterate` function again generates an infinite sequence. We can use the `take` function to see the first so many elements without having to interrupt the program.

```
Prelude> replicate 10 "a"
["a","a","a","a","a","a","a","a","a","a"] :: [[Char]]
Prelude> take 10 (iterate (2*) 1)
[1,2,4,8,16,32,64,128,256,512] :: [Integer]
```

The infinite lists touch another fundamental property of Haskell: lazy evaluation. Each expression is evaluate only if it is needed and only once if it is needed multiple times. In the case of infinite lists, the list is not constructed before it can be used (this would take a little bit too long). Instead, the next value is generated exactly when it is needed.

17.2.4. Types and Classes

In contrast to ML, we can apply the functions `times2` and `sign` defined above to integers as well as floating point numbers.

```
Main> times2 5.5
11.0 :: Double
```

This makes us curious about the type of `times2`. In ML, the compiler would have inferred from the integer constant and the multiplication that the function is an integer function. Haskell is smarter and uses less restrictive types.

```
Main> :type times2
times2 :: Num a => a -> a
Main> :type (*)
(*) :: Num a => a -> a -> a
```

Our new function is of type `Num a => a -> a`. What is this supposed to mean? The right hand side `a -> a` tell us that it is a function mapping a value of some type `a` to the same type. The left hand side is a type restriction. It tells us that `a` has to be a numeric type, that is, a type of class `Num`. Similarly, the type of the multiplication is a (curried) function with two arguments, both of the same numeric type and resulting in the same numeric type. How is `Num` defined? The `:info` command tells us all about it.

```
Main> :info Num
-- type class
infixl 6 +
infixl 6 -
infixl 7 *
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  fromInt :: Int -> a

-- instances:
instance Num Int
instance Num Integer
instance Num Float
instance Num Double
instance Integral a => Num (Ratio a)
```

We see that a numeric type has to support the basic arithmetic operators and conversion functions from integers. The command also shows the currently available types of class `Num`. There is more information hidden in the class statement `class (Eq a, Show a)`. The class `Num` is derived from `Eq`. In plain words, each numeric type must also be an equality type.

```
Main> :info Eq
-- type class
infix 4 ==
infix 4 /=
```

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

-- instances:
instance Eq ()
instance Eq Char
...
```

Checking the class definition, we find out that an equality type has to support - surprise, surprise - the equality and inequality operators.

17.2.5. User Defined Types

Haskell is not restricted to the build-in types and classes like `Integer` and `Num`. They are nothing special and we can define our own types and classes and use them as if they were part of the standard modules. Let's start with our own types. First, we can define shortcuts for existing types. This doesn't give us anything new besides a more readable way to deal with complex type constructors.

```
type ShopItem = (String, Float)
type Basket = [ShopItem]

shopSum :: Basket -> Float
shopSum [] = 0.0
shopSum ((_,i):xs) = i + shopSum xs
Main> :reload
...
Main> shopSum2 [("apple", 1.5), ("pear", 2.5)]
4.0 :: Float
```

We could have defined the function directly using the type expression `[(String, Float)]` in the type declaration for `shopSum`, or, even shorter, without any type declaration.

```
shopSum2 [] = 0.0
shopSum2 ((_,i):xs) = i + shopSum xs
```

Having seen ML, user defined types in Haskell are nothing new. As with function definitions, we can't type them interactively, but have to put the definitions in a module. We'll use our test module again. We see the familiar data constructors, tuple types, union types, and recursive types.

17.2.6. Imperative Programming

Haskell touts itself as a *pure* functional language, and for now we have restricted ourselves to pure definitions without any side effects. In the real world, however, we would like to read a file or print a message among other useful actions causing side effects.

How can we introduce imperative features without compromising the functional core of the language? The basics idea is to consider actions as values of special types. These "action values" can be combined in a controlled fashion to form larger actions. The simplest example takes us back to our favorite message.

```
Prelude> print "Hello World"
"Hello World"
 :: IO ()
```

This looks like a pure statement taken from an imperative language (it works in Perl and Python, for example). The only surprising thing is the type `IO ()` of the expression. The expression `print "Hello World"` is a I/O action. The interpreter happens to treat actions by executing them resulting in the output "Hello World" (as the side effect of the action).

The situation get clearer when looking at the type of the function `print`.

```
Prelude> :type print
print :: Show a => a -> IO ()
```

The function `print` takes a showable value (that is, of type class `Show`) and returns an I/O action of type `IO ()`. `IO` is a built-in type constructor for input and output actions. The type `IO a` is an I/O action returning a value of type `a`. The function `getLine`, for example, is of type `IO String`, since is corresponds to an action resulting in a string.

```
Prelude> :type getLine
getLine :: IO String
Prelude> getLine
1234
"1234" :: IO String
```

Since the `print` action does not return anything, it is of type `IO ()` with the unit type `()`.

We can work with actions like with any other value, for example, store them in collections or pass them to functions.

```
Prelude> [print "Hello", print "World"]
[<<IO action>>,<<IO action>>] :: [IO ()]
Prelude> [print "Hello", print "World"] !! 1
"World"
 :: IO ()
```

```
Prelude> (print "Hello", getLine)
(<<IO action>>,<<IO action>>) :: (IO (),IO String)
Prelude> snd (print "Hello", getLine)
asdf
"asdf" :: IO String
```

A first simple way to combine actions is an `if` expression:

```
Prelude> if 1 < 2 then print "Hello" else print "World"
"Hello"
 :: IO ()
```

Another natural operation for actions is sequencing (which is at the core of imperative programming). Haskell's `do` construct lets us combine multiple actions so that the resulting action executes the combined actions in sequence.

```
Prelude> do print "Hello"; print "World"
"Hello"
"World"
 :: IO ()
```

Here is a more elaborate example using `do` to recursively perform an action multiple times.

```
times n action
 = if n <= 1
      then action
      else do action
              times (n-1) action
```

This way, we can easily print our message many times.

```
Main> times 3 (print "Hello World")
"Hello World"
"Hello World"
"Hello World"
 :: IO ()
```

What is missing until now is a link between I/O actions type `IO a`) and the underlying values of type `a`. Within a `do` block, we can bind the result of an action to a symbol with the `<-`. Here is an action which echos a line read from standard input.

```
echo = do line <- getLine
          print line
```

Superficially it looks like an assignment, but it is just giving the result of an action (here `getLine`) a name which can be used in the next actions (here `print`) of the `do` construct.

```
Main> echo
```

```
asdf
"asdf"
 :: IO ()
```

Note that the `<-` is restricted to `do` blocks. This way, the results of actions can not spoil the functional program.

Going in the opposite directory, the `return` function turns any value into an I/O action which does nothing (no side effects) but return the original value.

```
Prelude> return "Hello" :: IO String
"Hello" :: IO String
```

This does not accomplish much by itself, but is sometimes needed to manipulate values within actions. As an example, we have a look at an extended `echo` program which echos line after line until the end of file is reached. First, we need a `while` loop which runs an action until a test (of type `IO Bool`) becomes true.

```
while test action
  = do res <- test
       if res then do action
                      while test action
              else return ()
```

Note the `return ()` expression which is the I/O action which does absolutely nothing, but is needed as the `else` expression.

Next, we apply the while loop to the end-of-file condition. The built-in action `isEOF` is of type `IO Bool`, but unfortunately returns the opposite of the test we need in the while loop. Therefore, we have to create a new test action which gets the result and returns the negated value.

```
import IO
echo
  = while (do result <- isEOF
              return (not result))
          (do line <- getLine
              print line)
```

[1]

Alternatively, we can first define a new function `ioNot` which negates I/O actions

```
ioNot test
  = do result <- test
       return (not result)
```

and then use it in the condition of the while loop.

```
echo2
  = while (ioNot isEOF)
          (do line <- getLine
              putStrLn line)
```

So, what actually is this type constructor `IO`? What makes it special so that we can use it with sequencing (`do`), naming of results (`<-`) and `return` constructs? Checking the type of the `return` function we see that it is defined for any type `a b` where the type constructor `a` is a *Monad*.

```
Main> :type return
return :: Monad a => b -> a b
```

Similarly, Haskell's type inference determines that the type of our `times` function relies on a monad.

```
Main> :type times
times :: (Ord a, Num a, Monad b) => a -> b c -> b c
```

The number of times we want to iterate must be an ordered number (type classes `Ord` and `Num`), and the action to be repeated is of type `b c` where the type constructor `b` must be a monad again.

A monad? What is this? No worries, we will not get into its origin in category theory (also known as "abstract nonsense"). In practice, it is enough to know that monads generalize the concept of I/O actions we have seen above to all the "action types" for which the "action operations" such as sequencing and `return` make sense.

17.3. More Features

17.3.1. Modules and Visibility

For now, we have used the definitions of the Prelude and defined our own in a test file which by default became the `Main` module. Haskell supports large scale development using modules and explicit export and import expressions. As a first step, we can give our test module `test.hs` a new name using the `module` construction. To be able to load the module by its module than (rather than its file name), we store the code in a file `Test.hs` corresponding to the module name.

```
module Test where

fac 0 = 1
fac n = n * fac(n-1)
```

When we now load the module, the prompt changes to `Test`.

```
Prelude> :load Test
Reading file "Test.hs":

Hugs session for:
/usr/share/hugs98/lib/Prelude.hs
Test.hs
Test> fac 5
120
```

We can now import the module `Test` from another module.

```
module NewTest where
import Test
f = fac
```

The `import` instruction makes all the public symbols of the imported module available in the importing module. Using the term "public symbols", it is clear that there must be a way to restrict the symbols exported by a module. All we need to do is place the list of exported symbols behind the name of the module.

```
module Test (x, y) where
x = 55
y = 66
z = 77
```

In this (trivial) example, only the symbols `x` and `y` are exported, whereas `z` is only visible by the module itself. Trying to use `z` in another module importing `Test`

```
module NewTest where
import Test
a = x
c = z
```

leads to an error:

```
Prelude> :load NewTest
Reading file "NewTest.hs":
Reading file "Test.hs":
Reading file "NewTest.hs":
Dependency analysis
ERROR NewTest.hs:4 - Undefined variable "z"
```

Haskell's `import` directive has a number of other options such as renaming the imported module, importing just a few symbols, or enforcing the use of qualified names (such as `Test.fac`). The Haskell report explain the module system in detail.

17.3.2. Lazy Evaluation

17.4. Discussion

Haskell introduces a whole range of features such as type classes, lazy evaluation, and monads for imperative programming. Judging from the number of libraries available for Haskell (especially the Glasgow Haskell Compiler GHC), we can go a long way with this unique programming model. It remains to be seen if these concepts will be adopted by mainstream programming languages (or if Haskell becomes one of them). As a first indication we saw that Python definitily benefits a lot from the adoption of Haskell's list comprehensions.

Haskell's treatment of operators combines the "normal" infix notation and strong typing with the advantages of Scheme when passing operators as (function) values to other functions.

References

Simon Thompson's Haskell book [THOMPSON99]> is a very readable and up-to-date introduction to Haskell.

Simon Thompson, 0-201-34275-8, Addison-Wesley, 1999, *Haskell - The Craft of Functional Programming: Second Edition*.

Notes

1. If may have trouble running this program with Hugs, since the Hugs version I've used did not implement the `isEOF` action yet.

Chapter 18. Python

Python was created by Guido van Rossum in the late 1980s and first published in 1991.

18.1. Software and Installation

There are two implementation of Python, the original one in C (also known as C-Python) and Jython, the implementation on top of the Java virtual machine. We will be using C-Python for the examples, but the two implementations are very close to each other (with Jython slightly lagging behind regarding new features) so that most examples should work as well using Jython. To install Python on Windows, just download the latest version from http://www.python.org and run the installer. This will install Python including the documentation and the little integrated development environment IDLE on your computer.

18.2. Quick Tour

Python's interactive programming environment helps us to explore the language by example. I'm using the interactive shell which is part of IDLE, the IDE which comes with the Python installation. Starting it shows the following output that invites us to enter Python commands at the prompt >>>.

```
Python 2.2.2 (#37, Oct 14 2002, 17:02:34) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>>
```

18.2.1. Expressions and Variables

Let's answer this friendly invitation with the unavoidable "Hello World!".

```
>>> print "Hello World"
Hello World
```

This was easy (how many lines do you need in Java?). In fact, the interactive shell prints the return value of the expressions we enter so that the plain "Hello World" is enough.

```
>>> "Hello World"
'Hello World'
```

Strings use single or double quotes as delimiters or "triple double quote" if the string contains newlines characters.

```
>>> """Hello
```

```
World"""
'Hello\nWorld'
```

All these forms of string literals behave the same. What about some calculations?

```
>>> 4*5+2**3
28
```

The only thing one has to know here is that ** means "to the power of". Can we save the result for later use?

```
>>> x = 4*5+2**3
>>> x
28
```

Python's variables don't need to be declared in advance. A variable is automatically created the first time a value is assigned to it. Referring to an undefined variable creates an error.

```
>>> y
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in ?
    y
NameError: name 'y' is not defined
```

All variables are local to the current file or function they are defined in unless declared otherwise (see below).

If you ask why scripting languages are so much more productive than more "traditional" languages, you are often referred to the powerful built-in string processing and collections. We can compare strings, concatenate and repeat them, and extract individual characters or substrings with straight-forward operators.

```
>>> "blah" == "blub"
0
>>> "blah" < "blub"
1
>>> "blah " + "blub"
'blah blub'
>>> 3 * "blah "
'blah blah blah'
>>> "Hello"[2]
'l'
>>> "Hello"[-1]
'o'
>>> "Hello"[1:4]
'ell'
```

Note that Python's indexes always start at zero, and negative indexes count from the end. We extract substrings with the slice operator which contains the limits, separated by a colon, in square brackets. Python ranges always uses the semantics of a right half open interval, that is, including the lower bound and excluding the upper bound. Once you get used to it, this turns out to be a useful convention, since you can always use the length of a structure as the upper bound.

```
>>> x = "blah"
>>> print x, x[0:len(x)], x[0:], x[:len(x)], x[:]
blah blah blah blah blah
```

Besides these operators, there is a large number of functions and methods manipulating strings.

```
>>> len("blah")
4
>>> "blah".upper()
'BLAH'
>>> "blahblah".replace("ah", "ub")
'blubblub'
>>> "blah".center(10)
'    blah    '
>>> "blahblah".find("ah")
2
>>> "blahblah".count("a")
2
```

This is the first example using method calls. The syntax is identical to method calls in most popular object based languages (C++, Java, JavaScript). The interpretation, however, is different. Calling a method is a two step process. First, we find the field whose name is the method.

```
>>> "blah".find
<built-in method upper of str object at 0x00824C98>
```

This field has to be a "callable object". Next, this object is "called" by passing the arguments in parentheses. Therefore, the parentheses of the method call are significant and can't be omitted (like, e.g., in Perl and Ruby). They make the difference between retrieving the method and actually calling it. We can perform the two steps explicitly using an intermediate variable to store the callable object.

```
>>> f = "blah".find
>>> f("la")
1
```

Python's API is not fully "object-oriented", although it is steadily moving in this direction. The length of a any collection (including strings) is still obtained with the `len` function although it could be a method of the collection. Most string manipulations are now methods of the string type (they were originally only available as functions in the string module).

The examples showed only a small subset of the available string methods. If you would like to see all of them, try `dir("")` which shows you the list of all methods a string has to offer.

18.2.2. Control Flow

Python supports the usual conditional and loop statements of procedural languages.

```
>>> i = 0
>>> while i<5:
        print i
        i += 1

0
1
2
3
4
```

Here we see one of the most debated characteristics of Python: there are no braces, no begin or end to denote the scope of the function. Instead the white space tells the compiler which statements belong together. Consecutive lines with the same identation comprise a block. It does not matter how the lines are indented (TABs or spaces, number of spaces), but the indentation has to be exactly the same for all lines in a block. Like it or not, it is at least a very compact way to define blocks of statements. In general, a complex Python statement such as a function definition, a class definition, or a control statement (if, while, exception handling) starts with a keyword followed by some expression, a colon, and an indented block. Here is a conditional statement.

```
>>> x = 50
>>> if x < 10:
        print "small"
elif x < 100:
        print "medium"
else:
        print "big"

medium
```

Here is another example showing a conditional statement nested in a while loop.

```
>>> i = 0
>>> while i<3:
if i%2 == 0:
print i, "is even"
else:
print i, "is odd"
i += 1
0 is even
1 is odd
2 is even
```

Python's most common loop is the `for` statement. A `for` loop uses an iterator to walk through a sequence of values. An iterator is an object which knows how to get the next element in the sequence and

when to stop. As an example, the standard string iterator walks through the characters of a string one by one.

```
>>> i = iter("ab")
>>> i.next()
'a'
>>> i.next()
'b'
>>> i.next()
Traceback (most recent call last):
  File "<pyshell#205>", line 1, in ?
    i.next()
StopIteration
```

We get the next element by calling the iterator's `next` method, and the iterator tells us to stop by throwing a `StopIteration` exception. The `for` loop assigns the result of the `next` method to the iteration variable until this exception is encountered.

```
>>> for i in iter("ab"):
        print i

a
b
```

The call to the `iter` function is optional. The `for` loop tries to find a suitable iterator automatically if it is not given one directly.

```
>>> for i in "ab":
        print i
```

We will see more examples in the context of collections and generators.

Python strictly distinguishes expressions from statements. Expressions compute values and statements such as `print`, `if`, or `while` control the program flow or cause other side effects. In Python, statements do not return any value and therefore can not be used in expressions. This is in strong contrast to the functional languages we will cover later where everything (including control statements) is an expression with a well-defined value. The newer scripting language Ruby follows the functional model as well. Programmers working with C-family languages will especially miss an equivalent of the "functional if" operator `?:` in Python. [1]

There is another difference to languages of the C family (an all other languages with proper lexical scoping). Python's control statements do not introduces new scopes. A variable defined in a block of a control statement is also visible outside.

```
>>> if x % 2 == 0:
        result = "even"
else:
        result = "odd"
```

```
>>> result
'even'
>>> for i in "ab": pass
>>> i
'b'
```

18.2.3. Collections

Python has three built-in collection types: list, map (dictionary) and tuple. A list is a sequence of values with fast (constant time) random access. In other languages, this kind of collection is often called a vector. A map associates values with keys. As a generalization of pairs, triples, and so forth, a tuple is sequence of fixed length. All collections can contain any kind of value, including other collections. Let's start with some list examples.

```
>>> l = [1, 2, "a", "b"]
>>> l[2]
'a'
>>> l[1:3]
[2, 'a']
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
>>> ["a", "b"] * 3
['a', 'b', 'a', 'b', 'a', 'b']
>>> len([1, "a", [2, 3]])
3
```

List literals are comma separated elements enclosed in square brackets. We recognize all the operators we have already used for strings. For lists, the subscript and slice operators can also be used to change the list.

```
>>> l = [1, 2, 3, 4]
>>> l[2] = 100
>>> l
[1, 2, 100, 4]
>>> l[1:3] = ["a", "b", "c"]
>>> l
[1, 'a', 'b', 'c', 4]
>>> del l[2]
>>> l
[1, 'a', 'c', 4]
>>> del l[2:]
[1, 'a']
>>> del l
>>> l
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in ?
    l
NameError: name 'l' is not defined
```

The only unusual syntax is the `del` operator which deletes the following object from its contained. In the example we use it to delete a list element, a slice, and the list variable itself.

Lists have a number of methods manipulating the list object in-place (also called destructive methods).

```
>>> l = [0, 1, 2, 3, 4]
>>> l.reverse()
>>> l
[4, 3, 2, 1, 0]
>>> l.sort()
>>> l
[0, 1, 2, 3, 4]
>>> l.extend(['a', 'b'])
[0, 1, 2, 3, 4, 'a', 'b']
>>> l.append('c')
>>> l
[0, 1, 2, 3, 4, 'a', 'b', 'c']
>>> l.pop()
'c'
>>> l.remove(4)
>>> l
[0, 1, 2, 3, 'a', 'b']
```

The `extends` method is the destructive version of the plus operator. The methods `append` and `pop` let us view a list as a LIFO stack. Note that all the destructive methods return `None`. This is a Python convention. Since nothing is returned, the methods can not be used in a context which assumes that the method returns a new changed object and leaves the original one unaltered. Keep in mind that the underlying implementation is an array and not a linked list which means that some methods might take longer for long lists than you expect, because elements have to be shifted or copied.

Dictionaries (also called maps) are not only crucial to many scripts, but also to Python's internal implementation. A map literal is a sequence of key-value pairs enclosed in curly braces. Key and value of each pair are separated by a colon. Keys and values can be of any type.

```
>>> m = {"John": 55, "Joe": [1, 2, 3]}
>>> m
{'John': 55, 'Joe': [1, 2, 3]}
>>> m["Joe"]
[1, 2, 3]
>>> m["Joe"] = 66
{'John': 55, 'Joe': 66}
>>> m
>>> del m["John"]
>>> m
{'Joe': 66}
>>> m["John"]
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in ?
    m["John"]
KeyError: John
```

```
>>> m.has_key("John")
0
```

Accessing elements in a map is like indexing a list, just with arbitrary keys instead of integers (and there are no slices).

The three methods `keys`, `values` and `items` gives us the keys, values, and key-value pairs as lists.

```
>>> print m.keys()
['John', 'Joe']
>>> m.values()
[55, [1, 2, 3]]
>>> m.items()
[('John', 55), ('Joe', [1, 2, 3])]
```

The same information can also be obtained more efficiently in the form of iterators with the `iterkeys`, `itervalues`, and `iteritems` methods. These methods will become useful when iterating through the entries in a map.

A data structure which is less common in other languages is the tuple. Tuple literals are comma separated values in parentheses (just like the arguments of a function). A trailing comma is allowed. It is mandatory for a one-tuple, since it distinguishes the one-tuple from a simple expression in parentheses.

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> t[1]
2
>>> t[1]
Traceback (most recent call last):
  File "<pyshell#181>", line 1, in ?
    t[1] = 55
TypeError: object doesn't support item assignment
>>> ("blah",)
('blah',)
>>> ("blah")
('blah')
```

Like strings, tuples can't be changed, they are immutable objects. But you may changed the objects contained in the tuple if they are mutable.

```
>>> t = (1, [])
>>> t[1].append("blah")
>>> t
(1, ["blah"])
```

Whenever it makes sense, Python automatically packs a sequence of comma separated values into a tuple and vice versa. This can, for example, be used combine multiple assignments into one.

```
>>> 1, 2
(1, 2)
>>> x, y = 1, 2
>>> x, y
(1, 2)
```

As we will see this feature also simplifies the iteration through maps and allows for multiple return values of functions.

How do we iterate throug a collection? The `for` loop walks through a list without the need to construct the loop explicitly.

```
>>> for i in ["a", "b", "c"]:
        print "i:", i
i: a
i: b
i: c
>>> i
'c'
```

Note that the loop variable is visible after the loop. Together with the automatic unpacking of pairs, we can easily iterate through a list of pairs.

```
>>> for a, b in [(1, 'a'), (2, 'b')]:
        print a, b
1 a
2 b
```

Combine this with the `items` method of a map and you get a convenient way to walk through the map's name-value pairs.

```
>>> m = {"John": 55, "Joe": 44}
>>> for key, value in m.items():
        print key, value

John 55
Joe 44
```

If we want to avoid the intermediate creation of the list, we can use the iterator instead.

```
>>> m = {"John": 55, "Joe": 44}
>>> for key, value in m.iteritems():
        print key, value
```

The default iterator gives us the keys of the map (and probably the most intuitive way to walk through the map).

```
>>> for key in m:
```

```
        print key, m[key]
```

18.2.4. Functions

Up to now, we have only used built-in functions, but eventually we will need our own. Here is an exciting one.

```
>>> def times2(x):
        "Multiply argument by two"
        return 2*x
>>> times2(5)
10
```

A function is defined with the `def` keyword followed by the name of the function, the parameter list, a colon and the indented function body. If the body starts with a string, the string is interpreted as the documentation of the function. Here the body just consists of the documentation string and the return statement. Unless left with an explicit return value, a function returns `None`, Python's equivalent of "nothing" or "undefined".

The interesting part is how this definition is handled by the Python interpreter. It creates a new function object and assigns it to the variable whose name is the name of the function. The function object is a first class object with its attributes and methods. We can, for example, ask the function for its name and documentation using the special attributes `__name__` and `__doc__`.

```
>>> times2
<function times2 at 0x0097D1E0>
>>> times2.__name__
'times2'
>>> times2.__doc__
'Multiply argument by two'
```

We can even add new attributes dynamically, for example, to add more meta information to the function such as permissions.

```
>>> times2.permissions = ["everybody"]
>>> times.permissions
['everybody']
```

Now that we have this function object, we can assign it to another variable, pass it to another ("higher order") function, and so forth.

```
>>> f = times2
>>> f(3)
6
>>> def printResult(f, x):
        print "%s(%d)=%d" % (f.__name__, x, f(x))
```

```
>>> printResult(times2, 5)
times2(5)=10
```

Higher order functions can often replace explicit loops. Python also supports a number of functions which are well-known for list oriented languages such as Lisp. As a first example, lists can be processed element-wise using the map function.

```
>>> def times2(x): return 2*x
>>> map(times2, [1, 2, 3])
[2, 4, 6]
>>> map(lambda x, y: x + y, [1, 2, 3], [2, 3, 4])
[3, 5, 7]
>>> map(lambda x, y: (x, y), [1, 2, 3], ["a", "b", "c"])
[(1, 'a'), (2, 'b'), (3, 'c')]
```

The `map` function if the first example of a higher order function (also called a functional), that is, a function which takes other functions as arguments. Since functions are first class objects in Python, passing functions to other functions is not different from passing any other kind of value. The last expression combines several lists into a list of tuples and can be more easily written using the built-in `zip` function:

```
>>> zip([1, 2, 3], ["a", "b", "c", "d"])
[(1, 'a'), (2, 'b'), (3, 'c')]
```

It is also possible to extract a sub-list using a filter:

```
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> filter(lambda x: x % 2 == 0, l)
[0, 2, 4, 6, 8]
```

As another example of list processing, you can recursively compute a value from a list using reduce.

```
>>> reduce(lambda x, y: x+y, [1, 2, 3, 4])
10
>>> reduce(lambda x, y: x + ", " + y, ["Joe", "John", "Mary"])
'Joe, John, Mary'
```

Like the `map` function, it takes a function and a list. The function is first applied to the first two elements of the list, then to the result of this computation and the third element, and so forth. Optionally, one can provide a start value so that the recursion starts by applying the function to the start value and the first value of the list.

```
>>> reduce(lambda x, y: x+y, [], 3)
3
>>> reduce(lambda x, y: x+y, [1], 3)
4
```

`Map` and `filter` are used less often in new programs because of a recent extension of the syntax for list literals known as list comprehensions. Remember the definition of sets from other set in your math class? Something like "all f(x) where x in X and x satisfies some condition"? Here is the Python version of this kind of list (instead of set) definition.

```
>>> [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

The expression constructs the list of squares of all even integers between zero and ten (not including). List comprehensions may combine multiple lists, e.g.:

```
>>> firstNames = ["John", "Joe", "Mary"]
>>> lastNames = ["Miller", "Smith"]
>>> [(f, l) for f in firstNames for l in lastNames]
[('John', 'Miller'), ('John', 'Smith'), ('Joe', 'Miller'),
 ('Joe', 'Smith'), ('Mary', 'Miller'), ('Mary', 'Smith')]
```

Two more features a C/C++ programmer misses when moving to Java are variable argument lists and default arguments. Python takes the C/C++ functionality one step further by allowing arguments to be passed by name.

```
>>> def f(s="blah", n=1): print n*s
>>> f("x", 5)
xxxxx
>>> f(n=2)
blahblah
>>> f(s="x")
x
```

Variable argument lists are declared with an asterisk and passed as a tuple to the function body.

```
>>> def f(s="blah", *args): print s, args
>>> f("blub", 1, 2, 3)
blub (1, 2, 3)
```

Similarly, keyword arguments can be passed as a generic argument map to a functions.

```
>>> def f(x, **kw): print x, kw
>>> f(x=1, y=2)
1 {'y': 2}
```

Putting it all together we can define a function with normal arguments, optional arguments, a variable argument tuple, and a keyword argument map.

```
>>> def f(x, y=5, *args, **kw):
        print x, y, args, kw
>>> f(1, 2, 3, a=4)
1 2 (3,) {'a': 4}
```

We can also do the opposite and call a function with an argument tuple and keyword argument list.

```
>>> def f(x, y=2, z=3): print x, y, z
>>> args = (100, 200)
>>> kw = {'z': 55}
>>> f(*args, **kw)
100 200 55
```

This feature comes in handy when passing arguments in a generic context from one function to another. As an application, we can now define the higher order function `compose` which combines two functions to a new function applying one function after the other.

```
>>> def compose(f, g): return lambda *args, **kw: f(g(*args, **kw))
>>> def times2(x): return 2*x
>>> def add(x, y): return x + y
>>> h = compose(times2, add)
>>> h(3, 4)
14
```

18.2.5. Objects and Classes

What's the fastest way to teach object-oriented programming? I mean, after drawing some diagrams explaining objects, classes, inheritance, polymorphism and so on. Let's type a Python class and see.

```
>>> class Person:
def __init__(self, name):
self.name = name
def sayHello(self):
print "Hello, I'm", self.name
>>> andy = Person("Andy")
>>> andy.sayHello()
Hello, I'm Andy
```

Ok, this it not very sophisticated, but we have defined a class `Person`, created an instance of this class, and called the method `sayHello`, all in seven lines of code. To do this, we had to know two things: method definitions looks like functions taking the instance `self` as the first argument (you don't need to call it `self`, but everybody does), and the constructor is called `__init__`. This name is one of the function names with special meaning in Python, all starting and ending with two underscore characters. To continue our study of object orientation, let's see what the newly introduced variables are:

```
>>> type(Person)
<type 'class'>
>>> andy
<__main__.Person instance at 0x009FD0B0>
>>> andy.sayHello
<bound method Person.sayHello of <__main__.Person instance at 0x009FD0B0>>
>>> Person.sayHello
<unbound method Person.sayHello>
```

Here we can see precisely what we have defined. `Person` is a class, and `andy` is an instance of this class. There seem to be two kinds of methods: `andy.sayHello` is a bound to the object `andy`, whereas `Person.sayHello` is not bound to an instance of the class `Person` yet. Since we can print all these objects, we can also use them in all kinds of expressions.

```
>>> f = andy.sayHello
>>> f()
Hello, I’m Andy
>>> F = Person.sayHello
>>> F(andy)
Hello, I’m Andy
>>> def evalThreeTimes(f):
for i in range(3): f()
>>> evalThreeTimes(andy.sayHello)
Hello, I’m Andy
Hello, I’m Andy
Hello, I’m Andy
```

The next example demontrates polymorphism. We define a new class `Employee` derived from `Person` which adds another attribute for the employee number which the obedient employee has to mention whenever saying hello.

```
>>> class Employee(Person):
def __init__(self, name, number):
Person.__init__(self, name)
self.number = number
def sayHello(self):
print "Hello, I’m", self.name, "also known as number", self.number
>>> homer = Employee("Homer", 1234)
>>> homer.sayHello()
Hello, I’m Homer also known as number 1234
```

It looks like all methods in Python can be polymorphic, since we don’t have to do anything special to define the new behavior (similar to Smalltalk and the default semantic (non-final method) in Java). The next example tells us more about the way Python class work.

```
>>> def f(self):
        print "Hi there, I’m", self.name
>>> Person.sayHello = f
>>> andy.sayHello()
Hi there, I’m Andy
>>> homer.sayHello()
Hello, I’m Homer also known as number 1234
```

We changed the `sayHello` method by assigning a new function to the unbound method, and indeed, calling `sayHello` on the instance `andy` now gives new answer, but `homer` (being an `Employee`) does not change his behaviour. This shows that methods are just callable members of a class. When calling a method, Python looks for a member with the name of the called method and then executes the "call operator" on this objects. The same happens when calling a function or a class: Python first looks for the object (like for any other variable) in the current environment and then passes the function’s arguments to

the "call" method of this callable object. You can turn any of your own objects into callable objects which behave like functions by implementing the special `__call__` method.

```
>>> class A:
def __init__(self, n): self.n = n
def __call__(self, x): return self.n * x
>>> a = A(5)
>>> a(3)
15
```

Existing Python classes can not be extended as easily as classes in other dynamic object oriented languages (Smalltalk, Objective C, CLOS). When we define a class again, a new class is created (in Ruby, the new class members are added to the existing class).

```
>>> class A:
def __init__(self, n): self.n = n
>>> A(5).n
5
>>> class A: pass
>>> A(5)
Traceback (most recent call last):
  File "<pyshell#244>", line 1, in ?
    A(5)
TypeError: this constructor takes no arguments
```

However, since methods are just callable member of the class, we can attach functions as methods to an existing class.

```
>>> class A:
def __init__(self, name): self.name = name

>>> def hello(self): return "My name is " + self.name
>>> A.hello = hello
>>> A("Homer").hello()
'My name is Homer'
```

An interesting application of keyword arguments is a "universal constructor" for objects.

```
>>> class Object:
def __init__(self, **kw):
self.__dict__.update(kw)
>>> joe = Object(name="Joe", age=25)
>>> joe.__dict__
{'age': 25, 'name': 'Joe'}
>>> print "my name is", joe.name
my name is Joe
```

18.3. More Features

18.3.1. Exceptions

Exceptions are by now well accepted as a good means to separate the main logic of a program from error handling.

```
>>> try:
1/0
    except ZeroDivisionError, e:
print e
integer division or modulo by zero
```

The `except` statement takes the name of the exception class we want to handle and the name of the variable in which the exception should be stored. Python comes with its own hierarchy of exception classes, and we can easily add our own.

18.3.2. Nested Definitions

Most Python statements can occur anywhere in the code. You can nest function definitions, class definitions, import statements within each other or other control statements. In the first example, a function returns another (local) function.

```
>>> def f(a):
def g(x): return a + x
return g
>>> h = f(10)
>>> h(5)
15
```

The argument `a` passed to the first function `f` is used in the local function `g`. This example only works in the new versions of Python supporting nested scopes.

Similarly, the next function creates a class on the fly and returns it to the caller.

```
>>> def f(x):
class B:
def __init__(self):
self.x = x
return B
>>> c = f(5)
>>> d = c()
>>> d
<__main__.B instance at 0x00923D60>
```

```
>>> d.x
5
```

18.3.3. Generators

Generators are another recent addition to Python, probably in response to Ruby (which copied it from Icon). A generator is a function-like object which can act as an iterator. The generator interrupts its processing, returns a value, and later continues at the same place while keeping its state (local variables, etc.) between the calls. To activate the generator feature one has to import it from the future (Python's way to gradually introduce changes which could break some code, in this case because of the new keyword "yield").

```
>>> from __future__ import generators
>>> def f(x):
yield x
yield 2*x
yield 3*x
>>> for i in f(5): print i
5
10
15
```

Generators are rather useful when implementing iterators on complex data structures (e.g., a tree walk).

18.3.4. String Formatting

If there is one feature I'm missing in Java, it is a formatting function as powerful as C's `printf`. This is even more true, since I know Python's merge operator `%`. The percent operator merges a format string (a printf-like pattern) with the values on the right hand side (either a single value or a tuple of values).

```
>>> "my name is %s" % "Joe"
'my name is Joe'
>>> "name: %s, age: %d" % ("Joe", 25)
'name: Joe, age: 25'
```

Many times, inserting the values in the order in which they appear is not what you want. Imagine a template for a letter in which you would like to insert the name of the recipient multiple times. In this case you can use named placeholders in the format string and pass the values as a dictionary:

```
>>> "name: %(name)s %(age)d" % {"name": "Joe", "age": 25}
'name: Joe 25'
```

Remembering that the attributes of an object are accessible as a dictionary, this results in a nice way to insert objects into templates:

```
>>> class Person:
def __init__(self, name, age):
self.name = name
self.age = age
>>> joe = Person("Joe", 25)
>>> "name: %(name)s, age: %(age)d" % joe.__dict__
'name: Joe, age: 25'
```

18.3.5. Operator Overloading

All the operators we have used so far were predefined in the language core. If we want operators to work with our own objects, we have to override the associated special methods. However, you can not change the behavior of a built-in class and redefine, say, the meaning of the operator + for integers.

18.3.6. Class Methods and Properties

Version 2.2 adds two kinds of methods to Python's toolset, static methods and class methods. Static methods are just functions which are made part of the class so that we can call them with the class or an instance. Unlike the other methods, the functions have no special argument (such as the instance passed to instance methods). Class methods are like static method, but get the class passed as the first argument.

```
>>> class A(object):
        def f(*args):
            print args
        fStatic = staticmethod(f)
        fClass = classmethod(f)
>>> class B(A): pass
>>> b = B()
>>> A.fStatic(55)
(55,)
>>> b.fStatic(55)
(55,)
>>> A.fClass(55)
(<class '__main__.A'>, 55)
>>> b.fClass(55)
(<class '__main__.B'>, 55)
```

Both types of methods are created by first defining the function inside of the class just like an instance method and then converting this function to a static or class method the built-in functions `staticmethod` and `classmethod`, respectively.

Properties look like attributes to the outside world, but are implemented with accessor methods. Prior to Python 2.2, one had to implement the special methods `__getattr__` and `__setattr__` to achieve this behavior. As of Python 2.2, the new `property` function accomplishes the same effect much more easily.

```
>>> class A(object):
def setx(self, x):
print "setting x"
self._x = x
def getx(self):
print "geeting x"
return self._x
x = property(getx, setx)

>>> a = A()
>>> a.x = 55
setting x
>>> a.x
geeting x
55
```

18.3.7. Visibility

By default, all members of a class are public. There is no equivalent to the visibility qualifiers of C++ or Java. However, it is possible to make the access to a class member more difficult. Whenever the name of a member starts with two underscore characters but does not end with two underscores (like Python's special members), Python considers the member private and adds an underscore and the class name as a prefix (this process is known as name mangling).

```
>>> class A:
        def __init__(self, name): self.__name = name
        def getName(self): return self.__name

>>> a = A("Homer")
>>> dir(a)
['_A__name', '__doc__', '__init__', '__module__', 'getName']
>>> a.getName()
'Homer'
>>> a.__name
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in ?
    a.__name
AttributeError: A instance has no attribute '__name'
>>> a._A__name
'Homer'
```

Within the class, we can still use the original name `__name`, but the member is actually stored under the name `_A__name`. Of course this does not prevent you from accessing this attribute directly, but it makes it a lot harder. Note that this approach works for attributes and method alike, since methods are just callable attributes.

18.4. Libraries and Common Examples

18.4.1. Modules and Packages

Up to now we have used the core language only, and as usual this is just the tip of the iceberg. Most of the useful functions are hidden in libraries. Python organizes libraries in modules and packages. Modules are just files (with the .py suffix) containing python code (you can also implement modules in C, but this is out of scope of this presentation), and packages are directories containing other modules or packages.

```
# test.py
def foo(x):
    return 2*x

>>> import test
>>> test.foo(5)
10
>>> from test import foo
>>> foo(3)
6
```

18.4.2. File I/O

Python's I/O libraries were developed as an abstraction of the UNIX system. File objects are input/output streams which can be created from files, sockets, and in memory strings. Here are some examples:

```
>>> f = open("test.txt", "w")
>>> for name in ["Joe", "John", "Mary"]:
f.write("my name is %s\n" % name)
>>> f.close()
>>> open("test.txt").read()
'my name is Joe\nmy name is John\nmy name is Mary\n'
>>> for line in open("test.txt"):
print "line:", line.strip()
line: my name is Joe
line: my name is John
line: my name is Mary
```

18.4.3. Regular Expressions

No scripting language can succeed without a sophisticated regular expression library. Python's (second) regular expression module is modeled after Perl's killer feature, but uses a plain object based interface.

```
>>> import re
```

```
>>> pattern = re.compile("hello[ \t]*(.*)")
>>> match = pattern.match("hello world")
>>> match
<_sre.SRE_Match object at 0x0145A190>
>>> match.group(1)
'world'
>>> match.group(0)
'hello world'
>>> pattern.match("bla")
>>> print pattern.match("bla")
None
```

18.4.4. SQL Database Access

After some time, Python now also provides a standard interface for the access to an SQL database including drivers for the most common databases (Oracle, Sybase, PostgreSQL, mySQL, SAP DB, etc.).

```
>>> import sys, psycopg
>>> connection = psycopg.connect(
       "host=127.0.0.1 user=test password=test dbname=testdb")
>>> cursor = conn.cursor()
>>> try:
        cursor.execute("create table person (firstname varchar(20), lastname varchar(20))")
        cursor.execute("insert into person values ('Homer', 'Simpson')")
        cursor.execute("select * from person")
        print cursor.fetchall()
    finally:
        curs.execute("drop table person")
```

18.5. Discussion

Now, that we have seen many aspects of Python, what are its main characteristics? What causes the language to be very expressive and easy to learn? Are these properties related to the bad performance and dangerous "openess" of the langage?

Variables don't have to be declared, and you can assign anything to a variable. This is one of the features which make life very convenient in the beginning, but may cause trouble for large systems. Not requiring variables to be declared saves typing, but can turn simple typos into hard-to-find bugs. It is difficult to automatically optimize Python programs (e.g., using a just-in-time compiler), because a variable may contain any type of object, and the type can change during the execution of the program.

Python has no concept of an "interface", that is a declaration of of a function or class without an implementation. This makes large scale development harder, since you only find out at runtime if an

object has the expected attributes and methods. In this sense, Python's object orientation is very similar to Smalltalk.

Without interfaces, it is almost impossible to define APIs precisely. Most of Python's standard APIs are nonetheless quite easy to learn, but it all depends on conventions and good documentation. The definition of complex APIs such as database access has definitely suffered from the lack of interfaces. Not surprisingly, the probably most complex Python application, the web publishing (content management) systen Zope has introduced its own concept of interfaces.

Python strongly distinguished statements (e.g., control statements) and expressions. Statements don't represent a value. This prevents a more functional programming style in some situations. For example, there is no "functional if" statement such as the question operator in C. Ideally one would like to write

```
>>> y = if x < 0 then -1 else 1
SyntaxError: invalid syntax
```

but this is unfortunately not valid Python code (Ruby does better in this area).

References

Here is a selection of books about Python. First one has to mention that the documentation which is shipped with the language itself it very good. The two most useful parts are the tutorial and the library manual, the latter sometimes being too terse.

Alex Martelli's "Python in a Nutshell" [MARTELLI03]> is a good description of the language including the newer features. The best introduction I have seen is "Learning Python" Lutz99>. The best reference is in my opinion David M. Beazley's (also known as the developer of the interface generator SWIG) "Python Essential Reference" Beazley01>. It starts with an introductory chapter which is sufficient to learn Python from scratch. If you have some old Python book and want to learn about the new features, the pocket reference Lutz01> is a good resource. There are also a number of books covering special Python topics such as GUI programming (the german "Python und GUI Toolkits" Lauer02> is a careful presentation of the most important GUI libraries), web programming, and especially the Python-based content management system Zope.

Alex Martelli, 0596001886, O'Reilly & Associates, Inc., 2003, *Python in a Nutshell*.

[Lutz99] Mark Lutz and David Ascher, 1-56592-464-9, O'Reilly & Associates, Inc., *Learning Python*.

[Beazley01] David M. Beazley, 0-7357-1091-0, New Riders, *Python Essential Reference, Second Edition*.

[Lutz01] Mark Lutz, 0-5960-0189-4, New Riders, 2001, *Python Pocket Reference, Covers Python 2*.

[Martelli02] Alex Martelli and David Ascher, 0-596-00167-3, O'Reilly & Associates, Inc., *Python Cookbook*.

[Lauer02] Michael Lauer, 3-8266-0844-5, mitp-Verlag, *Python und GUI-Toolkits*.

[Lutz02] Mark Lutz, 3-89721-240-4, O'Reilly & Associates, Inc., *Python Pocket Reference, Second Edition*.

Notes

1. A functional "if" is about to be added to Python in the next release.

Chapter 19. Java

SUN microsystems started developing Java (originally under the name "Oak") in 1991 as a programming environment for small devices such as set-top boxes for interactive TV. Java was officially announced in 1995 and quickly gained popularity together with the internet (partly because of the integration of Java in the Netscape and Microsoft web browsers). Especially after the introduction of the Java 2 platform end of 1998 (Java 1.2), the language (and programming environment) took the software industry by storm entering more and more areas from enterprise server applications to mobile phones.

Java is more than just a programming language. It also defines a run-time environment with a well-defined byte-code language, the Java virtual machine, on top of which other languages can be implemented (such as JavaScript or Jython). In this chapter we will focus on Java, the programming language.

19.1. Software and Installation

We will be using the SUN's Java software development kit (J2SDK), version 1.4.2, which you can download for a number of platforms from the main Java (http://java.sun.com) site. We mainly need the compiler `javac` to compile the source code into the intermediate byte code, and the interpreter `java` to run this byte code.

19.2. Quick Tour

19.2.1. Hello World

As usual in a more traditional, compiled language we need some scaffolding before we can enjoy the result of the our "Hello World" program.

```
public class Hello {
  public static final void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

To run this program, we put this code into a file called `Hello.java`, call the compiler with the command `javac Hello.java`, and finally execute the resulting byte code (contained in the class file `Hello.class`) with the Java interpreter using `java -classpath . Hello` (no `.class` suffix here!). The `classpath` option tells the interpreter where to look for classes. In our case, we want it to pick up the class we have just compiled in the current directory.

Two things catch our attention immediately. First, a Java program looks a lot like C (or C++), and second, Java seems to be serious about object orientation in the sense that nothing goes without a class. C's `main` function as the entry point to a program becomes a static method of a class, that is, a method bound to a class and not an instance of a class. In our example, the surrounding class provides no more than a namespace for the function, since it does not use the class at all.

Java's classes are directly mapped to the file system. A Java source file typically contains a single class (to be precise, exactly one public, non-nested class), whose name is reflected in the name of the source file by just adding the `.java` suffix. The same holds for packages, Java's means to organize the source code in a hierarchical manner. The package hierarchy maps to the directory tree of the underlying file system. As an example, we can put our "Hello World" program in the `sample` package by adding the package declaration as the first line to the source code and putting the source file in the subdirectory `sample`.

```
package sample;

public class Hello {
  public static final void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Of course, we also have to tell the compiler which new source file to compile using the command `javac sample/Hello.java`. Similarly, we have to supply the fully qualified class name `sample.Hello` when running the program with the Java interpreter, that is, use the command `java -classpath . sample.Hello`.

With this knowledge about packages, we can continue dissecting the program. The actual print statement `System.out.println("Hello World");` is a call to the method `println` of the standard output stream which is available as a class member (static member in Java parlance) of the `System` class. The fully qualified name of this class is actually `java.lang.System`, that is, it is defined in the standard library package `java.lang`. The same is true for the `String` class used in the signature of the `main` method. Hence, we could have written

```
public class Hello {
  public static final void main(java.lang.String[] args) {
    java.lang.System.out.println("Hello World");
  }
}
```

Alternatively, we could have imported the standard package with the `import` statement before the class definition.

```
import java.lang.*;
public class Hello {
  public static final void main(String[] args) {
    System.out.println("Hello World");
  }
```

```
}
```

Since the classes of the standard package `java.lang` are used everywhere in a Java program, the compiler uses the import statement `import java.lang.*` implicitly.

Before we finally leave this example for more exciting tasks, I would like to point out two differences to the early member of the C family: There is separation of declaration and definition as in C++ and Objective C, and the array brackets are normally placed behind the type rather than the variable (although the latter is still allowed).

19.2.2. Types

The fact that Java is an object oriented programming language does not mean that everything is an object like in Smalltalk. Instead, following its C/C++ ancestors, there is a clear divide between primitive types such as integers on the one hand and objects on the other.

Primitive values are plain scalar values. They have no methods, live on the stack, and use value semantics (no direct pointers in Java!). Their type is only known at compile time and they can't be used in a generic context (e.g., a collection).

```
public class Sample {
  public static void main(String[] args) {
    double x = 1.25;
    double y = 2.5 / x + 1;
    System.out.println(Double.toString(y)); // true

    int m = 5, n = 10;
    System.out.println(Boolean.toString(m+2*n == 25)); // true
  }
}
```

Java supports all of C's scalar types with the exception of unsigned integers. As an addition, there is also a `boolean` type. Since one of Java's main goals is portability, the representation of the primitive types is well-defined. A `long` integer, for example, is always a 64 bit long independent of the underlying operating system.

Note that Java follows C++ and allows us to define variables in middle of block of statements. To print the results, we use the static `toString` methods of the standard classes `Double` and `Boolean`.

All other data in Java (including arrays, structures, etc.) is represented as objects. Objects are quite the opposite of primitive values. They have attributes and methods, live on the heap, and use reference semantics. You can ask for an object's type (the class) at run-time and do most of the things we know

from Smalltalk. Objects are created using the new operator (another C++ heritage) and automatically deleted by Java's garbage collection if there is no reference to the object anymore and the virtual machine needs more memory.

```
public class Sample {
  public static void main(String[] args) {
    String a = new String("blah");
    String b = new String("blah");
    String c = a;

    System.out.println(Boolean.toString(a == b));      // false
    System.out.println(Boolean.toString(a == c));      // true
    System.out.println(Boolean.toString(a.equals(b))); // true
  }
}
```

In its effort to radically simplify C++, Java dropped all features deemed complex or dangerous: pointers, operator overloading, parameterized types (generics, templates), default arguments, variable argument lists, macros, and so forth. As a consequence of the lack of operator overloading, the equality operator `==` always refers to object identity. To test equality in terms of the contents of the object, one has to call the `equals` method. Especially for strings this is one of the gotchas for the beginning Java developer. [1]

Strings are not a very good example for objects, since they play a special role in Java. Unlike most other objects, strings are immutable, that is, they can be changed once they have been created. Furthermore, Java supports strings them with some special syntax. We can directly assign a string literal to a string object (reference variable, to be precise), and the plus operator is overloaded with string concatenation including the conversion of any other type to a string. Therefore, we could have written the last example as follows.

```
public class Sample {
  public static void main(String[] args) {
    String a = "blah";
    String b = "blah";
    String c = a;

    System.out.println("" + (a == b));      // false
    System.out.println("" + (a == c));      // true
    System.out.println("" + (a.equals(b))); // true
  }
}
```

None of these special syntax features is available to user defined classes.

19.2.3. Classes and Interfaces

It is about time to define our first Java class.

19.2.4. Exceptions

Like most modern languages, Java uses exceptions to handle error conditions. Exceptions a objects extending the `Throwable` class. To handle an exception, the code in question has to be placed in a `try` block, followed by an arbitrary number of `catch` blocks handling different exceptions (matching by inheritance), and an optional `finally` block containing actions (such as closing a resource) to be performed after the try block whether an exceptions occurs or not.

```
public class ExceptionSample {
  static void testException(int i) {
    try {
      System.out.println("start try " + i);
      if (i == 0) {
throw new MyException(i);
      }
      else if (i == 1) {
throw new Exception("oops");
      }
      System.out.println("end try");
    }
    catch (MyException e) {
      System.out.println("caught my exception: " + e.getMessage());
    }
    catch (Exception e) {
      System.out.println("caught exception: " + e.getMessage());
    }
    finally {
      System.out.println("clean up");
    }
  }

  public static void main(String[] args) {
    testException(0);
    testException(1);
    testException(2);
  }
}

class MyException extends Exception {
  MyException(int i) {
    super("i=" + i);
  }
}
```

The example defines a custom exception class `MyException` which is thrown during the first run of the `testException` method and caught by the first catch block. During the second call, the general exception passes the first catch block, but is caught by the second one. In any case, the clean-up message of the finally clause is printed.

```
start try 0
start try 0
```

```
caught my exception: i=0
clean up
start try 1
caught exception: oops
clean up
start try 2
end try
clean up
```

Due to the garbage collection, the finally block is much more important than in C++, since we can't rely on destructors to release resources.

As a peculiarity, Java distinguishes checked and unchecked exceptions. If a method may throw a checked exception (and not catch it), it must declare the exception with a `throws` clause.

```
public class ExceptionSample1 {
  static void throwingMethod(int i) throws MyException {
    if (i == 0) {
      throw new MyException("oops");
    }
  }

  public static void main(String[] args) {
    try {
      throwingMethod(0);
    }
    catch (Exception e) {
      System.out.println("exception: " + e.getMessage());
    }
  }
}

class MyException extends Exception {
  MyException(String message) {
    super(message);
  }
}
```

In other words, checked exceptions become part of the signature or a method. An exception is a check

19.3. More Features

19.3.1. Collections

With version 1.2, Java obtained a thoroughly designed collection library similar to ones available for Smalltalk.

```
Set s = new HashSet();
s.add("a");
s.add("b");
s.add("c");
for (Iterator i=s.iterator(); i.hasNext(); ) {
  System.out.println(s.next());
}
```

19.3.2. Inner Classes

19.3.3. Reflection

Java's version 1.1 introduced another important enhancement: the possibility to obtain detailed information about classes at run-time. Being a no-brainer for Smalltalk, Java's predecessor C++ to this date provides only very limited "run-time type information" (RTTI) which does not give access to attributes or method.

19.3.4. Applets

If only for historical reasons, an introduction to Java must cover browser-based Java clients using applets. An applet is a Java program which runs in the Java virtual machine contained in a web browser. One problem with applets is that their evolution basically stopped at java 1.1.8 as the last supported by Microsoft's Internet Explorer out of the box (that is, without downloading a plugin). Therefore, we do not have the much improved libraries of the Java2 platform at our disposal, such as the new collections and the Swing user interface.

19.4. Discussion

Java is a compromise. It is result of the frustration with C++ and can be seen as a great simplification of this highly complex, multi-paradigm language. It is much harder to write a desastrous program (crashing

a system) in Java than in C or C++.

Java checked exceptions are appealing at first sight, but soon turn out to lead to meaningless code mapping the checked exception of one API to the checked exception of another API, often accompanied by cascading stack traces in the resulting log files. Not surprisingly, the C# designers dropped this features.

Although the core language is relatively simple (probably with the exception of inner classes), some of the APIs are not, which may indicate that certain functionality is not easy to express in such a restricted language. It is interesting to observe, how Java is evolving. Partly driven by the evolution of C#, version 1.5 will introduce features such as generic types which were originally deemed too complex.

Considering, Java was designed as a new language from scratch, is not famous for the consistency of its API. As an example, the length of something is expressed in (at least) three different ways:

```
int n = "blah".length();
n = (new int[10]).length;
n = (new ArrayList()).size();
```

The naming of classes and methods, however, is in general readable, since it avoids abbreviations. Unfortunately, the naming standard does not define how acronyms are written as part of identifiers. Are they all caps or just the first letter? The standard library demonstrates that people were still debating even when writing a single class (e.g., HttpURLConnection).

References

If you would like to feel the original spirit of the Java "movement", have a look into Hoff96>. To learn Java, especially with a C/C++ background, the Nutshell book Flanagan02> is still the best in my opinion. If you want to get a deeper understanding of how to write good Java programs (and you have more time since you are facing 1400 pages), Eckel02> is for you. This book is particularly interesting if viewed in the context of Bruce Eckel’s C++ book and his more recent move towards Python.

[Hoff96] Arthur van Hoff, Sami Shaio, and Orca Starbuck, 0-201-48837-X, Addison-Wesley, 1996, *Hooked on Java: Creating hot Web sites with Java applets*.

[Flanagan02] David Flanagan, 0596002831, O’Reilly, 2002, *Java in a Nutshell, 4th edition*.

[Eckel02] Bruce Eckel, 0131002872, Prentice Hall, 2002, *Thinking in Java, 3rd edition*.

Notes

1. The `equals` method does not really belong to any of the two objects involved. The underlying equality test is a symmetric operation and should be implemented depending on both objects. It is therefore a good example for a situation where a generic function makes more sense. In the Java `equals` implementations one will instead always find a cast operator (if not an explicit type check).

Chapter 20. JavaScript

I was debating whether to include JavaScript in this book or not when I had to use JavaScript on an HTML form to enable or disable fields depending on the value of another field. Doing so on a simple web page is straightforward, but I was pleasantly surprised that the dynamic object-oriented features of JavaScript allowed me to implement this functionality in a generic way as part of an XML/XSL based web application framework. The second reason I decided to cover JavaScript is the excellent reference card (http://www.javascript-reference.info) I stumbled upon while solving this problem. This card shows JavaScript as a small yet powerful language.

The main difference between JavaScript and the other languages in this book it its original as a scripting language for internet browsers. JavaScript (originally called "LiveScript") was created by Netscape in 1995 to allow web designers to use the newly introduced Java applets without coding Java. Although the name stresses the relationship to Java, JavaScript is a completely different language, much closer to scripting languages such as Perl and (apart from the curly braces) Python. Rather than controlling applets, it turned out to be very useful to access and manipulate the parts of HTML pages (by means of the underlying document object model, DOM).

Microsoft first tried to position Visual Basic as an alternative browser scripting language in the form of VBScript and then added its own JavaScript implementation called JScript to the Internet Explorer in 1996. Because the differences of these implementations, a standardization process was started with ECMA which lead to the first ECMAScript standard in 1997.

We will treat JavaScript in this chapter as an independent programming language without the context of a web browser.

20.1. Software and Installation

We use Netscape's JavaScript implementation in Java, called Rhino (http://www.mozilla.org/rhino). At the time of this writing, version 1.5 release 4.1 is the most current stable release. It implements edition 3 of the ECMA standard. The implementation includes also an interpreter with an interactive shell that we can use for our experiments. The shell is started with the command

```
java -classpath js.jar org.mozilla.javascript.tools.shell.Main
```

The Java archive `js.jar` contains the JavaScript implementation including the `org.mozilla.javascript.tools.shell.Main` class which contains the interactive shell. When running the command, the intepreter greets you with a short message.

```
Rhino 1.5 release 4.1 2003 04 21
js>
```

20.2. Quick Tour

20.2.1. Expressions

If it takes more than one line to print "Hello World", JavaScript should not call itself a scripting language. In fact, it is the shortest "Hello World" program in this book (tied with Lisp, Ruby, Haskell, and Python):

```
js> "Hello World"
Hello World
```

Of course, we are cheating, since we just use the shell to repeat the string value (so we did for the other languages). The real "Hello World" program looks like this:

```
js> print('Hello World')
Hello World
```

This starts revealing some information about JavaScript. We can call functions (`print` being one of them) using the "standard" (mathematical) function notation, and strings literals can be defined with single quotes or double quotes (as in the first example).

The `print` function can take any number of arguments, and these arguments can be of any type.

```
js> print('result:', 4+5*3)
result: 19
```

This also shows us that we can perform arithmetic, again using the standard (i.e., mathematical) syntax. As in Python, the printed values are separated by spaces.

JavaScript follows Java's expression syntax including update operators (`+=`, `-=`, and so forth), increment and decrement operators, as well as bit-wise operations (e.g., `&` for bit-wise "and" and << for "shift left".

Like in Java, numbers can be entered as integer, floating point (with or without exponent), hexadecimal (starting with `0x`), or octal (starting with `0`) literals.

```
js> 1.5e-2
0.015
js> 0x9d
157
js> 011
9
```

JavaScript knows only one number type (simply called "number") that is stored as a 64-bit floating point number (C's "double" type). Therefore, division does not distinguish between integers and real numbers, and the result of `3/2` is `1.5` and not `1`.

```
js> typeof(55)
number
js> typeof(5.5)
number
js> 3 / 2
1.5
js> 5 % 2
1
js> 5.5 % 2
1.5
js> 5.5 % 2.5
0.5
```

Boolean expressions also use Java's syntax with the constants `true` and `false` and the C operators `!`, `&&`, and `||` for the three Boolean operations "not", "and", and "or". But in contrast to strongly typed languages such as Java, they can be applied to any expression. A value is considered false if it is the boolean constant `false`, an empty string, a numerical zero, or a null reference. Everything else (including empty lists) is considered true.

Semantically, the boolean "and" and "or" operators work as shortcut operators returning one of their operands (like in Python). The result of an "and" expression is the first operand that is false or, if all operands are true, the last that is true. Similarly, the result of an "or" expression is the first operand that is true or, if all operands are false, the last that is false.

```
js> "a" && "b"
b
js> 0 && "b"
0
js> "" || 0
0
```

This way, we can also simulate a functional "if-then-else" expression, but we do not have to resort to this trick, since JavaScript also support question mark operator.

```
js> 4 < 5 && "b" || "c"
b
js> 4 > 5 && "b" || "c"
c
js> 4 < 5 ? "b" : "c"
b
```

20.2.2. Control Statements

JavaScript offers Java's conditional and loop statements.

```
js> i = 3;
3
js> while (i > 0) { print(i); i -= 1; }
3
2
1
0
js> do {
  print(i);
  i += 1;
} while (i < 4)
0
1
2
3
4
js> for (i=0; i<3; i++) {
  print(i);
}
0
1
2
js> if (i < 3) {
  print("less than three");
} else if (i > 3) {
  print("greater than three");
} else {
  print("exactly three");
}
exactly three
```

We can also use a `break` statement to escape from the innermost loop or switch clause.

JavaScript's `switch` statement supports numbers and strings (like C#).

20.2.3. Collections

Let's see if we can detect more similarities to Python. Defining lists looks exactly the same.

```
js> x = [1, 2, "blah"]
```

```
1,2,blah
js> x[0]
1
js> x[2]
blah
js> x.slice(1,2)
2
js> x.slice(0,2)
1,2
```

As we see, list literals and access to elements use the same syntax as Python, and as in any other language with some C heritage, indexing starts at zero (think pointers and offsets). JavaScript also allows us to extracts parts of a list using slices with the `slice` method. This is the first time we see a method call in JavaScript, and again, there is no surprise. Note that the slice indexes describe half-open intervals, that is, the left boundary is included and the right one is not.

As for maps, the main difference is that they are printed as objects rather than showing the keys and values (we will see shortly why).

```
js> m = {"blah": 55, "blub": 66}
[object Object]
js> m["blah"]
55
js> m["xxx"]
js>
```

We also notice that a missing key does not raise an exception, but just returns nothing (which is called `null` in JavaScript).

20.2.4. Functions

JavaScript treats functions as first class objects, almost like a functional language. One way to define a function is to create an anonymous function and assign it to a variable. The three periods in the following examples denote the repeated function definition as printed by the interactive shell.

```
js> add = function(x, y) { return x + y; }
...
js> add(1, 2)
3
```

The anonymous function is defined with the `function` keyword followed by the argument list and the code block defining the function's body. Since JavaScript is dynamically typed, there is no need to define the argument and return types. To return a value, we call the `return` operator followed by the expression we want to return (no implicit return like in functional languages).

You can also use the more conventional syntax with the function's name following the `function` keyword.

```
js> function add(x, y) { return x + y; }
js> add(1,2)
3
```

The result is exactly the same. The second syntax is just a shortcut for the first one (saving as much as a single character, the assignment operator =).

Having defined a function using anonymous function objects and assignment, it is clear that we easily pass functions as arguments to other functions. In the following example, we define the function `times` that calls a call function a given number of times.

```
js> function times(n, f) {
  for (i=0; i<n; ++i) {
    f()
  }
}
...
js> times(5, function() { print("blah") })
blah
blah
blah
blah
blah
```

We can use the same mechanism to define higher order functions which return functions themselves, such as the composition of functions.

```
js> function compose(f, g) {
  return function(x) { return f(g(x)) }
}
js> function times2(x) { return 2*x }
js> function plus10(x) { return x + 10; }
js> compose(times2, plus10)(10)
40
```

The `compose` function defined above works only for single arguments. For the general case we need to apply a function to an argument list without knowning the actual number of arguments in advance. As a first observation, we can call a JavaScript function with more than the required number of arguments. We can, for example, pass three instead of two arguments to the `add` function defined above.

```
js> add(1, 2, 3)
3
```

The third argument is simply ignored. In the body of a function we have access to array of arguments under the special variable `arguments`. This allows us to generalize the `add` function to arbitrarily many arguments.

```
js> function sum() {
  var result = 0;
  for (var i=0; i<arguments.length; i++) {
    result += arguments[i];
  }
  return result;
}
js> sum(1, 2, 3)
6
```

If we don't know the number of arguments in advance, we can call a function using its apply method which takes the object the function belongs to (more on this in the next section) as the first argument and the array of arguments as the second.

```
js> sum.apply(this, [1, 2, 3, 4])
10
```

Putting all these pieces together, we can now define the general composition function.

```
js> function compose(f, g) {
  return function() {
    return f(g.apply(this, arguments));
  }
}
js> compose(times2, sum)(1, 2, 3, 4)
20
```

Here, `this` is another implicit variable pointing to the object for which the function was called.

There is a third way to define a function which leads us directly to the object-oriented aspects of JavaScript. We can construct a function dynamically using strings for the argument names as well as the function's body by passing these strings to the `Function` constructor.

```
js> add = new Function("x", "y", "return x+y");
...
js> add(1, 2)
3
```

As you can imagine, this built-in code generation capability is extremely powerful (and dangerous).

20.2.5. Objects

Most of the chapters in this book explaining the object-oriented features of a language are called "Objects and Classes", just like object-oriented programming could by called "class-oriented" in most languages. JavaScript is one of the few exceptions. There are no classes. Its object-oriented features are based on prototypes rather than classes, an approach which dates back to the Self (http://research.sun.com/self/language.html) language.

To start with, let us revisit dictionaries (maps), now viewed as objects.

```
js> person = { "name": "Homer" }
[object Object]
js> person.name
Homer
js> person["name"]
Homer
js> person.age = 66
66
js> person["age"]
66
js> delete person.name
true
js> person.name
```

In JavaScript, objects and maps are basically the same thing. You can view the direct access to an object's properties (such as `name` and `age`) as syntactic sugar for the index operator. Obviously, this only works as long as the name of the property is a proper JavaScript identifier (starting with a letter, dollar sign, or underscore character).

```
js> person["1"] = 123
123
js> person["1"]
123
js> person.1
js: "<stdin>", line 52: uncaught JavaScript exception: SyntaxError: missing ; before statem
js: person.1
js: .......^
```

Like in Python, properties can be any kind of value including functions (in Python "callable objects"). A method or operation in other object-oriented languages corresponds to a function property in JavaScript.

```
js> person.hello = function() { print("Hello, I'm", this.name); }
...
js> person.hello()
Hello, I'm Homer
```

We obviously do not want to set the properties for each new object again and again. In JavaScript, we have two means to define the properties of all objects of a certain kind (I almost wrote "class"): constructors and prototypes.

A constructor is a function that sets the properties of a new object. It is called when constructing an object with the `new` operator.

```
js> function Person(name) { this.name = name; }
js> person = new Person("Homer");
[object Object]
js> person.name
Homer
js> person.constructor

function Person(name) {
    this.name = name;
}
```

We can ask an object for its constructor using the `constructor` attribute. In a sense, JavaScript's classes are the constructor functions. This becomes even more apparent when we look at prototypes. First, we should remark that functions are first class objects which can have properties themselves (again just like in Python).

```
js> function add(x, y) { return x + y; }
js> add.description = "I'm adding two numbers"
I'm adding two numbers
js> add.description
I'm adding two numbers
```

When constructing an object, it is linked to its constructor function, and each constructor function has the `prototype` property. When looking for a property of an object, JavaScript first checks the object itself. If the property is defined in the object, its value is returned. If it is not defined in the object itself, JavaScript checks the prototype of the object, that is, the `prototype` property of the object's constructor function. If the prototype object defined the requested property, its value is returned. Otherwise, the result is null. The following example demonstrates the different situations.

```
js> person.age
js> Person.prototype.age = 55
55
js> person.age
55
js> person.age = 66
66
js> person.age
66
js> delete person.age
true
js> person.age
55
```

We first check the `person`'s `age` property which turns out to be empty. We then set the `age` property of the constructor's prototype to 55. Now, the `person` object still does not define the `age` property itself, but JavaScript finds it in the prototype and prints 55. After setting the `person`'s `age` property explicitly, we see the new property value. Finally, after deleting the property again, the prototype's value is retrieved.

The nice thing about this prototype approach is that it treats attributes and methods exactly the same way. All properties, whether they are normal values (attributes) or functions (method), follow the same lookup rules.

Static or class methods of other object-oriented languages become function properties of JavaScript's constructors.

```
js> function Person(name, age) { this.name = name; this.age = age; }
js> Person.compareAge = function(a, b) { return a.age - b.age; }
...
js> homer = new Person("Homer", 55);
[object Object]
js> bart = new Person("Bart", 11);
[object Object]
js> Person.compareAge(homer, bart)
44
```

20.3. More Features

20.3.1. Exceptions

20.3.2. Regular Expressions

JavaScript, being a true scripting language, contains a powerful regular expression library. This is not surprising since JavaScript is often used to validate HTML forms.

Bibliography

Yes, David Flanagan has also written (besides his much acclaimed "Java in a Netshell") one of the standard JavaScript books ([FLANAGAN01]>).

David Flanagan, O'Reilly, 2001, 0-596-00048-0, *JavaScript: The Definitive Guide, 4th Edition*.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995, 0-201-63361-2, *Design Patterns: Elements of Reusable Object-Oriented Software*.

Chapter 21. Ruby

Ruby is a relatively new scripting language designed by Yukuhiro Matsumoto as an alternative to Perl and Python ("more powerful than Perl, more object-oriented than Python" according the inventor). It was first released in 1995 and quickly gained popularity in Japan. Outside of Japan, the interest in Ruby was mainly sparked by "pragmatic programmers" (see [HUNT00]>), Andrew Hunt and David Thomas, with their book about Ruby [THOMAS00]>. Ruby combines aspects of such different origins as Perl and Smalltalk. It is considered by many the cleanest scripting language.

21.1. Software and Installation

Since there is only one Ruby implementation (the original one by the inventor) available for now, we don't have to choose between different dialects. We use the standard Windows distribution (which comes with an installer) as well as the Linux version. Part of the installation is an interactive shell called `irb` (for interactive Ruby) which is similar to the interactive environments we have used for Python, Lisp, and so forth. On Debian Linux, the interactive shell is contained in a second package (called `irb`) that you have to install together with the Ruby package itself.

Using the option `--simple-prompt` changes the prompt of the interactive shell to `>>` which is more suitable for our presentation than the longwinded standard prompt indicating module and line number. Like in Perl, we can manipulate the way the print statement separates fields and records using the special variables `$,` and `$\`. We will set the record separator to a space to simplify the examples.

21.2. Quick Tour

21.2.1. Expression

We start our journey at the usual place.

```
>> "Hello World"
=> "Hello World"
>> puts "Hello World\n"
Hello World
=> nil
>> 4 + 5 * 6
=> 34
```

The first impression is typical for a "scripting" language: no surprise. As an interesting detail we notice that the interactive shell always prints the value of the statement after the arrow "=>". Even the print statement returns something (albeit nil). This is a first indication that Ruby treats everything as

expressions (like a functional language). This impression is quickly confirmed trying the "if" statement (or rather expression), something we lamented about when using Python.

```
>> if 1 < 2 then 5 else 6 end
=> 5
```

Even better, the same is possible with a case statement.

```
>> case 30
    when 0..10 then "apple"
    when 11..20 then "pear"
    else "banana"
  end
=> "banana"
```

Many developers will welcome another difference from Python: indentation does not matter; Ruby uses the "end" keyword to denote the end of a block if necessary. Concerning variables, we expect from a scripting language that they don’t have to be declared and that they can change their type any time.

```
>> x = "blah"
=> "blah"
>> x = 55
=> 55
```

21.2.2. Collections

What about built-in collections as another typical scripting feature? Ruby’s lists feel like Python lists (or Perl’s anonymous arrays) with Perl’s slices.

```
>> l = [1, 2, 3]
=> [1, 2, 3]
>> l[1]
=> 2
>> l[-1]
=> 3
>> l[1..2]
=> [2, 3]
>> l[1..-1]
=> [2, 3]
```

The syntax for (hash) maps corresponds to Perl’s anonymous hashes (when used in a sensible way with the arrow notation), but the subscript operator is (fortunately) the same as for lists.

```
>> m = {"a" => 1, "b" => 2}
=> {"a"=>1, "b"=>2}
>> m["a"]
=> 1
```

Ruby is truely object oriented. All values are first class objects (including integers, floating point numbers, strings) and all functionality which can be attached to an object is provided as a method of the object (and not a function as in Python in many cases). Here are a few examples using strings and lists.

```
>> "blah".length
=> 4
>> "blah".length()
=> 4
>> "blah".index('l')
=> 1
>> -1234.abs
=> 1234
>> "blah".capitalize
=> Blah
>> "BlAh.downcase
=> blah
>> [1, 2, 3].join(";")
=> 1;2;3
```

Like Perl, Ruby allows to omit the parentheses around function or method arguments unless required to ensure the correct precedence. And like Perl, Ruby has to pay for this debatable convenience with a more complex handling of function objects as we will see below.

Ruby has adopted Scheme's convention for method suffixes. Predicates, that is, methods which return a boolean, end with a question mark, and destructive methods, which change the associated object, end with an exclamation mark.

```
>> [1, 2, 3].reverse!
=> [3, 2, 1]
>> [1, 2, 3].include?(1)
=> true
>> [1, 2, 3].include? 4
=> false
>> [].empty?
=> true
>> [1, 2, 3, 2, 1].uniq!
=> [1, 2, 3]
```

21.2.3. Objects and Classes

Ruby's object oriented part combines Smalltalk and Python with a more common syntax. Here is the `Person` class again.

```
>> class Person
>>   def initialize(name, age)
>>     @name = name
>>     @age = age
>>   end
```

```
>> end
=> nil
>> person = Person.new("Homer", 55)
=> #<Person:0x2a42868 @age=55, @name="Homer">
```

Similar to Python, methods are functions inside a class. But in contrast to Python we see no explicit reference to the instance itself `self` and the initializer is simply called `initialize`. Ruby uses "funny symbols" to mark instance, class, and global variables. The "at" sign is used for instance variables, two "at" signs for class variables, and the dollar for global variables. In this example, we set the two instance variables `name` and `age` to the given values. Like in Python, attributes do not have to be declared but string into live at the first assignment.

Our class does not do much yet. As a first step, we will override Ruby's method `to_s` which is used by the print function to convert an object to a string. It is pleasant for our incremental presentation that we can add features to the existing `Person` class as we go.

```
class Person
  def to_s
    "Person(name=#{@name}, age=#{age})"
  end
end
=> nil
>> print person
Person(name=Homer, age=55)
=> nil
```

Since the attribute itself uses the `@` prefix, there is no confusion between the attribute and a method with the same name.

```
class Person
  def name
    @name
  end
end
=> nil
>> person.name
=> "Homer"
```

But how to distinguish getter and setter? The setter syntax show another example of Ruby's method suffixes. When assigning a value to a "field" of an object, the method with the same name as the field and the = suffix is called.

```
class Person
  def name=(aName)
    @name = aName
  end
end
=> nil
>> person.name = "Bart"
```

```
=> "Bart"
>> person.name
=> "Bart"
```

We don't need to code the standard accessor methods ourselves. Ruby provides a shortcut for it.

21.3. More Features

21.3.1. Exception Handling

```
def doSomething(x)
  if x < 10
    raise "value to low"
  end
end

begin
  doSomething(5)
rescue
  print "error: #{$!}\n"
ensure
  print "make sure this is done\n"
end


# user-defined exception

class ChainedException < Exception
  attr :no
  attr :cause
  def initialize(no, cause=nil)
    @no = no
    @cause = cause
  end

  def to_s()
    return "no=#{@no}"
  end
end

def raiseException()
  raise ChainedException.new(123)
end

begin
  raiseException()
```

```
rescue ChainedException
  print "error: #{$!}\n"
end
```

21.3.2. Modules and Mixins

Mixin defining the "to_s" method based on reflection.

```
module Display
  def to_s()
    s = ""
    for name in instance_variables
      s += "#{name[1..-1]}=#{instance_eval(name)}\n"
    end
    return s
  end
end

# Tiny address class using the Display mixin
class Address include Display
  attr_reader :street, :city, :country

  def initialize(street, city, country="Germany")
    @street, @city, @country = street, city, country
  end
end

address = Address.new("Am Seestern 4", "Duesseldorf", "Germany")
print address, "\n"
```

21.3.3. Operator Overloading

21.3.4. Regular Expressions

Here is a regular expression looking for the "src" parts in HTML image tags.

```
# regular expression looking for the "src" parts in HTML image tags
# The 'true' argument makes the pattern case insensitive
r = Regexp.compile('(?:input|img).*src=(\'|")(.*)\1', true)

s = "<input type='image' src='images/some.gif'>"
m = r.match(s)
assertEquals("images/some.gif", m[2])
```

```
# same in perl style
s =~ /(?:input|img).*src=(\'|")(.*)\1/
assertEquals("images/some.gif", m[2])
```

The `true` argument of the `compile` method makes the pattern case insensitive.

21.4. Libraries and Common Examples

21.4.1. Input and Output

```
dir = Dir.new("dir")
entries = []
dir.each { |file| entries << file }

assertEquals([".", ".."], entries)

entries = []
Dir.foreach("dir") do |file|
  if file[0..0] != "."
    entries << file
  end
end
assertEquals([], entries)
```

21.4.2. Leftovers

```
#!/usr/bin/env ruby
  l = []
  l << "bla"

x = ""
for i in 0..5 do x += i.to_s end
assertEquals("012345", x)

x = ""
i = 0
begin
  x += i.to_s
  i += 1
end while i <= 5
assertEquals("012345", x)

x = ""
```

```
i = 0
while i <= 5
  x += i.to_s
  i += 1
end
assertEquals("012345", x)

# Can we handle classes as objects? Yes
x = Address
a = x.new("Elfgenweg 14", "Duesseldorf")
print a, "\n"

assertEquals("Germany", a.country)

#----------------------------------------------------------------------
# blocks and iterators

s = ""
(0..5).each { |i| s += i.to_s }
assertEquals("012345", s)

s = ""
(0..5).to_a.reverse_each { |i| s += i.to_s }
assertEquals("543210", s)

assertEquals([2, 4], [1, 2].collect {|i| 2 * i})

#----------------------------------------------------------------------
# Scoping

def testScope()
  # As in python, "if" statements do not define a new scope
  if 1
    innerX = 5
  end
  assertEquals(5, innerX)

  # The same is true for loops
  for i in [1, 2] do
    innerY = 6
  end
  assertEquals(6, innerY)
end

testScope()

#----------------------------------------------------------------------
# Function "pointers"
#
# The function "proc" converts a block into a function (Proc) object.
# This includes the context (here the variable x).

class Button
```

```
  def initialize(label, action)
    @label = label
    @action = action
  end

  def press
    @action.call()
  end
end

x = "this is "
b = Button.new("bla", proc { x + "blub" })
assertEquals("this is blub", b.press)

#----------------------------------------------------------------------
#
```

21.5. Discussion

Ruby programs are among the most compact and readable in recent comparisons. The combination of the object-oriented API and the functional approach to statements and expressions pays off. On the downside is the Perl heritage. Although they might make Ruby more attractive for Perl programmers, the features copied over from Perl don't seem to fit Ruby's otherwise clean design. The same is true for the handling of function objects. Since functions can be called without parameter lists, a special syntax is required to refer to the function itself and calling a function object. Scheme and Python demonstrate with a consistent syntax that this complication is not necessary. You can avoid the special Perl features by the cleaner Ruby functions, but the function handling is an annoyance in an otherwise very interesting language.

References

The best book on Ruby is [THOMAS00]>. It is a passionate introduction to the language and introduced the language to a wider public (beyond people speaking Japanese). Their previous book [HUNT00]> pathed the way by telling developers to learn at least one new programming language every year.

Andrew Hunt and David Thomas, 020161622X, Addison Wesley Longman, 2000, *The Pragmatic Programmer: From Journeyman to Master*.

David Thomas and Andrew Hunt, 0201710897, Addison Wesley Longman, 2000, *Programming Ruby*.

Chapter 22. XSLT

As everybody (who is not completely immune against any IT industry hype) knows, XSLT is the transformation language for XML, the extensible markup language. Why do we tackle such a transformation language in a book about general purpose programming languages? First, of course, it adds a few million people to the potential audience for this book. More seriously, I realized when using XSLT for larger projects that programming XSLT is quite different from the kind of (mostly object-oriented) programming I was used to. Some things become very easy and others extremely tedious.

XSLT ows part of its expressiveness to the powerful XML query language XPath which we will therefore cover to some extent as well.

22.1. Software and Installation

There are many good open source libraries for XSLT available for several languages (most notably for the C-family C, C++, Java, C#). For the examples in this chapter, you can use any of these tools. Because of its simplicity and speed, I have used xsltproc (http://xmlsoft.org/XSLT/xsltproc2.html), the command line interface for the fast XSLT library libxslt (http://xmlsoft.org/XSLT/) written in C. Both, the library and the command line tool are part of every Linux distribution, but are also available as binaries for the Win32 platform.

22.2. Quick Tour

22.2.1. Hello World

XSLT is a transformation language and as such not meant to print "Hello World". However, we can show the required scaffolding by defining a transformation which turns any XML input into our friendly message.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/">Hello World</xsl:template>
</xsl:stylesheet>
```

We put this transformation (or "stylesheet") into a file called `hello.xsl`. To run the program, that is, to apply this transformation, we enter any XML document into a file called `hello.xml`, for example,

```
<?xml version="1.0"?>
<hello/>
```

and run the XSLT processor with the stylesheet and the XML document as the two arguments:

```
> xsltproc hello.xslt hello.xml
Hello World
```

We have seen many "Hello World" programs by now, but this is definitely different. First, an XSLT stylesheet is again an XML document. Similar to Lisp, this means that the same tools can be used for the data and the programs manipulating the data. We can, for example, use XSLT to generate or modify other XSLT stylesheets. On the downside, this also means that we have to deal with a syntax that was not invented for programs. A good XML editor such as James Clark's nxml (http://www.xmlhack.com/read.php?item=2061) mode for emacs (which I'm using to write this book) or one of the many graphical XML tools is indispensible.

After the XML preamble `<?xml version="1.0"?>`, the root element tells that we are defining an XSLT stylesheet and introduces the associated namespace, which is almost always (unless you generate another stylesheet) called `xsl`).

The body of consists of a number of global definitions followed by a list of rules. In our example, there is exactly one of each. The output statement, `<xsl:output method="text"/>`, determines how the output is formatted. The default output method is `xml`, since XSLT main pupose is to transform one XML document into another. The `xml` method ensures that the output is well-formed XML including proper nesting of elements and the escaping of special characters. Since we want to generate plain text, we use the `text` method. The text method is often used in XSLT based code generators, for example, to generate Java classes from XML schemas.

The `template` element defines the transformation rule. As usual, a rule consists of two parts: when to apply the rule and what to do in case the rule is applied. In XSLT, a rule is applied to all the XML nodes matching the XPath expression in the `match` attribute of the `template`. We will dive into XPath in the next paragraph. Our example uses the simplest possible XPath expression `/` matching the root element of an XML document.

Now, how do we get anything to the output stream? When the XSLT processor encounters a non-XSL element or text node in a template, the element or text is copied to the output document. In other words, the one and only rule of our "Hello World" stylesheet, is applied to the root node of the XML document which causes the "Hello World" message to be copied to the output stream.

22.2.2. XPath

Much of the power of XSLT derives from the ability to extract information from XML documents in a very concise way using XPath expressions. As test data for the following examples we take a small bibliography (as used in the docbook (http://www.docbook.org) source of this book) and store it in `biblio.xml`.

```
<?xml version="1.0" ?>
```

```
<bibliography id="biblio.xslt">
  <title>References</title>
  <biblioentry id="Eckstein01">
    <authorgroup>
      <author>
<firstname>Robert</firstname>
<surname>Eckstein</surname>
      </author>
      <author>
<firstname>Michel</firstname>
<surname>Casabianca</surname>
      </author>
    </authorgroup>
    <isbn>0596001339</isbn>
    <publisher>
      <publishername>O’Reilly</publishername>
    </publisher>
    <pubdate>2001</pubdate>
    <title>XML Pocket Reference</title>
  </biblioentry>

  <biblioentry id="Kay03">
    <author>
      <firstname>Michael</firstname>
      <surname>Kay</surname>
    </author>
    <isbn>0764543814</isbn>
    <publisher>
      <publishername>John Wiley</publishername>
    </publisher>
    <pubdate>2003</pubdate>
    <title>XSLT</title>
  </biblioentry>
</bibliography>
```

To start with, we apply an empty stylesheet to this data.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
```

To our surprise, the output is not empty, but contains all the text (that is, the contents of the text nodes) of the document.

```
<?xml version="1.0"?>
  References
Robert
Eckstein
Michel
Casabianca
    0596001339
      O’Reilly
```

```
2001
XML Pocket Reference
  Michael
  Kay
0764543814
  John Wiley
2003
XSLT
```

Whenever the XSLT processor encounters a node for which no template is defined in the stylesheet, it applies a default template. For an element node, the processor applies all templates to children of the element. The default template for text nodes copies the contents to the output. Together, this explains why we see all the text of the XML input document when applying the empty stylesheet.

Like paths in a file system, XPath expressions denote nodes in an XML document. In the simplest case, an XPath expression looks exactly like a (UNIX) directory path. The main difference is that a path may refer to multiple nodes, since an XML element may contain multiple children with the same name. Here is a stylesheet which extracts the titles from our bibliography.

22.2.3. Conditions and Loops

22.2.4. Functions

`template` elements are also used to define functions

22.3. More Features

22.4. Discussion

References

The best book on XML and XSL I know is also one of the shortest: the XSLT pocket reference. About half of the book (that is, just 50 pages) covers XSLT and XPath. If you want (or need) to spend more time on XSLT, have a look at [KAY03]>, written by the developer of the Saxon XSLT processor (who therefore must know XSLT in and out).

Robert Eckstein and Michel Casabianca, 0596001339, O'Reilly, 2001, *XML Pocket Reference: Second Edition*.

Michael Kay, 0764543814, John Wiley, 2003, *XSLT: Second Edition*.

Chapter 23. C#

Microsoft's answer to Java (the language, not the run-time environment) is C# (pronounced "see sharp"). Among the languages of the C family, it stands between Java and C++. C# definitely owes a lot to Java. It is probably easier to migrate from Java to C# than from any other programming language (including C++ and Visual Basic). In many areas C# goes beyond Java by resolving some of its problems (such as the rift between primitive values and objects) and preserving more of C++ features.

23.1. Software and Installation

I've used two environments to run the C# programs: Microsoft's .NET SDK version 1.1 and Mono (www.go-mono.org), Ximian's open source implementation of the .NET framework, version 0.23. In fact, I switched from one to the other without problems when moving the book to the Linux platform. Both compilers are available for free and are easy to install using a Windows installer or a Linux package (Debian package in my case). Microsoft's C# compiler is called `csc` and the Mono C# compiler is `mcs`. To run the test programs, just enter the code with your favorite text editor, apply the compiler with the file containing the C# code (e.g., `csc hello.cs` and start the created executable (e.g., `hello.exe`). There is an emacs mode (http://www.cybercom.net/~zbrad/DotNet/Emacs/)for C#.

23.2. Quick Tour

Because of the similarities to Java, we will concentrate on the differences and extensions C# has to offer.

23.2.1. Hello World

```
class Hello {
  static void Main() {
    System.Console.WriteLine("Hello World");
  }
}
```

The C# version of our favorite program looks almost like the Java code. There are only few visible differences. The `public` qualifiers for the class and the method are missing (but they exist) and the method names start (following Microsoft's C++ convention) with a capital letter. What looks like the direct access to the attribute `Console` is in fact the call of the getter method of a property (more on this below). Otherwise, the program works just like its Java counterpart.

Another difference is the file organization. The name of the source file is independent of the classes contained in the file, and you can put as many classes in a file as you want.

Moving on to expressions, there are not many differences either. However, C# has more than twice as many primitive types. First, C# (or better: the underlying .NET framework and its Common Language Runtime, CLR) supports unsigned integer types such as `uint`. Second, it has a built-in decimal type which is extremely useful for financial calculations (accountants hate rounding errors).

23.2.2. Control Statements

All the familiar control statements work in C# just like they do in the other languages of the C family.

```
class Control {
  static int sign(int x) {
    if (x < 0) {
      return -1;
    }
    else if (x == 0) {
      return 0;
    }
    else {
      return 1;
    }
  }

  static void Main() {
    System.Console.WriteLine("sign(55)={0}", sign(55));
  }
}
```

Just like Java, C# insists on boolean expressions for the conditions.

The standard loops also work exactly as in Java. In particular, we can can define loop variables inside the `for` statement. The loop variable have to be of the same type and must not already be used in the current scope.

```
class Control {
  static void Main() {
    int i = 0;
    while (i < 3) {
      System.Console.WriteLine("i={0}", i);
      i++;
    }

    i = 0;
    do {
      System.Console.WriteLine("i={0}", i);
      i++;
```

```
    } while (i < 3);

    for (int j=0, k=1; j<3; j++, k += 2) {
      System.Console.WriteLine("j={0}, k={1}", j, k);
    }
  }
}
```

Despite the similarities between Java and C#, we find some small but useful extension in almost every area of the language. With regards to control statements, the first one is the `foreach` loop which simplifies the iteration through a collection.

```
class Control {
  static void Main() {
    int[] numbers = { 1, 3, 5};
    foreach (int i in numbers) {
      System.Console.WriteLine("i={0}", i);
    }
  }
}
```

The little extension is so convenient that the Java language added it in version 1.5. In order not to upset the Java community by introducing a new keyword, Java replaced the "`in`" (which I personally use quite often for input streams) by a colon.

The second extension is the `switch` statement. In addition to integer expression, we can also use strings in the `switch` expression.

```
class Control {
  static void Main() {
    int i = 0;
    switch ("t" + "wo") {
    case "one":
      i = 1;
      break;
    case "two":
      i = 2;
      break;
    case "three":
      i = 3;
      break;
    }
    System.Console.WriteLine("i={0}", i);
  }
}
```

The strings in the `case` clauses must be constants. This allows for an efficient implementation of the `switch` expression using "interned" strings. Basically, the string constants are stored in a hash table, and the implementation of the `switch` statement evaluates the string expression and tries to find it in the

hash table. If it is not in the table, it is not interned and therefore does not match any of the `case` strings. If the string is found in the hash table, it can be efficiently compared to the `case` strings using the address of the interned strings.

23.2.3. Classes

The following definition of our person class again shows a lot of resemblance to the Java version, but also demonstrates the use of properties which goes beyond a naming convention as in Java.

```
public class Person {
  string name;
  int age;

  public Person(string name, int age) {
    this.name = name;
    this.age = age;
  }

  public string Name {
    get { return this.name; }
  }

  public int Age {
    get { return this.age; }
    set {
      System.Console.WriteLine("setting age to {0}", value);
      this.age = value;
    }
  }
}
```

The class defines two properties, a read-only property for the name and a read-write property for the age of the person. By omitting all the signatures, C# keeps the syntax short and consistent. The argument passed to the setter is implicitly available as the `value`. We can now use the property just like an attribute, but behind the scenes the getter and setter methods will be called.

```
class Test {
  static void Main() {
    Person person = new Person("Homer", 55);
    person.Age = 66;
    System.Console.WriteLine("name={0}, age={1}", person.Name, person.Age);
  }
}

result:
setting age to 66
name=Homer, age=66
```

With regards to inheritance, C# follows the C++ model more closely than Java. Methods, for example, do not use dynamic dispatching by default. Instead, you have to declare a method as `virtual` in the base class.

```
public class Person {
  string name;
  int age;

  public Person(string name, int age) {
    this.name = name;
    this.age = age;
  }

  public virtual string Hello() {
    return "Hello, I am " + this.name;
  }
}
```

In the derived class, we have to tell the compiler explicitly that we are about to override a virtual method.

```
public class Employee : Person {
  private int number;

  public Employee(string name, int age, int number) : base(name, age) {
    this.number = number;
  }

  public override string Hello() {
    return "Hello, I am number " + this.number;
  }
}
```

Also notice the C++ syntax with respect to the base class. Calling the constructor of the base class looks similar to C++ but with the name of the parent class replaced by `base`. At least the syntax is clearer than Java's call to super which looks like a normal method call, but has to be the first statement in a constructor. We can also call another constructor of the same class using `this` instead of `base`

```
public class Person {
  string name;
  int age;

  public Person(string name, int age) {
    this.name = name;
    this.age = age;
  }

  public Person(string name) : this(name, 0) {}
  ...
}
```

From a client perspective, the code looks the same in Java and C#.

```
class Test {
  static void Main() {
    Person person = new Employee("Homer", 55, 1234);
    System.Console.WriteLine(person.Hello());
  }
}

result:
Hello, I am number 1234
```

23.2.4. Collections

Like Java and in contrast to C++ and Eiffel, the first version of the .NET framework (and therefore C#) does not support generic classes. Hence, C#'s collection classes are similar to Java and the dynamically typed languages (e.g., Smalltalk). Here is an example demonstrating the basic usage of lists and iterators (called enumerators in C#).

```
using System;
using System.Collections;

public class ListTest {
  public static void Main() {
    IList list = new ArrayList();
    list.Add("blah");
    list.Add("blub");

    Console.WriteLine("list contains 'blah'? " + list.Contains("blah"));

    for (IEnumerator i=list.GetEnumerator(); i.MoveNext(); ) {
      string item = (string)i.Current;
      Console.WriteLine("item=" + item);
    }

    foreach (string item in list) {
      Console.WriteLine("item=" + item);
    }
  }
}
```

The enumerator semantics also look like a compromise between Java and C++ iterators. An enumerator starts in front of its first element. Each call to `MoveNext` moves it forward to the next element or returns

false if there is not element left. The `Current` property gives us access to the current element. There is also a `Reset` method which put the enumerator back to its initial position in front of the first element.

Note that we do not need to case when using the `foreach` loop. In interesting twist is added when storing value types in a collection. The collections always contain reference types, but the automatic boxing and simple unboxing make the difference almost invisible.

```
using System;
using System.Collections;

public class ListTest {
  public static void Main() {
    IList list = new ArrayList();
    list.Add(55);
    list.Add(66);

    Console.WriteLine("list contains 55? " + list.Contains(55));

    for (IEnumerator i=list.GetEnumerator(); i.MoveNext(); ) {
      int item = (int)i.Current;
      Console.WriteLine("item=" + item);
    }

    foreach (int item in list) {
      Console.WriteLine("item=" + item);
    }
  }
}
```

Besides lists, C# supports the whole range of collections such as dictionaries, sets, stacks, and so forth. Here is an example using the hash table implementation of a dictionary.

```
using System;
using System.Collections;

public class DictionaryTest {
  public static void Main() {
    IDictionary map = new Hashtable();
    map["blah"] = 55;
    map["blub"] = 66;

    foreach (string key in map.Keys) {
      Console.WriteLine("map[" + key + "]=" + map[key]);
    }
    for (IDictionaryEnumerator i=map.GetEnumerator(); i.MoveNext(); ) {
      Console.WriteLine("map[" + i.Key + "]=" + i.Value);
    }
  }
}
```

A dictionary gives us access to its keys and values as collections which can be used conveniently in a `foreach`. With the dictionary enumerator it is also possible to iterate through keys and values at the same time.

23.3. More Features

23.3.1. Delegates and Events

Java's inner classes (in particular anonymous ones) took me a while to understand when they were introduced with JDK 1.1 as part of the new event handling. Being spoiled by Python's "method objects", they seem like a rather complex way to handle callback functions. C#'s solution resembles Eiffel's agents, but with more syntax support.

```
class Test {
  public delegate double DoubleFunction(double x);

  static void Evaluate(DoubleFunction f, double x) {
    System.Console.WriteLine("f(" + x + ")=" + f(x));
  }

  static double Times2(double x) { return 2*x; }

  static void Main() {
    Evaluate(new DoubleFunction(Times2), 5.5);
  }
}
```

A `delegate` is a function type. In the example, we define the type `DoubleFunction` for functions with a single double argument and returning a double. We can now use this new type just like any other type to declare variables and method arguments as we have done in the `Evaluate` method. Calling a delegate looks just like a function call. There is no automatic transformation from a method to a delegate. Instead, we create the delegate (or function object) by passing the name of the method to the constructor of the delegate.

The previous example used static methods only, but we can do the same with instance methods.

```
class Test {
  public delegate double DoubleFunction(double x);

  static void Main() {
    LinearFunction f = new LinearFunction(2, 3);
```

```
    Evaluate(new DoubleFunction(f.Compute), 5.5);
  }

  static void Evaluate(DoubleFunction f, double x) {
    System.Console.WriteLine("f(" + x + ")=" + f(x));
  }
}

class LinearFunction {
  private double a, b;

  public LinearFunction(double a, double b) {
    this.a = a;
    this.b = b;
  }

  public double Compute(double x) {
    return a*x + b;
  }
}

result:
f(5.5)=14
```

Here, we define a class `LinearFunction` modeling scalar linear functions of the form `f(x)=a*x+b`. In the main program, we first construct a specific instance of a linear function and then pass its `Compute` method wrapped into the `DoubleFunction` delegate to the evaluation method.

For now, delegates look just like function objects, but C# goes a little further and allows us to add and subtract delegates. What are these operations supposed to mean for functions?

```
class Test {
  public delegate int Printer(string message);

  static void Main() {
    Printer print1 = new Printer(Print1);
    Printer print2 = new Printer(Print2);
    Printer printer = print1 + print2;
    int result = printer("Hello World");
    System.Console.WriteLine("result=" + result);

    printer -= print1;
    result = printer("Hello World");
    System.Console.WriteLine("result=" + result);
  }

  static int Print1(string message) {
    System.Console.WriteLine("Print1: " + message);
    return 1;
  }
```

```
  static int Print2(string message) {
    System.Console.WriteLine("Print2: " + message);
    return 2;
  }
}

result:

Print1: Hello World
Print2: Hello World
result=2
Print2: Hello World
result=2
```

When we add two delegates, and call the result, the two functions are executed one after the other. The return value of the combined delegate is the return value of the last function executed. Similarly, we can subtract a delegate from the sum and end up with the remaining one.

The main application of function objects in Windows programs are callbacks for graphical user interfaces. Their design often follows the observer (publish-subscribe) pattern. Hence, the "delegate arithmetic" is interpreted as adding or removing a function from a list of callbacks.

You may wonder at this point (like I did) what happens if we try and subtract the other delegate as well. The subtracting itself is carried out without any complaint, but calling the result causes a null reference exception.

C# gives the observer pattern its place in the syntax with the introduction of *events*. Besides attributes and methods, events are the third kind of member a class may have. An event wraps a delegate and presents it differently to the class itself and the outside world. From the outside, we can only add and remove callbacks with the `+=` and `-=` operators. Inside of the class containing the event, we can "fire" the event by calling it just like a delegate.

```
class Test {
  public delegate int Printer(string message);
  public event Printer OnPrint;

  static void Main() {
    new Test().run();
  }

  public void run() {
    Printer print1 = new Printer(Print1);
    Printer print2 = new Printer(Print2);

    OnPrint += print1;
    OnPrint += print2;
    System.Console.WriteLine("result=" + OnPrint("Hello World"));

    OnPrint -= print2;
```

```
    System.Console.WriteLine("result=" + OnPrint("Hello World"));


    OnPrint -= print1;
    if (OnPrint == null) {
      System.Console.WriteLine("empty callback list");
    }
  }
  ...
}

result:

Print1: Hello World
Print2: Hello World
result=2
Print1: Hello World
result=1
empty callback list
```

The event is declared in the class like a delegate attribute, but with the keyword `event` as an additional modifier. Unfortunately, we can not call an event if it has no subscribers. Doing so will result again in a null reference exception. However, we can prevent this situation, since an empty event evaluates to null.

23.3.2. Operator Overloading

As we have noticed already, C# keeps a lot more of the C++ features than Java. One more example is operator overloading. The underlying principle is that we should be able to define our own types which look and work the same way as the built-in types such as integers and strings. To this end, we need to define the meaning of operators for our own types.

```
public class Point {
  private double x, y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public static Point operator+(Point a, Point b) {
    return new Point(a.x + b.x, a.y + b.y);
  }

  public override string ToString() {
    return "(" + this.x + ", " + this.y + ")";
  }
}
```

```
public class Test {
  static void Main() {
    Point a = new Point(1.5, 2.5), b = new Point(2.5, 3.5);
    Point c = a + b;
    System.Console.WriteLine("c=" + c);
  }
}
```

23.3.3. Casting References

In a strongly typed object-oriented language (especially without generics), we are often forced to cast a reference. And although not proper OO-style, we also encounter situations where we need to check the type of an object and, if it is the expected type, cast the reference so that we can handle the special type. C# offers the two convenient operators `is` and `as` for this reason.

```
class Hello {
  static void Main() {
    object o = "hello";

    if (o is string) {
      System.Console.WriteLine("this is a string");
    }

    string s = o as string;

    if (s == null) {
      System.Console.WriteLine("not a string");
    }
    else {
      System.Console.WriteLine("s={0}", s);
    }
  }
}
```

```
this is a string
s=hello
```

The `is` operator is the crisp version of Java's `instanceof`.The `as` operator combines the type check with the casting (thus saving one type check). If the reference is of the expected type, it is cast to this type. Otherwise the `as` operator return a null reference.

23.3.4. Enumerated Types

Enumerated types are another example of a C/C++ feature which Java dropped to keep the language simple. C# follows Ada in supporting enumerated types as first class types. An enumerated type is

defined like in C, but the values are not just integers in disguise, but objects which allow us to get the numeric value as well as the string representation.

```
using System;
class Hello {
  enum Color {
    Red, Green, Blue
  }
  static void Main() {
    foreach (Color color in Enum.GetValues(typeof(Color))) {
      Console.WriteLine("color: {0} {1}", (int)color, color.ToString());
    }

    Color c = (Color)Enum.Parse(typeof(Color), "Green");
    Console.WriteLine("c: {0}", c.ToString());
  }
}
```

```
color: 0 Red
color: 1 Green
color: 2 Blue
c: Green
```

The loop walks through all the values of out enumerated type, which we obtain by applying the static `GetValues` method to the `Color` type. The print statement shows the conversion of the color value to an integer and a string. The last two lines demonstrate the opposite direction convertion a string to a `Color` value using the `Parse` method.

By adding the `Flags` attribute we can change the semantics of the `ToString` and `Parse` methods so that the enumerated type behaves like a bit set.

```
using System;
class Hello {
  [Flags]
  enum Permission {
    Read = 0x1, Write = 0x2, Execute = 0x4
  }
  static void Main() {
    Console.WriteLine(
      "permissions: {0}", Permission.Read | Permission.Execute);

    Permission p = (Permission)Enum.Parse(typeof(Permission), "Read, Write");
    Console.WriteLine("p: {0}", p.ToString());
  }
}
```

```
permissions: Read, Execute
p: Read, Write
```

The .NET framework library contains many examples for this kind of enumeration (e.g., `FileAttributes`).

23.3.5. Releasing Resources

Like in any programming environment with garbage collection we can't rely on destructors to release resources such as file handles or network connections, since we do not know when the destructors are called (if they are called at all). We therefore have to ensure manually that our resources are closed under all circumstances. In Java, this leads to the idiom of a try-finally statement which works as well in C#.

```
using System.IO;

class Hello {
  static void Main() {
    StreamWriter writer = null;
    try {
      writer = new StreamWriter("hello.txt", false);
      writer.WriteLine("Hello World");
    }
    finally {
      if (writer != null) {
writer.Close();
      }
    }
  }
}
```

Besides the required number of lines it is disturbing that we have to widen the scope of the `writer` variable only to be able to close it in the `finally` block. Fortunately, C# has a special syntax (similar to Lisp's `with-open-file`) which takes care of the closing of resources automatically just like a destructor in C++ would do.

```
using System.IO;

class Hello {
  static void Main() {
    using (StreamWriter writer = new StreamWriter("hello.txt", false)) {
      writer.WriteLine("Hello World");
    }
  }
}
```

The `using` statement creates a new scope for the variable `writer` demarcated by the curly braces. When this scope is left, whether by successfully executing the block or by throwing an exception, the `Dispose` method of the declared object is called. The mechanism works for all objects implementing the `IDisposable` interface. All the classes representing system resources such as files, sockets, database connections in the .NET framework library implement this interface.

23.4. Discussion

C# definitely looks like an improved Java. Many of the improvements are useful enough for Java to adopt them (as has been done with Java 1.5). For some of them such as properties and events would require too many changes to the Java language giving C# the advantage of being the newer language. To me, the most profound differences are the type model (of the CLR) supporting user-defined reference and value types, and the attributes that open a whole new range of possibilities for defining and using meta data in the code as opposed to the sometimes overwhelming XML configuration files in the J2EE world.

I tend to agree with Bertrand Meyer (and the Smalltalk and Java people) that dynamic dispatch should be the default in object oriented systems. Otherwise you need to change a base class (and recompile all classes depending on it) whenever you find out that a method needs to be virtual later in the development process.

C#'s delegates are definitely a step forward when compared to Java's inner classes and Eiffel's agents, but are still far away from the convenience of Smalltalk, Python, or the functional languages.

Chapter 24. Thoughts

No, this chapter does not announce the winner of the programming language competition. I'm not even attempting to compare the languages side by side. You will have noticed anyway which languages I find more interesting. This chapter just captures some thoughts that came to my mind when collecting the information about the programming languages describes in this book.

24.1. Paradigm

How much should a programming language guide a developer's thinking? Some of the languages clearly favor a specific paradigm, for example, Eiffel object-orientation and Haskell functional programming. Others are more open, most notable Lisp which has been able to swallow all the paradigms of the last 40 years (functional, procedural, object-oriented, meta-programming).

In my opinion, we are currently observing a convergence of the major approaches to programming. There is definitely a renaissance of functional programming after the object-oriented era. Object-orientation as a means to control state (and side effects) is good, but avoiding side effects wherever possible is better. I guess it is fair to say, that the C family is adding more and more features from the functional world. Just think about variable declarations anywhere in the code (almost a let expression), C#'s delegates and Java's inner classes (almost first class function objects and closures).

The modern scripting languages, Python and Ruby, are going into the same direction. Python's list comprehensions (adopted from Haskell), nested scopes, and iterators render a number of procedural constructs obsolete. Ruby started with a more functional approach (everything is an expression) from the very beginning. Python is still ahead in terms of "callable objects".

24.2. Typing

One of the big debates concerning programming languages is strong (static, compile-time) versus weak (dynamic, run-time) typing. Much of the success of the so-called scripting languages is due to dynamic typing adopted by Perl, Python, Ruby, and the like. Code size, readability, speed of development, learning curve are all definite advantages of this approach. The drawbacks of dynamic typing are mainly in the areas of run-time speed, robustness (run-time type errors), and well-defined interfaces (components).

Most statically typed languages are explicitly typed in the sense that the developer has to tell the compiler which types to use. One of the difficulties is the handling of container types such as collections. If an explicitly typed language does not want to give up the main benefits (speed, rubustness, strongly-typed interfaces) of strong typing for container types, it has to offer parametrized types. That's why Java and C# had to eventually add generics (with Java not gaining the performance benefits, since its virtual machine does not support the generic types). Obviously, this complicates the languages significantly with C++ being the "best" example.

ML and Haskell offer a different solution: implicit typing using type inference. Here, the developer provides only the type information that can not be derived from the context. The compiler always chooses the most general type applicable. In a way, ML and Haskell make templates (generic types) the default, but without any of the hazzle required in an explicitly typed language such as C++.

It looks like the difference between strong and weak typing is getting more and more blurred. A good Lisp compiler is able to generate code that is almost as fast as C. Obviously, the ML and Haskell dialects also produce executables that are as fast as their explicitly typed counterparts. It is possible that type inference will enter the mainstream just like automatic memory management (with garbage collection) has become a standard feature of modern programming languages.

24.3. Syntax

The syntax of programming languages mainly seems to be a matter of taste and habit. Lisp's minimal syntax with the equivalence between code and data still provides unmatched extensibility (just compare meta programming in C++ and Lisp), but suffers from readibility. Learning the basic syntax is simple, but keeping the thousands of forms in mind (without supporting syntax) is hard.

Using whitespace as part of the syntax (Python, Haskell), seems to cause too much of a (useless) debate. Personally, I think that Eiffel (apart from the bang-bang syntax for instantiation) has the most readible syntax of the "conventional" languages.

A simple means to avoid repetitive definitions of function is a flexible handling of arguments. In respect, Python stands out with its support of default arguments, named arguments, variable arguments lists and arbitrary keyword arguments. Combined with the (relatively new) syntax to apply a function to an argument list and keyword argument map, there is not room for improvement left. Among the other languages, Lisp probably comes closest. It is surprising to me, that Java and C# did not keep C++'s default arguments (VB.NET does support default arguments).