Functional Programming in CLEAN

DRAFT

JULY 1999

Functional Programming in CLEAN

Preface

Functional languages enable programmers to concentrate on the problem one would like to solve without being forced to worry too much about all kinds of uninteresting implementation details. A functional program can be regarded as an executable specification. Functional programming languages are therefore popular in educational and research environments. Functional languages are very suited for teaching students the first principles of programming. In research environments they are used for rapid prototyping of complex systems. Recent developments in the implementation techniques and new insights in the underlying concepts such as input/output handling make that modern functional languages can nowadays also be used successfully for the development of real world applications.

The purpose of this book is to teach practical programming skills using the state-of-the art pure functional language CONCURRENT CLEAN. CLEAN has many aspects in common with other modern functional languages like MIRANDA, HASKELL and ML. In addition CLEAN offers additional support for the development of stand-alone window based applications and it offers support for process communication and for the development of distributed applications.

This book on functional programming using CLEAN is split into three parts.

In the first part an introduction into functional programming is given. In six Chapters we treat the basic aspects of functional programming, functions, data structures, the type system and I/O handling. The idea is that you are able to write simple functions and applications as soon as possible. The intention is not to treat all available features of CLEAN but only the most important ones that are most commonly used. A complete description of all available language constructs can be found in the CLEAN language manual.

The main emphasis of this book lies in the second part in which several case studies are presented. Each case treats a tiny, but complete application in an illustrative problem domain. The case studies include applications like a simple database, an object-oriented drawing program, a data compression utility, an interpreter for a functional language. Each case furthermore illustrates a certain aspect of functional programming. Some case applications are reused in others to illustrate the reusability of code.

In the third part of this book we discuss the different kinds of programming development techniques for functional programming and efficiency aspects are treated.

So, a lot of material is presented in this book. However, one certainly does not have to work through all case studies. Depending on the programming experience already acquired and the time available one can use this book as a textbook for or one or two semester course on functional programming. The book can be used as an introductory textbook for people with little programming experience. It can also be used for people who already have programming experience in other programming paradigm (imperative, object-oriented or logical) and now want to learn how to develop applications in a pure functional language.

We hope that you enjoy the book and that it will stimulate you to use a functional language for the development of your applications.

Table of Contents

Preface i

Table of Contents iii

Introduction to Functional Programming 1

1.1 Functional languages 1
1.2 Programming with functions 2
1.2.1 The `Start' expression 2
1.2.2 Defining new functions 3
1.2.3 Program evaluation with functions 4
1.3 Standard functions 5
1.3.1 Names of functions and operators 5
1.3.2 Predefined functions on numbers 5
1.3.3 Predefined functions on Booleans 6
1.3.4 Predefined functions on lists 6
1.3.5 Predefined functions on functions 7
1.4 Defining functions 7
1.4.1 Definition by combination 7
1.4.2 Definition by cases 8
1.4.3 Definition using patterns 9
1.4.4 Definition by induction or recursion 10
1.4.5 Local definitions, scope and lay-out 11
1.4.6 Comments 12
1.5 Types 12
1.5.1 Sorts of errors 13
1.5.2 Typing of expressions 14
1.5.3 Polymorphism 15
1.5.4 Functions with more than one argument 15
1.5.5 Overloading 16
1.5.6 Type annotations and attributes 17
1.5.7 Well-formed Types 17
1.6 Synonym definitions 19
1.6.1 Global constant functions (CAF's) 19
1.6.2 Macro's and type synonyms 19
1.7 Modules 20
1.8 Overview 22
1.9 Exercises 22

Functions and Numbers 23

2.1 Operators 23
2.1.1 Operators as functions and vice versa 23
2.1.2 Priorities 23
2.1.3 Association 24
2.1.4 Definition of operators 25

2.2 Partial parameterization 26
2.2.1 Currying of functions 26
2.3 Functions as argument 27
2.3.1 Functions on lists 28
2.3.2 Iteration 28
2.3.3 The lambda notation 29
2.3.4 Function composition 30
2.4 Numerical functions 31
2.4.1 Calculations with integers 31
2.4.2 Calculations with reals 34
2.5 Exercises 36

Data Structures 37

3.1 Lists 37
3.1.1 Structure of a list 37
3.1.2 Functions on lists 40
3.1.3 Higher order functions on lists 44
3.1.4 Sorting lists 47
3.1.5 List comprehensions 48
3.2 Infinite lists 51
3.2.1 Enumerating all numbers 51
3.2.2 Lazy evaluation 52
3.2.3 Functions generating infinite lists 53
3.2.4 Displaying a number as a list of characters 53
3.2.5 The list of all prime numbers 54
3.3 Tuples 55
3.3.1 Tuples and lists 57
3.4 Records 58
3.4.1 Rational numbers 60
3.5 Arrays 61
3.5.1 Array comprehensions 62
3.5.2 Lazy, strict and unboxed arrays 63
3.5.3 Array updates 63
3.5.4 Array patterns 64
3.6 Algebraic datatypes 64
3.6.1 Tree definitions 66
3.6.2 Search trees 67
3.6.3 Sorting using search trees 69
3.6.4 Deleting from search trees 70
3.7 Abstract datatypes 70
3.8 Correctness of programs 71
3.8.1 Direct proofs 72
3.8.2 Proof by case distinction 73
3.8.3 Proof by induction 74
3.8.4 Program synthesis 76
3.9 Run-time errors 77
3.9.1 Non-termination 78
3.9.2 Partial functions 78
3.9.3 Cyclic dependencies 79
3.9.4 Insufficient memory 80
3.10 Exercises 82

The Power of Types **83**

4.1 Type Classes 83
4.1.2 A class for Rational Numbers 87
4.1.3 Internal overloading 88
4.1.4 Derived class members 88
4.1.5 Type constructor classes 89
4.2 Existential types 90
4.3 Uniqueness types 96
4.3.1 Graph Reduction 96
4.3.2 Destructive updating 98
4.3.4 Uniqueness information 99
4.3.5 Uniqueness typing 100
4.3.5 Nested scope style 102
4.3.6 Propagation of uniqueness 103
4.3.7 Uniqueness polymorphism 104
4.3.8 Attributed datatypes 105
4.3.9 Higher order uniqueness typing 107
4.3.10 Creating unique objects 108
4.4 Exercises 108

Interactive Programs **109**

5.1 Changing files in the World 109
5.2 Environment Passing Techniques 112
5.2.1 Nested scope style 113
5.2.2 Monadic style 114
5.2.3 Tracing program execution 115
5.3 Handling Events 116
5.3.1 Events 116
5.4 Dialogs 118
5.4.1 A Hello World Dialog 119
5.4.2 A File Copy Dialog 120
5.4.3 Function Test Dialogs 122
5.4.4 An Input Dialog for a Menu Function 126
5.4.5 Generic Notices 127
5.5 Windows 129
5.5.1 Hello World in a Window 130
5.5.2 Peano Curves 131
5.5.3 A Window to show Text 135
5.6 Timers 139
5.7 A Line Drawing Program 140
5.8 Exercises 147

Efficiency of programs **149**

6.1 Reasoning about efficiency 149
6.1.1 Upper bounds 150
6.1.2 Under bounds 151
6.1.3 Tight upper bounds 152
6.2 Counting reduction steps 152
6.2.1 Memorization 153
6.2.2 Determining the complexity for recursive functions 154

6.2.3 Manipulation recursive data structures 156
6.2.4 Estimating the average complexity 157
6.2.5 Determining upper bounds and under bounds 160
6.3 Constant factors 160
6.3.1 Generating a pseudo random list 162
6.3.2 Measurements 163
6.3.3 Other ways to speed up programs 164
6.4 Exploiting Strictness 166
6.5 Unboxed values 167
6.6 The cost of Currying 169
6.6.1 Folding to the right or to the left 171
6.7 Exercises 172

Index **173**

Part I
Chapter 1
Introduction to Functional Programming

1.1 Functional languages
1.2 Programming with functions
1.3 Standard functions
1.4 Defining functions
1.5 Types
1.6 Synonym definitions
1.7 Modules
1.8 Overview
1.9 Exercises

1.1 Functional languages

Many centuries before the advent of digital computers, functions have been used to describe the relation between input and output of processes. Computer programs, too, are descriptions of the way a result can be computed, given some arguments. A natural way to write a computer program is therefore to define some functions and applying them to concrete values.

We need not to constrain ourselves to *numeric* functions. Functions can also be defined that have, e.g., sequences of numbers as argument. Also, the result of a function can be some compound structure. In this way, functions can be used to model processes with large, structured, input and output.

The first programming language based on the notion of functions was LISP, developed in the early 60s by John McCarthy. The name is an abbreviation of `list processor', which reflects the fact that functions can operate on lists (sequences) of values. An important feature of the language was that functions themselves can be used as argument to other functions.

Experience with developing large programs has showed that the ability to *check* programs before they are ran is most useful. Apart from the syntactical correctness of a program, the compiler can check whether it actually makes sense to apply a given function to a particular argument. This is called type checking. For example, a program where the square root of a list is taken, is considered to be incorrectly typed and is therefore rejected by the compiler.

In the last decade, functional languages have been developed in which a *type system* ensures the type correctness of programs. Some examples are ML, MIRANDA, HASKELL, and CLEAN. As functions can be used as arguments of other functions, functions are `values' in some sense. The ability of defining functions operating on functions and having functions as a result (higher-order functions) is an important feature of these functional languages.

In this book, we will use the language CLEAN. Compared to the other languages mentioned above, CLEAN provides an accurate control over the exact manipulations needed to execute a program. There is a library that offers easy access to functions manipulating the user interface in a platform independent way. Also, the type system is enriched with uniqueness types, making it possible for implementations to improve the efficiency of program execution. Finally, the CLEAN development system is fast and generates very efficient applications.

1.2 Programming with functions

In a functional programming language like CLEAN one defines functions. The functions can be used in an expression, of which the value must be computed.

The CLEAN compiler is a program that translates a CLEAN program into an executable application. The execution of such an application consists of the evaluation of an indicated expression given the functions you have defined in the program.

1.2.1 The `Start' expression

The expression to be evaluated is named `Start`. By providing an appropriate definition for the function `Start`, you can evaluate the desired expression. For example:

```
Start = 5+2*3
```

When this `Start` expression is evaluated, the result of the evaluation, '`11`', will be shown to the user. For the evaluation of the start expression, other functions have to be applied. In this case the operators `+` and `*`. The operators `+` and `*` are actually special functions which have been predefined in the *standard library* which is part of the CLEAN system.

The standard library consists of several modules. Each module is stored in a separate file. Each module contains the definition of a collection of functions and operators that somehow belong to each other.

In the program you write you have to specify which of the predefined functions you would like to use in your program. For the time being you just simply add the line

```
import StdEnv
```

and all commonly used predefined functions from the standard library, called the *standard environment*, can be used. The program you write yourself is a kind of module as well. It therefore should have a name, say

```
module test
```

and be stored in a file which in that case must have the name `test.icl`. So, an example of a tiny but complete CLEAN program which can be translated by the compiler into an executable application is:

```
module test

import StdEnv
```

In the library commonly used mathematical functions are available, such as the square root function. For example, when the start expression

```
Start = 5+2*3
Start = sqrt(2.0)
```

is evaluated, the value `1.4142135623731` is displayed to the user.

Functions are, of course, heavily used in a functional language. To reduce notational complexity in expressions, the parentheses around the argument of a function is commonly omitted. Thus, the expression below is also valid:

```
Start = sqrt 2.0
```

This is a digression from mathematical practice that juxtaposition of expressions indicates multiplication. In CLEAN multiplication must be written explicitly, using the `*` operator. As function application occurs far more often than multiplication in functional programming practice, this reduces notational burden. The following would be a correct `Start` expression:

```
Start = sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3
```

A sequence of numbers can be put into a *list* in CLEAN. Lists are denoted with square brackets. There is a number of standard functions operating on lists:

```
Start = sum [1..10]
```

In this example `[1..10]` is the CLEAN notation for the list of numbers from `1` to `10`. The standard function `sum` can be applied to such a list to calculate the sum (`55`) of those numbers. Just as with `sqrt` and `sin` the (round) parentheses are redundant when calling the function `sum`.

A list is one of the ways to compose data, making it possible to apply functions to large amounts of data. Lists can also be the result of a function. Execution of the program

```
Start = reverse [1..10]
```

will display the list `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]` to the user. The standard function `reverse` reverses the order of a list.

There are more standard functions manipulating lists. What they do can often be guessed from the name: `length` determines the length of a list, `sort` sorts the elements of a list from small to large.

In an expression more functions can be combined. It is for example possible to first sort a list and then reverse it. The program

```
Start = reverse (sort [1,6,2,9,2,7])
```

will sort the numbers in the list, and then reverse the resulting list. The result `[9, 7, 6, 2, 2, 1]` is displayed to the user. As conventional in mathematical literature, `g (f x)` means that `f` should be applied to `x` and `g` should be applied to the result of that. The parentheses in this example are (even in CLEAN!) necessary, to indicate that `(f x)` is an argument to `g` as a whole.

1.2.2 Defining new functions

In a functional programming language it is possible to define new functions by yourself. The function can be used like the predefined functions from the standard environment, in the `Start` expression and in other function definitions. Definitions of functions are always part of a module. Such a module is always stored in a file.

For instance, a function `fac`, which calculates the factorial of a number, can be defined. The factorial of a number n is the product of all numbers between 1 and n. For example, the factorial of 4 is 1*2*3*4 = 24. The `fac` function and its use in the `Start` expression can be defined in a CLEAN program:

```
fac n = prod [1..n]

Start = fac 6
```

The value of the `Start` expression, `720`, will be shown to the user.

Functions that are defined can be used in other functions as well. A function that can make use of the `fac` function is `over`. It calculates the number of possibilities in which k objects can be chosen from a collection of n objects. According to statistics literature this number equals

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

These numbers are called binomial coefficients, $\binom{n}{k}$ is pronounced as *n over k*. The definition can, just as with `fac`, be almost literally been written down in CLEAN:

```
over n k = fac n / (fac k * fac (n-k))

Start = over 10 3
```


The *arguments* appearing on the left-hand side of a function definition, like `n` and `k` in the function `over`, are called the *formal arguments* or *formal parameters* of the function. For using it, one applies a function with *actual arguments* (also called *actual parameters*). For example, on the right-hand side of the start expression the function over is applied to the actual arguments `3` and `120`. The actual argument corresponding to `n` is `3`, and the actual argument corresponding to `k` is `120`.

When run, this program displays the number of ways a committee of three persons can be chosen from a group of ten people (`120`).

Apart from functions, also *constants* may be defined. This might be useful for definitions like

```
pi = 3.1415926
```

Another example of a constant is `Start`, which must be defined in every program. In fact, constants are just functions without arguments.

1.2.3 Program evaluation with functions

So, a functional program generally consists of a collection of function definitions and one initial expression (the `Start` expression). The execution of a functional program starts with the evaluation of the initial expression (the `Start` expression). This initial expression is repeatedly replaced by its result after evaluating a function application. This process of evaluation is called *reduction*. One step of the process, evaluating a single function application, is called a *reduction step*. This step consists of the replacement of a part of the expression which matches a function definition (this part is called the *redex*, a *red*ucable *ex*pression) with (a copy of) the function definition in which for the formal arguments uniformly the actual arguments are substituted. When the expression contains no redexes reduction cannot take place anymore: the expression is said to be in *normal form*. In principle, the normal form of the start expression is the result of the evaluation of the program.

Suppose we define a function as follows

```
extremelyUsefulFunction x = (x + 19) * x
```

A program using this function consists then of a start expression

```
Start = extremelyUsefulFunction 2
```

This expression will be reduced as follows (the arrow → indicates a reduction step, the redex which is reduced is underlined):

```
Start
→ extremelyUsefulFunction 2
→ (2 + 19) * 2
→ 21 * 2
→ 42
```

So, the result of evaluating this extremely useless program is `42`. In other words, `42` is the normal form of `extremelyUsefulFunction 2`.

1.3 Standard functions

1.3.1 Names of functions and operators

In the CLEAN standard environment, a large number of standard functions is predefined. We will discuss some of them in the subsections below.

The rules for names of functions are rather liberal. Function names start with a letter, followed by zero or more letters, digits, or the symbol _ or `. Both lower and upper case letters are allowed, and treated as distinct symbols. Some examples are:

```
f   sum x3   Ab   g`   to_the_power_of     AverageLengthOfTheDutchPopulation
```

The underscore sign is mostly used to make long names easier to read. Another way to achieve that is to start each word in the identifier with a capital. This is a common convention in many programming languages.

Numbers and back-quotes in a name can be used to emphasize the dependencies of some functions or parameters. However, this is only meant for the human reader. As far as the CLEAN compiler is concerned, the name `x3` is as related to `x2` as to `qX`a_y`. Although the names of functions and function arguments are completely irrelevant for the semantics (the meaning) of the program, it is important to chose these names carefully. A program with well-chosen names is much easier to understand and maintain than a program without meaningful names. A program with misleading names is even worse.

Another possibility to choose function names is combining one or more `funny' symbols from the set

```
~ @ # % ^ ? ! + - * < > \ / | & = :
```

Some examples of names that are allowed are:

```
+ ++ && || <= == <> . %
@@ -*- \/ /\ ... <+> ? :->
```

The names on the first of these two lines are defined in some of the standard modules. The operators on the second line are examples of other names that are allowed.

There is one exception to the choice of names. The following words are reserved for special purposes, and cannot be used as the name of a function:

```
Bool Char default definition case class code export from if implementation import in
infix infixl infixr instance Int let let! module of system where with
```

Also, the following symbol combinations are reserved, and may not be used as function name:

```
// \\ & : :: { } /* */ | ! & # #! . [ ] = =: :== => -> <- <-:
```

However, enough symbol combinations remain to attain some interesting graphic effects…

1.3.2 Predefined functions on numbers

There are two kinds of numbers available in CLEAN: Integer numbers, like `17`, `0` and `-3`; Floating-point numbers, like `2.5`, `-7.81`, `0.0`, `1.2e3` and `0.5e-2`. The character `e` in floating-point numbers means `times ten to the power of'. For example `1.2e3` denotes the number $1.2*10^3 = 1200.0$. The number `0.5e-2` is in fact $0.5*10^{-2} = 0.005$.

In the standard modules `StdInt` and `StdReal` some functions and operators are defined on numbers. The four mathematical operators addition (`+`), subtraction (`-`), multiplication (`*`) and division (`/`) can be used on both integer and real numbers, as in the programs

```
Start = 5-12
```

and

```
Start = 2.5 * 3.0
```

When dividing integer numbers, the fractional part is discarded:

```
Start = 19/4
```

displays the result `4` to the user. If exact division is required, real numbers must be used:

```
Start = 19.0/4.0
```

will show the result `4.75`. The arguments of an arithmetic operator must both be integer or both be real. The expression `1.5 + 2` is not accepted by the compiler. However, there are standard functions `toInt` and `toReal` that convert numbers to an integer or real, respectively.

Other standard functions on integer numbers include

`abs`	the absolute value of a number
`sign`	`-1` for negative numbers, `0` for zero, `1` for positive numbers
`gcd`	the greatest common divisor of two numbers

`^`	raising a number to a power

Some standard functions on real numbers are:

`sqrt`	the square root function
`sin`	the sine function
`ln`	the natural logarithm
`exp`	the exponential function (e-to-the-power-of)

1.3.3 Predefined functions on Booleans

The operator `<` determines whether a number is smaller than another number. The result is the constant `True` (if it is true) or the constant `False` (if it is false). For example, the value of `1<2` is `True`.

The values `True` and `False` are the only elements of the set of *truth values* or *Boolean values* (named after the English mathematician George Boole, who lived from 1815 till 1864). Functions (and operators) resulting such a value are called *Boolean functions* or *predicates*.

Next to `<` there is also an operator `>` (greater than), an operator `<=` (smaller or equal to), and an operator `>=` (greater or equal to). Furthermore, there is the operator `==` (equal to) and an operator `<>` (not equal to).

Results of Boolean functions can be combined with the operators `&&` (`and') and `||` (`or'). The operator `&&` only returns `True` if the results left *and* right are true:

```
Start = 1<2 && 3<4
```

will show the result `True` to the user. The `or' operator needs only one of the two statements to be true (both may be true as well), so `1==1 || 2==3` will yield `True`. There is a function `not` swapping `True` and `False`. Furthermore there is a function `isEven` which checks whether an integer number is even.

1.3.4 Predefined functions on lists

In the standard module `StdList` a number of functions operating on lists is defined. Some functions on lists have already been discussed: `length` determines the length of a list, `sum` calculates the sum of a list of whole numbers.

The operator `++` concatenates two lists. For example,

```
Start = [1,2] ++ [3,4,5]
```

will show the list `[1,2,3,4,5]`.

The function `and` operates on a list of which the elements are Booleans; `and` checks if all the elements in the list are `True`. For example, the expression `and [1<2, 2<3, 1==0]` returns `False`.

Some functions have two parameters. The function `take` operates on a number and a list. If the number is `n`, the function will return the first `n` elements of the list. For example, `take 3 [2..10]` returns the list `[2,3,4]`.

1.3.5 Predefined functions on functions

In the functions discussed so far, the parameters were numbers, Booleans or lists. However, the argument of a function can be a function itself too! An example of that is the function `map`, which takes two arguments: a function and a list. The function `map` applies the argument function to all the elements of the list. It is defined in the standard module `StdList`.

Some examples of the use of the `map` functions are:

```
Start = map fac [1,2,3,4,5]
```

Applying the `fac` function to all five numbers, this shows the list `[1,2,6,24,120]`.

Running the program

```
Start = map sqrt [1.0,2.0,3.0,4.0]
```

shows the list `[1.0, 1.41421, 1.73205, 2.0]`, and the program

```
Start = map isEven [1..8]
```

checks all eight numbers for even-ness, yielding a list of Boolean values: [False, True, False, True, False, True, False, True].

1.4 Defining functions

1.4.1 Definition by combination

The easiest way to define functions is by combining other functions and operators, for example by applying predefined functions that have been imported:

```
fac n    = prod [1..n]
square x = x * x
```

Functions can also have more than one argument:

```
over n k   =   fac n / (fac k * fac (n-k))
roots a b c =  [ (~b+sqrt(b*b-4.0*a*c)) / (2.0*a)
               , (~b-sqrt(b*b-4.0*a*c)) / (2.0*a)
               ]
```

The operator ~ negates its argument. See 1.3.2 and Chapter 2 for the difference between 2 and 2.0.

Functions without arguments are commonly known as `constants':

```
pi  = 3.1415926535
e   = exp 1.0
```

These examples illustrate the general form of a function definition:

- the name of the function being defined
- names for the formal arguments (if there are any)
- the symbol =
- an expression, in which the arguments may be used, together with functions (either functions imported form another module or defined elsewhere in the module) and elementary values (like 42).

In definitions of functions with a Boolean result, at the right hand side of the =-symbol an expression of Boolean value is given:

```
negative x  =   x < 0
positive x  =   x > 0
isZero   x  =   x == 0
```

Note, in this last definition, the difference between the =-symbol and the ==-operator. A single `equals'-symbol (=) separates the left-hand side from the right-hand side in function definitions. A double `equals'-symbol (==) is an operator with a Boolean result, just as < and >.

In the definition of the roots function in the example above, the expressions sqrt(b*b-4.0*a*c) and (2.0*a) occur twice. Apart from being boring to type, evaluation of this kind of expression is needlessly time-consuming: the identical subexpressions are evaluated twice. To prevent this, in CLEAN it is possible to name a subexpression, and denote it only once. A more efficient definition would be:

```
roots a b c =  [ (~b+s)/d
               , (~b-s)/d
               ]
where
    s = sqrt (b*b-4.0*a*c)
    d = 2.0*a
```

The word where is not the name of a function. It is one of the `reserved words' that where mentioned in subsection 1.3.1. Following the word where in the definition, again some definitions are given. In this case the constants s and d are defined. These constant may be used in the expression preceding the where. They cannot be used elsewhere in the program; they are *local definitions*. You may wonder why s and d are called `constants', although their

value can be different on different uses of the roots function. The word `constants' is justified however, as the value of the constants is fixed during each invocation of roots.

1.4.2 Definition by cases

In some occasions it is necessary to distinguish a number of cases in a function definition. The function that calculates the absolute value of a number is an example of this: for negative arguments the result is calculated differently than for positive arguments. In CLEAN, this is written as follows:

```
abs x
    | x<0   = ~x
    | x>=0  = x
```

You may also distinguish more than two cases, as in the definition of the signum function below:

```
signum x
    | x>0   = 1
    | x==0  = 0
    | x<0   = -1
```

The expressions in the three cases are `guarded' by Boolean expressions, which are therefore called *guards*. When a function that is defined using guarded expressions is called, the guards are tried one by one, in the order they are written in the program. For the first guard that evaluates to True, the expression at the right hand side of the =-symbol is evaluated. Because the guards are tried in textual order, you may write True instead of the last guard. For clarity, you can also use the constant otherwise, that is defined to be True in the StdBool module.

```
abs x
    | x<0       = ~x
    | otherwise = x
```

A guard which yields always true, like True or otherwise, is in principle superfluous and may be omitted.

```
abs x
    | x<0   = ~x
            = x
```

The description of allowed forms of function definition (the `syntax' of a function definition) is therefore more complicated than was suggested in the previous subsection. A more adequate description of a function definition is:

- the name of the function;
- the names of zero or more arguments;
- an =-symbol and an expression, or: one ore more `guarded expressions';
- (optional:) the word where followed by local definitions.

A `guarded expression' consists of a |-symbol, a Boolean expression, a =-symbol, and an expression. But still, this description of the syntax of a function definition is not complete…

1.4.3 Definition using patterns

Until now, we used only variable names as formal arguments. In most programming languages, formal arguments may only be variables. But in CLEAN, there are other possibilities: a formal argument may also be a *pattern*.

An example of a function definition in which a pattern is used as a formal argument is

```
h [1,x,y] = x+y
```

This function can only be applied to lists with exactly three elements, of which the first must be 1. Of such a list, the second and third elements are added. Thus, the function is not defined for shorter and longer list, nor for lists of which the first element is not 1. It is a common phenomenon that functions are not defined for all possible arguments. For ex-

ample, the `sqrt` function is not defined for negative numbers, and the `/` operator is not defined for `0` as its second argument. These functions are called *partial functions*.

You can define functions with different patterns as formal argument:

```
sum []      = 0
sum [x]     = x
sum [x,y]   = x+y
sum [x,y,z] = x+y+z
```

This function can be applied to lists with zero, one, two or three elements (in the next subsection this definition is extended to lists of arbitrary length). In each case, the elements of the list are added. On use of this function, it is checked whether the actual argument `matches' one of the patterns. Again, the definitions are tried in textual order. For example, the call `sum [3,4]` matches the pattern in the third line of the definition: The `3` corresponds to the `x` and the `4` to the `y`.

As a pattern, the following constructions are allowed:

- numbers (e.g. `3`);
- the Boolean constants `True` and `False`;
- names (e.g. `x`);
- list enumeration's, of which the elements must be patterns (e.g. `[1,x,y]`);
- lists patterns in which a distinction is made between the first element and the rest of the list (e.g. `[a:b]`).

Using patterns, we could for example define the logical conjunction of two Boolean functions:

```
AND False False =   False
AND False True  =   False
AND True  False =   False
AND True  True  =   True
```

By naming the first element of a list, two useful functions can be defined, as is done in the module `StdList`:

```
hd [x:y] = x
tl [x:y] = y
```

The function `hd` returns the first element of a list (its `head'), while the function `tl` returns all but the first element (the `tail' of the list). These functions can be applied to almost all lists. They are not defined, however, for the empty list (a list without elements): an empty list has no first element, let alone a `tail'. This makes `hd` and `tl` partial functions.

Note the difference in the patterns (and expressions) `[x:y]` and `[x,y]`. The pattern `[x:y]` denotes a list with first element (head) `x` and rest (tail) `y`. This tail can be any list, including the empty list `[]`. The pattern `[x,y]` denotes a list of exactly two elements, the first one is called `x`, and the other one `y`.

1.4.4 Definition by induction or recursion

In definitions of functions, other functions may be used. But also the function being defined may be used in it's own definition! A function which is used in its own definition is called a *recursive function* (because its name `re-(oc)curs' in its definition). Here is an example of a recursive definition:

```
fac n
  | n==0 = 1
  | n>0  = n * fac (n-1)
```

The name of the function being defined (`fac`) occurs in the defining expression on the right hand side of the `=`-symbol.

Another example of a recursively defined function is `raising to an integer power'. It can be defined as:

```
power x n
  | n==0 = 1
  | n>0  = x * power x (n-1)
```


Also, functions operating on lists can be recursive. In the previous subsection we introduced a function to determine the length of some lists. Using recursion we can define a function `sum` for lists of arbitrary length:

```
sum list
  | list == []   = 0
  | otherwise    = hd list + sum (tl list)
```

Using patterns we can also define this function in an even more readable way:

```
sum []              = 0
sum [first: rest]   = first + sum rest
```

Using patterns, you can give the relevant parts of the list a name directly (like `first` and `rest` in this example). In the definition that uses guarded expressions to distinguish the cases, auxiliary functions `hd` and `tl` are necessary. In these auxiliary functions, eventually the case distinction is again made by patterns.

Using patterns, we can define a function `length` that operates on lists:

```
length []           = 0
length [first:rest] = 1 + length rest
```

The value of the first element is not used (only the fact that a first element exists). For cases like this, it is allowed to use the '_' symbol instead of an identifier:

```
length []       = 0
length [_:rest] = 1 + length rest
```

Recursive functions are generally used with two restrictions:

- for a base case there is a non-recursive definition;
- the actual argument of the recursive call is *closer to the base case* (e.g., numerically smaller, or a shorter list) than the formal argument of the function being defined.

In the definition of `fac` given above, the base case is `n==0`; in this case the result can be determined directly (without using the function recursively). In the case that `n>0`, there is a recursive call, namely `fac (n-1)`. The argument in the recursive call (`n-1`) is, as required, smaller than `n`.

For lists, the recursive call must have a shorter list as argument,. There should be a non-recursive definition for some finite list, usually the empty list.

1.4.5 Local definitions, scope and lay-out

If you want to define a function to solve a certain problem you often need to define a number of additional functions each solving a part of the original problem. Functions following the keyword *where* are *locally* defined which means that they only have a meaning within the surrounding function. It is a good habit to define functions that are only used in a particular function definition, locally to the function they belong. In this way you make it clear to the reader that these functions are not used elsewhere in the program. The *scope* of a definition is the piece of program text where the definition can be used. The box in figure 1.1 shows the scope of a local function definition, i.e. the area in which the locally defined function is known and can be applied. The figure also shows the scope of the arguments of a function. If a name of a function or argument is used in an expression one has to look for a corresponding definition in the smallest surrounding scope (box). If the name is not defined there one has to look for a definition in the nearest surrounding scope and so on.

```
function args
         | guard1 = expression1
         | guard2 = expression2
         where
             function args = expression
```

Figure 1.1: Defining functions and graphs locally for a function alternative.

With a *let* statement one can locally define new functions which only have a meaning within a certain expression.

```
roots a b c =  let s = sqrt (b*b-4.0*a*c)
                   d = 2.0*a
               in  [(~b+s)/d , (~b-s)/d ]
```

A let statement is allowed in any expression on the right-hand side of a function or graph definition. The scope of a let expression is illustrated in Figure 1.2.

```
let function args = expression
in expression
```

Figure 1.2: Defining functions and graphs locally for a certain expression.

Layout

On most places in the program extra whitespace is allowed, to make the program more readable for humans. In the examples above, for example, extra spaces have been added in order to align the =-symbols. Of course, no extra whitespace is allowed in the middle of an identifier or a number: `len gth` is different from `length`, and `1 7` is different from `17`.

Also, newlines can be added in most places. We did so in the definition of the `roots` function, because the line would be very long otherwise. However, unlike most other programming languages, newlines are not entirely meaningless. Compare these two `where`-expressions:

```
where              | where
    a = f x y      |     a   = f x
    b = g z        | y b = g z
```

The place where the new line is inserted (between the `y` and the `b`, or between the `x` and the `y`) does make a difference: in the first situation `a` and `b` are defined while in the second situation `a` and `y` are defined (`y` has `b` as formal argument).

The CLEAN compiler uses the criteria below for determining which text groups together:

- a line that is indented *exactly as much* as the previous line, is considered to be a new definition;
- a line that is indented *more* belongs to the expression on the previous line;
- a line that is indented *less* does not belong to the same group of definitions any more.

The third rule is necessary only when `where`-constructions are nested, as in:

```
f x y = g (x+w)
where
    g u = u + v
    where
        v = u * u
    w = 2 + y
```

Here, `w` is a local definition of `f`, not of `g`. This is because the definition of `w` is indented less than the definition of `v`; therefore it doesn't belong to the local definitions of `g`. If it would be indented even less, it would not be a local definition of `f` anymore as well. This would result in an error message, because `y` is not defined outside the function `f` and its local definitions.

All this is rather complicated to explain, but in practice everything works fine if you adhere to the rule:

definitions on the same level should be indented the same amount

This is also be true for global definitions, the global level starts at the very beginning of a line.

Although programs using this layout rule are syntactical appealing, it is allowed to define the scope of definitions explicitly. For example:

```
f x y = g (x+w)
where { g u =u + v
        where { v = u * u
              };
        w = 2 + y
      };
```

This form of layout cannot be mixed with the layout rule within a single module.

1.4.6 Comments

On all places in the program where extra whitespace is allowed (that is, almost everywhere) comments may be added. Comments are neglected by the compiler, but serve to elucidate the text for human readers. There are two ways to mark text as comment:

- with symbols `//` a comment is marked that extends to the end of the line
- with symbols `/*` a comment is marked that extends to the matching symbols `*/`.

Comments that are built in the second way may be nested, that is contain a comment themselves. The comment is finished only when every `/*` is closed with a matching `*/`. For example in

```
/* /* hello */ f x = 3 */
```

There is no function `f` defined: everything is comment.

1.5 Types

All language elements in CLEAN have a type. These types are used to group data of the same kind. We have seen some integers, like `0`, `1` and `42`. Another kind of values are the Boolean values `True` and `False`. The type system of CLEAN prevents that these different kinds of data are mixed. The type system of CLEAN assigns a type to each and every element in the language. This implies that basic values have a type, compound datatypes have a type and functions have a type. The types given to the formal arguments of a function specify the *domain* the function is defined on. The type given to the function result specifies the *range* (*co-domain*) of the function.

The language CLEAN, and many (but not all) other functional languages, have a *static type system*. This means that the compiler checks that type conflicts cannot occur during program execution. This is done by assigning types to all function definitions in the program.

1.5.1 Sorts of errors

To err is human, especially when writing programs. Fortunately, the compiler can warn for some errors. If a function definition does not conform to the syntax, this is reported by the compiler. For example, when you try to compile the following definition:

```
isZero x = x=0
```

the compiler will complain: the second `=` should have been a `==`. Since the compiler does not know your intention, it can only indicate that there is something wrong. In this case the error message is (the part `[…]` indicates the file and line where the error is found):

```
Syntax error […]: <global definition> expected instead of '='
```

Other examples of syntax errors that are detected by the compiler are expressions in which not every opening parenthesis has a matching closing one, or the use of reserved words (such as `where`) in places where this is not allowed.

A second sort of errors for which the compiler can warn is the use of functions that are neither defined nor included from another module. For example, if you define, say on line 20 of a CLEAN module called `test.icl`

```
Start = Sqrt 5
```

the compiler notices that the function `Sqrt` was never defined (if the function in the module `StdReal` was intended, it should have been spelled `sqrt`). The compiler reports:

```
Error [test.icl,20,Start]: Sqrt not defined
```

The next check the compiler does is *type checking*. Here it is checked whether functions are only used on values that they were intended to operate on. For example, functions which operate on numbers may not be applied to Boolean values, neither to lists. Functions which operate on lists, like `length`, may in turn not be applied to numbers, and so on.

If in an expression the term `1+True` occurs, the compiler will complain:

```
Type error […]: "argument 2 of +" cannot unify demanded type Int with Bool
```

The `[…]` replaces the indication of the file, the line and the function of the location where the error was detected. Another example of an error message occurs when the function `length` is applied to anything but a list, as in `length 3`:

```
Type error […]: "argument 1 of length" cannot unify demanded type [x] with Int
```

The compiler uses a technique called *unification* to verify that, in any application, the actual types match the corresponding types of the formal arguments. This explains the term 'unify' in the type error messages if such a matching fails. Only when a program is free of type errors, the compiler can generate code for the program. When there are type errors, there is no program to be executed.

In strongly typed languages like CLEAN, all errors in the type of expressions are detected by the compiler. Thus, a program that survives checking by the compiler is guaranteed to be type-error free. In other languages only a part of the type correctness can be checked at compile time. In these languages a part of the type checks are done during the execution of the generated application when function is actually applied. Hence, parts of the program that are not used in the current execution of the program are not checked for type consistency. In those languages you can never be sure that at run time no type errors will pop up. Extensive testing is needed to achieve some confidence in the type correctness of the program. There are even language implementations where all type checks are delayed until program execution.

Surviving the type check of the compiler does not imply that the program is correct. If you used multiplication instead of addition in the definition of `sum`, the compiler will not complain about it: it has no knowledge of the intentions of the programmer. These kind of errors, called `logical errors', are among the hardest to find, because the compiler does not warn you for them.

1.5.2 Typing of expressions

Every expression has a type. The type of a constant or function that is defined can be specified in the program. For example:

```
Start :: Int
Start = 3+4
```

The symbol `::` can be pronounced as `is of type'.

There are four basic types:

- `Int`: the type of the integer numbers (also negative ones);
- `Real`: the type of floating-point numbers (an approximation of the Real numbers);

- `Bool`: the type of the Boolean values `True` and `False`;
- `Char`: the type of letters, digits and symbols as they appear on the keyboard of the computer.

In many programming languages string, sequence of `Char`, is a predefined or basic type. Some functional programming languages use a list of Char as representation for string. For efficiency reasons Clean uses an unboxed array of `Char`, `{#Char}`, as representation of strings. See below.

Lists can have various types. There exist lists of integers, lists of Boolean values, and even lists of lists of integers. All these types are different:

```
x :: [Int]
x =  [1,2,3]

y :: [Bool]
y =  [True,False]

z :: [[Int]]
z =  [[1,2],[3,4,5]]
```

The type of a list is denoted by the type of its elements, enclosed in square brackets: `[Int]` is the type of lists of integers. All elements of a list must have the same type. If not, the compiler will complain.

Not only constants, but also functions have a type. The type of a function is determined by the types of its arguments and its result. For example, the type of the function `sum` is:

```
sum :: [Int] -> Int
```

That is, the function sum operates on lists of integers and yields an integer. The symbol `->` in the type might remind you of the arrow symbol (→) that is used in mathematics. More examples of types of functions are:

```
sqrt    :: Real -> Real
isEven  :: Int -> Bool
```

A way to pronounce lines like this is `isEven is of type Int to Bool' or 'isEven is a function from Int to Bool'.

Functions can, just as numbers, Booleans and lists, be used as elements of a list as well. Functions occurring in one list should be of the same type, because elements of a list must be of the same type. An example is:

```
trigs :: [Real->Real]
trigs =  [sin,cos,tan]
```

The compiler is able to determine the type of a function automatically. It does so when type checking a program. So, if one defines a function, it is allowed to leave out its type definition. But, although a *type declaration* is strictly speaking superfluous, it has two advantages to specify a type explicitly in the program:

- the compiler checks whether the function indeed has the type intended by the programmer;
- the program is easier to understand for a human reader.

It is considered a very good habit to supply types for all important functions that you define. The declaration of the type has to precede to the function definition.

1.5.3 Polymorphism

For some functions on lists the concrete type of the elements of the list is immaterial. The function `length`, for example, can count the elements of a list of integers, but also of a list of Booleans, and –why not– a list of functions or a list of lists. The type of length is denoted as:

```
length :: [a] -> Int
```

This type indicates that the function has a list as argument, but that the concrete type of the elements of the list is not fixed. To indicate this, a *type variable* is written, `a` in the example. Unlike concrete types, like `Int` and `Bool`, type variables are written in lower case.

The function `hd`, yielding the first element of a list, has as type:

```
hd :: [a] -> a
```

This function, too, operates on lists of any type. The result of `hd`, however, is of the same type as the elements of the list (because it is the first element of the list). Therefore, to hold the place of the result, the same type variable is used.

A type which contains type variables is called a *polymorphic type* (literally: a type of many shapes). Functions with a polymorphic type are called polymorphic functions, and a language allowing polymorphic functions (such as CLEAN) is called a polymorphic language. Polymorphic functions, like `length` and `hd`, have in common that they only need to know the *structure* of the arguments. A non-polymorphic function, such as `sum`, also uses properties of the elements, like `addibility'. Polymorphic functions can be used in many different situations. Therefore, a lot of the functions in the standard modules are polymorphic.

Not only functions on lists can be polymorphic. The simplest polymorphic function is the identity function (the function that yields its argument unchanged):

```
id :: a -> a
id x = x
```

The function `id` can operate on values of any type (yielding a result of the same type). So it can be applied to a number, as in `id 3`, but also to a Boolean value, as in `id True`. It can also be applied to lists of Booleans, as in `id [True,False]` or lists of lists of integers: `id [[1,2,3],[4,5]]`. The function can even be applied to functions: `id sqrt` of `id sum`. The argument may be of any type, even the type `a->a`. Therefore the function may also be applied to itself: `id id`.

1.5.4 Functions with more than one argument

Functions with more arguments have a type, too. All the types of the arguments are listed before the arrow. The function `over` from subsection 1.4.1 has type:

```
over :: Int Int -> Int
```

The function `roots` from the same subsection has three floating-point numbers as arguments and a list of floats as result:

```
roots :: Real Real Real -> [Real]
```

Operators, too, have a type. After all, operators are just functions written between the arguments instead of in front of them. Apart from the actual type of the operator, the type declaration contains some additional information to tell what kind of infix operator this is (see section 2.1). You could declare for example:

```
(&&) infixr 1 :: Bool Bool -> Bool
```

An operator can always be transformed to an ordinary function by enclosing it in brackets. In the type declaration of an operator and in the left-hand side of its own definition this is obligatory.

1.5.5 Overloading

The operator `+` can be used on two integer numbers (`Int`) giving an integer number as result, but it can also be used on two real numbers (`Real`) yielding a real number as result. So, the type of `+` can be both `Int Int->Int` and `Real Real->Real`. One could assume that + is a polymorphic function, say of type `a a->a`. If that would be the case, the operator could be applied on arguments of any type, for instance `Bool` arguments too, which is not the case. So, the operator `+` seems to be sort of polymorphic in a restricted way.

However, + is not polymorphic at all. Actually, there exists not just one operator +, but there are several of them. There are different operators defined which are all carrying the same name: `+`. One of them is defined on integer numbers, one on real numbers, and there may be many more. A function or operator for which several definitions may exist, is called *overloaded*.

In CLEAN it is generally not allowed to use the same name for different functions. If one wants to use the same name for different functions, one has to explicitly define this via a *class* declaration. For instance, the overloaded use of the operator `+` can be declared as (see `StdOverloaded`):

```
class (+) infixl 6 a :: a a -> a
```

With this declaration + is defined as the name of an overloaded operator (which can be used in infix notation and has priority `6`, see chapter 2.1). Each of the concrete functions (called *instances*) with the name + must have a type of the form `a a -> a`, where `a` is the *class variable* which has to be substituted by the concrete type the operator is defined on. So, an instance for `+` can e.g. have type `Int Int -> Int` (substitute for the class variable `a` the type `Int`) or `Real Real -> Real` (substitute for `a` the type `Real`). The concrete definition of an instance is defined separately (see `StdInt`, `StdReal`). For instance, one can define an instance for + working on Booleans as follows:

```
instance + Bool
where
    (+) :: Bool Bool -> Bool
    (+) True b = True
    (+) a    b = b
```

Now one can use `+` to add Booleans as well, even though this seems not to be a very useful definition. Notice that the class definition ensures that all instances have the same kind of type, it does not ensure that all the operators also behave uniformly or behave in a sensible way.

When one uses an overloaded function, it is often clear from the context, which of the available instances is intended. For instance, if one defines:

```
increment n = n + 1
```

it is clear that the instance of `+` working on integer numbers is meant. Therefore, `increment` has type:

```
increment :: Int -> Int
```

However, it is not always clear from the context which instance has to be taken. If one defines:

```
double n = n + n
```

it is not clear which instance to choose. Any of them can be applied. As a consequence, the function `double` becomes overloaded as well: it can be used on many types. More precisely, it can be applied on an argument of any type under the condition that there is an instance for `+` for this type defined. This is reflected in the type of `double`:

```
double :: a a -> a | + a
```

As said before, the compiler is capable of deducing the type of a function, even if it is an overloaded one. More information on overloading can be found in Chapter 4.

1.5.6 Type annotations and attributes

The type declarations in CLEAN are also used to supply additional information about (the arguments of) the function. There are two kinds of annotations

- *Strictness* annotations indicate which arguments will always be needed during the computation of the function result. Strictness of function arguments is indicated by the `!`-symbol in the type declaration.

- *Uniqueness* attributes indicate whether arguments will be shared by other functions, or that the function at hand is the only one using them. Uniqueness is indicated by a `.`-symbol, or a variable and a `:`-symbol in front of the type of the argument.

Some examples of types with annotations and attributes from the standard environment:

```
isEven              :: !Int -> Bool          // True if argument is even
spaces              :: !Int -> .[Char]       // Make list of n spaces
(++) infixr 0       :: ![.a] u:[.a] -> u:[.a]  // Concatenate two lists
class (+) infixl 6 a  :: !a !a -> a          // Add arg1 to arg2
```

Strictness information is important for efficiency; uniqueness is important when dealing with I/O (see Chapter 5). For the time being you can simply ignore both strictness annotations and uniqueness attributes. The compiler has an option that switches off the strictness analysis, and an option that inhibits displaying uniqueness information in types.

More information on uniqueness attributes can be found in Chapter 4, the effect of strictness is explained in more detail in Part III, Chapter 2.

1.5.7 Well-formed Types

When you specify a type for a function the compiler checks whether this type is correct or not. Although type errors might look boring while you are trying to compile your program, they are a great benefit. By checking the types in your program the compiler guarantees that errors caused by applying functions to illegal arguments cannot occur. In this way the compiler spots a lot of the errors you made while your were writing the program before you can execute the program. The compiler uses the following rules to judge type correctness of your program:

1) all alternatives of a function should have the same type;
2) all occurrences of an argument in the body of a function should have the same type;
3) each function used in an expression should have arguments that fits the corresponding formal arguments in the function definition;
4) a type definition supplied should comply with the rules given here.

An actual argument fits the formal argument of function when its type is equal to, or more specific than the corresponding type in the definition. We usually say: the type of the actual argument should be an *instance* of the type of the formal argument. It should be possible to make the type of the actual argument and the type of the corresponding formal argument equal by replacing variables in the type of the formal argument by other types.

Similarly, it is allowed that the type of one function alternative is more general that the type of an other alternative. The type of each alternative should be an instance of the type of the entire function. The same holds within an alternative containing a number of guarded bodies. The type of each function body ought to be an instance of the result type of the function.

We illustrate these rules with some examples. In these examples we will show how the CLEAN compiler is able to derive a type for your functions. When you are writing functions, you know your intentions and hence a type for the function you are constructing. Consider the following function definition:

```
f 1 y = 2
f x y = y
```

From the first alternative it is clear the type of `f` should be `Int t -> Int`. The first argument is compared in the pattern match with the integer `1` and hence it should be an integer. We do not know anything about the second argument. Any type of argument will do. So, we use a type variable for the type. The body is an `Int`, hence the type of the result of this function is `Int`. The type of the second alternative is `u v -> v`. We do not know any thing about the type of the arguments. When we look to the body of the function alternative we can only decide that its type is equal to the type of the second argument. For the type of

the entire function types `Int t -> Int` and `u v -> v` should be equal. From the type of the result we conclude that `v` should be `Int`. We replace the type variable `v` by `Int`. The type of the function alternatives is now `Int t -> Int` and `u Int -> Int`. The only way to make these types equal is to replace `t` and `u` by `Int` as well. Hence the type derived by the compiler for this function is `Int Int -> Int`.

Type correctness rule 4) implies that it is allowed to specify a more restricted than the most general type that would have been derived by the compiler. As example we consider the function `Int_Id`:

```
Int_Id :: Int -> Int
Int_Id i = i
```

Here a type is given. The compiler just checks that this type does not cause any conflicts. When we assume that the argument is of type `Int` also the result is of type `Int`. Since this is consistent with the definition this type is correct. Note that the same function can have also the more general type `v -> v`. Like usual the more specific type is obtained by replacing type variables by other types. Here the type variable `v` is replaced by `Int`.

Our next example illustrates the type rules for guarded function bodies. We consider the somewhat artificial function `g`:

```
g 0 y z = y
g x y z
  | x == y    = y
  | otherwise = z
```

In the first function alternative we can conclude that the first argument should be an `Int` (due to the given pattern), the type of the result of the function is equal to its second argument: `Int u v -> u`.

In the second alternative, the argument `y` is compared to `x` in the guard. The `==`-operator has type `x x -> Bool`, hence the type of the first and second argument should be equal. Since both `y` and `z` occur as result of the guarded bodies of this alternative, their types should be equal. So, the type of the second alternative is `t t t -> t`.

When we unify the type of the alternatives, the type for these alternatives must be made equal. We conclude that the type of the function `g` is `Int Int Int -> Int`.

Remember what we have told in section 1.5.5 about overloading. It is not always necessary to determine types exactly. It can be sufficient to enforce that some type variables are part of the appropriate classes. This is illustrated in the function `h`.

```
h x y z
  | x == y    = y
  | otherwise = x+z
```

Similar to the function `g`, the type of argument `x` and `y` should be equal since these arguments are tested for equality. However, none of these types are known. It is sufficient that the type of these arguments is member of the type class `==`. Likewise, the last function body forces the type of the arguments `x` and `z` to be equal and part of the type class `+`. Hence, the type of the entire function is `a a a -> a | + , == a`. This reads: the function `h` takes three values of type `a` as arguments and yields a value of type `a`, provided that `+` and `==` is defined for type `a` (`a` should be member of the type classses `+` and `==`). Since the type `Int Int Int -> Int` is an instance of this type, it is allowed to specify that type for the function `h`.

You might be confused by the power of CLEAN's type system. We encourage you to start specifying the type of the functions you write as soon as possible. These types helps you to understand the function to write and the functions you have written. Moreover, the compiler usually gives more appropriate error messages when the intended type of the functions is known.

1.6 Synonym definitions

1.6.1 Global constant functions (CAF's)

We have seen in the definition of the `roots` function given in subsection 1.4.1 that one can define local constants (e.g. `s = sqrt(b*b-4.0*a*c)`). By using such a local constant efficiency is gained because the corresponding expression will be evaluated only once, even if it is used on several places.

It is also possible to define such constants on the *global* level, e.g. a very large list of integers is defined by:

```
biglist :: [Int]
biglist =: [1..100000]
```

Notice that one has to use the `=:` symbol to separate left-hand side from the right-hand side of the global constant definition (the `=:`-symbol can also be used as alternative for `=` in local constant definitions). Constant functions on the global level are also known as *constant applicative forms* (CAF's). Global constants are evaluated in the same way as local constants: they are evaluated only once. The difference with local constants is that a global constant can be used anywhere in the program. The (evaluated) constant will be remembered during the whole life time of the application. The advantage is that if the same constant is used on several places, it does not has to be calculated over and over again. The disadvantage can be that an evaluated constant might consume much more space than an unevaluated one. For instance the unevaluated expression `[1..100000]` consumes much less space than an evaluated list with `100000` elements in it. If you rather would like to evaluate the global constant each time it is used to save space, you can define it as:

```
biglist :: [Int]
biglist = [1..100000]
```

The use of `=:` instead of `=` makes all the difference.

1.6.2 Macro's and type synonyms

It is sometimes very convenient to introduce a new name for a given expression or for an existing type. Consider the following definitions:

```
:: Color :== Int

Black    :== 1
White    :== 0

invert :: Color -> Color
invert Black = White
invert White = Black
```

In this example a new name is given to the type `Int`, namely `Color`. By defining

```
:: Color :== Int
```

`Color` has become a *type synonym* for the type `Int`. `Color -> Color` and `Int -> Int` are now both a correct type for the function `invert`.

One can also define a synonym name for an expression. The definitions

```
Black   :== 1
White   :== 0
```

are examples of a *macro definition*. So, with a type synonym one can define a new name for an existing type, with a macro one can define a new name for an expression. This can be used to increase the readability of a program.

Macro names can begin with a lowercase character, an uppercase character or a funny character. In order to use a macro in a pattern, it should syntactical be equal to a constructor; it should begin with an uppercase character or a funny character. All identifiers beginning with a lowercase character are treated as variables.

Macro's and type synonyms have in common that whenever a macro name or type synonym name is used, the compiler will replace the name by the corresponding definition before the program is type checked or run. Type synonyms lead to much more readable code. The compiler will try to use the type synonym name for its error messages. Using macro's instead of functions or (global) constants leads to more efficient programs, because the evaluation of the macro will be done at compile time while functions and (global) constants are evaluated at run-time.

Just like functions macro's can have arguments. Since macro's are 'evaluated' at compile time the value of the arguments is usually not known, nor can be computed in all circumstances. Hence it is not allowed to use patterns in macro's. When the optimum execution speed is not important you can always use an ordinary function instead of a macro with arguments. We will return to macro's in chapter 6.

1.7 Modules

CLEAN is a modular language. This means that a CLEAN program is composed out of modules. Each module has a unique name. A module (say you named it `MyModule`) is in principle split into two parts: a CLEAN *implementation module* (stored in a file with extension `.icl`, e.g. `MyModule.icl`) and a CLEAN *definition module* (stored in a file with extension `.dcl`, e.g. `MyModule.dcl`).

Function definitions can only be given in implementation modules. A function defined in a specific implementation module by default only has a meaning inside that module. It cannot be used in another module, unless the function is exported. To export a function (say with the name `MyFunction`) one has to declare its type in the corresponding definition module. Other implementation modules now can use the function, but to do so they have to import the specific function. One can explicitly import a specific function from a specific definition module (e.g. by declaring: `from MyModule import MyFunction`). It is also possible to import all functions exported by a certain definition module with one import declaration (e.g. by declaring: `import MyModule`).

For instance, assume that one has defined the following implementation module (to be stored in file `Example.icl`):

```
implementation module Example

increment :: Int -> Int
increment n = n + 1
```

In this example the operator `+` needs to be imported from module `StdInt`. This can be done in the following way:

```
implementation module Example

from StdInt import +

increment :: Int -> Int
increment n = n + 1
```

And indeed, the operator `+` is exported from `StdInt` because its type definition appears in the definition module of `StdInt`. It is a lot of work to import all functions explicitly. One can import *all* standard operators and functions with one declaration in the following way:

```
implementation module Example

import StdEnv

increment :: Int -> Int
increment n = n + 1
```

The definition module of `StdEnv` looks like:

```
definition module StdEnv
```

```
import
    StdOverloaded, StdClass,
    StdBool, StdInt, StdReal, StdChar,
    StdList, StdCharList, StdTuple, StdArray, StdString, StdFunc, StdMisc,
    StdFile, StdEnum
```

When one imports a module as a whole (e.g. via `import StdEnv`) not only the definitions exported in that particular definition module will be imported, but also all definitions which are on their turn imported in that definition module, and so on. In this way one can import many functions with just one statement. This can be handy, e.g. one can use it to create your own 'standard environment'. However, the approach can also be dangerous because a lot of functions are automatically imported this way, perhaps also functions are imported one did not expect at first glance. Since functions must have different names, name conflicts might arise unexpectedly (the compiler will spot this, but it can be annoying).

When you have defined a new implementation module, you can export a function by repeating its type (not its implementation) in the corresponding definition module. For instance:

```
definition module Example

increment :: Int -> Int
```

In this way a whole hierarchy of modules can be created (a cyclic dependency between definition modules is not allowed). Of course, the top-most implementation module does not need to export anything. That's why it does not need to have a corresponding definition module. When an implementation module begins with

```
module …
```

instead of

```
implementation module …
```

it is assumed to be a top-most implementation module. No definition module is expected in that case. Any top-most module must contain a `Start` rule such that it is clear which expression has to be evaluated given the (imported) function definitions.

The advantage of the module system is that implementation modules can be compiled separately. If one changes an implementation module, none of the other modules have to be recompiled. So, one can change implementations without effecting other modules. This reduces compilation time significantly. If, however, a definition module is changed, all implementation modules importing from that definition module have to be recompiled as well to ensure that everything remains consistent. Fortunately, the CLEAN compiler decides which modules should be compiled when you compile the main module and does this reasonably fast…

1.8 Overview

In this chapter we introduced the basic concepts of functional programming. Each functional program in CLEAN evaluates the expression `Start`. By providing appropriate function definitions, any expression can be evaluated.

Each function definition consists of one or more alternatives. These alternatives are distinguished by their patterns and optionally by guards. The first alternative that matches the expression is used to rewrite it. Guards are used to express conditions that cannot be checked by a constant pattern (like `n>0`). When you have the choice between using a pattern and a guard, use a pattern because it is more clearer.

It is also possible to define a choice using the conditional function `if`, or by a case expression. These possibilities are used for small definitions which do not deserve an own function definition, or when writing function patterns becomes boring.

The static type system guarantees that dynamic type problems cannot occur: a program that is approved by the compiler cannot fail during execution due to type problems. The type system allows many powerful constructs like:

- *higher-order functions*: the possibility to use functions as argument and result of other functions;
- *polymorphism*: the possibility to use one function for many different types;
- *overloading*: several different functions with the same name can be defined, the compiler has to determine which of these functions fits the current type of arguments. A collection of functions with the same name is called a class.

In the chapters to come we will discuss these topics in more detail and we will show the benefits of these language constructs.

1.9 Exercises

1. Make sure the CLEAN system is installed on your computer. The system can be downloaded from www.cs.kun.nl/~clean. Write and execute a program that prints the value 42.
2. Write a function that takes two arguments, say n and x, and computes their power, x^n. Use this to construct a function that squares its argument. Write a program that computes the square of 128.
3. Define the function `isum :: Int -> Int` which adds the digits of its argument. So,
   ```
   isum 1234 = 10
   isum 0    = 0
   isum 1001 = 2
   ```
   You may assume that `isum` is applied to an argument which is not negative.
4. Use the function `isum` to check whether a number can be divided by 9.
5. Define a function `max` with two arguments that delivers the maximum of the two.
6. Define a function `min` that has two arguments that delivers the minimum of the two.
7. Define a function `MaxOfList` that calculates the largest element of a list.
8. Define a function `MinOfList` that calculates the smallest element of a list.
9. Define a function `Last` that returns the last element of a list.
10. Define a function `LastTwo` that returns the last two elements of a list.
11. Define a function `Reverse` that reverses the elements in a list.
12. Define a function `Palindrome` which checks whether a list of characters is a palindrome, i.e. when you reverse the characters you should get the same list as the original.

Part I
Chapter 2
Functions and Numbers

2.1 Operators
2.2 Partial parameterization
2.3 Functions as argument
2.4 Numerical functions
2.5 Exercises

2.1 Operators

An operator is a function of two arguments that is written between those arguments ('infix' notation) instead of in front of them ('prefix' notation). We are more used to write `1 + 2` instead of writing `+ 1 2`. The fact that a function is an operator is indicated at its type definition, between its name and concrete type.

For example, in the standard module `StdBool`, the definition of the conjunction operator starts with:

```
(&&) infixr 3 :: Bool Bool -> Bool
```

This defines `&&` to be an operator that is written in between the arguments ('`infix`'), associates to the right (hence the 'r' in `infixr`, see also section 2.1.3), and has priority `3` (see section 2.1.2).

2.1.1 Operators as functions and vice versa

Sometimes it can be more convenient to write an operator before its arguments, as if it were an ordinary function. You can do so by writing the operator name in parentheses. It is thus allowed to write `(+) 1 2` instead of `1 + 2`. This notation with parentheses is obligatory in the operators type definition and in the left-hand-side of the operators function definition. That is why `&&` is written in parentheses in the definition above.

Using the function notation for operators is extremely useful in partial parametrization and when you want to use an operator as function argument. This is discussed in section 2 and 3 of this chapter.

2.1.2 Priorities

In primary school we learn that 'multiplication precedes addition'. Put differently: the priority of multiplication is higher than that of addition. CLEAN also knows about these priorities: the value of the expression `2*3+4*5` is `26`, not `50`, `46`, or `70`.

There are more levels of priority in CLEAN. The comparison operators, like `<` and `==`, have lower priority than the arithmetical operators. Thus the expression `3+4<8` has the meaning that you would expect: `3+4` is compared with `8` (with result `False`), and not: `3` is added to the result of `4<8` (which would be a type error).

Altogether there are twelve levels of priority (between `0` and `11`). The two top most priorities are reserved for selection of elements out of an array or record (see subsection 3.4 and 3.5) and for function application and cannot be taken by any ordinary operator. The priorities of the operators in the standard modules are:

level `11`	*reserved*: array selection, record selection
level `10`	*reserved*: function application
level `9`	`o ! %`
level `8`	`^`
level `7`	`* / mod rem`
level `6`	`+ - bitor bitand bitxor`
level `5`	`++ +++`
level `4`	`== <> < <= > >=`
level `3`	`&&`
level `2`	`||`
level `1`	`:=`
level `0`	`` `bind` ``

As of yet, not all these operators have been discussed; some of them will be discussed in this chapter or in later ones.

Since the priority of `*` (7) is higher than the priority of `+` (6), the expression `2*3+4*5` is interpreted as `(2*3)+(4*5)`. To override these priorities you can place parentheses in an expression around subexpressions that must be calculated first: in `2*(3+4)*5` the subexpression `3+4` is calculated first, despite the fact that `*` has higher priority than `+`.

Applying functions to their actual argument or parameters (the 'invisible' operator between `f` and `x` in `f x`) has almost topmost priority. The expression `square 3 + 4` therefore calculates `3` squared, and then adds `4`. Even if you write `square 3+4` first the function `square` is applied, and only then the addition is performed. To calculate the square of `3+4` parentheses are required to override the high priority of function calls: `square (3+4)`. Only selection from an array or record has higher priority than function application. This makes it possible to select a function from a record and apply it to some arguments without using parentheses (see chapter 3).

2.1.3 Association

The priority rule still leaves undecided what happens when operators with equal priority occur in an expression. For addition, this is not a problem, but for e.g. subtraction this is an issue: is the result of `8-5-1` the value `2` (first calculate `8` minus `5`, and subtract `1` from the result), or `4` (first calculate `5` minus `1`, and subtract that from `8`)?

For each operator in CLEAN it is defined in which order an expression containing multiple occurrences of it should be evaluated. In principle, there are four possibilities for an operator, say ⊕:

- the operator ⊕ *associates to the left, i.e.* a ⊕ b ⊕ c is interpreted as (a ⊕ b) ⊕ c;
- the operator ⊕ *associates to the right, i.e.* a ⊕ b ⊕ c is evaluated as a ⊕ (b ⊕ c);
- the operator ⊕ is *associative, i.e.* it doesn't matter in which order a ⊕ b ⊕ c is evaluated (this cannot be indicated in the language: a choice between left or right has to be made);
- the operator ⊕ is *non-associative, i.e.* it is not allowed to write a ⊕ b ⊕ c; you always need parentheses to indicated the intended interpretation.

For the operators in the standard modules the choice is made according to mathematical tradition. When in doubt, the operators are made non-associative. For associative operators a more or less arbitrary choice is made for left- or right-associativity (one could e.g. select the more efficient of the two).

Operators that associate to the *left* are:

- the 'invisible' operator function application, so `f x y` means `(f x) y` (the reason for this is discussed in section 2.2).
- the special operator '`.`' which is used for selection of an element from an array or from a record, e.g. `a.[i].[j]` means `(a.[i]).[j]` (select element `i` from array `a` which gives an array from which element `j` is selected, see also subsection 3.6).
- the operator `-`, so the value of `8-5-1` is `2` (as usual in mathematics), not `4`.

Operators that associate to the *right* are:

- the operator `^` (raising to the power), so the value of `2^2^3` is 2^8=`256` (as usual in mathematics), not 4^3=`64`;

Non associative operators are:

- the operator `/` and the related operators `div`, `rem` en `mod`. The result of `64/8/2` is therefore neither `4` nor `16`, but undefined. The compiler generates the error message:
  ```
  Error […]: / conflicting or unknown associativity for infix operators
  ```
- the comparison operators `==`, `<` etcetera: most of the time it is meaningless anyway to write `a==b==c`. To test if `x` is between `2` and `8`, don't write `2<x<8` but `2<x && x<8`.

Associative operators are:

- the operators `*` and `+` (these operators are evaluated left-associative according to mathematical tradition);
- the operators `++`, `+++`, `&&` and `||` (these operators are evaluated right-associative for reasons of efficiency);
- the function composition operator `o` (see subsection 2.3.4)

2.1.4 Definition of operators

If you define an operator yourself, you have to indicate its priority and order of association. We have seen some operator definitions in 1.5.5. As an example, we look at the way in which the power-operator can be declared, specifying that it has priority `8` and associates to the right:

```
(^) infixr 8 :: Int Int -> Int
```

For operators that should associate to the left you use the reserved word `infixl`, for non-associative operators `infix`:

```
(+) infixl 6 :: Int Int -> Int
(==) infix 4 :: Int Int -> Int
```

By defining the priorities cleverly, it is possible to avoid parentheses as much as possible. Consider for instance the operator '`n` over `k`' from subsection 1.4.1:

```
over n k = fac n / (fac k * fac (n-k))
```

We can define it as an operator by:

```
(!^!) n k = fac n / (fac k * fac (n-k))
```

Because at some time it might be useful to calculate $\binom{a+b}{c}$, it would be handy if `!^!` had a lower priority than `+`; then you could leave out parentheses in `a+b!^!c`. On the other hand, expressions like $\binom{a}{b} < \binom{c}{d}$ may be useful. By giving `!^!` a higher priority than `<`, again no parentheses are necessary.

For the priority of `!^!`, it is therefore best to choose e.g. `5` (lower than `+` (`6`), but higher than `<` (`4`)). About the associativity: as it is not very customary to calculate `a!^!b!^!c`, it is best to make the operator non-associative. Therefore the type declaration for our new operator will be:

```
(!^!) infix 5 :: Int Int -> Int
```

If you insist, you can define an infix operator without specifying its type:

```
(!^!) infix 5
```


Using this operator you can write a program that decides whether there are more ways to select 2 out of 4 people, or one person more from a group that consists of two persons more:

```
Start = 4 !^! 2 > 4+2 !^! 2+1
```

You can use any legal function name as name for an infix operator. So, the operator defined above can also be called `over`. In order to prevent confusion with ordinary functions, it is common to use names with funny symbols for infix operators.

2.2 Partial parameterization

2.2.1 Currying of functions

Suppose `plus` is a function adding two integer numbers. In an expression this function can be called with two arguments, for instance `plus 3 5`.

In CLEAN it is also allowed to call a function with *less* arguments than is specified in its type. If `plus` is provided with only `one` argument, for example `plus 1`, a function remains which still expects an argument. This function can be used to define other functions:

```
successor :: (Int -> Int)
successor =  plus 1
```

Calling a function with fewer arguments than it expects is known as *partial parameterization*.

If one wants to apply operators with fewer arguments, one should use the prefix notation with parentheses (see section 2.1). For example, the successor function could have been defined using the operator `+` instead of the function `plus`, by defining

```
successor = (+) 1
```

A more important use of a partially parameterized function is that the result can serve as a parameter for another function. The function argument of the function `map` (applying a function to all elements of a list) for instance, often is a partially parameterized function:

```
map (plus 5) [1,2,3]
```

The expression `plus 5` can be regarded as 'the function adding `5` to something'. In the example, this function is applied by `map` to all elements of the list `[1,2,3]`, yielding the list `[6,7,8]`.

The fact that `plus` can accept one argument rather than two, suggests that its type should be:

```
plus :: Int -> (Int->Int)
```

That is, it accepts something like `5` (an `Int`) and returns something like the successor function (of type `Int->Int`).

For this reason, the CLEAN type system treats the types `Int Int -> Int` and `Int -> (Int -> Int)` as equivalent. To mimic a function with multiple arguments by using intermediate (anonymous) functions having all one argument is known as *Currying*, after the English mathematician Haskell Curry. The function itself is called a *curried* function. (This tribute is not exactly right, because this method was used earlier by M. Schönfinkel).

As a matter of fact these types are not treated completely equivalent in a type declaration of functions. The CLEAN system uses the types also to indicate the arity (number of arguments) of functions. This is especially relevant in definition modules. The following increment functions do not differ in behavior, but do have different types. The only difference between these functions is their arity. In expressions they are treated as equivalent.

```
inc1 :: (Int -> Int)                    // function with arity zero
inc1 = plus 1

inc2 :: Int -> Int                      // function with arity one
inc2 n = plus 1 n
```

Apart from a powerful feature Currying is also a source of strange type error messages. Since it is in general perfectly legal to use a function with fewer or more arguments as its definition the CLEAN compiler cannot complain about forgotten or superfluous arguments. However, the CLEAN compiler does notice that there is an error by checking the type of the expression. Some typical errors are:

```
f x = 2 * successor
```

Which causes the error message

```
Type error [...f]: "argument 2 of *" cannot unify demanded type Int with Int -> Int
```

and

```
f x = 2 * successor 1 x
```

The CLEAN type system gives the error message

```
Type error [...f]: "argument 1 of successor" cannot unify demanded type x -> y with Int
```

2.3 Functions as argument

In a functional programming language, functions behave in many aspects just like other values, like numbers and lists. For example:

- functions have a *type*;
- functions can be the *result* of other functions (which is exploited in Currying);
- functions can be used as an *argument* of other functions.

With this last possibility it is possible to write general functions, of which the specific behavior is determined by a function given as an argument.

Functions which take a function as an argument or which return a function as result are sometimes called *higher-order functions*, to distinguish them from first-order functions like e.g. the common numerical functions which work on values.

The function `twice` is a higher-order function taking another function, `f`, and an argument, `x`, for that function as argument. The function `twice` applies `f` two times to the argument `x`:

```
twice :: (t->t) t -> t
twice f x = f (f x)
```

Since the argument `f` is used as function it has a function type: `(t->t)`. Since the result of the first application is used as argument of the second application, these types should be equal. The value `x` is used as argument of `f`, hence it should have the same type `t`.

We show some examples of the use of `twice` using `inc n = n+1`. The arrow → indicates a single reduction step, the symbol $\rightarrow_*$ indicates a sequence of reduction steps (zero or more). We underline the part of the expression that will be rewritten:

```
twice inc 0
→  inc (inc 0)
→  inc (0+1)
→  inc 1
→  1+1
→  2

twice twice inc 0                    // f is bound to twice, and x is bound to inc.
→  twice (twice inc) 0
→  (twice inc) ((twice inc) 0)
→* (twice inc) 2                     // as in the previous example
→  inc (inc 2)
→* inc 3
→* 4
```

The parentheses in the type declaration of higher-order functions can be necessary to indicate which arguments belong to the function and which arguments belong to the type of the higher-order function, i.e. to distinguish `x (y->z) -> u`, `(x y->z) -> u` and `x y -> (z->u)`. Without parentheses, types associate to the right. This implies that `x y -> z -> u` means `x y -> (z->u)`. It is always allowed to insert additional parentheses to indicate the association more clearly.

2.3.1 Functions on lists

The function `map` is another example of a higher-order function. This function takes care of the principle of 'handling all elements in a list'. What has to be done to the elements of the list, is specified by the function, which, next to the list, is passed to `map`.

The function `map` can be defined as follows (the first rule of the definition states the type of the function (you can ask the CLEAN system to derive the types for you). It expresses that `map` takes two arguments, a function (of arguments of type `a` to results of type `b`) and a list (of elements of type `a`); the result will be a list of elements of type `b`) :

```
map :: (a->b) [a] -> [b]
map f []       = []
map f [x:xs]   = [f x : map f xs]
```

The definition uses patterns: the function is defined separately for the case the second argument is a list without elements, and for the case the list consists of a first element `x` and a remainder `xs`. The function is recursive: in the case of a non-empty list the function `map` is applied again. In the recursive application, the argument is shorter (the list `xs` is shorter than the list `[x:xs]`); finally the non-recursive part of the function will be applied.

Another frequently used higher-order function on lists is `filter`. This function returns those elements of a list, which satisfies some condition. The condition to be used is passed as an argument to `filter`. Examples of the use of `filter` are (here `[1..10]` is the CLEAN short-hand for the list `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`):

```
filter isEven [1..10]
```

which yields `[2, 4, 6, 8, 10]`, and

```
filter ((>)10) [2,17,8,12,5]
```

which yields `[2,8,5]` (because e.g. `(>) 10 2` is equivalent with `10 > 2`). Note that in the last example the operator `>` is Curried. If the list elements are of type `a`, then the function parameter of `filter` has to be of type `a->Bool`. Just as `map`, the definition of `filter` is recursive:

```
filter :: (a->Bool) [a] -> [a]
filter p [] = []
filter p [x:xs]
    | p x       = [x : filter p xs]
    | otherwise = filter p xs
```

In case the list is not empty (so it is of the form `[x:xs]`), there are two cases: either the first element `x` satisfies `p`, or it does not. If so, it will be put in the result; the other elements are (by a recursive call) 'filtered'.

2.3.2 Iteration

In mathematics *iteration* is often used. This means: take an initial value, apply some function to that, until the result satisfies some condition.

Iteration can be described very well by a higher-order function. In the standard module `StdFunc` this function is called `until`. Its type is:

```
until :: (a->Bool) (a->a) a -> a
```

The function has three arguments: the property the final result should satisfy (a function `a->Bool`), the function which is to be applied repeatedly (a function `a->a`), and an initial value (of the type `a`). The final result is also of type `a`. The call `until p f x` can be read as: 'until `p` is satisfied, apply `f` to `x`'.

The definition of `until` is recursive. The recursive and non-recursive cases are this time not distinguished by patterns, but by Boolean expression:

```
until p f x
    | p x       = x
    | otherwise = until p f (f x)
```

If the initial value `x` satisfies the property `p` immediately, then the initial value is also the final value. If not the function `f` is applied to `x`. The result, `(f x)`, will be used as a new initial value in the recursive call of `until`.

Like all higher-order functions `until` can be conveniently called with partially parameterized functions. For instance, the expression below calculates the first power of two which is greater than `1000` (start with `1` and keep on doubling until the result is greater than `1000`):

```
until ((<)1000) ((*)2) 1
```

The result is the first power of two that is bigger than `1000`, that is `1024`.

In contrast to previously discussed recursive functions, the argument of the recursive call of `until` is not 'smaller' than the formal argument. That is why `until` does not always terminate with a result. When calling `until ((>)0) ((+)1) 1` the condition is never satisfied (note that `0` is the left argument of `>`); the function `until` will keep on counting indefinitely, and it will never return a result.

If a program does not yield an answer because it is computing an infinite recursion, the running program has to be interrupted by the user. Often the program will interrupt itself when its memory resources (stack space or heap space) are exhausted.

The function `iterate` behaves like an unbounded until. It generates a list of elements obtained by applying a given function `f` again and again to an initial value `x`.

```
iterare :: (t->t) t -> [t]
iterate f x = [x: iterate f (f x)]
```

An application of this function will be shown in section 2.4.2.

2.3.3 The lambda notation

Sometimes it is very convenient to define a tiny function 'right on the spot' without being forced to invent a name for such a function. For instance assume that we would like to calculate x^2+3x+1 for all x in the list `[1..100]`. Of course, it is always possible to define the function separately in a `where` clause:

```
ys = map f [1..100]
where
    f x = x*x + 3*x + 1
```

However, if this happens too much it gets a little annoying to keep on thinking of names for the functions, and then defining them afterwards. For these situations there is a special notation available, with which functions can be created without giving them a name:

```
\ pattern -> expression
```

This notation is known as the *lambda notation* (after the greek letter λ; the symbol `\` is the closest approximation for that letter available on most keyboards…).

An example of the lambda notation is the function `\x -> x*x+3*x+1`. This can be read as: 'the function that, given the argument x, will calculate the value of x^2+3x+1'. The lambda notation is often used when passing functions as an argument to other functions, as in:

```
ys = map (\x->x*x+3*x+1) [1..100]
```

Lambda notation can be used to define function with several arguments. Each of these arguments can be an arbitrary pattern. However, multiple alternatives and guards are not allowed in lambda notation. This language construct is only intended as a short notation for fairly simple functions which do not deserve a name.

With a *lambda expression* a new scope is introduced. The formal parameters have a meaning in the corresponding function body.


```
\ args -> body
```

Figure 2.1: Scope of lambda expression.

Local definitions in a lambda expression can be introduced through a let expression. A ridiculous example is a very complex identity function:

```
difficultIdentity :: !a -> a
difficultIdentity x = (\y -> let z = y in z) x
```

2.3.4 Function composition

If f and g are functions, than $g.f$ is the mathematical notation for '*g after f*': the function which applies f first, and then g to the result. In CLEAN the operator which composes two functions is also very useful. It is simply called `o` (not `.` since the `.` is used in real denotations and for selection out of a record or an array), which may be written as an infix operator. This makes it possible to define:

```
odd         = not o isEven
closeToZero = ((>)10) o abs
```

The operator `o` can be defined as a higher-order operator:

```
(o) infixr 9 :: (b -> c) (a -> b) -> (a -> c)
(o) g f = \x -> g (f x)
```

The lambda notation is used to make `o` an operator defined on the desired two arguments. It is not allowed to write `(o) g f x = g (f x)` since an infix operator should have exactly two arguments. So, we have to define the function composition operator `o` using a lambda notation or a local function definition. The more intuitive definition

```
comp g f x = g (f x)
```

has three arguments and type `(b -> c) (a -> b) a -> c`. Although this is a perfectly legal function definition in CLEAN, it cannot be used as an infix operator.

Without lambda notation a local function should be used:

```
(o) g f = h
where   h x = g (f x)
```

Not all functions can be composed to each other. The *range* of `f` (the type of the result of `f`) has to be equal to the *domain* of `g` (the type of the argument of `g`). So if `f` is a function `a -> b`, `g` has to be a function `b -> c`. The composition of two functions is a function which goes directly from `a` to `c`. This is reflected in the type of `o`.

The use of the operator `o` may perhaps seem limited, because functions like `odd` can be defined also by

```
odd x = not (isEven x)
```

However, a composition of two functions can serve as an argument for another higher order function, and then it is convenient that it need not be named. The expression below evaluates to a list with all odd numbers between `1` and `100`:

```
filter (not o isEven) [1..100]
```

Using function composition a function similar to `twice` (as defined in the beginning of section 2.3) can be defined:

```
Twice :: (t->t) -> (t->t)
Twice f = f o f
```

In the standard module `StdFunc` the function composition operator is pre-defined. The operator is especially useful when many functions have to be composed. The programming can be done at a function level; low level things like numbers and lists have disappeared from sight. It is generally considered much nicer to write

```
f   = g o h o i o j o k
```

rather than

```
f x = g(h(i(j(k x))))
```

2.4 Numerical functions

2.4.1 Calculations with integers

When dividing integers (`Int`) the part following the decimal point is lost: `10/3` equals `3`. Still it is not necessary to use `Real` numbers if you do not want to loose that part. On the contrary: often the *remainder* of a division is more interesting than the decimal fraction. The remainder of a division is the number which is on the last line of a long division. For instance in the division `345/12`

```
1 2 / 3 4 5 \ 2 8
      2 4
      1 0 5
        9 6
          9
```

is the quotient `28` and the remainder `9`.

The remainder of a division can be determined with the standard operator `rem`. For example, `345 rem 12` yields `9`. The remainder of a division is for example useful in the next cases:

- Calculating with times. For example, if it is now `9` o'clock, then `33` hours later the time will be `(9+33) rem 24 = 20` o'clock.
- Calculating with weekdays. Encode the days as 0=Sunday, 1=Monday, …, 6=Saturday. If it is day 3 (Wednesday), then in 40 days it will be `(3+40) rem 7 = 1` (Monday).
- Determining divisibility. A number *m* is divisible by *n* if the remainder of the division by *n* equals zero; `m rem n == 0`.
- Determining decimal representation of a number. The last digit of a number `x` equals `x rem 10`. The last but one digit equals `(x/10) rem 10`. The second next equals `(x/100) rem 10`, etcetera.

As a more extensive example of calculating with whole numbers two applications are discussed: the calculation of a list of prime numbers and the calculation of the day of the week on a given date.

Calculating a list of prime numbers

A number can be divided by another number, if the remainder of the division by that number, equals zero. The function `divisible` tests two numbers on divisibility:

```
divisible :: Int Int -> Bool
divisible t n = t rem n == 0
```

The denominators of a number are those numbers it can be divided by. The function `denominators` computes the list of denominators of a number:

```
denominators :: Int -> [Int]
denominators x = filter (divisible x) [1..x]
```

Note that the function `divisible` is partially parameterized with `x`; by calling `filter` *those* elements are filtered out `[1..x]` by which `x` can be divided.

A number is a prime number iff it has exactly two divisors: `1` and itself. The function `prime` checks if the list of denominators indeed consists of those two elements:

```
prime :: Int -> Bool
prime x = denominators x == [1,x]
```

The function `primes` finally determines all prime numbers up to a given upper bound:

```
primes :: Int -> [Int]
primes x = filter prime [1..x]
```

Although this may not be the most efficient way to calculate primes, it is the easiest way: the functions are a direct translation of the mathematical definitions.

Compute the day of the week

On what day will be the last New Year's Eve this century? Evaluation of


```
day 31 12 1999
```

will yield `"Friday"`. If the number of the day is known (according to the mentioned coding `0`=Sunday etc.) the function `day` is very easy to write:

```
:: Day   :== Int
:: Month :== Int
:: Year  :== Int

day :: Day Month Year -> String
day d m y = weekday (daynumber d m y)

weekday :: Day -> String
weekday 0 = "Sunday"
weekday 1 = "Monday"
weekday 2 = "Tuesday"
weekday 3 = "Wednesday"
weekday 4 = "Thursday"
weekday 5 = "Friday"
weekday 6 = "Saturday"
```

The function `weekday` uses seven patterns to select the right text (a quoted word is a *string*; for details see subsection 3.6).

When you do not like to introduce a separate function `weekday` with seven alternatives you can also use a case expression:

```
day :: Day Month Year -> String
day d m y =   case daynumber d m y of
                  0 -> "Sunday"
                  1 -> "Monday"
                  2 -> "Tuesday"
                  3 -> "Wednesday"
                  4 -> "Thursday"
                  5 -> "Friday"
                  6 -> "Saturday"
```

The first pattern in the case that matches the value of the expression between `case` and `of` is used to determine the value of the expression. In general a case expression consists of the key word `case`, an expression, the key word `of` and one or more alternatives. Each alternative consists of a pattern the symbol `->` and an expression. Like usual you can use a variable to write a pattern that matches each expression. As in functions you can replace the variable pattern by _ when you are not interested in its value.

A *case expression* itroduces a new scope. The scope rules are identical to the scope rules of an ordinary function definition.

```
case expression of
      args -> body
      args -> body
```

Figure 2.2: Scopes in a case expression.

When you find even this definition of `day` to longwinded you can use the daynumber as list selector. The operator `!!` selects the indicated element of a list. The first element has index `0`.

```
day :: Day Month Year -> String
day d m y  = ["Sunday","Monday","Tuesday","Wednesday"
             ,"Thursday","Friday","Saturday"] !! daynumber d m y
```

The function `daynumber` chooses a Sunday in a distant past and adds:

- the number of years passed since then times `365`;
- a correction for the elapsed leap years;
- the lengths of this years already elapsed months;
- the number of passed days in the current month.

Of the resulting (huge) number the remainder of a division by 7 is determined: this will be the required day number.

As origin of the day numbers we could choose the day of the calendar adjustment. But it will be easier to extrapolate back to the fictional day before the very first day of the calendar: day Zero, i.e. the day before the First of January Year 1, which is, ofcourse, day 1. That fictional day Zero will have daynumber 0 and would then be on a Sunday. Accordingly, the first day of the calendar (First of January, Year 1) has daynumber 1 and is on a Monday, etcetera. The definition of the function daynumber will be easier by this extrapolation.

```
daynumber :: Day Month Year -> Int
daynumber d m y
    =   ( (y-1)*365                        // days in full years before this year
        + (y-1)/4                          // ordinary leap year correction
        - (y-1)/100                        // leap year correction for centuries
        + (y-1)/400                        // leap year correction for four centuries
        + sum (take (m-1) (months y))      // days in months of this year
        + d
        ) rem 7
```

The call take n xs returns the first n elements of the list xs. The function take is defined in the StdEnv. It can be defined by:

```
take :: Int [a] -> [a]
take 0 xs      = []
take n [x:xs]  = [x : take (n-1) xs]
```

The function months should return the lengths of the months in a given year:

```
months :: Year -> [Int]
months y = [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
where
    feb | leap y    = 29
        | otherwise = 28
```

You might find it convenient to use the predefined conditional function if to eliminate the local definition feb in months. The definition becomes:

```
months :: Year -> [Int]
months y = [31, if (leap y) 29 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

The function if has a special definition for efficiency reasons. Semantically it could have been defined as

```
if :: !Bool t t -> t
if condition then else
    | condition = then
    | otherwise = else
```

Since the calendar adjustment of pope Gregorius in 1752 the following rule holds for leap years (years with 366 days):

- a year divisible by 4 is a leap year (e.g. 1972);
- but: if it is divisible by 100 it is no leap year (e.g. 1900);
- but: if it is divisible by 400 it *is* a leap year (e.g. 2000).

```
leap :: Year -> Bool
leap y = divisible y 4 && (not(divisible y 100) || divisible y 400)
```

Another way to define this is:

```
leap :: Year -> Bool
leap y
    | divisible y 100   = divisible y 400
    | otherwise         = divisible y 4
```

With this the function day and all needed auxiliary functions are finished. It might be sensible to add to the function day that it can only be used for years after the calendar adjustment:

```
day :: Day Month Year -> String
day d m y
    | y>1752 = weekday (daynumber d m y)
```


calling day with a smaller year yields automatically an error. This definition of day is an example of a *partial function*: a function which is not defined on some values of its domain. An error will be generated automatically when a partial function is used with an argument for which it is not defined.

```
Run time error, rule 'day' in module 'testI2' does not match
```

The programmer can determine the error message by making the function a *total function* and generating an error with the library function abort. This also guarantees that, as intended, daynumber will be called with positive years only.

```
day :: Day Month Year -> String
day d m y
    | y>1752    = weekday (daynumber d m y)
    | otherwise = abort ("day: undefined for year "+++toString y)
```

When designing the prime number program and the program to compute the day of the week two different strategies were used. In the second program the required function day was immediately defined. For this the auxiliary function weekday and daynumber were needed. To implement daynumber a function months was required, and this months needed a function leap. This approach is called *top-down*: start with the most important, and gradually fill out all the details.

The prime number example used the *bottom-up* approach: first the function divisible was written, and with the help of that one the function denominators, with that a function prime and concluding with the required prime.

It does not matter for the final result (the compiler does not care in which order the functions are defined). However, when designing a program it can be useful to determine which approach you use, (bottom-up or top-down), or that you even use a mixed approach (until the 'top' hits the 'bottom').

2.4.2 Calculations with reals

When calculating Real numbers an exact answer is normally not possible. The result of a division for instance is rounded to a certain number of decimals (depending on the calculational preciseness of the computer): evaluation of 10.0/6.0 yields 1.6666667, not 1 2/3. For the computation of a number of mathematical operations, like sqrt, also an approximation is used. Therefore, when designing your own functions which operate on Real numbers it is acceptable the result is also an approximation of the 'real' value. The approximation results in rounding errors and a maximum value. The exact approximation used is machine dependent. In chapter 1 we have seen some approximated real numbers. You can get an idea of the accuracy of real numbers on your computer by executing one of the follwing programs.

```
Start = "e  = " +++ toString (exp 1.0) +++ "\npi = " +++ toString (2.0*asin 1.0)

Start = takeWhile ((<>) 0.0) (iterate (\x -> x/10.0) 1.0)

takeWhile::(a -> Bool) [a] -> [a]
takeWhile f []  = []
takeWhile f [x:xs]
    | f x       = [x:takeWhile f xs]
    | otherwise = []
```

The first program computes the value of some well known constants.

The second program generates a list of numbers. The second program uses the function takeWhile which yields the longest prefix of the list argument for which the elements satisfy the predicate f. takeWhile gets a list in which each number is 10.0 times smaller than its predecessor. The result list ends when the number cannot be determined to be different from 0. Without approximations in the computer, this program will run forever.

The derivative function

An example of a calculation with reals is the calculation of the derivative function The mathematical definition of the derivative f' of the function f is:

$$f'(x) \quad = \quad \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

The precise value of the limit cannot be calculated by the computer. However, an approximated value can be obtained by using a very small value for h (not *too* small, because that would result in unacceptable rounding errors).

The operation 'derivative' is a higher-order function: a function goes in and a function comes out. The definition in CLEAN could be:

```
diff :: (Real->Real) -> (Real->Real)
diff f = derivative_of_f
where
    derivative_of_f x  = (f (x+h) - f x) / h
    h                  = 0.0001
```

The function `diff` is very amenable to partial parameterization, like in the definition:

```
derivative_of_sine_squared :: (Real->Real)
derivative_of_sine_squared = diff (square o sin)
```

The value of `h` in the definition of `diff` is put in a `where` clause. Therefore it is easily adjusted, if the program would have to be changed in the future (naturally, this can also be done in the expression itself, but then it has to be done twice, with the danger of forgetting one).

Even more flexible it would be, to define the value of `h` as a parameter of `diff`:

```
flexDiff :: Real (Real->Real) Real -> Real
flexDiff h f x = (f (x+h) - f x) / h
```

By defining `h` as the first parameter of `flexDiff`, this function can be partially parameterized too, to make different versions of `diff`:

```
roughDiff   :: (Real->Real) Real -> Real
roughDiff   = flexDiff 0.01

fineDiff    = flexDiff 0.0001
superDiff   = flexDiff 0.000001
```

In mathematics you have probably learned to compute the derivative of a function symbolically. Since the definition of functions cannot be manipulated in languages like CLEAN, symbolic computation of derivatives is not possible here.

Definition of square root

The function `sqrt` which calculates the square root of a number, is defined in standard module `StdReal`. In this section a method will be discussed how you can make your own root function, if it would not have been built in. It demonstrates a technique often used when calculating with `Real` numbers.

For the square root of a number `x` the following property holds:

if y is an approximation of $\sqrt{x}$

then $\frac{1}{2}(y + \frac{x}{y})$ is a better approximation.

This property can be used to calculate the root of a number x: take 1 as a first approximation, and keep on improving the approximation until the result is satisfactory. The value y is good enough for $\sqrt{x}$ if y^2 is not too different from x.

For the value of $\sqrt{3}$ the approximations y_0, y_1 etc. are as follows:

$y_0 = 1 \qquad = 1$

$y_1 = 0.5*(y_0+3/y_0) \qquad = 2$
$y_2 = 0.5*(y_1+3/y_1) \qquad = 1.75$
$y_3 = 0.5*(y_2+3/y_2) \qquad = 1.732142857$
$y_4 = 0.5*(y_3+3/y_3) \qquad = 1.732050810$
$y_5 = 0.5*(y_4+3/y_4) \qquad = 1.732050807$

The square of the last approximation only differs 10^{18} from 3.

For the process 'improving an initial value until good enough' the function `until` from subsection 2.3.2 can be used:

```
root :: Real -> Real
root x = until goodEnough improve 1.0
where
    improve y    = 0.5*(y+x/y)
    goodEnough y = y*y ~=~ x
```

The operator `~=~` is the 'about equal to' operator, which can be defined as follows:

```
(~=~) infix 5 :: Real real -> Bool
(~=~) a b = a-b <h && b-a<h
where
    h = 0.000001
```

The higher-order function `until` operates on the *functions* `improve` and `goodEnough` and on the initial value `1.0`.

Although `improve` is next to `1.0`, the function `improve` is not applied immediately to `1.0`; instead of that both will be passed to `until`. This is caused by the Currying mechanism: it is like if there were parentheses as in `(((until goodEnough) improve) 1.0)`. Only when looking closely at the definition of `until` it shows that `improve` is still applied to `1.0`.

There are some other quite interesting observations possible with `improve` and `goodEnough`. These functions can, except for their parameter `y`, make use of `x`. For these functions `x` is an implicit parameter.

2.5 Exercises

1. Combining operators. Define the function `odd` using the operators even, `+` and `o`.
2. Finite precision of reals. Execute the start expression given in section 2.4.2. Rewrite the program such that it only prints the smallest number that is different from zero using the function until.
3. Counting days. Write a function that given the current day and your date of birth determines how many days you have to wait for your birthday.
4. Define the function `mapfun` that applies a list of functions to its second argument returning the results of these function applications into a list. So, e.g.

   ```
   mapfun [f,g,h] x = [f x, g x, h x]
   ```

Part I
Chapter 3
Data Structures

3.1 Lists
3.2 Infinite lists
3.3 Tuples
3.4 Records
3.5 Arrays
3.6 Algebraic datatypes
3.7 Abstract datatypes
3.8 Correctness of programs
3.9 Run-time errors
3.10 Exercises

Data structures are used to store and manipulate collections of data and to represent specific data values. The example of a datatype representing specific values that we have seen is the datatype Bool which contains the values `True` and `False`. In section 3.6 we teach you how to define this kind of algebraic datatypes.

The lists that we have used every now and then are an example of a recursive algebraic datatype. In principle it is possible to define all datatypes directly in CLEAN. Since a number of these datatypes are used very frequently in functional programming they are predefined in CLEAN. In order to make the manipulation of these datatypes easier and syntactically nicer special purpose notation is introduced for a number of datatypes.

Lists are by far the most used recursive datatype used in functional programming. Lists hold an arbitrary number of elements of the same type. They are discussed in section 3.1 and 3.2. Tuples hold a fixed number of data values that can be of different types. The use of tuples is treated in section 3.3. Records are similar to tuples. The difference between a record and a tuple is that fields in a record are indicated by their name, while in a tuple they are indicated by their position. Records are discussed in section 3.4. The last predefined datatype discussed in this chapter are arrays. Arrays are similar to fixed length lists. In contrast to lists an array element can be selected in constant time. Usually, it is only worthwhile to use arrays instead of lists when this access time is of great importance.

3.1 Lists

In the previous chapters we have seen some lists. A list is a sequence of elements of the same type. The elements of the list can have any type, provided that each element has the same type. The elements of a list are written between the square brackets `[` and `]`. The elements are separated by a comma. For example the list of the first five prime numbers is `[2,3,5,7,11]`. You can construct a new list of an element and a list by the infix operator `:`. For example `[1:[2,3,5,7,11]]`. This list can also be written as `[1,2,3,5,7,11]`. Both notations can be used in the patterns of functions manipulating lists. In this section we will elaborate on lists and list processing functions.

3.1.1 Structure of a list

Lists are used to group a number of elements. Those elements should be of the *same type*.

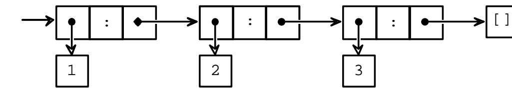


Figure 3.1: Pictorial representation of the list `[1,2,3]`.

A list in CLEAN should be regarded as a *linked list*: a chain of `:`-boxes (called the *spine of a list*) referring to each other. The most simple list is the empty list `[]` which indicates the end of a list. A non-empty list is of shape `[x:xs]` where `x` refers to a list element and `xs` refers to a list. A pictorial representation of a list is given in figure 3.1.

For every type there exists a type `list of that type'. Therefore there are lists of integers, lists of reals and lists of functions from `Int` to `Int`. But also a number of lists of the same type can be stored in a list; in this way you get lists of lists of integers, lists of lists of lists of Booleans and so forth.

The type of a list is denoted by the type of its elements between square brackets. The types listed above can thus be expressed shorter by `[Int]`, `[Real]`, `[Int->Int]`, `[[Int]]` and `[[[Bool]]]`.

There are several ways to construct a list: by enumeration, by construction using `:` and by specification of an interval.

Enumeration

Enumeration of the elements often is the easiest method to build a list. The elements must be of the same type. Some examples of list enumerations with their types are:

```
[1,2,3]              :: [Int]
[1,3,7,2,8]          :: [Int]
[True,False,True]    :: [Bool]
[sin,cos,tan]        :: [Real->Real]
[[1,2,3],[1,2]]      :: [[Int]]
```

For the type of the list it doesn't matter how many elements there are. A list with three integer elements and a list with two integer elements both have the type `[Int]`. That is why in the fifth example the lists `[1,2,3]` and `[1,2]` can in turn be elements of one list of lists.

The elements of a list need not be constants; they may be determined by a computation:

```
[1+2,3*x,length [1,2]]  :: [Int]
[3<4,a==5,p && q]       :: [Bool]
```

The expressions used in a list must all be of the same type.

There are no restrictions on the number of elements of a list. A list therefore can contain just one element:

```
[True]                  :: [Bool]
[[1,2,3]]               :: [[Int]]
```

A list with one element is also called a *singleton list*. The list `[[1,2,3]]` is a singleton list as well, for it is a list of lists containing one element (the list `[1,2,3]`).

Note the difference between an *expression* and a *type*. If there is a type between the square brackets, the whole is a type (for example `[Bool]` or `[[Int]]`). If there is an expression between the square brackets, the whole is an expression as well (a singleton list, for example `[True]` or `[3]`).

Furthermore the number of elements of a list can be zero. A list with zero elements is called the *empty list*. The empty list has a polymorphic type: it is a `list of whatever'. At positions in a polymorphic type where an arbitrary type can be substituted type variables are used (see subsection 1.5.3); so the type of the empty list is `[a]`:

```
[] :: [a]
```

The empty list can be used in an expression wherever a list is needed. The type is then determined by the context:

`sum []`	`[]` is an empty list of numbers
`and []`	`[]` is an empty list of Booleans
`[[],[1,2],[3]]`	`[]` is an empty list of numbers
`[[1<2,True],[]]`	`[]` is an empty list of Booleans
`[[[1]],[]]`	`[]` is an empty list of lists of numbers
`length []`	`[]` is an empty list (doesn't matter of what type)

Construction using :

Another way to build a list is by using the notation involving `:` . This notation most closely follows the way lists are actually represented internally in the CLEAN system. For example, the list `xs = [1,2,3]` is actually a shorthand for `xs = [1:[2:[3:[]]]]`. One can imagine this list to be constructed internally as shown in figure 3.2.

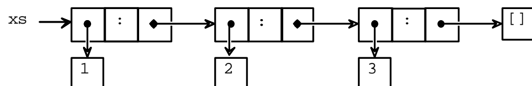


Figure 3.2: Pictorial representation of the list defined as `xs = [1,2,3]`.

If `xs` is a list (say `xs = [1,2,3]`), `[0:xs]` is a list as well, the list `[0,1,2,3]`. The new list is constructed by making a new box to store `[x:xs]`, where `x` refers to a new box containing `0` and `xs` refers to the old list. In figure 3.3 the pictorial representation is shown.

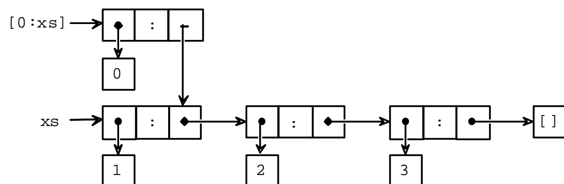


Figure 3.3: Pictorial representation of the list `[0,xs] where xs = [1,2,3]`.

The operator `:` is often called *cons*. In the same jargon the empty list `[]` is called *nil*.

Enumerable intervals

The third way to construct a list is the interval notation: two numeric expression with two dots between and square brackets surrounding them: the expression `[1..5]` evaluates to `[1,2,3,4,5]`. The expression `[1..5]` is a special notation, called a *dot-dot expression*. Another form of a dot-dot expression is `[1,3..9]` in which the interval is `2` (the difference between `3` and `1`). The dot-dot expression is internally translated to a function calculating the interval. For instance, the expression `[first,second..upto]` is translated to `from_then_to first second upto`, which in the case of `[1,3..9]` evaluates to `[1,3,5,7,9]`. `from_then_to` is a predefined function (see `StdEnum`) which is defined as follows:

```
from_then_to : a a a -> [a] | Enum a
from_then_to n1 n2 e
    | n1 <= n2  = _from_by_to n1 (n2-n1) e
    | otherwise = _from_by_down_to n1 (n2-n1) e
where
    from_by_to n s e
        | n <= e    = [n : _from_by_to (n+s) s e]
        | otherwise = []
```



```
    from_by_down_to n s e
        | n >= e    = [n : _from_by_down_to (n+s) s e]
        | otherwise = []
```

The dot-dot expression `[1..5]` can be seen as a special case of the expression `[1,2..5]`. When the step size happens to be one the element indicating the step size can be omitted.

When a list does not have an upperbound, it can be omitted. In this way one can specify a list with an infinite number of elements. Such alist list generated will be evaluated as far as necessary. See also section 3.2. Some examples are:

`[1..]`	generates the list	`[1,2,3,4,5,6,7,8,9,10,…`
`[1,3..]`	generates the list	`[1,3,5,7,9,11,13,15,…`
`[100,80..]`	generates the list	`[100,80,60,40,20,0,-20,-40,…`

Besides for integer numbers a dot-dot expression can also be used for other enumerables (class `Enum`, see chapter 4), such as real numbers and characters. E.g. the expression `['a'..'c']` evaluates to `['a','b','c']`.

3.1.2 Functions on lists

Functions on lists are often defined using *patterns*: the function is defined for the empty list `[]` and the list of the form `[x:xs]` separately. For a list is either empty or has a first element `x` in front of a (possibly empty) list `xs`.

A number of definitions of functions on lists have already been discussed: `hd` and `tl` in subsection 1.4.3, `sum` and `length` in subsection 1.4.4, and `map` and `filter` in subsection 2.3.1. Even though these are all standard functions defined in the standard environment and you don't have to define them yourself, it is important to look at their definitions. Firstly because they are good examples of functions on lists, secondly because the definition often is the best description of what a standard function does.

In this paragraph more definitions of functions on lists follow. A lot of these functions are recursively defined, which means that in the case of the pattern `[x:xs]` they call themselves with the (smaller) parameter `xs`. This is a direct consequence of the fact that lists themselves are recursively defined.

Selecting parts of lists

In the standard environment a number of functions are defined that select parts of a list. As a list is built from a head and a tail, it is easy to retrieve these parts again:

```
hd :: [a] -> a
hd [x:_] = x

tl :: [a] -> [a]
tl [_:xs] = xs
```

These functions perform pattern matching on their parameters, but observe that both functions are partial: there are no separate definitions for the pattern `[]`. If these functions are applied to an empty list, the execution will be aborted with an error message generated at run time:

```
hd of []
```

It is a little bit more complicated to write a function that selects the *last* element from a list. For that you need recursion:

```
last :: [a] -> a
last [x]    = x
last [x:xs] = last xs
```

The pattern `[x]` is just an abbreviation of `[x:[]]`. Again this function is undefined for the empty list, because that case is not covered by the two patterns. Just as `hd` goes with `tl`, `last` goes with `init`. The function `init` selects everything *but* the last element. Therefore you need recursion again:

```
init :: [a] -> [a]
init [x]    = []
init [x:xs] = [x:init xs]
```

Figure 3.4 gives a pictorial overview of the effect of applying the functions hd, tl, init and last to the list [1,2,3]. Notice that hd, tl and last simply return (a reference to) an existing list or list element, while for init new *cons* boxes have to be constructed (a new spine) referring to existing list elements. Have again a close look to the definition of these functions. The functions hd, tl and last yield a function argument as result while in the init function new list parts are being constructed on the right-hand side of the function definition.

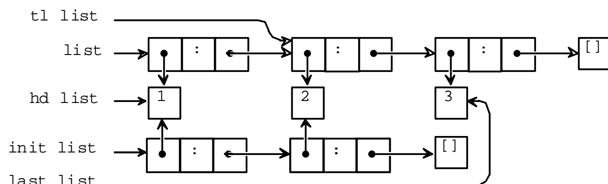


Figure 3.4: Pictorial representation of the list list = [1,2,3], and the result of applying the functions hd, tl, init and last to this list.

In subsection 2.4.1 a function take was presented. Apart from a list take has an integer as an argument, which denotes how many elements of the list must be part of the result. The counterpart of take is drop that deletes a number of elements from the beginning of the list. Finally there is an operator !! that select one specific element from the list. Schematic this is shown in figure 3.5.

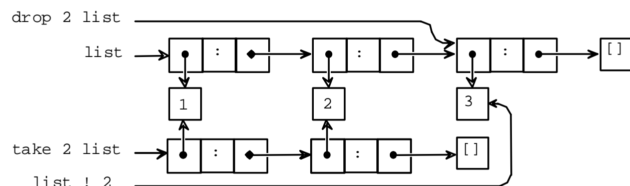


Figure 3.5: Pictorial representation of the list list = [1,2,3], and the result of applying the functions drop 2, take 2 and !! 2 to this list.

These functions are defined as follows:

```
take :: Int [a] -> [a]
take n [] = []
take n [x:xs]
    | n < 1     = []
    | otherwise = [x:take (n-1) xs]

drop :: int [a] -> [a]
drop n [] = []
drop n [x:xs]
    | n < 1     = [x:xs]
    | otherwise = drop (n-1) xs
```

Whenever a list is too short as much elements as possible are taken or left out respectively. This follows from the first line in the definitions: if you give the function an empty list, the result is always an empty list, whatever the number is. If these lines were left out of the definitions, then take and drop would be undefined for lists that are too short. Also with

respect to the number of elements to take or drop these functions are foolproof[1]: all negative numbers are treated as 0.

The operator !! selects one element from a list. The head of the list is numbered `zero' and so xs!!3 delivers the *fourth* element of the list xs. This operator cannot be applied to a list that is too short; there is no reasonable value in that case. The definition is similar to:

```
(!!) infixl 9 :: [a] Int -> a
(!!) [x:xs] n
    | n == 0    = x
    | otherwise = xs!!(n-1)
```

For high numbers this function costs some time: the list has to be traversed from the beginning. So it should be used economically. The operator is suited to fetch one element from a list. The function weekday from subsection 2.4.1 could have been defined this way:

```
weekday d = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"] !! d
```

However, if all elements of the lists are used successively, it's better to use map or foldr.

Reversing lists

The function reverse from the standard environment reverses the elements of a list. The function can easily be defined recursively. A reversed empty list is still an empty list. In case of a non-empty list the tail should be reversed and the head should be appended to the end of that. The definition could be like this:

```
reverse :: [a] -> [a]
reverse []     = []
reverse [x:xs]  = reverse xs ++ [x]
```

The effect of applying reverse to the list [1,2,3] is depicted in figure 3.6.

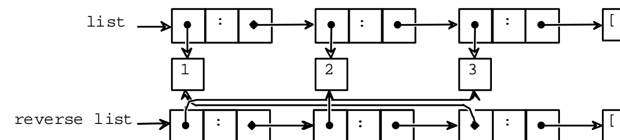


Figure 3.6: Pictorial representation of the list list = [1,2,3], and the effect of applying the function reverse to this list.

Properties of lists

An important property of a list is its length. The length can be computed using the function length. In the standard environment this function is defined equivalent with:

```
length :: [a] -> Int
length []      = 0
length [_:xs]  = 1 + length xs
```

Furthermore, the standard environment provides a function isMember that tests whether a certain element is contained in a list. That function isMember can be defined as follows:

```
isMember :: a [a] -> Bool | == a
isMember e xs = or (map ((==)e) xs)
```

The function compares all elements of xs with e (partial parameterization of the operator ==). That results in a list of Booleans of which or checks whether there is at least one equal to True. By the utilization of the function composition operator the function can also be written like this:

```
isMember :: a -> ([a] -> Bool) | == a
isMember e = or o map ((==) e)
```

The function notMember checks whether an element is not contained in a list:

```
notMember e xs = not (isMember e xs)
```

[1]In this respect the functions shown here differ from the function provided in the StdEnv of Clean 1.3.

Comparing and ordering lists

Two lists are equal if they contain exactly the same elements in the same order. This is a definition of the operator `==` which tests the equality of lists:

```
(==) infix 4 :: [a] [a] -> Bool | == a
(==) []     []     = True
(==) []     [y:ys] = False
(==) [x:xs] []     = False
(==) [x:xs] [y:ys] = x==y && xs==ys
```

In this definition both the first and the second argument can be empty or non-empty; there is a definition for all four combinations. In the fourth case the corresponding elements are compared (`x==y`) and the operator is called recursively on the tails of the lists (`xs==ys`).

As the overloaded operator `==` is used on the list elements, the equality test on lists becomes an overloaded function as well. The general type of the overloaded operator `==` is defined in `StdOverloaded` as:

```
(==) infix 4 a :: a a -> Bool
```

With the definition of `==` on lists an new instance of the overloaded operator `==` should be defined with type:

```
instance == [a] | == a
where
    (==) infix 4 :: [a] [a] -> Bool | == a
```

which expresses the `==` can be used on lists under the assumption that `==` is defined on the elements of the list as well. Therefore lists of functions are not comparable, because functions themselves are not. However, lists of lists of integers are comparable, because lists of integers are comparable (because integers are).

If the elements of a list can be ordered using `<`, then lists can also be ordered. This is done using the *lexicographical ordering* (`dictionary ordering'): the first elements of the lists determine the order, unless they are same; in that case the second element is decisive unless they are equal as well, etcetera. For example, `[2,3]<[3,1]` and `[2,1]<[2,2]` hold. If one of the two lists is equal to the beginning of the other then the shortest one is the `smallest`, for example `[2,3]<[2,3,4]`. The fact that the word `etcetera' is used in this description, is a clue that recursion is needed in the definition of the function `<` (less than):

```
(<)  infix  4 :: [a] [a] -> Bool | < a
(<) []      []      = False
(<) []      _       = True
(<) _       []      = False
(<) [x:xs]  [y:ys]  = x < y || (x == y && xs < ys)
```

When the functions `<` and `==` have been defined, others comparison functions can easily be defined as well: `<>` (not equal to), `>` (greater than), `>=` (greater than or equal to) and `<=` (smaller than or equal to):

```
(<>) x y = not (x==y)
(>)  x y = y < x
(>=) x y = not (x<y)
(<=) x y = not (y<x)
max  x y = if (x<y) y x
min  x y = if (x<y) x y
```

For software engineering reasons, the other comparison functions are in CLEAN actually predefined using the derived class members mechanism (see chapter 4.1). The class `Eq` contains `==` as well as the derived operator `<>`, the class `Ord` contains `<` as well as the derived operators `>`, `>=`, `<=`, `max` and `min`.

Joining lists

Two lists with the same type can be joined to form one list using the operator `++`. This is also called *concatenation* (`chaining together'). E.g.: `[1,2,3]++[4,5]` results in the list `[1,2,3,4,5]`. Concatenating with the empty list (at the front or at the back) leaves a list unaltered: `[1,2]++[]` gives `[1,2]` and `[]++[1,2]` gives also `[1,2]`.

The operator `++` is a standard operator (see `StdList`) defined as:

```
(++) infixr 5 :: [a] [a] -> [a]
(++) []     ys = ys
(++) [x:xs] ys = [x:xs++ys]
```

There is another function for joining lists called `flatten`. It acts on a *list* of lists. All lists in the list of lists which are joined to form one single list. For example

```
flatten [[1,2,3],[4,5],[],[6]]
```

evaluates to `[1,2,3,4,5,6]`. The definition of `flatten` is as follows:

```
flatten :: [[a]] -> [a]
flatten []       = []
flatten [xs:xss] = xs ++ flatten xss
```

The first pattern, `[]`, the empty list, is an empty list *of lists* in this case. The result is an empty list *of elements*. In the second case of the definition the list of lists is not empty, so there is a list, `xs`, in front of a rest list of lists, `xss`. First all the rest lists are joined by the recursive call of `flatten`; then the first list `xs` is put in front of that as well.

Note the difference between `++` and `flatten`: the operator `++` acts on *two* lists, the function `flatten` on *a list* of lists. Both are popularly called `concatenation'. (Compare with the situation of the operator `&&`, that checks whether two Booleans are both `True` and the function `and` that checks whether a whole list of Booleans only contains `True` elements).

3.1.3 Higher order functions on lists

Functions can be made more flexible by giving them a function as a parameter. A lot of functions on lists have a function as a parameter. Therefore they are higher-order functions.

In the standard library some handy higher-order functions on lists are predefined, like `map`, `filter` and `foldr`, which can often be used to replace certain types of recursive function definitions. Most people write ordinary recursive functions when they are new in functional programming. When they have more experience they start to recognize that they can equally well define these functions by applications of the standard general list processing functions.

map and filter

Previously `map` and `filter` were discussed. These functions process elements of a list. The action taken depends on the function argument. The function `map` applies its function parameter to each element of the list:

```
            xs  =  [  1  ,  2  ,  3  ,  4  ,  5  ]
                      ↓     ↓     ↓     ↓     ↓
     map square xs  →  [  1  ,  4  ,  9  ,  16 ,  25 ]
```

The `filter` function eliminates elements from a list that do not satisfy a certain Boolean predicate:

```
            xs  =  [  1  ,  2  ,  3  ,  4  ,  5  ]
                            ↓           ↓
   filter isEven xs  →  [        2  ,        4        ]
```

These three standard functions are defined recursively in the standard environment. They were discussed earlier in subsection 2.3.1.

```
map :: (a->b) [a] -> [b]
map f []     = []
map f [x:xs] = [f x : map f xs]

filter :: (a->Bool) [a] -> [a]
filter p []     = []
filter p [x:xs]
    | p x       = [x : filter p xs]
    | otherwise = filter p xs
```

By using these standard functions extensively the recursion in other functions can be hidden. The `dirty work' is then dealt with by the standard functions and the other functions look neater.

takewhile and dropwhile

A variant of the `filter` function is the function `takeWhile`. This function has, just like `filter`, a predicate (function with a Boolean result) and a list as parameters. The difference is that `filter` always looks at all elements of the list. The function `takeWhile` starts at the beginning of the list and stops searching as soon as an element is found, which does not satisfy the given predicate. For example: `takeWhile isEven [2,4,6,7,8,9]` gives `[2,4,6]`. Different from `filter` the `8` does not appear in the result, because the `7` makes `takeWhile` stop searching. The standard environment definition reads:

```
takeWhile :: (a->Bool) [a] -> [a]
takeWhile p [] = []
takeWhile p [x:xs]
    | p x        = [x : takeWhile p xs]
    | otherwise = []
```

Compare this definition to that of `filter`.

Like `take` goes with a function `drop`, `takeWhile` goes with a function `dropWhile`. This leaves out the beginning of a list that satisfies a certain property. For example: `dropWhile isEven [2,4,6,7,8,9]` equals `[7,8,9]`. Its definition reads:

```
dropWhile :: (a->Bool) [a] -> [a]
dropWhile p [] = []
dropWhile p [x:xs]
    | p x        = dropWhile p xs
    | otherwise = [x:xs]
```

There are several variants of the fold function. In this section we will compare them and give some hints on their use.

foldr

Folding functions can be used to handle the often occurring recursive operation on the elements of a list. There are several variants of these functions like `foldr` and `foldl`. The `foldr` function inserts an operator between all elements of a list starting at the right hand with a given value. ':' is replaced by the given operator and `[]` by the supplied value:

```
                 xs  =  [   1   : [ 2   : [ 3   : [ 4   : [ 5   :   []  ]]]]]
                            ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
      foldr (+) 0 xs  →  (   1   + ( 2   + ( 3   + ( 4   + ( 5   +   0   )))))
```

The definition in the standard environment is semantically equivalent to:

```
foldr :: (a->b->b) b [a] -> b
foldr op e []       = e
foldr op e [x:xs]   = op x (foldr op e xs)
```

By using standard functions extensively the recursion in other functions can be hidden. The `dirty work' is then dealt with by the standard functions and the other functions look neater.

For instance, take a look at the definitions of the functions `sum` (calculating the sum of a list of numbers), `product` (calculating the product of a list of numbers), and `and` (checking whether all elements of a list of Boolean values are all `True`):

```
sum []       = 0
sum [x:xs]   = x + sum xs

product []      = 1
product [x:xs]  = x * product xs

and []       = True
and [x:xs]   = x && and xs
```


The structure of these three definitions is the same. The only difference is the value which is returned for an empty list (`0`, `1` or `True`), and the operator being used to attach the first element to the result of the recursive call (`+`, `*` or `&&`). These functions can be defined more easily by using the `foldr` function:

```
sum     = foldr (+) 0
product = foldr (*) 1
and     = foldr (&&) True
```

A lot of functions can be written as a combination of a call to `foldr` and to `map`. A good example is the function `isMember`:

```
isMember e = foldr (||) False o map ((==)e)
```

The fold functions are in fact very general. It is possible to write map and filter as applications of fold:

```
mymap :: (a -> b) [a] -> [b]
mymap f list = foldr ((\h t -> [h:t]) o f) [] list

mymap2 :: (a -> b) [a] -> [b]
mymap2 f list = foldr (\h t -> [f h:t]) [] list

myfilter :: (a -> Bool) [a] -> [a]
myfilter f list = foldr (\h t -> if (f h) [h:t] t) [] list
```

As a matter of fact, it is rather hard to find list manipulating functions that cannot be written as an application of fold.

foldl

The function `foldr` puts an operator between all elements of a list and starts with this at the end of the list. The function `foldl` does the same thing, but starts at the beginning of the list. Just as `foldr`, `foldl` has an extra parameter that represents the result for the empty list.

Here is an example of `foldl` on a list with five elements:

```
                xs  =       [   1   : [ 2   : [ 3   : [ 4   : [ 5   : []]]]]
                                ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
     foldl (+) 0 xs  =  (((((0 + 1)  +   2)  +   3)  +   4)  +   5)
```

The definition can be written like this:

```
foldl::(a -> (b -> a)) !a ![b] -> a
foldl op e []       = e
foldl op e [x:xs]   = foldl op (op e x) xs
```

The element `e` has been made strict in order to serve as a proper accumulator.

In the case of associative operators like `+` it doesn't matter that much whether you use `foldr` or `foldl`. Of course, for non-associative operators like `-` the result depends on which function you use. In fact, the functions `or`, `and`, `sum` and `product` can also be defined using `foldl`.

From the types of the functions `foldl` and `foldr` you can see that they are more general then the examples shown above suggest. In fact nearly every list processing function can be expressed as a fold over the list. As example we show the reverse function:

```
reversel :: [a] -> [a]
reversel l = foldl (\r x -> [x:r]) [] l
```

These examples are not intended to enforce you to write each and every list manipulation as a fold, they are just intended to show you the possibilities.

3.1.4 Sorting lists

All functions on lists discussed up to now are fairly simple: in order to determine the result the lists is traversed once using recursion.

A list manipulation that cannot be written in this manner is the sorting (putting the elements in ascending order). The elements should be completely shuffled in order to accomplish sorting. However, it is not very difficult to write a sorting function. There are differ-

ent approaches to solve the sorting problem. In other words, there are different *algorithms*. Two algorithms will be discussed here. In both algorithms it is required that the elements can be ordered. So, it is possible to sort a list of integers or a list of lists of integers, but not a list of functions. This fact is expressed by the type of the sorting function:

```
sort :: [a] -> [a] | Ord a
```

This means: `sort` acts on lists of type `a` for which an instance of class `Ord` is defined. This means that if one wants to apply `sort` on an object of certain type, say `T`, somewhere an instance of the overloaded operator `<` on `T` has to be defined as well. This is sufficient, because the other members of `Ord` (`<=`, `>`, etcetera) can be derived from `<`.

Insertion sort

Suppose a sorted list is given. Then a new element can be inserted in the right place using the following function[1]:

```
Insert :: a [a] -> [a] | Ord a
Insert e [] = [e]
Insert e [x:xs]
    | e<=x      = [e,x : xs]
    | otherwise = [x : Insert e xs]
```

If the list is empty, the new element `e` becomes the only element. If the list is not empty and has `x` as its first element, then it depends on whether `e` is smaller than `x`. If this is the case, `e` is put in front of the list; otherwise, `x` is put in front and `e` must be inserted in the rest of the list. An example of the use of `Insert`:

```
Insert 5 [2,4,6,8,10]
```

evaluates to `[2,4,5,6,8,10]`. Observe that when `Insert` is applied, the parameter list has to be sorted; only then the result is sorted, too.

The function `Insert` can be used to sort a list that is not already sorted. Suppose `[a,b,c,d]` has to be sorted. You can sort this list by taking an empty list (which is sorted) and insert the elements of the list to be sorted one by one. The effect of applying the sorting function `isort` to our example list should be:

```
isort [a,b,c,d] = Insert d (Insert c (Insert b (Insert a [])))
```

Therefore one possible sorting algorithm is:

```
isort :: [a] -> [a] | Ord a
isort []    = []
isort [a:x] = Insert a (isort x)
```

with the function `insert` as defined above. This algorithm is called *insertion sort*.

The function `isort` could also be defined as follwes, using a `foldr` function:

```
isort :: [a] -> [a] | Ord a
isort = foldr Insert []
```

Merge sort

Another sorting algorithm makes use of the possibility to merge two sorted lists into one. This is what the function `merge` does[2]:

```
merge :: [a] [a] -> [a] | Ord a
merge []     ys = ys
merge xs     [] = xs
merge [x:xs] [y:ys]
    | x <= y    = [x : merge xs [y:ys]]
    | otherwise = [y : merge [x:xs] ys]
```

If either one of the lists is empty, the other list is the result. If both lists are non-empty, then the smallest of the two head elements is the head of the result and the remaining elements are merged by a recursive call to `merge`.

[1]We use `Insert` instead of `insert` to avoid name conflist with the function defined in `StdList`.
[2]This function definition is included in `StdList`.

In the last alternative of the function merge the arguments are taken apart by patterns. However, the lists are also used as a whole in the right-hand side. CLEAN provides a way to prevent rebuilding of these expressions in the body of the function. The pattern being matched can also be given a name as a whole, using the special symbol `=:`, as in the definition below:

```
merge :: [a] [a] -> [a] | Ord a
merge []       ys          = ys
merge xs       []          = xs
merge p=:[x:xs] q=:[y:ys]
     | x <= y              = [x : merge xs q]
     | otherwise           = [y : merge p ys]
```

Just like `insert`, `merge` supposes that the arguments are sorted. In that case it makes sure that also the result is a sorted list.

On top of the `merge` function you can build a sorting algorithm, too. This algorithm takes advantage of the fact that the empty list and singleton lists (lists with one element) are always sorted. Longer lists can (approximately) be split in two pieces. The two halves can be sorted by recursive calls to the sorting algorithm. Finally the two sorted results can be merged by `merge`.

```
msort :: [a] -> [a] | Ord a
msort xs
     | len <= 1  = xs
     | otherwise = merge (msort ys) (msort zs)
where
    ys     = take half xs
    zs     = drop half xs
    half   = len / 2
    len    = length xs
```

This algorithm is called *merge sort*. In the standard environment `insert` and `merge` are defined and a function `sort` that works like `isort`.

3.1.5 List comprehensions

In set theory the following notation to define sets is often used:

$$V = \{ x^2 \mid x \leftarrow N, x \bmod 2 = 0 \}.$$

This expression is called a *set comprehension*. The set V above consists of all squares of x (x^2), where x comes from the set N ($x \in N$), such that x is even ($x \bmod 2 = 0$). Analogously, in CLEAN a similar notation is available to construct lists, called a *list comprehension*. A simple example of this notation is the following expression:

```
Start :: [Int]
Start = [x*x \\ x <- [1..10]]
```

This expression can be read as `x squared for all `x` from `1` to `10`'. A list comprehension consists of two parts separated by a double backslash (\\). The left part consists of an expression denoting the elements of the result list. This expression might contain variables, introduced in the right part of the list comprehension. The latter is done via *generators*, i.e. expressions of the form `x<-xs` indicating that `x` ranges over all values in the list `xs`. For each of these values the value of the expression in front of the double backslash is computed.

Thus, the example above has the same value as

```
Start :: [Int]
Start = map (\x -> x*x) [1..10]
```

The advantage of the comprehension notation is that it is clearer.

Similar to set comprehensions we can add an additional predicate to the values that should be used to compute elements of the resulting list. The constraint is separated from the generator by a vertical bar symbol. The list of the squares of all integers between 1 and 10 that are even is computed by the following program.

```
Start :: [Int]
Start = [x*x \\ x <- [1..10] | x mod 2 == 0]
```

In a list comprehension after the double backslash more than one generator can appear separated by a `,`. This is called a *nested combination of generators*. With a nested combination of generators, the expression in front of the double backslash is computed for every possible combination of the corresponding bound variables. For example:

```
Start :: [(Int,Int)]
Start = [(x,y) \\ x<-[1..2], y<-[4..6]]
```

evaluates to the list

```
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]
```

By convention the last variable changes fastest: for each value of `x`, `y` traverses the list `[4..6]`.

Another way of combining generators is *parallel combination of generators*, indicated by separating the generators with a `&`-symbol instead of the `,`-symbol. With parallel combination of generators, the ith element is drawn from several lists at the same time. For example:

```
Start :: [(Int,Int)]
Start = [(x,y) \\ x<-[1..2] & y<-[4..6]]
```

evaluates to the list

```
[(1,4),(2,5)]
```

When the shortest list is exhausted, all generators combined with the `&`-operator stop.

In analogy to mathematical set comprehensions multiple generators can be combined with constraints. The constraint is separated from the generators by a vertical bar symbol. This is used in:

```
Start :: [(Int,Int)]
Start = [(x,y) \\ x<-[1..5], y<-[1..x] | isEven x]
```

which evaluates to

```
[(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```

In the resulting list only those values of `x` are stored for which `isEven x` evaluates to `True`. The scope of the variable `x` is not only restricted to the left-hand side of the comprehension but extends to the right-hand side of the generator introducing `x`. This explains why `x` can also be used in `isEven x` and in `[1..x]`. It is not allowed to use `y` in the generators preceding it. `y` can only be used in `(x,y)` and in the constraint.

In a *list comprehension* new variables are introduced when *generators* are specified. Each generator can generate a selector which can be tested in a guard and used to generate the next selector and finally in the resulting expression. The scope of the variables introduced by the generators is indicated in figure 3.7.

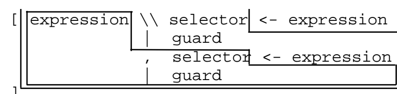


Figure 3.7: Scopes in a list comprehension.

The back-arrow (<-) is a special notation reserved for defining list comprehension and cannot be used as a common operator in an arbitrary expression.

Strictly speaking the list comprehension notation is superfluous. You can reach the same effect by combinations of the standard functions `map`, `filter` and `flatten`. However, especially in difficult cases the comprehension notation is more concise and therefore much easier to understand. Without it the example above should be written like


```
Start :: [(Int,Int)]
Start  =  flatten (map f (filter isEven [1..5]))
          where
              f x = map g [1..x]
              where
                  g y = (x,y)
```

which is less intuitive.

List comprehensions are very clear. For instance, all Pythagorean triangles with sides less or equal than 100 can be defined as follows:

```
triangles :: [(Int,Int,Int)]
triangles = [ (a,b,c)   \\ a <- [1..max]
                        ,  b <- [a..max]
                        ,  c <- [b..max]
                        |  a^2 + b^2 == c^2
            ]
            where max = 100
```

By using the list generators `[a..max]` and `[b..max]` we prevented that permutations of triangles are found.

The compiler translates the list comprehensions into an equivalent expression with `map`, `filter` and `flatten`. Just like the interval notation (the dot-dot expression), the comprehension notation is purely for the programmer's convenience. Using list comprehensions it is possible to define many list processing functions in a very clear and compact manner.

```
map :: (a->b) [a] -> [b]
map f l = [f x \\ x<-l]

filter :: (a->Bool) [a] -> [a]
filter p l = [x \\ x<-l | p x]
```

However, functions destructing the structure of the list (like `sum`, `isMember`, `reverse` and `take`) are impossible or hard to write using list comprehensions.

Quick sort

List comprehensions can be used to give a very clear definition of yet another sorting algorithm: *quick sort*. Similar to merge sort the list is split into two parts which are sorted separately. In merge sort we take the first half and second half of the list and sort these separately. In quick sort we select all elements less or equal to a median and all elements greater than the median and sort these separately. This has the advantage that the sorted sub-lists can be "merged" by the append operator `++`. We use the first element of the list as median to split the lists into two parts.

```
qsort :: [a] -> [a] | Ord a
qsort []     = []
qsort [a:xs] = qsort [x \\ x<-xs | x<a] ++ [a] ++ qsort [x \\ x<-xs | x>=a]
```

3.2 Infinite lists

3.2.1 Enumerating all numbers

The number of elements in a list can be infinite. The function `from` below returns an infinitely long list:

```
from :: Int -> [Int]
from n = [n : from (n+1)]
```

Of course, the computer can't store or compute an infinite number of elements. Fortunately you can already inspect the beginning of the list, while the rest of the list is still to be built. Execution of the program `Start = from 5` yields:

```
[5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,...
```

If an infinite list is the result of the program, the program will not terminate unless it is interrupted by the user.

An infinite list can also be used as an intermediate result, while the final result is finite. For example this is the case in the following problem: `determine all powers of three smaller than `1000`'. The first ten powers can be calculated using the following call:

```
Start :: [Int]
Start = map ((^)3) [1..10]
```

The result will be the list

```
[3,9,27,81,243,729,2187,6561,19683,59049]
```

The elements smaller than `1000` can be extracted by `takeWhile`:

```
Start :: [Int]
Start = takeWhile ((>) 1000) (map ((^)3) [1..10])
```

the result of which is the shorter list

```
[3,9,27,81,243,729]
```

But how do you know beforehand that `10` elements suffice? The solution is to use the infinite list `[1..]` instead of `[1..10]` and so compute *all* powers of three. That will certainly be enough…

```
Start :: [Int]
Start = takeWhile ((>) 1000) (map ((^)3) [1..])
```

Although the intermediate result is an infinite list, in finite time the result will be computed.

This method can be applied because when a program is executed functions are evaluated in a lazy way: work is always postponed as long as possible. That is why the outcome of `map ((^)3) (from 1)` is not computed fully (that would take an infinite amount of time). Instead only the first element is computed. This is passed on to the outer world, in this case the function `takeWhile`. Only when this element is processed and `takewhile` asks for another element, the second element is calculated. Sooner or later `takeWhile` will not ask for new elements to be computed (after the first number `>= 1000` is reached). No further elements will be computed by `map`. This is illustrated in the following trace of the reduction process:

```
takeWhile ((>) 5) (map ((^)) 3) [1..])
→  takeWhile ((>) 5) (map ((^) 3) [1:[2..]])
→  takeWhile ((>) 5) [(^) 3 1:map ((^) 3) [2..]]
→  takeWhile ((>) 5) [3:map ((^) 3) [2..]]
→  [3:takeWhile ((>) 5) (map ((^) 3) [2..])]          since (>) 5 3 → True
→  [3:takeWhile ((>) 5) (map ((^) 3) [2:[3..]])]
→  [3:takeWhile ((>) 5) [(^) 3 2: map ((^) 3) [3..]]]
→  [3:takeWhile ((>) 5) [9: map ((^) 3) [3..]]]
→  [3:[]]                                             since (>) 5 9 → False
```

As you might expect list comprehensions can also be used with infinite lists. The same program as above can be written as:

```
Start :: [Int]
Start = takeWhile ((>) 1000) [x^3 \\ x <- [1..]]
```

However, be careful not to write:

```
Start :: [Int]
Start = [x^3 \\ x <- [1..] | x^3 < 1000]
```

This is equivalent to

```
Start :: [Int]
Start = filter ((>) 1000) [x^3 \\ x <- [1..]]
```

Where the function `takeWhile` yields the empty list as soon as the predicate fails once, the function `filter` checks each element. For an infinite list, there are infinitely many elements to test. Hence this program will not terminate.

3.2.2 Lazy evaluation

The evaluation method (the way expressions are calculated) of CLEAN is called *lazy evaluation*. With lazy evaluation an expression (or part of it) is only computed when it is certain that its value is *really* needed for the result. The opposite of lazy evaluation is *strict evaluation*,

also called *eager evaluation*. With eager evaluation, before computing the a function's result, first all actual arguments of the function are evaluated.

Infinite lists can be defined thanks to lazy evaluation. In languages that use strict evaluation (like all imperative languages and some older functional languages) infinite lists are not possible.

Lazy evaluation has a number of other advantages. For example, consider the function `prime` from subsection 2.4.1 that tests whether a number is prime:

```
prime :: Int -> Bool
prime x = divisors x == [1,x]
```

Would this function determine *all* divisors of `x` and then compare that list to `[1,x]`? No, that would be too much work! At the call `prime 30` the following happens. To begin, the first divisor of `30` is determined: `1`. This value is compared with the first element of the list `[1,30]`. Regarding the first element the lists are equal. Then the second divisor of `30` is determined: `2`. This number is compared with the second value of `[1,30]`: the second elements of the lists are not equal. The operator `==` `knows' that two lists can never be equal again as soon as two different elements are encountered. Therefore `False` can be returned immediately. The other divisors of `30` are never computed!

The lazy behavior of the operator `==` is caused by its definition. The recursive line from the definition in subsection 3.1.2 reads:

```
(==) [x:xs] [y:ys] = x==y && xs==ys
```

If `x==y` delivers the value `False`, there is no need to compute `xs==ys`: the final result will always be `False`. This lazy behavior of the operator `&&` is clear from its definition:

```
(&&) False x = False
(&&) True  x = x
```

If the left argument is `False`, the value of the right argument is not needed in the computation of the result.

Functions that need all elements of a list, cannot be used on infinite lists. Examples of such functions are `sum` and `length`.

At the call `sum (from 1)` or `length (from 1)` even lazy evaluation doesn't help to compute the answer in finite time. In that case the computer will go into trance and will never deliver a final answer (unless the result of the computation isn't used anywhere, for then the computation is of course never performed…

A function argument is called *strict* when its value is needed to determine the result of the function in every possible application of that function. For instance the operator `+` is strict in both arguments, both numbers are needed to compute their sum. The operator `&&` is only strict in its first argument, when this argument in `False` the result of the function is `False` whatever the value of the second argument is. In CLEAN it is possible to indicate strictness of arguments by adding the annotation `!` to the argument in the type definition. The CLEAN system evaluates arguments that are indicated to be strict eagerly. This implies that their value is computed before the function is evaluated. In general it is not needed to put strictness annotations in the type definition. The compiler will be able to derive most strictness information automatically.

3.2.3 Functions generating infinite lists

In the standard module `StdEnum` some functions are defined that result in infinite lists. An infinite list which only contains repetitions of the same element can be generated using the function `repeat`:

```
repeat :: a -> [a]
repeat x = [x : repeat x]
```

The call `repeat 't'` returns the infinite list `['t','t','t','t',...`

An infinite list generated by `repeat` can be used as an intermediate result by a function that does have a finite result. For example, the function `repeatn` makes a finite number of copies of an element:

```
repeatn :: Int a -> [a]
repeatn n x = take n (repeat x)
```

Thanks to lazy evaluation `repeatn` can use the infinite result of `repeat`. The functions `repeat` and `repeatn` are defined in the standard library.

The most flexible function is again a higher-order function, which is a function with a function as an argument. The function `iterate` has a function and a starting element as arguments. The result is an infinite list in which every element is obtained by applying the function to the previous element. For example:

```
iterate (+1) 3      is  [3,4,5,6,7,8,…
iterate (*2) 1      is  [1,2,4,8,16,32,…
iterate (/10) 5678  is  [5678,567,56,5,0,0,…
```

The definition of `iterate`, which is in the standard environment, reads as follows:

```
iterate :: (a->a) a -> [a]
iterate f x = [x : iterate f (f x)]
```

This function resembles the function `until` defined in subsection 2.3.2. The function `until` also has a function and a starting element as arguments. The difference is that `until` stops as soon as the value satisfies a certain condition (which is also an argument). Furthermore, `until` only delivers the last value (which satisfies the given condition), while `iterate` stores all intermediate results in a list. It has to, because there is no last element of an infinite list…

3.2.4 Displaying a number as a list of characters

A function that can convert values of a certain type into a list of characters is very handy. Suppose e.g. that the function `intChars` is able to convert a positive number into a list of characters that contains the digits of that number. For example: `intChars 3210` gives the list `['3210']`. With such a function you can combine the result of a computation with a list of characters, for example as in `intChars (3*14)++[' lines']`.

The function `intChars` can be constructed by combining a number of functions that are applied one after another. Firstly, the number should be repeatedly divided by `10` using `iterate`.

The infinite tail of zeroes is not interesting and can be chopped off by `takeWhile`. Now the desired digits can be found as the last digits of the numbers in the list; the last digit of a number is equal to the remainder after division by `10`. The digits are still in the wrong order, but that can be resolved by `reverse`. Finally the digits (of type `Int`) must be converted to the corresponding digit characters (of type `Char`). For this purpose we have to define the function `digitChar`:

```
digitChar :: Int -> Char
digitChar n
    | 0 <= n && n <= 9  = toChar (n + toInt '0')
```

An example clarifies this all:

```
           3210
             ↓  iterate (\x -> x / 10)
     [3210,321,32,3,0,0,…
             ↓  takeWhile (\y -> y <> 0)
       [3210,321,32,3]
             ↓  map (\z -> z rem 10)
          [3,2,1,0]
             ↓  reverse
          [0,1,2,3]
             ↓  map digitChar
      ['0','1','2','3']
```


The function `intChars` can now be simply written as the composition of these five steps. Note that the functions are written down in reversed order, because the function composition operator (`o`) means `after':

```
intChars :: (Int -> [Char])
intChars =  map digitChar
            o reverse
            o map (\z -> z rem 10)
            o takeWhile (\y -> y <> 0)
            o iterate (\x -> x / 10)
```

Functional programming can be really programming with functions!

Of course it is also possible to write a recursive function that does the same job. Actually, the function shown here works also for negative numbers and zero.

```
intToChars :: Int -> [Char]
intToChars 0 = ['0']
intToChars n | n<0  = ['-':intToChars (~n)]
             | n<10 = [digitChar n]
                    = intToChars (n/10) ++ [digitChar (n rem 10)]
```

3.2.5 The list of all prime numbers

In subsection 2.4.1 `prime` was defined that determines whether a number is prime With that the (infinite) list of all prime numbers can be generated by

```
filter prime [2..]
```

The `prime` function searches for the divisors of a number. If such a divisor is large, it takes long before the function decides a number is not a prime.

By making clever use of `iterate` a much faster algorithm is possible. This method also starts off with the infinite list `[2..]`:

```
[2,3,4,5,6,7,8,9,10,11,…
```

The first number, `2`, can be stored in the list of primes. Then `2` and all its multiples are crossed out. What remains is:

```
[3,5,7,9,11,13,15,17,19,21,…
```

The first number, `3`, is a prime number. This number and its multiples are deleted from the list:

```
[5,7,11,13,17,19,23,25,29,31,…
```

The same process is repeated, but now with `5`:

```
[7,11,13,17,19,23,29,31,37,41,…
```

And you go on and on. The function `cross out multiples of first element' is always applied to the previous result. This is an application of `iterate` using `[2..]` as the starting value:

```
iterate crossout [2..]
where
    crossout [x:xs] = filter (not o multiple x) xs
    multiple x y    = divisible y x
```

The number `y` is a multiple of `x` if `y` is divisible by `x`. The function `divisible` was defined in section 2.4.1 as: `divisible t n = t rem n == 0`. As the starting value is a infinite list. the result of this is an *infinite list of infinite lists*. That super list looks like this:

```
[[2,3,4,5,6,7,8,9,10,11,12,13,14,…
,[3,5,7,9,11,13,15,17,19,21,23,25,27,…
,[5,7,11,13,17,19,23,25,29,31,35,37,41,…
,[7,11,13,17,19,23,29,31,37,41,43,47,49,…
,[11,13,17,19,23,29,31,37,41,43,47,51,53,…
,…
```

You can never see this thing as a whole; if you try to evaluate it, you will only see the beginning of the first list. But you need the complete list to be visible: the desired prime numbers are the first elements of the lists. So the prime numbers can be determined by taking the `head` of each list:

```
primenums :: [Int]
primenums = map head (iterate crossout [2..])
where
    crossout [x:xs] = filter (not o (multiple x)) xs
```

Thanks to lazy evaluation only that part of each list is calculated that is needed for the desired part of the answer. If you want to know the next prime, more elements of every list are calculated as far as necessary.

Often it is hard (as in this example) to imagine what is computed at what moment. But there is no need: while programming you can just pretend infinite lists really exist; the evaluation order is automatically optimized by lazy evaluation.

This algorithm to compute prime numbers is called the sieve of Eratosthenes. Eratosthenes was a greek mathematician born in Cyrene who lived 276-196 BC. His algorithm can be expressed slightly more elegant using list comprehensions:

```
primes :: [Int]
primes = sieve [2..]

sieve :: [Int] -> [Int]
sieve [prime:rest] = [prime: sieve [i \\ i <- rest | i mod prime <> 0]]
```

3.3 Tuples

All elements in a list have to be of the same type, e.g. it is not possible to store both an integer and a string in one and the same list. Sometimes one needs to group information of different types together. A tuple can be used for this.

A *tuple* consists of a fixed number of values that are grouped together (see figure 3.8). The values may be of different types.

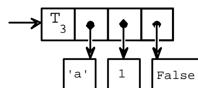


Figure 3.8: Pictorial representation of the tuple `('a',1,False)` of type `(Char,Int,Bool)`.

Tuples are denoted by round parentheses around the elements. Examples of tuples are:

- `(1,'a')` a tuple with as elements the integer `1` and the character `'a'`;
- `("foo",True,2)` a tuple with three elements: the string `foo`, the Boolean `True` and the number `2`;
- `([1,2],sqrt)` a tuple with two elements: the list of integers `[1,2]` and the function from real to real `sqrt`;
- `(1,(2,3))` a tuple with two elements: the number `1` and a tuple containing the numbers `2` and `3`.

The tuple of each combination types is a distinct type. The order in which the components appear is important, too. The type of tuples is written by enumerating the types of the elements between parentheses. The four expressions above can be types as follows:

```
(1,'a')        :: (Int,Char)
("foo",True,2) :: (String,Bool,Int)
([1,2],sqrt)   :: ([Int],Real->Real)
(1,(2,3))      :: (Int,(Int,Int))
```

A tuple with two elements is called a 2-tuple or a *pair*. Tuples with three elements are called 3-tuples etc. There are no 1-tuples: the expression `(7)` is just an integer; for it is allowed to put parentheses around every expression.

The standard library provides some functions that operate on tuples. These are good examples of how to define functions on tuples: by pattern matching.

```
fst :: (a,b) -> a
fst (x,y) = x
```



```
snd :: (a,b) -> b
snd (x,y) = y
```

These functions are all polymorphic, but of course it is possible to write your own functions that only work for a specific type of tuple:

```
f :: (Int,Char) -> Int
f (n,c) = n + toInt c
```

Tuples come in handy for functions with multiple results. Functions can have several arguments. However, functions have only a single result. Functions with more than one result are only possible by `wrapping' these results up in some structure, e.g. a tuple. Then the tuple as a whole is the only result.

An example of a function with two results is `splitAt` that is defined in the standard environment. This function delivers the results of `take` and `drop` at the same time. Therefore the function could be defined as follows:

```
splitAt :: Int [a] -> ([a],[a])
splitAt n xs = (take n xs,drop n xs)
```

However, the work of both functions can be done simultaneously. That is why in the standard library `splitAt` is defined as:

```
splitAt :: Int [a] -> ([a],[a])
splitAt 0 xs     = ([] ,xs)
splitAt n []     = ([] ,[])
splitAt n [x:xs] = ([x:ys],zs)
where
    (ys,zs) = splitAt (n-1) xs
```

The result of the recursive call of `splitAt` can be inspected by writing down a `right-hand side pattern match', which is called a *selector*:

```
splitAt n [x:xs]    = ([x:ys],zs)
where
    (ys,zs) = splitAt (n-1) xs
```

The tuple elements thus obtained can be used in other expressions, in this case to define the result of the function `splitAt`.

The call `splitAt 2 ['clean']` gives the 2-tuple `(['cl'],['ean'])`. In the definition (at the recursive call) you can see how you can use such a result tuple: by exposing it to a pattern match (here `(ys,zs)`).

Another example is a function that calculates the average of a list, say a list of reals. In this case one can use the predefined functions `sum` and `length`: `average = sum / toReal length`. Again this has the disadvantage that one walks through the list twice. It is much more efficient to use one function `sumlength` which just walks through the list once to calculate both the sum of the elements (of type `Real`) as well as the total number of elements in the list (of type `Int`) at the same time. The function `sumlength` therefore returns one tuple with both results stored in it:

```
average :: [Real] -> Real
average list = mysum / toReal mylength
where
    (mysum,mylength) = sumlength list 0.0 0

    sumlength :: [Real] Real Int -> (Real,Int)
    sumlength [x:xs] sum length = sumlength xs (sum+x) (length+1)
    sumlength []     sum length = (sum,length)
```

Using type classes this function can be made slightly more general:

```
average :: [t] -> t | +, zero, one t
average list = mysum / mylength
where
    (mysum,mylength) = sumlength list zero zero
```

```
sumlength :: [t] t t -> (t,t)  | +, one t
sumlength [x:xs] sum length = sumlength xs (sum+x) (length+one)
sumlength []     sum length = (sum,length)
```

3.3.1 Tuples and lists

Tuples can of course appear as elements of a list. A list of two-tuples can be used e.g. for searching (dictionaries, telephone directories etc.). The search function can be easily written using patterns; for the list a `non-empty list with as a first element a 2-tuple' is used.

```
search :: [(a,b)] a -> b | == a
search [(x,y):ts] s
    | x == s    = y
    | otherwise = search ts s
```

The function is polymorphic, so that it works on lists of 2-tuples of arbitrary type. However, the elements should be comparable, which is why the functions `search` is overloaded since `==` is overloaded as well. The element to be searched is intentionally defined as the second argument, so that the function `search` can easily be partially parameterized with a specific search list, for example:

```
telephoneNr = search telephoneDirectory
translation = search dictionary
```

where `telephoneDirectory` and `dictionary` can be separately defined as constants.

Another function in which 2-tuples play a role is the `zip` function. This function is defined in the standard environment. It has two lists as arguments that are chained together element-wise in the result. For example: `zip [1,2,3] ['abc']` results in the list `[(1,'a'),(2,'b'),(3,'c')]`. If the two lists are not of equal length, the shortest determines the size of the result. The definition is rather straightforward:

```
zip :: [a] [b] -> [(a,b)]
zip []      ys      = []
zip xs      []      = []
zip [x:xs]  [y:ys]  = [(x,y) : zip xs ys]
```

The function is polymorphic and can thus be used on lists with elements of arbitrary type. The name *zip* reflects the fact that the lists are so to speak `zipped'. The functions `zip` can more compactly be defined using a list comprehension:

```
zip :: [a] [b] -> [(a,b)]
zip as bs = [(a,b) \\ a <- as & b <- bs]
```

If two values of the same type are to be grouped, you can use a list. Sometimes a tuple is more appropriate. A point in the plane, for example, is described by two `Real` numbers. Such a point can be represented by a list or a 2-tuple. In both cases it possible to define functions working on points, e.g. `distance to the origin'. The function `distanceL` is the list version, `distanceT` the tuple version:

```
distanceL :: [Real] -> Real
distanceL [x,y] = sqrt (x*x+y*y)

distanceT :: (Real,Real) -> Real
distanceT (x,y) = sqrt (x*x+y*y)
```

As long as the function is called correctly, there is no difference. But it could happen that due to a typing error or a logical error the function is called with three coordinates. In the case of `distanceT` this mistake is detected during the analysis of the program: a tuple with three numbers is of another type than a tuple with two numbers. However, using `distanceL` the program is well-typed. Only when the function is really used, it becomes evident that `distanceL` is undefined for a list of three elements. Here the use of tuples instead of lists helps to detect errors.

3.4 Records

Often one would like to group information of possibly different type on a more structural way simply because the information belongs together. Information in a person database

may consist for example of a string (name), a Boolean (male), three integers (date of birth) and a Boolean again (CLEAN user). If one wants to use such a kind of record, one first has to declare its type in a *type definition*, e.g.:

```
:: Person = {   name        :: String
            ,   birthdate   :: (Int,Int,Int)
            ,   cleanuser   :: Bool
            }
```

Type definitions in CLEAN always start with a `::` at the beginning of a line. With this particular type definition a new type is declared, called a *record type*. A record is a kind of tuple. The record elements can be of different type, just like in tuples. However, in a record type, a name (the *field name*) is used to refer to a record element (see also figure 3.9). This fieldname must be used to identify the corresponding record element.

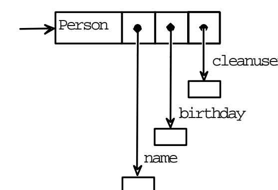


Figure 3.9: Pictorial representation of a record `Person`.

Once the type is defined, a record of that type can be created, e.g. in the following way:

```
SomePerson :: Person
SomePerson =    {   name        = "Rinus"
                ,   birthdate   = (10,26,1952)
                ,   cleanuser   = True
                }
```

Each of the record elements (identified by `fieldname =`) must get a value of the type as indicated in the record type declaration. The order in which the fields are specified is irrelevant, but all fields of the record have to get a value.

An important difference between a tuple and a record is that a tuple field always has to be selected by its position, as in:

```
fst :: (a,b) -> a
fst (x,y) = x
```

while a record field is selected by field name. For instance, one can define:

```
:: Pair a b =   {   first   :: a
                ,   second  :: b
                }
```

This is a polymorphic record named `Pair` with two record elements. One element is named `first` and is of type `a`, the other is named `second` and is of type `b`. A function which selects the first element can be defined as:

```
fstR :: (Pair a b) -> a
fstR {first} = first
```

The example illustrates how the pattern matching mechanism can be used to select one or more record elements. The nice thing about this feature is that one only needs to name the fields one is interested in.

```
IsCleanUser :: Person -> String
IsCleanUser {cleanuser = True}  = "Yes"
IsCleanUser _                   = "No"
```

There is a special selection operator, '.', to select an element from a record. It expects a expression yielding a record and a field name to select an expression of that record. For instance:

```
GetPersonName :: Person -> String
GetPersonName person = person.name
```

Finally, there is a special language construct which enables you to create a new record given another existing record of the same type. Consider:

```
ChangePersonName :: Person String -> Person
ChangePersonName person newname = {person & name = newname}
```

The new record is created by making a copy of the old record `person`. The contents of the fields of the new record will be exactly the same as the contents of the fields of the old record, with exception of the field `name` which will contain the new name `newname`. The operator `&` is called the *functional update operator*. Do not confuse it with a destructive update (assignment) as is present in imperative languages (like in C, C++, PASCAL). Nothing is changed, a complete new record is made which will be identical to the old one with exception of the specified new field values. The old record itself remains unchanged.

The CLEAN system determines the type of a record from the field names used. When there are several records with the used field names determining the type fails. The user should explicitly specify the type of the record inside the record. It is not sufficient to that the type of the record can be deduced from the type of the function. It is always allowed to indicate the type of a record explicitly.

```
AnotherPerson :: Person
AnotherPerson = {   Person
                |   name        = "Pieter"
                ,   birthdate   = (7,3,1957)
                ,   cleanuser   = True
                }
```

The records in CLEAN make it possible to define functions which are less sensible for changes. For instance, assume that one has defined:

```
:: Point = {   x :: Real
           ,   y :: Real
           }
MovePoint :: Point (Real,Real) -> Point
MovePoint p (dx,dy) = {p & x = p.x + dx, y = p.y + dy}
```

Now, lets assume that in a later state of the development of the program one would like to add more information to the record `Point`, say

```
:: Point = {   x :: Real
           ,   y :: Real
           ,   c :: Color
           }
```

where `Color` is some other type. This change in the definition of the record `Point` has no consequences for the definition of `MovePoint`. If `Point` would be a tuple, one would have to change the definition of `MovePoint` as well, because `Point` would change from a 2-tuple to a 3-tuple which has consequences for the pattern match as well as for the construction of a new point.

So, for clarity and ease of programming we strongly recommend the use of records instead of tuples. Only use tuples for functions that return multiple results.

3.4.1 Rational numbers

An application in which records can be used is an implementation of the *Rational numbers*. The rational numbers form the mathematical set $\mathcal{Q}$, numbers that can be written as a *ratio*. It is not possible to use `Real` numbers for calculations with ratios: the calculations must be *exact*, such that the outcome of $\frac{1}{2} + \frac{1}{3}$ is the fraction $\frac{5}{6}$ and not the `Real 0.833333`.

Fractions can be represented by a numerator and a denominator, which are both integer numbers. So the following type definition is obvious:

```
:: Q = { num :: Int
       , den :: Int
       }
```

Next a number of frequently used fractions get a special name:

```
QZero    = {num = 0, den = 1}
QOne     = {num = 1, den = 1}
QTwo     = {num = 2, den = 1}
QHalf    = {num = 1, den = 2}
QThird   = {num = 1, den = 3}
Qquarter = {num = 1, den = 4}
```

We want to write some functions that perform the most important arithmetical operations on fractions: multiplication, division, addition and subtraction. Instead of introducing new names for these functions we use the overloading mechanism (introduced in section 1.5.5 and explained in more detail in section 4.1) in order to use the obvious operator symbols: `*`, `/`, `+`, `-`.

The problem is that one value can be represented by different fractions. For example, a half can be represented by `{num=1,den=2}`, but also by `{num=2,den=4}` and `{num=17,den=34}`. Therefore the outcome of two times a quarter might `differ' from `half'. To solve this problem a function `simplify` is needed that can simplify a fraction. By applying this function after every operation on fractions, fractions will always be represented in the same way. The result of two times a quarter can then be safely compared to a half: the result is `True`.

The function `simplify` divides the numerator and the denominator by their *greatest common divisor*. The greatest common divisor (`gcd`) of two numbers is the greatest number by which both numbers are divisible. For negative numbers we want a negative nominator. When the denominator is zero the fraction is not defined. The definition of `simplify` reads as follows:

```
simplify :: Q -> Q
simplify {num=n,den=d}
    | d == 0    = abort "denominator of Q is 0!"
    | d < 0     = {num = ~n / g, den = ~d / g}
    | otherwise = {num =  n / g, den =  d / g}
where
    g = gcd n d
```

A simple definition of `gcd x y` determines the greatest divisor of `x` that also divides `y` using `divisors` and `divisible` from subsection 2.4.1.

```
gcd :: Int Int -> Int
gcd x y = last (filter (divisible (abs y)) (divisors (abs x)))
```

(In the standard library a more efficient version of `gcd` is defined:

```
gcd :: Int Int -> Int
gcd x y = gcd' (abs x) (abs y)
where
    gcd' x 0 = x
    gcd' x y = gcd' y (x mod y)
```

This algorithm is based on the fact that if `x` and `y` are divisible by `d` then so is `x mod y` (=`x-(x/y)*y`)).

Using `simplify` we are now in the position to define the mathematical operations. Due to the number of places where a record of type `Q` must be created and simplified it is convenient to introduce an additional function `mkQ`.

```
mkQ :: x x -> Q | toInt x
mkQ n d = simplify {num = toInt n, den = toInt d}
```

To multiply two fractions, the numerators and denominators must be multiplied ($\frac{2}{3} * \frac{5}{4} = \frac{10}{12}$). Then the result can be simplified (to $\frac{5}{6}$):

```
instance * Q
where (*) q1 q2 = mkQ (q1.num*q2.num) (q1.den*q2.den)
```

Dividing by a number is the same as multiplying by the inverse:

```
instance / Q
where (/) q1 q2 = mkQ (q1.num*q2.den) (q1.den*q2.num)
```

Before you can add two fractions, their denominators must be made the same first. ($\frac{1}{4}+\frac{3}{10} = \frac{10}{40}+\frac{12}{40} = 22$). The product of the denominator can serve as the common denominator. Then the numerators must be multiplied by the denominator of the other fraction, after which they can be added. Finally the result must be simplified (to $\frac{11}{20}$).

```
instance + Q
where (+) q1 q2 = mkQ (q1.num * q2.den + q1.den * q2.num) (q1.den * q2.den)

instance - Q
where (-) q1 q2 = mkQ (q1.num * q2.den - q1.den * q2.num) (q1.den * q2.den)
```

The result of computations with fractions is displayed as a record. If this is not nice enough, you can define a function `toString`:

```
instance toString Q
where
toString q
    | sq.den==1 = toString sq.num
    | otherwise = toString sq.num +++ "/" +++ toString sq.den
where
    sq = simplify q
```

3.5 Arrays

An *array* is a predefined data structure that is used mainly for reasons of efficiency. With a list an array has in common that *all* its elements have to be of the *same* type. With a tuple/record-like data structure an array has in common that it contains a fixed number of elements. The elements of an array are numbered. This number, called the *index*, is used to identify an array element, like field names are used to identify record elements. An array index is an integer number between 0 and the number of array elements - 1.

Arrays are notated using curly brackets. For instance,

```
MyArray :: {Int}
MyArray = {1,3,5,7,9}
```

is an array of integers (see figure 3.10). It's type is indicated by `{Int}`, to be read as 'array of `Int`'.

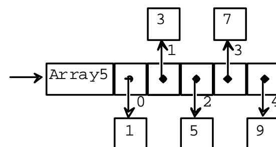


Figure 3.10: Pictorial representation of an array with 5 elements `{1,3,5,7,9}`.

Compare this with a list of integers:

```
MyList :: [Int]
MyList = [1,3,5,7,9]
```

One can use the operator `!!` to select the ith element from a list (see subsection 3.1.2): For instance

```
MyList !! 2
```

will yield the value `5`. To select the ith element from array `a` one writes `a.[i]`. So,

```
MyArray.[2]
```


will also yield the value `5`. Besides the small difference in notation there is big difference in the efficiency between an array selection and a list selection. To select the ith element from a list, one has to recursively walk through the spine of the list until the ith list element is found (see the definition of `!!` in subsection 3.1.2). This takes *i* steps. The ith element of an array can be found directly in *one* step because all the references to the elements are stored in the array box itself (see figure 3.10). Selection can therefore be done very efficiently regardless which element is selected in *constant time.*

The big disadvantage of selection is that it is possible to use an index out of the index range (i.e. index < 0 or index □ *n*, where *n* is the number of list/array elements). Such an index error generally cannot be detected at compile-time, such that this will give rise to a run-time error. So, selection both on arrays as on lists is a very dangerous operation because it is a partial function and one easily makes mistakes in the calculation of an index. Selection is the main operation on arrays. The construction of lists is such that selection can generally be avoided. Instead one can without danger recursively traverse the spine of a list until the desired element is found. Hitting on the empty list a special action can be taken. Lists can furthermore easily be extended while an array is fixed sized. Lists are therefore more flexible and less error prone. Unless ultimate efficiency is demanded, the use of lists above arrays is recommended.

But, arrays can be very useful if time and space consumption is becoming very critical, e.g. when one uses a huge and fixed number of elements that are frequently selected and updated in a more or less random order.

3.5.1 Array comprehensions

To increase readability, CLEAN offers array comprehensions in the same spirit as list comprehension's. For instance, if `ArrayA` is an array and `ListA` a list, then

```
NewArray = {elem \\ elem <- ListA}
```

will create an new array with the same (amount of) elements as in `ListA`. Conversion the other way around is easy as well:

```
NewList = [elem \\ elem <-: ArrayA]
```

Notice that the `<-` symbol is used to draw elements from a list while the `<-:` symbol is used to draw elements from an array.

Also a map-like function on an array can be defined in a straightforward manner:

```
MapArray f a = {f e \\ e <-: a}
```

3.5.2 Lazy, strict and unboxed arrays

To obtain optimal efficiency in time and space, several kinds of arrays can be defined in CLEAN. By default an array is a *lazy array* (say, of type `{Int}`), i.e. an array consists of a contiguous block of memory containing references to the array elements (see figure 3.10). The same representation is chosen if a *strict array* is used (prefix the element type with a `!`, e.g. `{!Int}`). Strict arrays have to property that its elements will always be evaluated whenever the array is used. For elements of basic type only (`Int`, `Real`, `Char`, `Bool`) an *unboxed array* can be defined (prefix the element type with a `#`, e.g. `{#Int}`). So, by explicitly specifying the type of the array upon creation one can indicate which representation one wants: the default one (lazy), or the strict or the unboxed version of the array.

Unboxed arrays are more efficient than lazy or strict arrays because the array elements themselves are stored in the array box. No references to other boxes have to be regarded. For instance, the following array

```
MyUnboxedArray :: {#Int}
MyUnboxedArray = {1,3,5,7,9}
```

is an unboxed array (due to its type specification) of integers. Compare its representation in figure 3.11 with the default representation given in figure 3.9.

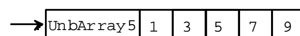


Figure 3.11: Pictorial representation of an unboxed array with 5 elements `{1,3,5,7,9}`.

Lazy, strict and unboxed arrays are regarded by the CLEAN compiler as objects of *different* types. This means for instance that a function that is expecting an *unboxed* array of `Char` cannot be applied to a *lazy* array of `Char` or the other way around. However, most predefined operations on arrays (like array selection) are overloaded such that they can be used on lazy, strict as well as on unboxed arrays.

A *string* is equivalent with an unboxed array of character `{#Char}`. A type synonym is defined in module `StdString`. For programming convenience, there is special syntax to denote strings. For instance, the string denotation

```
"abc"
```

is equivalent with the unboxed array `{'a','b','c'}`. Compare this with `['abc']` which is equivalent with the list of characters `['a','b','c']`.

3.5.3 Array updates

It is also possible to update an array, using the same notation as for records (see subsection 3.4). In principle a new array is constructed out of existing one. One has to indicate for which index the new array differs from the old one. Assume that `Array5` is an integer array with 5 elements. Then an array with elements `{1,3,5,7,9}` can be created as follows:

```
{Array5 & [0]=1,[1]=3,[2]=5,[3]=7,[4]=9}
```

As with record updating, the order in which the array elements are specified is irrelevant. So, the following definition

```
{Array5 & [1]=3,[0]=1,[3]=7,[4]=9,[2]=5}
```

is also fine.

One can even combine array updates with array comprehension's. So the next two expressions will also yield the array `{1,3,5,7,9}` as result.

```
{Array5 & [i]=2*i+1 \\ i <- [0..4]}
{Array5 & [i]=elem \\ elem <- [1,3..9] & i <- [0..4]}
```

As said before, arrays are mainly important to achieve optimal efficiency. That is why updates of arrays are in CLEAN only defined on unique arrays, such that the update can always be done *destructively* ! This is semantically sound because the original unique array is known not to be used anymore. Uniqueness is explained in more detail in chapter 4 and 5.

3.5.4 Array patterns

Array elements can be selected in the patterns of a function. This is similar to the selection of the fields of a record. This is illustrated by the following functions.

```
CopyFirst :: Int *{#a} -> {#a} | ArrayElem a
CopyFirst j   a=:{[0]=a0} = {a & [j] = a0}

CopyElem :: Int Int *{#a} -> {#a} | ArrayElem a
CopyElem  i j a=:{[i]=ai} = {a & [j] = ai}

CopyCond :: Int Int *{#a} -> {#a} | ArrayElem, ==, zero a
CopyCond  i j a=:{[i]=ai, [j]=aj}
    | a.[0] == zero = {a & [j] = ai}
    | otherwise     = {a & [i] = aj}
```

The selection of elements specified in the pattern is done before the right hand side of the rule is constructed. This explains why the given examples are allowed. When the `CopyElem` function is written as

```
CopyElem2 :: Int Int *{#a} -> {#a} | ArrayElem a
CopyElem2  i j a = {a & [j] = a.[i]}
```


it will be rejected by the CLEAN system. An array can only be updated when it is unique. The reference to the old array, `a.[i]`, in the array update spoils the uniqueness properties of the array. Without selection in the pattern this function should be written with a `let!` construct:

```
CopyElem3 :: Int Int *{#a} -> {#a} | ArrayElem a
CopyElem3  i j a = let! ai = a.[i] in {a & [j] = ai}
```

The graphs specified in the strict let part are evaluated before the expression after the keyword `in` is evaluated. This implies that the element `ai` is selected from the array before the array is updated.

3.6 Algebraic datatypes

We have seen several `built-in' ways to structure information: lists, tuples, records and arrays. In some cases these data structures are not appropriate to represent the information. Therefore it has been made possible to define a new, so-called *algebraic datatype* yourself.

An algebraic datatype is a type that defines the way elements can be constructed. In fact, all built-in types are predefined algebraic datatypes. A `list', for instance, is an algebraic type. Lists can be constructed in two ways:

- as the empty list;
- by prepending an element to an existing list using the `[x:xs]` notation.

In the case distinction in definitions of functions that operate on lists these two way construction methods reappear, for example:

```
length :: [t] -> Int
length []       = 0
length [x:xs]   = 1 + length xs
```

By defining the function for both cases, the function is totally defined.

If you want to use a new data structure in CLEAN, you have to define its type in an algebraic datatype definition. For instance:

```
:: Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

is an algebraic datatype definition introducing a new type, named `Day`. It moreover introduces seven new constants that have this type `Day` (`Mon,Tue,Wed,Thu,Fri,Sat,Sun`). These constants are called *data constructors*. Once defined in an algebraic type, the data constructors can be used in function definitions. They can appear in expressions to construct new objects of the specified algebraic type. They can appear in patterns, for instance to discriminate between objects of the same algebraic type.

```
IsWeekend Sat   = True
IsWeekend Sun   = True
IsWeekend _     = False
```

A data constructor can only belong to *one* algebraic datatype definition. As a consequence, the CLEAN system can directly tell the type of each data constructor. So, `Mon :: Day`, `Tue :: Day`, and so on. And therefore, the type of `IsWeekend` is:

```
IsWeekend :: Day -> Bool
```

The algebraic type `Day` is called an *enumeration type*: the type definition just enumerates all possible values. In chapter 2 we used integers to represent the days of the week. This has both advantages and disadvantages:

- An advantage of the algebraic datatype is that well chosen names avoids confusion. When you use integers you have to decide and remember whether the week starts on Sunday or on Monday. Moreover, there is the question whether the first day of the week has number `0` or number `1`.
- An other advantage of the algebraic type is that the type checker is able to verify type correctness. A function that expects or delivers an element of type `Day` will always use one of the listed values. When you use integers, the compiler is only able to verify that

an integer is used at each spot a `Day` is expected. It is still possible to use the value `42` where a `Day` is expected. In addition using algebraic can prevent confusion between enumerated types. When we use this definition of `Day` and a similar definition of `Month` it is not possible to interchange the arguments of `daynumber` by accident without making a type error.

- An advantage of using integers to represent days is that the definition of operations like addition, comparison and equality can be reused. In chapter 2 we saw how pleasant this is. For an algebraic type all the needed operations have to be defined.

The balance between advantages and disadvantages for the application at hand determines whether it is better to use an algebraic enumeration type or to use integers as encoding (Booleans can be used for types with two values). Unless there are strong reasons to use something else we recommend generally to use an algebraic datatype. In the next section we show that it is possible to equip the constructors with arguments and to define recursive types. This is far beyond the possibilities of an encoding of types in integers.

As usual, it is possible to combine algebraic types with other types in various ways. For example:

```
:: Employee = { name       :: String
              , gender     :: Gender
              , birthdate  :: Date
              , cleanuser  :: Bool
              }
:: Date     = { day     :: Int
              , month   :: Int
              , year    :: Int
              }
:: Gender   = Male
            | Female
```

These types can be used in functions like:

```
WeekDayOfBirth :: Employee -> Day
WeekDayOfBirth {birthdate={day,month,year}}
 = [Sun, Mon, Tue, Wed, Thu, Fri, Sat] !! daynumber day month year
```

Where we use the function `daynumber` as defined in chapter 2. An example of a function generating a value of the type `Employee` is:

```
AnEmployee :: Employee
AnEmployee =    { name      = "Pieter"
                , gender    = Male
                , birthdate = {year = 1957, month = 7, day = 3}
                , cleanuser = True
                }
```

3.6.1 Tree definitions

All data structures can be defined using an algebraic datatype definition. In this way one can define data structures with certain properties. For instance, a list is a very flexible data structure. But, it also has a disadvantage. It is a linear structure; as more elements are prepended, the chain (spine) becomes longer (see figure 3.3). Sometimes such a linear structure is not appropriate and a *tree structure* would be better. A (binary) tree can be defined as:

```
:: Tree a   = Node a (Tree a) (Tree a)
            | Leaf
```

You can pronounce this definition as follows. `A tree with elements of type `a` (shortly, a tree of `a`) can be built in two ways: (1) by applying the constant `Node` to three arguments (one of type `a` and two of type tree of `a`), or (2) by using the constant `Leaf`.' `Node` and `Leaf` are two new constants. `Node` is a data constructor of arity three (`Node :: a (Tree a) (Tree a) -> (Tree a)`), `Leaf` is a data constructor of arity zero (`Leaf :: Tree a`). The algebraic type definition also states that the new type `Tree` is polymorphic.

You can construct trees by using the data constructors in an expression (this tree is also drawn in the figure 3.12).


```
Node 4  (Node 2 (Node 1 Leaf Leaf)
                (Node 3 Leaf Leaf)
        )
        (Node 6 (Node 5 Leaf Leaf)
                (Node 7 Leaf Leaf)
        )
```

You don't have to distribute it nicely over the lines; the following is also allowed:

```
Node 4(Node 2(Node 1 Leaf Leaf)(Node 3 Leaf Leaf))
      (Node 6(Node 5 Leaf Leaf)(Node 7 Leaf Leaf))
```

However, the layout of the first expression is clearer.

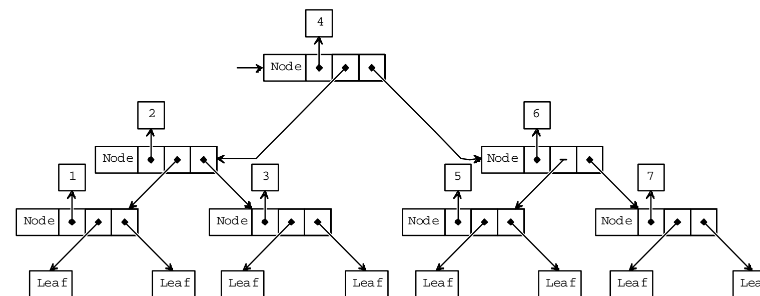


Figure 3.12: Pictorial representation of a tree.

Not every instance of the type tree needs to be as symmetrical as the tree show above. This is illustrated by the following example.

```
Node 7  (Node 3 (Node 5 Leaf
                        Leaf
                )
                Leaf
        )
        Leaf
```

An algebraic datatype definition can be seen as the specification of a grammar in which is specified what legal data objects are of a specific type. If you don't construct a data structure as specified in the algebraic datatype definition, a type error is generated at compile time.

Functions on a tree can be defined by making a pattern distinction for every data constructor. The next function, for example, computes the number of `Node` constructions in a tree:

```
sizeT :: Tree a -> Int
sizeT Leaf          = 0
sizeT (Node x p q)  = 1 + sizeT p + sizeT q
```

Compare this function to the function `length` on lists.

There are many more types of trees possible. A few examples:

- Trees in which the information is stored in the leaves (instead of in the nodes as in `Tree`):
```
:: Tree2 a  = Node2 (Tree2 a) (Tree2 a)
            | Leaf2 a
```
  Note that even the minimal tree of this type contains one information item.
- Trees in which information of type `a` is stored in the nodes and information of type `b` in the leaves:
```
:: Tree3 a b    = Node3 a (Tree3 a b) (Tree3 a b)
                | Leaf3 b
```
- Trees that split in three branches at every node instead of two:

```
:: Tree4 a  = Node4 a (Tree4 a) (Tree4 a) (Tree4 a)
            | Leaf4
```

- Trees in which the number of branches in a node is variable:
  ```
  :: Tree5 a  = Node5 a [Tree5 a]
  ```
  In this tree you don't need a separate constructor for a `leaf', because you can use a node with no outward branches.
- Trees in which every node only has one outward branch:
  ```
  :: Tree6 a  = Node6 a (Tree6 a) | Leaf6
  ```
  A `tree' of this type is essentially a list: it has a linear structure.
- Trees with different kinds of nodes:
  ```
  :: Tree7 a b    = Node7a Int a (Tree7 a b) (Tree7 a b)
                  | Node7b Char (Tree7 a b)
                  | Leaf7a b
                  | Leaf7b Int
  ```

3.6.2 Search trees

A good example of a situation in which trees perform better than lists is searching (the presence of) an element in a large collection. You can use a *search tree* for this purpose.

In subsection 3.1.2 a function `isMember` was defined that delivered `True` if an element was present in a list. Whether this function is defined using the standard functions `map` and `or`

```
isMember :: a [a] -> Bool | Eq a
isMember e xs = or (map ((==)e) xs)
```

or directly with recursion

```
isMember e []      = False
isMember e [x:xs]  = x==e || isMember e xs
```

doesn't affect the efficiency that much. In both cases all elements of the list are inspected one by one. As soon as the element is found, the function immediately results in `True` (thanks to lazy evaluation), but if the element is not there the function has to examine all elements to reach that conclusion.

It is somewhat more convenient if the function can assume the list is sorted, i.e. the elements are in increasing order. The search can then be stopped when it has `passed' the wanted element. As a consequence the elements must not only be comparable (class `Eq`), but also orderable (class `Ord`):

```
elem' :: a [a] -> Bool | Eq, Ord a
elem' e []     = False
elem' e [x:xs] = e == x || (e > x && elem' e xs)
```

A much larger improvement can be achieved if the elements are not stored in a list, but in *search tree*. A search tree is a kind of `sorted tree'. It is a tree built following the definition of `Tree` from the previous paragraph:

```
:: Tree a       = Node a (Tree a) (Tree a)
                | Leaf
```

At every node an element is stored and two (smaller) trees: a `left' subtree and a `right' subtree (see figure 3.12). Furthermore, in a search tree it is required that all values in the left subtree are *smaller* or equal to the value in the node and all values in the right subtree *greater*. The values in the example tree in the figure are chosen so that it is in fact a search tree.

In a search tree the search for an element is very simple. If the value you are looking for is equal to the stored value in an node, you are done. If it is smaller you have to continue searching in the left subtree (the right subtree contains larger values). The other way around, if the value is larger you should look in the right subtree. Thus the function `elemTree` reads as follows:

```
elemTree :: a (Tree a) -> Bool | Eq, Ord a
elemTree e Leaf = False
elemTree e (Node x le ri)
    | e==x  = True
```



```
    | e<x   = elemTree e le
    | e>x   = elemTree e ri
```

If the tree is well-balanced, i.e. it doesn't show big holes, the number of elements that has to be searched roughly halves at each step. And the demanded element is found quickly: a collection of thousand elements only has to be halved ten times and a collection of a million elements twenty times. Compare that to the half million steps `isMember` costs on average on a collection of a million elements.

In general you can say the complete search of a collection of `n` elements costs `n` steps with `isMember`, but only ^{2}log n steps with `elemTree`.

Search trees are handy when a large collection has to be searched many times. Also e.g. `search` from subsection 3.3.1 can achieve enormous speed gains by using search trees.

Structure of a search tree

The form of a search tree for a certain collection can be determined `by hand'. Then the search tree can be typed in as one big expression with a lot of data constructors. However, that is an annoying task that can easily be automated.

Like the function `insert` adds an element to a sorted list (see subsection 3.1.4), the function `insertTree` adds an element to a search tree. The result will again be a search tree, i.e. the element will be inserted in the right place:

```
insertTree :: a (Tree a) -> Tree a | Ord a
insertTree e Leaf   = Node e Leaf Leaf
insertTree e (Node x le ri)
    | e<=x  = Node x (insertTree e le) ri
    | e>x   = Node x le (insertTree e ri)
```

If the element is added to a `Leaf` (an `empty' tree), a small tree is built from `e` and two empty trees. Otherwise, the tree is not empty and contains a stored value `x`. This value is used to decide whether `e` should be inserted in the left or the right subtree. When the tree will only be used to decide whether an element occurs in the tree there is no need to store duplicates. It is straight forward to change the function `insertTree` accordingly:

```
insertTree :: a (Tree a) -> Tree a | Ord, Eq a
insertTree e Leaf = Node e Leaf Leaf
insertTree e node=:(Node x le ri)
    | e<x   = Node x (insertTree e le) ri
    | e==x  = node
    | e>x   = Node x le (insertTree e ri)
```

By using the function `insertTree` repeatedly, all elements of a list can be put in a search tree:

```
listToTree :: [a] -> Tree a | Ord, Eq a
listToTree []     = Leaf
listToTree [x:xs] = insertTree x (listToTree xs)
```

The experienced functional programmer will recognise the pattern of recursion and replace it by an application of the function `foldr`:

```
listToTree :: ([a] -> Tree a) | Ord, Eq a
listToTree = foldr insertTree Leaf
```

Compare this function to `isort` in subsection 3.1.4.

A disadvantage of using `listToTree` is that the resulting search tree is not always well-balanced. This problem is not so obvious when information is added in random order. If, however, the list which is turned into a tree is already sorted, the search tree `grows cooked'. For example, when running the program

```
Start = listToTree [1..7]
```

the output will be

```
Node 7 (Node 6 (Node 5 (Node 4 (Node 3 (Node 2 (Node 1 Leaf Leaf) Leaf) Leaf) Leaf) Leaf)
Leaf) Leaf
```

Although this is a search tree (every value is between values in the left and right subtree) the structure is almost linear. Therefore logarithmic search times are not possible in this tree. A better (not `linear') tree with the same values would be:

```
Node 4  (Node 2 (Node 1 Leaf Leaf)
                (Node 3 Leaf Leaf)
        )
        (Node 6 (Node 5 Leaf Leaf)
                (Node 7 Leaf Leaf)
        )
```

3.6.3 Sorting using search trees

The functions that are developed above can be used in a new sorting algorithm: *tree sort*. For that one extra function is necessary: a function that puts the elements of a search tree in a list preserving the ordering. This function is defined as follows:

```
labels :: (Tree a) -> [a]
labels Leaf            = []
labels (Node x le ri)  = labels le ++ [x] ++ labels ri
```

The name of the function is inspired by the habit to call the value stored in a node the *label* of that node.

In contrast with `insertTree` this function performs a recursive call to the left *and* the right subtree. In this manner every element of the tree is inspected. As the value `x` is inserted in the right place, the result is a sorted list (provided that the argument is a search tree).

An arbitrary list can be sorted by transforming it into a search tree with `listToTree` and than summing up the elements in the right order with `labels`:

```
tsort :: ([a] -> [a]) | Eq, Ord a
tsort = labels o listToTree
```

In chapter 6 we will show how functions like `labels` can be implemented more efficiently using a continuation.

3.6.4 Deleting from search trees

A search tree can be used as a database. Apart from the operations enumerate, insert and build, which are already written, a function for deleting elements comes in handy. This function somewhat resembles the function `insertTree`; depending on the stored value the function is called recursively on its left or right subtree.

```
deleteTree :: a (Tree a) -> (Tree a) | Eq, Ord a
deleteTree e Leaf = Leaf
deleteTree e (Node x le ri)
     | e<x   = Node x (deleteTree e le) ri
     | e==x  = join le ri
     | e>x   = Node x le (deleteTree e ri)
```

If, however, the value is found in the tree (the case `e==x`) it can't be left out just like that without leaving a `hole'. That is why a function `join` that joins two search trees is necessary . This function takes the largest element from the left subtree as a new node. If the left subtree is empty, joining is of course no problem:

```
join :: (Tree a) (Tree a) -> (Tree a)
join Leaf b2 = b2
join b1   b2 = Node x b1' b2
where
     (x,b1') = largest b1
```

The function `largest`, apart from giving the largest element of a tree, also gives the tree that results after deleting that largest element. These two results are combined in a tuple. The largest element can be found by choosing the right subtree over and over again:

```
largest :: (Tree a) -> (a,(Tree a))
largest (Node x b1 Leaf) = (x,b1)
largest (Node x b1 b2)   = (y,Node x b1 b2')
where
     (y,b2') = largest b2
```


As the function `largest` is only called from `join` it doesn't have to be defined for a `Leaf`-tree. It is only applied on non-empty trees, because the empty tree is already treated separately in `join`.

3.7 Abstract datatypes

In subsection 1.6 we have explained the module structure of CLEAN. By default a function only has a meaning inside the implementation module it is defined in. If you want to use a function in another module as well, the type of that function has to be repeated in the corresponding definition module. Now, if you want to export a type, you simply repeat the type declaration in the definition module. For instance, the type `Day` of subsection 3.4.1 is exported by repeating its complete definition

```
definition module day

:: Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

in the definition module.

For software engineering reasons it often much better only to export the *name* of a type but not its concrete definition (the right-hand side of the type definition). In CLEAN this is done by specifying only the left-hand side of a type in the definition module. The concrete definition (the right-hand side of the type definition) remains hidden in the implementation module, e.g.

```
definition module day

:: Day
```

So, CLEAN's module structure can be used to hide the actual definition of a type. The actual definition of the type can be an algebraic datatype, a record type or a synonym type (giving a new name to an existing type).

A type of which the actually definition is hidden is called an *abstract datatype*. The advantage of an abstract datatype is that, since its concrete structure remains invisible for the outside world, an object of abstract type can only be created and manipulated with help of functions that are exported by the module as well. The outside world can only pass objects of abstract type around or store them in some data structure. They cannot create such an abstract object nor change its contents. The exported functions are the only means with which the abstract data can be created and manipulated.

Modules exporting an abstract datatype provide a kind of data encapsulation known from the object oriented style of programming. The exported functions can be seen as the methods to manipulate the abstract objects.

The most well-known example of an abstract datatype is a stack It can be defined as:

```
definition module stack

:: Stack a

Empty   ::   (Stack a)
isEmpty ::   (Stack a) -> Bool
Top     ::   (Stack a) -> a
Push    :: a (Stack a) -> Stack a
Pop     ::   (Stack a) -> Stack a
```

It defines an abstract datatype (object) of type 'Stack of anything'. `Empty` is a function (method) which creates an empty stack. The other functions can be used to push an item of type `a` on top of a given stack yielding a stack (`Push`), to remove the top element from a

given stack (Pop), to retrieve the top element from a given stack (Top), and to check whether a given stack is empty or not (isEmpty).

In the corresponding implementation module one has to think of a convenient way to represent a stack, given the functions (methods) on stacks one has to provide. A stack can very well be implemented by using a list. No new type is needed. Therefore, a stack can be defined by using a synonym type.

```
implementation module stack

::Stack a :== [a]

Empty :: (Stack a)
Empty = []

isEmpty :: (Stack a) -> Bool
isEmpty [] = True
isEmpty s  = False

Top :: (Stack a) -> a
Top [e:s] = e

Push :: a (Stack a) -> Stack a
Push e s = [e:s]

Pop :: (Stack a) -> Stack a
Pop [e:s] = s
```

3.8 Correctness of programs

It is of course very important that the functions you have defined work correctly on all circumstances. This means that each function has to work correctly for all imaginable values of its parameters. Although the type system ensures that a function is called with the correct kind of parameters, it cannot ensure that the function behaves correctly for all possible values the arguments can have. One can of course try to test a function. In that case one has to choose representative test values to test the function with. It is often not easy to find good representative test values. When case distinctions are made (by using patterns or guards) one has to ensure that all possibilities are being tested. However, in general there are simply too many cases. Testing can increase the confidence in a program. However, to be absolutely sure that a function is correct one needs a formal way to reason about functions and functional programs. One of the nice properties of functional programming is that functions are side-effect free. So, one can reason about functional programs by using simple standard mathematical formal reasoning techniques like uniform substitution and induction.

3.8.1 Direct proofs

The simplest form of proofs are direct proofs. A direct proof is a obtained by a sequence of rewrite steps. For a simple example we consider the following definitions:

```
I :: t -> t
I x = x                      // This function is defined in StdEnv.

twice :: (t->t) t -> t
twice f x = f (f x)          // This function is defined in StdEnv.

f :: t -> t
f x = twice I x
```

When we want to show that for all `x`, `f x = x`, we can run a lot of tests. However, there are infinitely many possible arguments for `f` . So, testing can build confidence, but can't show that truth of `f x = x`. A simple proof shows that `f x = x` for all `x`. We start with the function definition of `f` and apply reduction steps to its body.


```
f x = twice I x // The function definition
    = I (I x)             // Using the definition of twice
    = I x                 // Using the definition of I for the outermost function I
    = x                   // Using the definition of I                          □
```

This example shows the style we will use for proofs. The proof consists of a sequence of equalities. We will give a justification of the equality as a comment and end the proof with the symbol □.

Even direct proofs are not always as simple as the example above. The actual proof consists usually of a sequence of equalities. The crux of constructing proofs is to decide which equalities should be used.

For the same functions it is possible to show that the functions `f` and `I` behave equal. It is tempting to try to prove `f = I`. However, we won't succeed when we try to proof the function `f` equal to `I` using the same technique as above. It is not necessary that the function bodies can be shown equivalent. It is sufficient that we show that functions `f` and `I` produce the same result for each argument: `f x = I x`. In general: *two functions are considered to be equivalent when they produce the same answer for all possible arguments.* It is very simple to show this equality for our example:

```
f x = twice I x       // The function definition
    = I (I x)         // Using the definition of twice
    = I x             // Using the definition of I for the outermost function I     □
```

As you can see from this example it is not always necessary to reduce expressions as far as you can (to normal form). In other proofs it is needed to apply functions in the opposite direction: e.g. to replace `x` by `I x`.

A similar problem arises when we define the function `g` as:

```
g :: (t -> t)
g = twice I
```

And try to prove that `g x = x` for all `x`. We can't start with the function definition and apply rewrite rules. In order to show this property we have to supply an arbitrary argument `x` to the function `g`. After invention of this idea the proof is simple and equivalent to the proof of `f x = x`.

3.8.2 Proof by case distinction

When functions to be used in proof are defined consists of various alternatives or contain guards its is not always possible to use a single direct proof. Instead of one direct proof we use a direct proof for all relevant cases.

As an example we will show that for all integer elements `x`, `abs x  0`, using

```
abs :: Int -> Int
abs n
    | n < 0     = ~n
    | otherwise = n
```

Without assumptions on the argument the proof can't be made. The cases to distinguish are indicated by the function under consideration. Here the guard determines the cases to distinguish.

Proof

Case `x < 0`

```
abs x = ~x            // Using the definition of abs
```

Since `x < 0`, `~x` will be greater than `0`, using the definitions in the class `Eq`, `x > 0` implies `x  0`. Hence, `abs x  0` when `x < 0`.

Case `x  0`

```
abs x = x             // Using the definition of abs
```

Since `x  0` and `abs x = x` we have `abs x  0`.

```
Each x is either < 0 or   0. For both cases we have abs x   0. So, abs x   0 for all x. □
```

The last line is an essential step in the proof. When we do not argue that we have covered all cases there is in fact no proof. Nevertheless, this last step is often omitted. It is fairly standard and it is supposed to be evident for the reader that all cases are covered.

The trouble of proving by cases is that you have to be very careful to cover all possible cases. A common mistake for numbers is to cover only the cases `x<0` and `x>0`. The case `x=0` is erroneously omitted.

Proof by case can be done for any datatype. Sometimes we can handle many values at once (as is done in the proof above), in other situation we must treat some or all possible values separately. Although a proof by case can have many cases, the number of cases should at least be finite.

As additional example we show that `Not (Not b) = b` for all Booleans b using:

```
Not :: Bool -> Bool
Not True  = False
Not False = True
```

Proof of `Not (Not b) = b`.

Case `b == True`

```
Not (Not b)              // The value to be computed.
    = Not (Not True)     // Using the assumption of this case.
    = Not False          // Using the definition of Not for the innermost application.
    = True               // Using the definition of Not.
```

Case `b == False`

```
Not (Not b)              // The value to be computed.
    = Not (Not False)    // Using the assumption of this case.
    = Not True           // Using the definition of Not for the innermost application.
    = False              // Using the definition of Not.
```

Each Boolean (`Bool`) is either `True` or `False`. So, this proof covers all cases and proves that `Not (Not b) = b` for all Booleans `b`. □

3.8.3 Proof by induction

As stated in the previous section proof by cases works only when there are a finite amount of cases to be considered. When there are in principle infinitely many cases to consider we can often use a proof by induction. The principle of proving properties by inductionis very well known in mathematics. In mathematics we prove that some property `P n` holds for all natural numbers `n` by showing two cases :

- Base case: Prove `P 0`
- Induction step: Prove `P (n+1)` assuming that `P n` holds.

The principle of this proof is very similar to recursion. Using the base case and the induction step we can prove `P 1`. Using `P 1` and the induction step we show that `P 2` holds. In the same way we can prove `P 3`, `P 4`, `P 5`, …. Using this machinery we can prove `P n` for any `n`. Since this is a common proof method, it suffices to show the base case and the induction step. The fact that the property can be proven for any value from these parts is taken for granted when you refer to induction.

As example we will show that the following efficient definition of the Fibonacci function and the naive definition are equivalent for all non-negative integers.

```
tupleFib :: Int -> Int
tupleFib n = fibn
where
    (fibn,_) = tf n
    tf 0 = (1,1)
    tf n = (y,x+y) where (x,y) = tf (n-1)
```



```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Proof of `tupleFib n = fib n`

The key step here is to understand that `tf n = (fib n, fib (n+1))` for all integers `n >= 0`. Once we have seen that this is the goal of our proof, it can be proven by induction.

Case `n == 0`

We have to prove that `tf 0 == (fib 0, fib (0+1))`. This is done by rewriting `tf 0`.

```
tf 0                         // Expression to be proven equal to (fib 0, fib (0+1))
    = (1,1)                  // Definition of tf.
    = (fib 0,1)              // Using first alternative of the definition of fib.
    = (fib 0,fib 1)          // Using second alternative of the definition of fib.
    = (fib 0,fib (0+1))      // Arithmetic.
```

This proves that `tf n = (fib n, fib (n+1))` for `n = 0`.

Case `n + 1`

We have to show that `tf (n+1) = (fib (n+1), fib (n+1+1))` assuming that `tf n = (fib n, fib (n+1))`. We will prove this by rewriting the expression `tf (n+1)`.

```
tf (n+1)                                  // Initial expression.
    = (y,x+y) where (x,y) = tf (n+1-1)    // Definition of tf, assuming that n>0.
    = (y,x+y) where (x,y) = tf n          // Arithmetic
    = (y,x+y) where (x,y) = (fib n, fib (n+1))  // Induction hypothesis
    = (fib (n+1),fib n + fib (n+1))       // Rewriting the expression.
    = (fib (n+1),fib (n+1+1))             // Last alternative of function fib.
```

This proves that `tf (n+1) = (fib (n+1), fib (n+1+1))` assuming that `tf n = (fib n, fib (n+1))`. These case together prove by induction that `tf n = (fib n, fib (n+1))` for all positive `n`. Using this result, proving `tupleFib n = fib n` is done by a direct proof.

```
tupleFib n
    = fibn where (fibn,_) = tf n                  // Definition of tupleFib.
    = fibn where (fibn,_) = (fib n, fib (n+1))    // Using the result proven above.
    = fib n                                       // Rewriting the expression. □
```

The key step in designing proofs is to find the appropriate sub-goals. Proofs can become very complex, by having many sub-goals. These sub-goals can require additional induction proofs and sub-goals, and so on.

As additional example we will prove the following Fibonacci function that avoids tuples equivalent to the function `fib`.

```
fastFib :: Int -> Int
fastFib n = f n 1 1
where
    f 0 a b = a
    f n a b = f (n-1) b (a+b)
```

Proof of `fastFib n = fib n` for all `n   0`.

We will first prove that `f n 1 1 = f (n-m) (fib m) (fib (m+1))` for all `m` such that `m 0 && m n`. This is proven again by induction:

Proof of `f n 1 1 = f (n-m) (fib m) (fib (m+1))`

Case `m == 0` of `f n 1 1 = f (n-m) (fib m) (fib (m+1))`

```
f (n-m) (fib m) (fib (m+1))        // Start with right-hand side.
    = f n (fib 0) (fib 1)          // Use m == 0.
    = f n 1 1                      // Use function fib.
```

Case `m + 1` of `f n 1 1 = f (n-m) (fib m) (fib (m+1))`

```
f n 1 1                                           // Left-hand side of goal.
    = f (n-m) (fib m) (fib (m+1))                 // Use induction hypothesis.
    = f (n-(m+1)) (fib (m+1)) (fib m + fib (m+1)) // Rewrite according to f.
    = f (n-(m+1)) (fib (m+1)) (fib ((m+1)+1))     // Using fib in reverse.
```

This proves that `f n 1 1 = f (n-m) (fib m) (fib (m+1))`. Now we can use this result to prove that `fastFib n = fib n`.

```
fastFib n                                  // Left-hand side of equality to prove.
    = f n 1 1                              // Definition of fastFib.
    = f (n-m) (fib m) (fib (m+1))          // Using the sub-goal proven above.
    = f 0 (fib n) (fib (n+1))              // Use m = n and arithmetic.
    = fib n                                // According to definition of f.  □
```

Inductive proofs for recursive datatypes

There is no reason to limit induction proofs to natural numbers. In fact induction proofs can be given for any ordered (data)type, under the assumption that the datastructures are finite. A well-known example in functional languages is the datatype list. The base case is `P([])`, the induction step is `P([x:xs])` assuming `P(xs)`.

As example we will show the equivalence of the following functions to reverse lists. The function `rev` is simple and a clear definition. In chapter III.3 we will show that `reverse` is more efficient.

```
rev :: [t] -> [t]                // Quadratic in the length of the argument list.
rev []    = []
rev [a:x] = rev x ++ [a]

reverse :: [t] -> [t]            // Linear in the length of the argument list.
reverse l = r l []
where
    r []    y = y
    r [a:x] y = r x [a:y]
```

Proof of `rev l = reverse l` for every (finite) list `l`.

In order to prove this we first prove an auxiliary equality: `r xs ys = rev xs ++ ys`. This is proven by induction to `xs`.

Proof of `r xs ys = rev xs ++ ys`.

Case `xs == []` of `r xs ys = rev xs ++ ys`.

```
r xs ys
    = r [] ys                    // Using xs == [].
    = ys                         // Definition of the function r.
    = [] ++ ys                   // Definition of the operator ++ in reverse.
    = rev [] ++ ys               // The first alternative of rev in reverse.
```

Case `xs == [a:x]` of `r xs ys = rev xs ++ ys`. Assuming `r x ys = rev x ++ ys`.

```
r xs ys
    = r [a:x] ys                 // Using xs == [a:x].
    = r x [a:ys]                 // According to function r.
    = rev x ++ [a:ys]            // Using the induction hypothesis
    = rev x ++ [a] ++ ys         // Using operator ++ in reverse.
    = rev [a:x] ++ ys            // Associativity of ++ and last alternative of rev.
```

This proves `r xs ys = reverse xs ++ ys` by induction. We will use this auxiliary result to show our main equality: `reverse l = rev l`.

```
reverse l
    = r l []                     // According to the definition of reverse.
    = rev l ++ []                // Using the auxiliary result.
    = rev l                      // Using l ++ [] = l for any list l.   □
```

Actually the proofs presented above are not complete. Also the "obvious" properties of the operators should be proven to make the proofs complete. This is a topic of the exercises.

After you have found the way to prove a property, it is not difficult to do. Nevertheless, proving function correct is much work. This is the reason that it is seldom done in practise, despite the advantages of the increased confidence in the correctness of the program. Since also proofs can be wrong, you should always be a little bit careful in using the results of your program, even if you have proven your program "correct".

3.8.4 Program synthesis

In the previous section we treated the development of function definitions and proving their equivalence as two separate activities. In program synthesis these actions are integrated. We start with a specification, usually a naive and obviously correct implementation, and syntheses a new function definition. The reasoning required in both methods is essentially equal. In program synthesis we try to construct the function in a systematic way. Without synthesis these programs should be created out of thin air. In practise we often need to create the key step for program synthesis out of thin air. This key step is usually exactly equivalent to the step needed to construct the proof afterwards.

In program synthesis we use only a very limited number of transformations: *rewrite* according to a function definition (also called unfold [Burstall 87?]), *introduction of patterns and guards* (similar to the cases in proofs), *inverse rewrite steps* (this replacing of an expression by a function definition is called fold), and finally the introduction of *eureka definitions*.

In order to demonstrate program synthesis we will construct a recursive definition for `reverse` from the definition using the toolbox function `foldl`. The definition for `foldl` is repeated here to since it will be used in the transformation.

```
reverse :: [t] -> [t]
reverse l = foldl (\xs x -> [x:xs]) [] l    // The specification

foldl :: (a -> (b -> a)) a [b] -> a         // Toolbox function equivalent to StdEnv
foldl f r []    = r
foldl f r [a:x] = foldl f (f r a) x
```

The first step is to introduce patterns for the argument list in `reverse`. This is necessary since no rewrite step can be done without assumption on the form of the argument.

Case `l == []`

```
reverse l                                   // The function to transform
    = foldl (\xs x -> [x:xs]) [] l          // Using the specification.
    = foldl (\xs x -> [x:xs]) [] []         // The assumption of this case.
    = []                                    // Using alternative 2 of foldl.
```

Case `l == [a:x]`

```
reverse l                                                   // The function to transform.
    = foldl (\xs x -> [x:xs]) [] l                          // Using the specification.
    = foldl (\xs x -> [x:xs]) [] [a:x]                      // Assumption l == [a:x].
    = foldl f (f [] a) x where f = \xs x -> [x:xs]          // Alternative 1 of foldl.
    = foldl (\xs x -> [x:xs]) [a] x                         // Lambda reduction.
    = foldl (\xs x -> [x:xs]) ([]++[a]) x                   // Properties of ++.
    = foldl (\xs x -> [x:xs]) [] x ++ [a]                   // Function foldl and λ-term.
    = reverse x ++ [a]                                      // Fold to call of reverse.
```

Collecting the cases we have obtained:

```
reverse :: [t] -> [t]
reverse []    = []
reverse [a:x] = reverse x ++ [a]
```

In order to turn this in an algorithm that is linear in the length of the list we need the eureka definition `revAcc xs ys = reverse xs ++ ys`. Note that this is exactly equivalent to the key step of the proof in the previous section.

```
reverse l                        // Function to transform.
    = reverse l ++ []            // Property of ++.
    = revAcc l []                // Eureka definition.
```

To obtain a definition for `r` we use again pattern introduction:

Case `l == []`

```
revAcc [] ys = reverse [] ++ ys  // Eureka definition and l == [].
    = [] ++ ys                   // Definition of reverse.
    = ys                         // Definition of operator ++.
```

Case `l == [a:x]`

```
revAcc [a:x] ys = reverse [a:x] ++ ys          // Eureka definition, l==[a:x].
    = reverse x ++ [a] ++ ys                   // Recursive definition reverse.
    = reverse x ++ [a:ys]                      // Associativity of ++.
    = revAcc x [a:ys]                          // Fold using eureka definition.
```

Collecting the cases we obtain:

```
reverse :: [t] -> [t]
reverse l = revAcc l []

revAcc :: [t] [t] -> [t]
revAcc [] ys    = ys
revAcc [a:x] ys = revAcc x [a:ys]
```

Since we used exactly the same eureka definition as was used in the proof, it is not surprising to see that we have obtained an completely equivalent definition of `reverse`. The key step in this kind of program synthesis is to discover the proper eureka definitions.

When you use uniqueness information in your functions you have to pay additional attention to the transformations. It is not sufficient that the uniqueness properties are preserved, but the compiler must also be able to verify them. Transformations using the first three steps are can in principle be automated to a certain extend [Wadler 88, Koopman ??] and will perhaps be incorporated into function language implementations in the future.

3.9 Run-time errors

The static type system of CLEAN prevents run-time type errors. The compiler ensures that it is impossible to apply a function to arguments of the wrong type. This prevents a lot of errors during program execution. Nevertheless, the compiler is not able to detect all possible errors. In this section we discuss some of the errors that can still occur.

3.9.1 Non-termination

It is perfectly possible to write programs in CLEAN that can run forever. Sometimes this is the intention of the programmer, in other situations this is considered an error. A very simple example of a program that will not terminate is:

```
Start :: String
Start = f 42

f :: t -> u
f x = f x
```

There is nothing wrong with the type of this program, but it will never produce a result. Programs that yields an infinite data structure are other examples of programs that does not terminate:

```
Start :: [Int]
Start = ones where ones = [1:ones]
```

As we have seen, there are programs manipulating infinite data structures that do terminate. In general it is undecidable whether a given program will terminate or not. So, the compiler is not able to warn you that your program does not terminate.

In large programs it may be pretty complicated to detect the reason why a program does not terminate. When a critical observation of your program does not indicate the error you should isolate the error by breaking your program into pieces that are tested individually. You can prevent a lot these problems by making it a habit to inspect the termination properties of each function you have written immediately after you have written it down. As we have seen there are many valid programs that use infinite data structures. For instance the first 20 prime numbers are computed by (see section 3.2.5):

```
Start :: [Int]
Start = take 20 primes
```


3.9.2 Partial functions

Many of the functions that you write are partial functions: the result of the function is only defined for some of the arguments allowed by the type system. Some examples are:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

depth :: (Tree a) -> Int
depth (Node _ l r) = max (depth l) (depth r)
```

The result of the function `fac` is only defined for integers greater or equal to `0`. For negative arguments the function does not terminate. The function `depth` is only defined for trees that are not a single leaf. There is no rule alternative applicable to the expression `depth Leaf`. Whenever CLEAN tries to evaluate this expression an appropriate error message is generated:

```
Run time error, rule 'depth' in module 'test' does not match
```

The CLEAN compiler analyses functions during compilation. Whenever the compiler cannot decide that there is always a rule alternative applicable it generates an appropriate warning:

```
Warning [test.icl,35,depth]: function may fail
```

Partial functions may result in a run-time error, either defined by the programmer or generated by the compiler in case none of the alternatives of a function matches..

```
fac1 :: Int -> Int
fac1 0 = 1
fac1 n
    | n>0       = n * fac1 (n-1)
    | otherwise = abort "fac1 has an negative argument"

fac2 :: Int -> Int
fac2 0 = 1
fac2 n
    | n>0 = n * fac2 (n-1)
```

When called with a negative argument these functions respectively will cause the following error messages:

```
fac1 has an negative argument
Run time error, rule 'fac2' in module 'test' does not match
```

Although the error is easy to detect in this way it might be a problem to detect the reason why this error was generated. You should make it a habit to consider what will happen when the function is called with 'wrong' arguments. With respect to detecting problems the functions `fac1` and `fac2` are considered better than `fac`. When you are worried about the additional runtime taken by the additional test you might consider doing the test for appropriate arguments once and for all:

```
fac3 :: Int -> Int
fac3 n
| n >= 0 = f n
where
    f 0 = 1
    f n = n * f(n-1)
```

In general it is worse to receive a wrong answer than receiving no answer at all. When you obtain no result you are at least aware of the fact that there is a problem. So, **do not write**:

```
fac4 n
    | n < 1     = 0
    | otherwise = n * fac4 (n-1)
```

A relateted error is:

```
Index out of range
```

This error is caused by selecting an element out of a list that does not exists, using the operator `!!`. A way to generetate the index out of range error is:

```
Start = [1..5] !! 10
```

3.9.3 Cyclic dependencies

When your program uses its own results before they can be computed you have by a nasty error known as *cycle in spine* or *black hole*. The origin of the name of this error is found a possible implementation of functional languages, see part III. This kind of errors can be very hard to find.

We will illustrate this kind of error by a program that generates a sorted list of all numbers of the form 2^n3^m. Computing these numbers is known as the Hamming problem. We will generate Hamming numbers by observing that a new Hamming number can be computed by multiplying and existing number by 2 or 3. Since we generate an infinite list of these numbers we cannot use on ordinary sorting function to sort Hamming numbers We sort these numbers by an adapted version of the function `merge`.

```
ham :: [Int] // definition below gives cycle in spine error
ham =: merge [n*2 \\ n <- ham] [n*3 \\ n <- ham]
where
    merge l=:[a:x] m=:[b:y]
        | a<b       =  [a:merge x m]
        | a==b      =  merge l y
        | otherwise =  [b: merge l y]

Start::[Int]
Start = take 100 Ham
```

Here it is no problem that the function `merge` is only defined for non-empty lists, it will only be used to merge infinite lists. Execution of this program yields:

```
Run Time Warning: cycle in spine detected
```

The source of the error is that the program is not able to generate a first Hamming number. When we know this and observe that 1 is the first hamming number ($1 = 2^0 3^0$), it is easy to give a correct version of this function:

```
ham :: [Int] // correct definition with first number in initial cycle
ham =: [1:merge [n*2 \\ n <- ham] [n*3 \\ n <- ham]]
where
    merge  l=:[a:x] m=:[b:y]
    | a<b       = [a:merge x m]
    | a==b      = merge l y
    | otherwise = [b: merge l y]
```

When we do not use the computed Hamming numbers to generate new Hamming numbers, but compute these numbers again as in:

```
ham :: [Int] // no cycle but function def, = instead of :=, gives heap full error
ham = merge [n*2 \\ n <- ham] [n*3 \\ n <- ham]
where
    merge l=:[a:x] m=:[b:y]
    | a<b       = [a:merge x m]
    | a==b      = merge l y
    | otherwise = [b: merge l y]
```

we obtain a '`heap full`' message instead of the '`cycle in spine`'. For each occurrence of `ham` the expression is evaluated again. Since none of these functions is able to generate a first element, an infinite expression will be generated. The heap will always be too small to hold this expression.

3.9.4 Insufficient memory

The heap is a piece of memory used by the CLEAN program to evaluate expressions. When this memory is exhausted the program tries to regain memory by removing parts of the expression that are not needed anymore. This process is called *garbage collection*. When it is not possible to find sufficient unused memory during garbage collection, the program is aborted and the error message '`heap full`' is displayed. The size of the heap used by programs written in CLEAN can be determined in the CLEAN system. When you program dis-

plays the '`heap full`' message you can try it again after you have increased the heap size. As shown in the previous paragraph it is also possible that a programming error causes this error message. No matter how large the heap is, the program will never behave as intended. In large programs it can be pretty tough to locate the source of this kind of error.

Apart from the heap, a program written in CLEAN uses some stacks. These stacks are used to maintain information on function arguments and parts of the expression currently under evaluation. Also these stacks can be too small. What happens when such a *stack overflow* occurs depends on the platform you are using and the options set in the CLEAN system. When you choose '`Check Stacks`' the program should notice that the stack space is exhausted and abort the program with an appropriate message. Stack checks cause a slight run-time overhead. Hence, people often switch stack checks off. Without these checks the stack can 'grow' within memory used for other purposes. The information that was kept there is spoiled. This can give error like '`illegal instruction`'.

Whether an erroneous program causes a heap full message or a stack overflow can depend on very small details. The following program will cause a '`heap full`' error:

```
Start :: String
Start = f 42

f :: t -> u
f x = f (f x)
```

We can understand this by writing the first lines of a trace

```
Start
→  f 42
→  f (f 42)
→  f (f (f 42))
→  …
```

It is clear that this expression will grow without bound. Hence execution will always cause a heap full error.

When we add a strictness annotation is added to the function `f`, the argument of `f` will be evaluated before the application of `f` itself is evaluated (see part III).

```
Start :: String
Start = f 42

f :: t -> u
f x = f (f x)
```

The trace looks very similar:

```
Start
→  f 42
→  f (f 42)
→  f (f (f 42))
→  …
```

In order to keep track of the function and its argument under evaluation some stack space is used. Now it depends on the relative size of the stack and the size of the memory which one is the first to be exhausted. CLEAN has a built in strictness analyzer that approximates the strictness properties of functions. A very small and semantically irrelevant change may change the derived strictness properties and hence cause the difference between a '`heap full`' or '`stack overflow`' error.

```
→   fib 4
→*  fib 3 + fib 2
→   fib 3 + fib 1 + fib 0
→   fib 3 + fib 1 + 1
→   fib 3 + 1 + 1
→   fib 3 + 2
→*  fib 2 + fib 1 + 2
→   fib 2 + 1 + 2
→*  fib 1 + fib 0 + 1 + 2
→   fib 1 + 1 + 1 + 2
```

```
→  1 + 1 + 1 + 2
→* 5
```

From this trace it is clear that the operator + evaluates its second argument first.

It is tempting to write a function `trace :: !String x -> x` that writes the string as trace to `stderr`. The definition of this function is somewhat more complicated as you might expect:

```
trace :: !String x -> x
trace s x
 #! y   = fwrites s stderr
 |  1<2 = K x y
 = abort "?"
```

The problem with a simple minded approach like:

```
trace s x = let! y = fwrites s stderr in K x y
```

is that the strictness analyser of CLEAN discovers that `x` is always needed. So, `x` is evaluated before the function `trace` is invoked. This will spoil the order of the trace information. Switching strictness analysis off prevents this, but it may change the order of evaluation of the program you are investigating.

Using the function `trace` we can write a version of the Fibonacci function that produces a trace as:

```
fib n = trace   ("fib "+toString n+" ")
                (if (n<2) 1 (fib (n-1) + fib (n-2)))
```

When we want to include also the result of reduction is the trace we have to be vary careful that the order of computations is not changed. For some programs changing the order of computations is not a real problem. For other programs, changing the order of reductions can cause non-termination.

When we write:

```
fib n = trace ("fib "+toString n+" = "+toString m+" ") m
    where m = if (n<2) 1 (fib (n-1) + fib (n-2))
```

the trace will be reversed! In order to write the first call of `fib`, we must evaluated this string. In order to evaluate this string we need the value of the outermost function, and hence all other occurrences of the `fib` function.

The module StdDebug that comes with the Clean system contains several predefined variants of the trace function.

3.10 Exercises

1. Define a function `CountOccurences` that counts the number of times a given element is occuring in a given list.

   ```
   CountOccurences :: a [a] -> Int | == a
   ```

2 Define the function MakeFrequenceTable

   ```
   MakeFrequenceTable [a] -> [(a, Int)] | == a
   ```

   That makes a frequence table. The table consists of a list of tuples. Each tuple consists of an element from the original list and its frequency (percentage). E.g.

   ```
   Frequentietabel [1,2,3,2] = [(1,25),(2,50),(3,25)]
   ```

3 Equality on tuples can be defined as:

   ```
   (==) (a,b) (c,d) = a == c && b == d
   ```

   Although the equality operator is also applied in the right-hand side expression this function is actually not recursive.

   What is the difference between this operator definition and the recursive definition of equality for lists in Section 3.1.2?

4 Define the function `flatten` (see Section 3.1.2) in terms of `foldr` and `++`.

5 Write a list comprehension for generating all permutations of some input list.

6 Describe the effect on the evaluation order of swapping `x==y` and `xs==ys` in the definition of `==` in Section 3.2.2.

7 Extend the set of operators on rational numbers with `==` and `<`.

8 Discuss how you can guarantee that rational numbers used in ordinary programs are always 'simplified'.

9 Define an appropriate datatype for AVL-trees and define functions for balancing, searching, inserting and deleting elements in such trees.

10 Proof that `abs (sign x) < 2` for all `x` using:

   ```
   sign x | x < 0  = -1
   | x == 0 = 0
   | x > 0  = 1
   ```

11 Proof that `fib n` `n` for all `n` `2`.

12 Proof that `l ++ [] = l`.

13 Proof that `x ++ (y ++ z) = (x ++ y) ++ z`. This is the associativity of `++`.

14 Proof that `rev (x ++ y) = rev y ++ rev x`.

15 Proof that `rev (rev xs) = xs` for every finite list.

16 Proof that `foldl (\xs x -> [x:xs]) ys x = foldl (\xs x -> [x:xs]) [] x ++ ys`.

17 Synthesize a recursive function to add elements of a list which is equivalent to:

   ```
   sum :: [t] -> t | + , zero t
   sum l = foldr (+) zero l
   ```

18 Design a Eureka rule to introduce an accumulator and transform the recursive function to a call of the addition function using the accumulator.

Part I
Chapter 4
The Power of Types

4.1 Type classes
4.2 Existential types
4.3 Uniqueness types
4.4 Exercises

CLEAN is a strongly typed language. This means that every expression in the language is typed and that type correctness can be verified by the compiler before the program is executed. Ill-typed programs are not accepted. Due to the type system, many errors can be found and reported at compile time. It is important to find programming errors in an early stage of the software development process. Correcting errors later generally takes much more effort and can cost a considerable amount of money. The type system certainly helps to find a large class of errors as soon as possible. However, the type system can only find certain programming errors which have to do with inconsistent use. E.g. it cannot find logical errors. Although the CLEAN's type system is very flexible there are some restrictions as well in order to make type correctness statically decidable. This implies that in some rare cases programs are rejected by the compiler although they are correct. We are willing to pay this price due to the very strong advantages of static decidability of type correctness.

Type systems can also be used to increase the expressive power of a language. In this chapter a number of language features which are related to the type system are explained. First we will explain the overloading mechanism of CLEAN which makes it possible to use the same function name for different functions performing similar kind of actions. It can be used to write (parts of) programs in such a way that the actual data structures being used can be chosen in a later state of the design (section 4.1). We explain how one can store objects of different types into a recursive data structure like a list using existentially quantified datatypes. In this way an object oriented style of programming can be achieved (section 4.2). Finally we treat an important feature of CLEAN: the uniqueness type system (section 4.3). It makes it possible to destructively update data structures like arrays and files without violating the pure functional semantics of the language.

4.1 Type Classes

When one defines a new function one has to give the function a new (meaningful) name different from all other function names which have been defined in the same scope However, sometimes it can be very convenient to reuse an existing name. An obvious example is the function '+'. One would like to have a '+' on integers, a '+' on reals and so on. So, sometimes one can increase the readability by using the same name for different functions doing similar kind of things albeit on different types. The mechanism which allows such functions to be defined is called *overloading* or *ad-hoc polymorphism*. Overloading occurs when several functions are defined in the same scope which have the same name. However, each

of these functions has a (slightly) different type. So, one and the same (overloaded) function name (e.g. `+`) is associated with a collection of different operations (`Int` addition, `Real` addition, etcetera). Notice the difference with a polymorphic function like `map` which is just one function defined over a range of types, acting in the same way for each concrete type.

The definition of an overloaded function consists of two parts:

1) the *signature* of the overloaded function, i.e. the name and the overall type the overloaded functions have in common;
2) a collection of (type dependent) concrete realizations: the *instances*.

For reasons of flexibility, these parts can be specified separately (e.g. in different modules). In CLEAN, a signature is introduced by a *class declaration*. This class declaration indicates that there may occur a number of functions with that name. In order to guarantee that all these functions are sufficient similar, the type of these functions should be an instance of the common type given in the signature. The signature can be seen as a blueprint all concrete instances have to obey. Examples of such signatures are the following (pre-defined) class declarations introducing some common overloaded operators.

```
class (+) infixl 6  a :: a a-> a
class (-) infixl 6  a :: a a-> a
class zero          a :: a

class (*) infixl 7  a :: a a-> a
class (/) infix 7   a :: a a-> a
class one           a :: a

class (==) infix 2  a :: a a-> Bool
class (<) infix 2   a :: a a-> Bool
```

In each class declaration, one of the type variables appearing in the signature is denoted explicitly. This *class variable* is used to relate the type of an overloaded operator to all the types of its instances. The latter are introduced by *instance declarations*. An instance declaration associates a function body with a concrete instance type. The type of this function is determined by substituting the instance type for the class variable in the corresponding signature. For example, we can define an instance of the overloaded operator `+` for strings, as follows.

```
instance + {#Char}
where
    (+) s1 s2 = s1 +++ s2
```

Since it is not allowed to define instances for type synonyms we have to define an instance for `{#Char}` rather than for `String`. Allowing instances for type synonyms would make it possible to have several different instances of some overloaded function which are actually instances for the same type. The Clean systems cannot distinguish which of these instances should be applied.

By substituting `{#Char}` for `a` in the signature of `+` one obtains the type for the newly defined operator, to wit `{#Char} {#Char}-> {#Char}`. In CLEAN it is permitted to specify the type of an instance explicitly, provided that this specified type is exactly the same as the type obtained via the above-mentioned substitution. Among other things, this means that the following instance declaration is valid as well.

```
instance + {#Char}
where
    (+) :: {#Char} {#Char} -> {#Char}
    (+) s1 s2 = s1 +++ s2
```

A large number of these operators and instances for the basic types and datatypes are predefined in `StdEnv`. In order to limit the size of the standard library only those operations that are considered the most useful are defined. It might happen that you have to define some instances of standard functions and operators yourself.

Observe that, what we have called an overloaded function is not a real function in the usual sense: An overloaded function actually stands for a whole family of functions. Therefore, if an overloaded function is applied in a certain context, the type system determines which concrete instance has to be used. For instance, if we define

```
increment n = n + 1
```

it is clear that the `Int` addition is meant leading to a substitution of this `Int` version for `+`. However, it is often impossible to derive the concrete version of an overloaded function from the context in which it is applied. Consider the following definition:

```
double n = n + n
```

Now, one cannot determine from the context which instance of `+` is meant. In fact, the function `double` becomes overloaded itself, which is reflected in its type:

```
double :: a -> a | + a
```

The *type context* `+` appearing in the type definition indicates the restriction that double is defined only on those objects that can be handled by a `+`.

Some other examples are:

```
instance + (a,b) | + a & + b
where
    (+) (x1,y1) (x2,y2) = (x1+x2,y1+y2)

instance == (a,b) | == a & == b
where
    (==) (x1,y1) (x2,y2) = x1 == x2 && y1 == y2
```

In general, a type context of the form `C a`, restricts instantiation of `a` to types for which an instance declaration of `C` exists. If a type context for `a` contains several class applications, it assumed that `a` is chosen from the instances types all these classes have in common.

One can, of course, use a more specific type for the function `double`. E.g.

```
double :: Int -> Int
double n = n + n
```

Obviously, `double` is not overloaded anymore: due to the additional type information, the instance of `+` to be used can now be determined.

Type contexts can become quite complex if several different overloaded functions are used in a function body. Consider, for example, the function `determinant` for solving quadratic equations.

```
determinant a b c = b * b - (fromInt 4) * a * c
```

The type of `determinant` is

```
determinant :: a a a -> a | *, -, fromInt a
```

To enlarge readability, it is possible to associate a new (class) name with a set of existing overloaded functions. E.g.

```
class Determinant a | *, -, fromInt a
```

The class `Determinant` consists of the overloaded functions `*`, `-` and `fromInt`. Using the new class in the type of `determinant` leads to the type declaration:

```
determinant :: a a a -> a | Determinant a.
```

Notice the difference between the function `determinant` and the class `Determinant`. The class `Determinant` is just a shorthand notation for a set of type restrictions. The name of such a type class should start with an uppercase symbol. The function `determinant` is just a function using the class `Determinant` as a compact way to define some restrictions on its type. As far as the CLEAN system is concerned it is a matter of coincidence that you find these names so similar.

Suppose `C1` is a new class, containing the class `C2`. Then `C2` forms a so-called *subclass* of `C1`. 'Being a subclass of' is a transitive relation on classes: if `C1` on its turn is a subclass of `C3` then also `C2` is also a subclass of `C3`.

A class definition can also contain new overloaded functions, the so-called *members* of the class. For example, the class `PlusMin` can be defined as follows.

```
class PlusMin a
where
    (+) infixl 6 :: a a -> a
    (-) infixl 6 :: a a -> a
    zero         :: a
```

To instantiate `PlusMin` one has to specify an instance for each of its members. For example, an instance of `PlusMin` for `Char` might look as follows.

```
instance PlusMin Char
where
    (+) x y = toChar ((toInt x) + (toInt y))
    (-) x y = toChar ((toInt x) - (toInt y))
    zero    = toChar 0
```

Some of the readers will have noticed that the definition of an overloaded function is essentially the same as the definition of a class consisting of a single member. Indeed, classes and overloaded operators are one and the same concept. Since operators are just functions with two arguments, you can use operators in type classes in the same way as ordinary functions.

As stated before, a class defines in fact a family of functions with the same name. For an overloaded function (a class member) a separate function has to be defined for each type instance. In order to guarantee that only a single instance is defined for each type, it is not allowed to define instances for type synonyms. The selection of the instance of the overloaded function to be applied is done by the CLEAN system based on type information. Whenever possible this selection is done at compile-time. Sometimes it is not possible to do this selection at compile-time. In those circumstances the selection is done at run-time. Even when the selection of the class member to be applied is done at run-time, the static type system still guarantees complete type consistency.

In CLEAN, the general form of a class definition is a combination of the variants discussed so far: A new class consists of a collection of existing classes extended with a set of new members. Besides that, such a class will appear in a type context of any function that uses one or more of its members, of which the actual instance could not be determined. For instance, if the `PlusMin` class is used (instead of the separate classes `+`, `-` and `zero`), the types of `double` and `determinant` will become:

```
double :: a -> a | PlusMin a
determinant :: a a a -> a | *, PlusMin, fromInt a
```

The CLEAN system itself is able to derive this kind of types with class restrictions.

The class `PlusMin` is defined in the standard environment (`StdClass`) is slightly different from the definition shown in this section. The definition is the standard environment is:

```
class PlusMin a | + , - , zero a
```

When you use the class `PlusMin` there is no difference between both definitions. However, when you define a new instance of the class you have to be aware of the actual definition of the class. When the class contains members, you have to create an instance for all member of the class as shown here. For a class that is defined by a class context, as `PlusMin` from `StdClass`, you define an instance by defining instances for all classes listed in the context. In the next section we show an example of the definition of an instance of this class.

4.1.2 A class for Rational Numbers

In chapter 3.4.1 we introduced a type `Q` for representing rational numbers. These numerals are records consisting of a numerator and a denominator field, both of type `Int`:

```
:: Q = {   num :: Int
       ,   den :: Int
       }
```

We define the usual arithmetical operations on `Q` as instances of the corresponding type classes. For example,

```
instance + Q
where
    (+) x y = mkQ (x.num * y.den + x.den * y.num) (x.den * y.den)

instance - Q
where
    (-) x y = mkQ (x.num * y.den - x.den * y.num) (x.den * y.den)

instance fromInt Q
where
    fromInt i = mkQ i 1

instance zero Q
where
    zero = fromInt 0
```

Using:

```
mkQ :: x x -> Q | toInt x
mkQ n d = simplify {num = toInt n, den = toInt d}

simplify :: Q -> Q
simplify {num=n,den=d}
    | d == 0    = abort "denominator of Q is 0!"
    | d < 0     = {num = ~n / g, den = ~d / g}
    | otherwise = {num =  n / g, den =  d / g}
where
    g = gcd n d
```

At first sight, it seems as if the definition of, for example, the instance for `+` is recursive, for, an application of `+` also appears in the body of this instance. However, from its context, it immediately follows that the actual operation that is meant is the `+` for values of type `Int`.

When a new datatype is introduced, it is often convenient if a string representation of this datatype is available. Amongst other things, this representation can be used for printing a concrete value of that type on the screen. For this purpose, the class `toString` is introduced in the standard environment:

```
class toString a :: a -> String
```

The corresponding instance of `toString` for `Q` might look as follows.

```
instance toString Q
where toString q
    | sq.den==1 = toString sq.num
    | otherwise = toString sq.num +++ "/" +++ toString sq.den
where
    sq = simplify q
```

By making `Q` an abstract datatype, the simplification of `q` in this function can be omitted. Such an abstract datatype guarantees that all rational numbers are simplified, provided that the functions in the abstract datatype always simplify a generated rational numbers.

By defining an instance of class `Enum` for the type `Q` it is even possible to generate list of rational numbers using dotdot expressions. Apart form the functions `+`, `-`, `zero` and `one`, the class `Enum` contains the ordering operator `<`. A suited instance declaration of `<` for `Q` is

```
instance < Q
where
    (<) x y = x.num * y.den < x.den * y.num
```

A program like

```
Start :: [String]
Start = [toString q \\ q <- [zero, mkQ 1 3 .. mkQ 3 2]]
```


is type correct. It's execution yields:

```
["0","1/3","2/3","1","4/3"]
```

4.1.3 Internal overloading

The execution of the program

```
Start :: String
Start = toString sum
where
    sum :: Q
    sum = zero + zero
```

results in the string `"0"`.

It seems as if it makes no difference if we would write

```
Start = toString (zero + zero)
```

However, in this situation it is not possible to determine the used instances of `zero`, `+` and `toString` uniquely, i.e. there are several concrete instances that can be applied. The problem is that the expression `toString (zero + zero)` is *internally* overloaded: the overloading is not reflected by its result type (which is simply `String`). Such an expression will cause the compiler to generate the error message:

```
Type error [...]: "zero" (internal) overloading is insolvable
```

When it is known which instance of, for example, `zero` should be used, one can deduce the concrete instances of `+` and `toString`. Internal overloading can always be solved by introducing auxiliary local definitions that are typed explicitly (like the `sum` function in the above example).

Another way to solve the ambiguity is to indicate one of the instances of the class as the *default instance*. In all situations where the overloading cannot be resolved, this default instance will be used. For instance, we can define the instance of type `Q` the default for the class `zero` by writing:

```
instance zero Q default
where
    zero = mkQ 0 1
```

Now it is allowed to write

```
Start = toString (zero + zero)
```

The context still does not determine which `zero` is used here, but now the CLEAN system picks the default one: the `zero` of type `Q`.

4.1.4 Derived class members

Sometimes, a member of a class is not really a new function, but defined in terms of other members (either of the same class or of a subclass). The standard environment, for example, introduces the class `Eq` containing the comparison operators `==` (already defined as class in `StdOverloaded`) and `<>` in the following way.

```
class Eq a | == a
where
    (<>) infix 2 :: a a -> Bool | Eq a
    (<>) x y :== not (x == y)
```

The `<>` operator is an example of, what is called, a *derived class member*: a member of which the body is included in the class definition itself. In contrast to other functional languages, like Haskell and Gofer, the instances of derived members are never specified in CLEAN; they are *inherited* from the classes corresponding to the used operators (`==` in the above example).

In the same style we can define a complete set of ordering operators based on `<`.

```
class Ord a | < a
where
    (>)  infix 2 :: a a -> Bool | Ord a
    (>)  x y :== y < x

    (<=) infix 2 :: a a -> Bool | Ord a
    (<=) x y :== not (y<x)

    (>=) infix 2 :: a a -> Bool | Ord a
    (>=) x y :== not (x<y)
```

In fact, also the equality operator `==` could be defined as a derived member, e.g. by specifying

```
class Eq a | < a
where
    (==) infix 2 :: a a -> Bool | Eq a
    (==) x y :== x <= y && x >= y

    (<>) infix 2 :: a a -> Bool | Eq a
    (<>) x y :== not (x == y)
```

By this mechanism, one obtains all ordering operations for a certain type, solely by defining an instance of `<` for this type. For efficiency reasons this is not done in the standard environment of CLEAN. In order to enable all possible comparison for some type `T` you should define an instance of `<` and `==`.

When defining instances of functions acting on polymorphic data structures, these instances are often overloaded themselves, as shown by the following example.

```
instance < [a] | < a
where
    (<) :: [a] [a] -> Bool | < a
    (<) _       []      = False
    (<} []      _       = True
    (<) [x:xs]  [y:ys]  = x < y || x == y && xs < ys
```

The instance type `[a]` is supplied with a type context which reflects that, in the body of the instance, the `<` operator is applied to the list elements. Observe that the specified type is, as always, the same as the type obtained from the signature of `<` after substituting `[a] | < a` for the class variable.

This example clearly shows the expressive power of the type classes. Suppose an instance `<` for some type `T` is available. With one single instance definition it possible to compare objects of type `[T]`, of type `[[T]]` and so on.

4.1.5 Type constructor classes

Until now, we assumed that each type constructor has a fixed arity indicating the number a type arguments, an application of that type constructor is supposed to have. For example the list constructor `[]` has arity 1, the 3-tuple constructor `(,,)` has arity 3, etcetera. *Higher-order types* are obtained by allowing type constructor applications in which the actual number of type arguments is less than the arity of the used constructor. In CLEAN it is possible to define classes with class variables ranging over such higher-order types. This leads to a so-called *type constructor class*. Type constructor classes can be used to define collections of overloaded higher-order functions. To explain the idea, consider the `map` function, defined as usual.

```
map :: (a -> b) [a] -> [b]
map f []     = []
map f [x:xs] = [f x:map f xs]
```

Experienced programmers will recognise that similar functions are often used for a wide range of other, mostly polymorphic data structures. E.g.

```
:: Tree a = Node a [Tree a]

mapTree :: (a -> b) (Tree a) -> Tree b
mapTree f (Node el ls) = Node (f el) (map (MapTree f) ls)
```



```
:: MayBe a = Just a | Nothing

MapMayBe :: (a -> b) (MayBe a) -> MayBe b
MapMayBe f (Just a) = Just (f a)
MapMayBe f Nothing  = Nothing
```

Since all of these variants for `map` have the same kind of behavior, it seems to be attractive to define them as instances of a single overloaded `map` function. Unfortunately, the overloading mechanism presented so far is not powerful enough to handle this case. For, an adequate class definition should be able to deal with (at least) the following type specifications:

```
(a -> b) [a] -> [b]
(a -> b) (Tree a) -> Tree b
(a -> b) (MayBe a) -> MayBe b.
```

It is easy to see, that a type signature for `map` such that all these type specifications can be obtained via the substitution of a single class variable by appropriate instance types, is impossible. However, by allowing class variables to be instantiated with higher-order instead of first-order types, such a type signature can be found, as indicated by the following class definition.

```
class map t :: (a -> b) (t a) -> t b
```

Here, the ordinary type variables `a` and `b` range over first-order types, whereas the class variable `t` ranges over higher-order types. To be more specific, the concrete instance types that can be substituted for `t` are (higher-order) types with one argument too few. The instance declarations that correspond to the different versions of `map` can now be specified as follows.

```
instance map []
where
    map f l = [f e \\ e <- l]

instance map Tree
where
    map f (Node el ls) = Node (f el) (map (map f) ls)

instance map MayBe
where
    map f (Just a) = Just (f a)
    map f Nothing  = Nothing
```

The following instance declaration for map is also valid.

```
instance map (,) a
where
    map :: (a -> b) (c,a) -> (c,b)
    map f (x,y) = (x,f y)
```

Here `(,) a` denotes the 2-tuple type constructor applied to a type variable `a`. Observe that an instance for type `(,)` (i.e. the same type constructor, but now with no type arguments) is impossible.

4.2 Existential types

Polymorphic algebraic datatypes offer a large flexibility when building new data structures. For instance, a list structure can be defined as:

```
:: List a = Cons a (List a) | Nil
```

This type definition can be used to create a list of integers, a list of characters, or even a lists of lists of something. However, according to the type definition, the types of the list elements stored in the list should all be equal, e.g. a list cannot contain both integers and characters. Of course, one can solve this problem *ad hoc*, e.g. by introducing the following auxiliary type.

```
:: OneOf a b = A a | B b
```

Indeed, a list of type `List (OneOf Int Char)` may contain integers as well as characters, but again the choice is limited. In fact, the amount of freedom is determined by the number of type variables appearing in the datatype definition. Of course, this can be extended to any finite number of types, e.g. `List (OneOf (OneOf Int (List Int)) (OneOf Char (Char -> Int)))`.

To enlarge applicability, CLEAN has been extended with the possibility to use so called *existentially quantified* type variables (or, for short, *existential* type variables) in algebraic datatype definitions. Existential type variables are not allowed in type specifications of functions, so data constructors are the only symbols with type specifications in which these special type variables may appear. In the following example, we illustrate the use of existential variables by defining a list data structure in which elements of different types can be stored.

```
:: List = E.a: Cons a List | Nil
```

The `E` prefix of `a` indicates that `a` is an existential type variable. In contrast to ordinary polymorphic (or, sometimes, called *universally quantified*) type variables, an existential type variable can be instantiated with a concrete type only when a data object of the type in question is created. Consider, for example, the function

```
newlist = Cons 1 Nil
```

Here, the variable `a` of the constructor `Cons` is instantiated with `Int`. Once the data structure is created this concrete type information, however, is *lost* which is reflected in the type of the result (`List`). This type allows us to build structures like `Cons 1 (Cons 'a' Nil)`.

However, when a data structure which is an instantiation of an existential quantified type variable is accessed e.g. in a pattern match of a function, it is not possible to derive its concrete type anymore. Therefore, the following function `Hd` which yields the head element of a list

```
Hd :: List -> ????         // this function cannot be typed statically
Hd (Cons x xs) = x
```

is illegal, for, it is unknown what the actual type of the returned list element will be. It can be of any type. The types of the list elements stored in the list are lost, and yielding a list element of an unknown type as function result cannot be allowed because the type checker cannot guarantee type correctness anymore. The function `Hd` is rejected by the type system. But, accessing the tail of the above list, e.g. by defining

```
Tl :: List -> List
Tl (Cons x xs) = xs
```

is allowed: one cannot do anything with `Tl`'s result that might disturb type safety.

One might conclude that the existential types are pretty useless. They are not, as shown below.

Creating objects using existential types

Clearly, a data structure with existential quantified parts is not very useful if there exist no way of accessing the stored objects. For this reason, one usually provides such a data structure with an *interface*: a collection of functions for changing and/or retrieving information of the hidden object. So, the general form of these data structures is

```
:: Object  = E.a:  { state :: a
                   , method_1 :: ... a ... -> ...
                   , method_2 :: ... -> ...a...
                   ,...
                   }
```

The trick is that, upon creation of the data structure, the type checker can verify the internal type consistency of the state and the methods working on this state which are stored together in the data structure created. Once created, the concrete type associated with the existentially quantified type variable is lost, but it can always be guaranteed that the stored

methods can safely be applied to the corresponding stored state whatever the concrete type is.

Those who are familiar with object oriented programming will recognise the similarity between the concept of *object-oriented data abstraction* and existentially quantified data structures in CLEAN.

We will illustrate the use of existentially quantified data structures with the aid of an example in which 'drawable objects' are represented as follows

```
:: Drawable = E.a: { state :: a
                   , move  :: Point a -> a
                   , draw  :: a Picture -> Picture
                   }
```

A `Drawable` contains a `state` field (e.g. the representation of a point, a line, a circle, etcetera) and two functions `move` and `draw` for moving and displaying the object on the screen, respectively. We use a number of graphical datatypes that are defined in the standard I/O library in the module `Picture`. Drawing pictures is explained in more detail in part II.

```
:: Point      :== (Int, Int)
:: Line       :== (Point, Point)
:: Rectangle  :== (Point, Point)         // bounding points
:: Oval       :== Rectangle              // bounding rectangle
:: Curve      :== (Oval, Int, Int)       // Oval with start and end angle
```

First, we define two auxiliary functions for creating the basic objects `Line` and `Curve`. The corresponding drawing routines are taken from the standard I/O lbrary `Picture`; moving is defined straightforwardly. In the definition of `move` we use `+` for tuples defined as follows:

```
instance + (x,y) | + x & + y
where (+) (x1,y1) (x2,y2) = (x1+x2,y1+y2)

MakeLine :: Line -> Drawable
MakeLine line = { state = line
                , move  = \dist line -> line + (dist,dist)
                , draw  = DrawLine
                }

MakeCurve :: Curve -> Drawable
MakeCurve curve =   { state = curve
                    , move  = \dist (oval,a1,a2) -> (oval + (dist,dist),a1,a2)
                    , draw  = DrawCurve
                    }
```

To illustrate the composition of these objects, a `Rectangle` is defined as a compound structure consisting of four lines, whereas a `Wedge` consists of two lines and a curve.

```
MakeRectangle :: Rectangle -> Drawable
MakeRectangle ((x1,y1),(x2,y2))
    =   { state =    [ MakeLine ((x1,y1),(x1,y2)), MakeLine ((x1,y2),(x2,y2))
                     , MakeLine ((x2,y2),(x2,y1)), MakeLine ((x2,y1),(x1,y1))
                     ]
        , draw  = \s p -> foldl (\pict {state,draw} -> draw state pict) p s
        , move  = \d -> map (MoveDrawable d)
        }

MakeWedge:: Curve -> Drawable
MakeWedge curve=:((begp,endp), a1, a2)
    =   { state = [ MakeLine (mid,mid+ epc1), MakeLine (mid,mid+ epc2)
                  , MakeCurve curve
                  ]
        , draw  = \s p -> foldl (\pict {state,draw} -> draw state pict) p s
        , move  = \d -> map (MoveDrawable d)
        }
where
    mid = (begp+endp)/(2,2)
    (epc1,epc2) = EndPointsOfCurve curve
```

Using a suitable implementation of `EndPointsOfCurve` and:

```
MoveDrawable :: Point Drawable -> Drawable
MoveDrawable p d=:{state,move} = {d & state = move p state}
```

Observe that moving and drawing of both compound objects is done in the same way. Moreover, due to the fact that (possibly different) `Drawables` can be stored in one list (for, the state of such objects is hidden) one can use standard list manipulating functions, such as `map` and `foldl` to perform these operations. Of course, the `Drawable` type is much too simple for being really useful: other functions have to added, e.g. predicates for testing whether a given point lies on the border, in the interior of a drawable or outside of it. Such extensions might be far from trivial, but nevertheless, the elegance of this method based on existentially quantified data structures is maintained. A full-fledged type `Drawable` is developed in part II of this book.

Extential types versus algebraic datatypes

Instead of using an existentially quantified data structure it is also possible to use an ordinary algebraic data structure:

```
:: AlgDrawable  = Line  Line
                | Curve Curve
                | Rect  [Line]
                | Wedge [AlgDrawable]
```

The manipulation of drawable objects now has to be done by separately defined functions. These functions use detailed knowledge of the exact construction of the various alternatives of the algebraic datatype `AlgDrawable`. Fortunately, the compiler generates a warning (`Function may fail`) when we accidentally omit one of the alternatives of `AlgDrawable`.

```
move :: Point AlgDrawable -> AlgDrawable
move p object = case object of
                  Line  line         -> Line  (line + (p,p))
                  Curve (rect,a1,a2) -> Curve (rect + (p,p),a1,a2)
                  Rect  lines        -> Rect  [line + (p,p) \\ line <- lines]
                  Wedge parts        -> Wedge (map (move p) parts)

draw :: AlgDrawable -> (Picture -> Picture)
draw object = case object of
                Line  line  -> DrawLine line
                Curve curve -> DrawCurve curve
                Rect  lines -> seq (map DrawLine lines)
                Wedge parts -> seq (map draw parts)
```

Although this is a way to handle objects that come in several different sorts, it has some drawbacks. The first disadvantage is that the properties and manipulation of drawable objects is distributed over a number of functions. For complicated types that are handled by many functions it becomes problematic to gather all information of that object. A second disadvantage of using an algebraic datatype is that it becomes difficult to maintain the code. When an additional object like `Oval` is introduced, it is difficult to be sure that all corresponding functions are updated to handle ovals.

Changing datatypes

When we change our mind and want to store a rectangle by its bounding points this is a very local change in the object-oriented approach. Only the function `MakeRectangle` needs to be changed:

```
MakeNewRectangle :: Rectangle -> Drawable
MakeNewRectangle rect = { state = rect
                        , move  = \p r -> r + (p,p)
                        , draw  = DrawRectangle
                        }
```

When we use the algebraic datatype `AlgDrawable` to represent drawable objects and we want to implement the equivalent change, we have to change the datatype and the corresponding manipulation functions.


```
:: AlgDrawable  = Line  Line
                | Curve Curve
                | Rect  Rectangle                              // changed
                | Wedge [AlgDrawable]

move :: Point NewAlgDrawable -> NewAlgDrawable
move p object = case object of
                  Line  line         -> Line  (line + (p,p))
                  Curve (rect,a1,a2) -> Curve (rect + (p,p),a1,a2)
                  Rect  rect         -> Rect  (rect + (p,p))     // changed
                  Wedge parts        -> Wedge (map (move p) parts)

draw :: NewAlgDrawable -> Picture -> Picture
draw object = case object of
                Line  line  -> DrawLine line
                Curve curve -> DrawCurve curve
                Rect  rect  -> DrawRectangle rect              // changed
                Wedge parts -> seq (map draw parts)
```

On the other-hand, adding an entirely new manipulation function is easier for the algebraic datatype. Only the new function has to be defined. In the object-oriented approach, each object creating function should be changed accordingly.

As example we will add an operation that determines the bounding rectangle of drawable objects. For the algebraic datatype approach we define the function

```
bounds :: AlgDrawable -> Rectangle
bounds object = case object of
                  Line  line  -> normalize line
                  Curve curve -> curveBounds curve
                  Rect  rect  -> normalize rect
                  Wedge parts -> foldl1 combine_bounds (map bounds parts)
where
    foldl1 :: (a a -> a) [a] -> a
    foldl1 f [x:xs] = foldl f x xs

    combine_bounds :: Rectangle Rectangle -> Rectangle
    combine_bounds ((tl1x,tl1y),(br1x,br1y)) ((tl2x,tl2y),(br2x,br2y))
       = ((min tl1x tl2x,min tl1y tl2y),(max br1x br2x,max br1y br2y))

    normalise :: Rectangle -> Rectangle
    normalise ((x1,y1),(x2,y2))=((min x1 x2, min y1 y2), (max x1 x2, max y1 y2))
```

For the object oriented approach we have to change the definition of `Drawable` and all functions generating objects of this type. We use the same supporting functions as above.

```
::  Drawable = E.a: { state  :: a
                    , move   :: Point a -> a
                    , draw   :: a Picture -> Picture
                    , bounds :: a -> Rectangle                // new
                    }

MakeLine :: Line -> Drawable
MakeLine line = { state  = line
                , move   = \dist line -> line + (dist,dist)
                , draw   = DrawLine
                , bounds = \l -> normalise l                  // new
                }

MakeCurve :: Curve -> Drawable
MakeCurve curve =   { state  = curve
                    , move   = \dist (rect,a1,a2) -> (rect + (dist,dist),a1,a2)
                    , draw   = DrawCurve
                    , bounds = \c -> curveBounds c            // new
                    }

MakeRectangle :: Rectangle -> Drawable
MakeRectangle ((x1,y1),(x2,y2))
 =  { state  = [ MakeLine ((x1,y1),(x1,y2)), MakeLine ((x1,y2),(x2,y2))
               , MakeLine ((x2,y2),(x2,y1)), MakeLine ((x2,y1),(x1,y1))
               ]
```

```
        , draw   = \s p -> foldl (\pict {state,draw} -> draw state pict) p s
        , move   = \d -> map (MoveDrawable d)
        , bounds = \parts -> foldl1 combine_bounds                    // new
                             (map (\{state,bounds} -> bounds state) parts)
        }

    MakeWedge :: Curve -> Drawable
    MakeWedge curve=:((tl,br), a1, a2)
     =  { state = [ MakeLine (mid, mid+epc1)
                  , MakeLine (mid, mid+epc2)
                  , MakeCurve curve
                  ]
        , draw   = \s p -> foldl (\pict {state,draw} -> draw state pict) p s
        , move   = \d -> map (MoveDrawable d)
        , bounds = \parts -> foldl1 combine_bounds                    // new
                             (map (\{state,bounds} -> bounds state) parts)
        }
    where
        mid = (tl + br)/ (2,2)
        (epc1,epc2) = EndPointsOfCurve curve
```

It is not possible to give a general rule when to use either the object-oriented approach or the algebraic datatype approach. As the examples above show both approaches have their advantages and disadvantages. Adding a new object type, or changing a specific object type is very easy and local in the object oriented approach. On the other hand, adding a new manipulation function, `method', is easy and much more local in the algebraic datatype approach. The decision should be based on the expected use and changes of the datatype. Fortunately, usually neither of the choices is really wrong. It is merely a matter of convenience.

A pipeline of functions

Existentially quantified data structures can also be used to solve the following problem. Consider the function `seq` which applies a sequence of functions to a given argument (see also Chapter 5).

```
seq :: [t->t] t -> t
seq []     s = s
seq [f:fs] s = seq fs (f s)
```

Since all elements of a list must have the same type, only (very) limited sequences of functions can be composed with `seq`. In general it is not possible to replace `f o g o h` by `seq [h, g, f]`. The types of the argument and the final result as well as the types of all intermediate results might all be different. However, by applying the `seq` function all those types are forced to become the same.

Existential types make it possible to hide the actual types of all intermediate results, as shown by the following type definition.

```
:: Pipe a b = Direct (a->b)
            | E.via: Indirect (a->via) (Pipe via b)
```

Using this `Pipe` datatype, it is possible to compose arbitrary functions in a real pipe-line fashion. The only restriction is that types of two consecutive functions should match. The function `ApplyPipe` for applying a sequence of functions to some initial value is defined as follows.

```
ApplyPipe:: (Pipe a b) a -> b
ApplyPipe (Direct f)        x   = f x
ApplyPipe (Indirect f pipe) x   = ApplyPipe pipe (f x)
```

The program:

```
Start :: Int
Start = ApplyPipe (Indirect toReal (Indirect exp (Direct toInt))) 7
```

is valid. The result is `1097`.

4.3 Uniqueness types

A very important property for reasoning about and analysing functional programs is *referential transparency*: functions always return the same result when called with the same arguments. Referential transparency makes it possible to reason about the evaluation of a program by substituting an application of a function with arguments by the functions definition in which for each argument in the definition uniformly the corresponding argument of the application is substituted. This principle of *uniform substitution*, which is familiar from high school mathematics, is vital for reasoning about functional programs.

Imperative languages like C, C++ and PASCAL allow data to be updated destructively. This feature is not only important for reasons of efficiency (the memory reserved for the data is re-used again). The possibility to destructively overwrite data is a key concept on any computer. E.g. one very much would like to change a record stored in a database or the contents of a file. Without this possibility no serious program can be written. Incorporating destructive updating without violating referential transparency property of a functional program takes some effort.

The price we have to pay in imperative languages is that there is no referential transparency; the value of a function application can be dependent on the effects of the program parts evaluated previously. These side-effects makes it very complicated to reason about imperative programs. Uniqueness types are a possibility to combine referential transparency and destructive updates.

4.3.1 Graph Reduction

Until now we have not been very precise about the model of computation used in the functional language CLEAN. Since the number of references to an expression is important to determine whether it is unique or not, we must become a little bit more specific.

The basic ingredients of execution, also called *reduction*, have been discussed. The first principle we have seen is *uniform substitution*: an application of a function with arguments is replaced by the function definition in which for each argument in the definition uniformly the corresponding argument of the application is substituted. The second principle is *lazy evaluation*: an expression is only evaluated when its value is needed to compute the initial expression.

Now we add the principle of *graph reduction*: all occurrences of a variable are replaced by one and the *same* expression during uniform substitution. The variables are either formal function arguments or expressions introduced as local definition. This implies that expressions are never copied and hence an expression is evaluated at most once. The corresponding variables can share the computed result. This is a sound thing to do due to referential transparency: the value of an expression is independed of the context in which it is evaluated. Due to the referential transparency there is no semantical difference between uniform substitution by copying or by sharing. Reduction of expressions that can be shared is called graph reduction. Graph reduction is generally more efficient than ordinary reduction because it avoids re-computation of expressions.

Graph reduction is illustrated by the following examples. A reduction step is indicated by the symbol →, a sequence of reduction steps is indicated by →*. Whenever we find it useful we underline the *redex* (*reducible expression*): the expression to be rewritten. Local definitions are used to indicate sharing.

```
Start = 3*7 + 3*7

Start
→  3*7 + 3*7
→  3*7 + 21
→  21 + 21
→  42
```

```
Start = x + x where x = 3*7

Start
→  x + x where x = 3*7
→  x + x where x = 21
→  42
```

Note that the sharing introduced in the rightmost program saves some work. These reduction sequences can be depicted as:

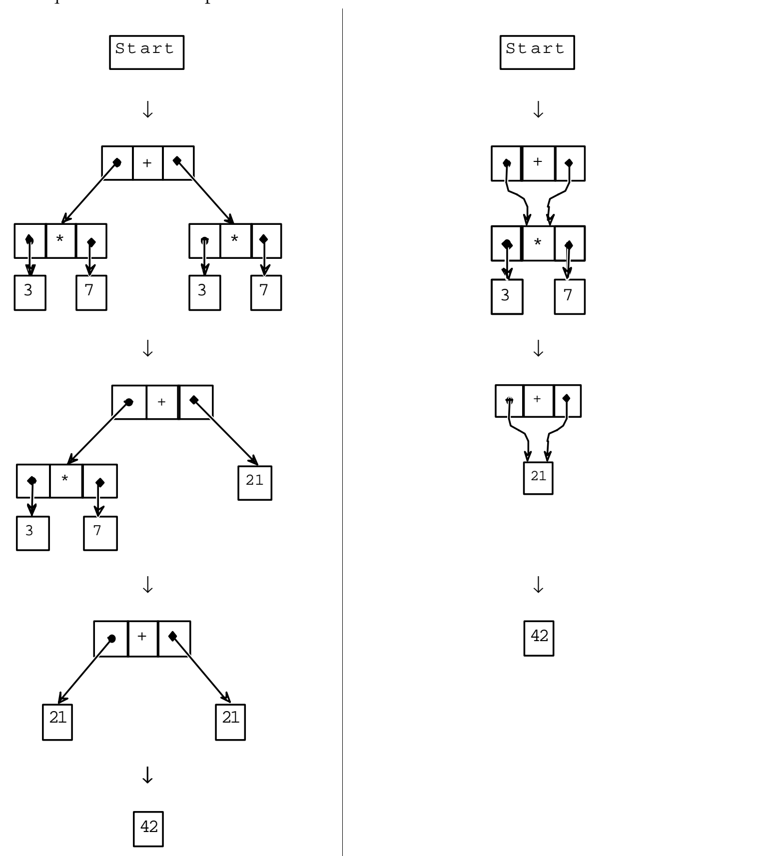


Figure 4.1: Pictorial representation of the reduction sequences shown above.

An other example where some work is shared is the familiar power function.

```
power :: Int Int -> Int
power x 0 = 1
power x n = x * power x (n-1)

Start :: Int
Start = power (3+4) 2
```

This program is executed by the following sequence of reduction steps.

```
Start
→  power (3+4) 2
→  x * power x (2-1) where x = 3+4
→  x * power x 1 where x = 3+4
→  x * x * power x (1-1) where x = 3+4
→  x * x * power x 0 where x = 3+4
→  x * x * 1 where x = 3+4
→  x * x * 1 where x = 7
```



```
→  x * 7 where x = 7
→  49
```

The number of references to an expression is usually called the *reference count*. From this example it is clear that the reference count can change dynamically. Initially the reference count of the expression `3+4` is one. The maximum reference count of this node is three. The result, the reduct, of the expression `3+4`, `7`, is used at two places.

4.3.2 Destructive updating

Consider the special data structure that represents a file. This data structure is special since it represents a structure on a disk that (usually) has to be updated destructively. So, a program manipulating such a data structure is not only manipulating a structure inside the program but it is also manipulating a structure (e.g. a file) in the outside world. The CLEAN run-time system takes care of keeping the real world object and the structure inside your program up to date. In your program you just manipulate the data structure.

Assume that one would have a function `fwritec` that appends a character to an existing file independent of the context from which it is called and returns the modified file. So, the intended result of such a function would be a file with the extra character in it:

```
fwritec :: Char File -> File
```

Such a function could be used by other functions:

```
AppendA :: File -> File
AppendA file = fwritec 'a' file
```

In fact, `File` is an abstract datatype similar to `stack` introduced in section 3.7. A function to push an `'a'` to the stack is:

```
pushA :: (Stack Char) -> Stack Char
pushA stack = push 'a' stack
```

We can push two characters on the stack by:

```
pushAandB :: (Stack Char) -> Stack Char
pushAandB stack = push 'b' (push 'a' stack)
```

In exactly the same way we can write two characters to a file:

```
AppendAandB :: File -> File
AppendAandB file = fwritec 'b' (fwritec 'a' file)
```

Problems with destructive updating

The fact that the abstract datatype `File` is mapped to a file on disc causes special care. Let us suppose that the following function `AppendAB` could be defined in a functional language.

```
AppendAB :: File -> (File, File)
AppendAB file = (fileA, fileB)
where
fileA = fwritec 'a' file
    fileB = fwritec 'b' file
```

What should then be the contents of the files in the resulting tuple `(fileA, fileB)`? There seem to be only two solutions, which both have unwanted properties.

The first solution is to assume that `fwritec` destructively changes the original file by appending a character to it (like in imperative languages). However, the value of the resulting tuple of `AppendAB` will now depend on the order of evaluation. If `fileB` is evaluated before `fileA` then `'b'` is appended to the file before `'a'`. If `fileA` is evaluated before `fileB` then the `'a'` will be written before `'b'`. This violates the rule of referential transparency in functional programming languages. So, just overwriting the file is rejected since loosing referential transparency would tremendously complicate analysing and reasoning about a program.

The second solution would be that in conformity with referential transparency the result is a tuple with two files: one extended with a character `'a'` and the other with the character `'b'`. This does not violate referential transparency because the result of the function calls

AppendA file and AppendB file is not influenced by the context. This means that each function call fwritec would have to be applied on a 'CLEAN' file, which in turn would mean that for the function call AppendAB two copies of the file have to be made. To the first copy the character 'a' is appended, and to the second copy the character 'b' is appended. If the original of the file is not used in any other context, it can be thrown away as garbage.

One could implement such a file by using a stack. For instance the function:

```
pushAB :: (Stack Char) -> (Stack Char, Stack Char)
pushAB stack = (push 'b' stack, push 'a' stack)
```

yields a tuple of two stacks. Although these stacks might be partially shared, there are now conceptually two stacks: on with a 'b' on top and another one with an 'a' on top.

This second solution however, does not correspond to the way operating systems behave in practice. It is rather impractical. This becomes even more obvious when one wants to write to a window on a screen: one would like to be able to perform output in an existing window. Following this second solution one would be obliged to construct a new window with each outputting command.

So, it would be nice if there would be a way to destructively overwrite files and the like *without* violating referential transparency. We require that the result of any expression is well defined *and* we want to update files and windows without making unnecessary copies.

4.3.4 Uniqueness information

The way to deal with this problem may be typical for the way language designers think: "If you don't like it, you don't allow it". So, it will not be allowed to update a data structure representing a real world object when there is more than one references to it. The definition of AppendAB above should therefore be rejected by the compiler.

Assume that we are able to guarantee that the reference count of the file argument of fwritec is always exactly one. We say that such an argument is *unique*. Now, when we apply fwritec to such a unique file we can observe the following. Semantically we should produce a new file. But we know that no other expression can refer to the old file: only fwritec has a reference to it. So, why not reuse the old file passed as argument to fwritec to construct the new file? In other words: when old file is unique it can simply be updated destructively by fwritec to produce the new file in the intended efficient and safe way.

Although the definition of AppendAB as shown above will be forbidden, it is allowed to write down the following:

```
WriteAB :: File -> File
WriteAB file = fileAB
where
    fileA  = fwritec 'a' file
    fileAB = fwritec 'b' fileA
```

Here, the data dependency is used to determine the order in which the characters are appended to the file (first 'a', then 'b'). This solution is semantically equal to the function AppendAandB introduced above. This programming style is very similar to the classical imperative style, in which the characters are appended by sequential program statements. Note however that the file to which the characters are appended is explicitly passed as an argument from one function definition to another. This technique of passing around of an argument is called *environment passing*. The functions are called *state transition* functions since the environment that is passed around can be seen as a state which may be changed while it is passed. The functions are called *state transition* functions since the environment which is passed can be seen as a state which may be changed while it is passed.

4.3.5 Uniqueness typing

Of course, somehow it must be guaranteed (and specified) that the environment is passed properly i.e. in such a way that the required updates are possible. For this purpose a type system is designed which derives the *uniqueness properties*. A function is said to have an argument of *unique type* if there will be just a single reference to the argument when the function will be evaluated. This property makes it safe for the function to re-use the memory consumed by the argument when appropriate.

In figure 4.1 all parts of the example of the left-hand side are unique. On the right-hand side the expression 3*7 is not unique since it is shared by both arguments of the addition. By drawing some pictures, it is immediately clear that the function WriteAB introduced above uses the file unique, while in AppendAB the reference count of the file is 2. Hence, the function AppendAB is rejected by the compiler.

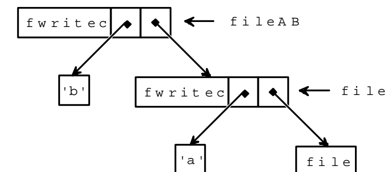


Figure 4.2: The result of WriteAB file

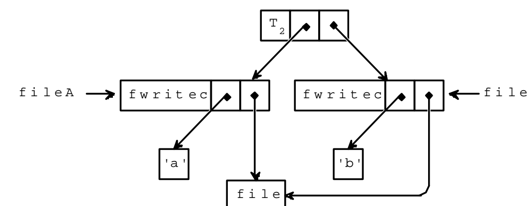


Figure 4.3: The result of AppendAB file

The function fwritec demands its second argument, the file, to be of unique type (in order to be able to overwrite it) and consequently it is derived that WriteAB must have a unique argument too. In the type this uniqueness is expressed with an asterisk which is attached as an attribute to the conventional type. This asterisk is used by the compiler; the type is only approved by the compiler when it can determine that the reference count of the corresponding argument will be exactly one when the function is executed.

```
fwritec :: Char *File -> *File
WriteAB :: *File -> *File
```

The uniqueness type system is an extension on top of the conventional type system. When in the type specification of a function an argument is attributed with the type attribute unique (*) it is guaranteed by the type system that, upon evaluation of the function, the function has private ("unique") access to this particular argument.

The correctness of the uniqueness type is checked by the compiler, like all other type information (except strictness information). Assume now that the programmer has defined the function AppendAB as follows:

```
AppendAB :: File -> (File, File)
AppendAB file = (fileA, fileB)
where
    fileA = fwritec 'a' file
    fileB = fwritec 'b' file
```

The compiler will reject this definition with the error message:

```
<conflicting uniqueness information due to argument 2 of fwritec>
```

This rejection of the definition is caused by the non-unique use of the argument file in the two local definitions (assuming the type `fwritec :: Char *File -> *File`).

It is important to know that there can be many references to the object before this specific access takes place. For instance, the following function definition will be approved by the type system, although there are two references to the argument file in the definition. When `fwritec` is called, however, the reference is unique.

```
AppendAorB :: Bool *File -> *File
AppendAorB cond file
    | cond      = fwritec 'a' file
    | otherwise = fwritec 'b' file
```

So, the concept of *uniqueness typing* can be used as a way to specify locality requirements of functions on their arguments: If an argument type of a function, say `F`, is *unique* then in any concrete application of `F` the actual argument will have reference count 1, so `F` has indeed 'private access' to it. This can be used for defining (inherent) destructive operations like the function `fwritec` for writing a character to a file.

Observe that uniqueness of result types can also be specified, allowing the result of an `fwritec` application to be passed to, for instance, another call of `fwritec`. Such a combination of uniqueness typing and explicit environment passing will guarantee that at any moment during evaluation the actual file has reference count 1, so all updates of the file can safely be done in-place. If no uniqueness attributes are specified for an argument type (e.g. the `Char` argument of `fwritec`), the reference count of the corresponding actual argument is generally unknown at run-time. Hence, no assumption can be made on the locality of that argument: it is considered as *non-unique.*

Offering a unique argument if a function requires a non-unique one is safe. More technically, we say that a unique object can be *coerced* to a non-unique one. Assume, for instance, that the functions `freadc and fwrites` have type

```
freadc:: File -> (Bool, Char, File)     // The Boolean indicates success or failure
fwrites :: String *File -> *File.
```

in the application

```
readwrite :: String *File -> (Bool, Char, File)
readwrite s f = freadc (fwrites s f)
```

the (unique) result `File` of `fwrites` is coerced to a non-unique one before it is passed to `freadc`.

Of course, offering a non-unique object if a function requires a unique one always fails. For, the non-unique object is possible shared, making a destructive update not well-defined. Note that an object may lose its uniqueness not only because uniqueness is not required by the context, but also because of sharing. This, for example, means that although an application of `fwritec` itself is always unique (due to its unique result type), it is considered as non-unique if there exist more references to it. To sum up, the *offered* type (by an argument) is determined by the *result* type of its outermost application and the reference count of that argument.

Until now, we distinguished objects with reference count 1 from objects with a larger reference count: only the former might be unique (depending on the object type itself). As we have seen in the example above the reference count is computed for each right-hand side separately. When there is an expression in the guards requiring an unique object this must be taken into account. This is the reason we have to write:

```
AppendAorB:: *File -> *File
AppendAorB file
    | fc == 'a' = fwritec 'a' nf
    | otherwise = fwritec 'b' nf
```



```
where
    (_,fc,nf) = sfreadc file
```

The function `sfwritec` reads a character form a unique file and yields a unique file:

```
sfreadc:: *File -> (Bool, Char, *File)
```

We assume here that reading a character from a file always succeeds. When the right-hand side of `AppendAorB` is evaluated, the guard is determined first (so the resulting access from `sfreadc` to `file` is not unique), and subsequently one of the alternatives is chosen and evaluated. Depending on the condition `fc == 'a'`, either the reference from the first `fwritec` application to `nf` or that of the second application is left unused, therefore the result is unique. As you might expect it is not allowed to use `file` instead of `nf` in the right-hand sides of the function `AppendAorB`. File manipulation is explained in more detail in chapter 5.

4.3.5 Nested scope style

A convenient notation for combining functions which are passing around environments, is to make use of nested scopes of *let-before* definitions (indicated by `let` or `#`). In that style the example `WriteAB` can be written as follows:

```
WriteAB :: File -> File
WriteAB file
    # fileA  = fwritec 'a' file
    # fileAB = fwritec 'b' fileA
    = fileAB
```

Let-before expressions have a special scope rule to obtain an imperative programming look. The variables in the left-hand side of these definitions do not appear in the scope of the right-hand side of that definition, but they do appear in the scope of the other definitions that follow (including the root expression, excluding local definitions in where blocks. This is shown in the figure 4.4:

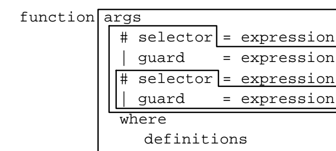


Figure 4.4: Scope of let-before expressions

Note that the scope of variables in the let before expressions does not extent to the definitions in the where expression of the alternative. The reverse is true however: definitions in the where expression can be used in the let before expressions.

Due to the nesting of scopes of the let-before the definition of `WriteAB` can be written as follows:

```
WriteAB :: File -> File
WriteAB file
    # file = fwritec 'a' file
    # file = fwritec 'b' file
    = file
```

So, instead of inventing new names for the intermediate files (`fileA`, `fileAB`) one can reuse the name `file`. The nested scope notation can be very nice and concise but, as is always the case with scopes, it can also be dangerous: the same name `file` is used on different spots while the meaning of the name is not always the same (one has to take the scope into account which changes from definition to definition). However, reusing the same name is rather safe when it is used for a threaded parameter of unique type. The type system will spot it (and reject it) when such parameters are not used in a correct single threaded man-

ner. We certainly do not recommend the use of let before expressions to adopt a imperative programming style for other cases.

The scope of the variables introduces by the `#`-definitions is the part of the right-hand side of the function following the `#`-definition. The right-hand side `#`-definition and the `where`-definitions are excluded from this scope. The reason to exclude the right-hand of the `#`-definition is obvious from the example above. When the body of the `#`-definition is part of the scope the variable `file` would be a circular definition. The reason to exclude the `where`-definitions is somewhat trickier. The scope of the `where`-definitions is the entire right-hand side of the function alternative. This includes the `#`-definitions. This implies that when we use the variable `file` in a `where`-definition of `WriteAB` it should be the original function argument. This is counter intuitive, you expect `file` to be the result of the last `freadc`. When you need local definitions in the one of the body of such a function you should use `let` or `with`. See the language manual and chapter 6 for a more elaborate discussion of the various local definitions.

4.3.6 Propagation of uniqueness

Pattern matching is an essential aspect of functional programming languages, causing a function to have access to 'deeper' arguments via 'data paths' instead of via a single reference. For example, the `head` function for lists, which is defined as

```
head :: [a] -> a
head [x:xs] = x
```

has (indirect) access to both `x` and `xs` This deeper access gives rise to, what can be called, indirect sharing: several functions access the same object (via intermediate data constructors) in spite of the fact that the object itself has reference count 1. Consider, for example the function `heads` which is defined as follows.

```
heads list = (head list, head list)
```

In the right-hand side of `heads`, both applications of `head` retrieve their result via the same list constructor. In the program

```
Start = heads [1,2]
```

the integer `1` does not remain unique, despite the fact that it has reference count 1. In this example the sharing is indicated by the fact that `list` has reference count 2. Sharing can be even more hidden as in:

```
heads2 list=:[x:_] = (head list, x)
```

If one wants to formulate uniqueness requirements on, for instance, the head argument of `head`, it is not sufficient to attribute the corresponding type variable `a` with `*`; the surrounding list itself should also become unique. One could say that uniqueness of list elements *propagates outwards*: if a list contains unique elements, the list itself should be unique as well. One can easily see that, without this propagation requirement, locality of object cannot be guaranteed anymore. E.g., suppose we would admit the following type for `head`.

```
head :: [*a] -> *a
```

Then, the definition of `heads` is typable, for, there are no uniqueness requirements on the direct argument of the two `head` applications. The type of `heads` is:

```
heads :: [*a] -> (*a,*a)
```

which is obviously not correct because the same object is delivered twice. However, applying the uniqueness propagation rule leads to the type

```
head :: *[*a] -> *a
```

Indeed, this excludes sharing of the list argument in any application of `head`, and therefore the definition of `heads` is no longer valid. This is exactly what we need.

In general, the propagation rule reflects the idea that if an unique object is stored in a larger data structure, the latter should be unique as well. This can also be formulated like: *an object stored inside a data structure can only be unique when the data structure itself is unique as well.*

In practice, however, one can be more liberal when the evaluation order is taken into account. The idea is that multiple references to an (unique) object are harmless if one knows that only one of the references will be present at the moment the object is accessed destructively. For instance, the compiler 'knows' that only one branch of the predefined conditional function `if` will be used. This implies that the following function is correct.

```
transform :: (Int -> Int) *{#Int} -> *{#Int}
transform f s
 | size s == 0 = s
 | otherwise   = if (s.[0] == 0)
                   {f i \\ i <-: s}
                   {f i \\ _ <-: s & i <- [s.[0]..]}
```

This example shows also that uniqueness of objects is not restricted to files and windows. Since arrays are usually large, it is in CLEAN only allowed to update unique arrays. Using the space of the old array, such an update can be done very efficient.

4.3.7 Uniqueness polymorphism

To indicate that functions leave uniqueness properties of arguments unchanged, one can use (*uniqueness*) *attribute variables*. The most simple example is the identity function which can be typed as follows:

```
id :: u:a -> u:a
```

Here `a` is an ordinary type variable, whereas `u` is an attribute variable. If `id` is applied to an unique object the result is also unique (in that case `u` is instantiated with the concrete attribute `*`). Of course, if `id` is applied to a non-unique object, the result remains non-unique. Note that we tacitly assume an attribute for 'non-unique' although there exists no denotation for it in CLEAN.

The next example shows that it is necessary to distinguish between type variables and attribute variables.

```
tup :: a -> (a, a)
tup x = (x, x)
```

Even if the argument of `tup` is unique, in the result this expression will be shared.

A more interesting example is the function `freadc` which is typed as

```
freadc :: u:File -> (Bool, Char, u:File)
```

Again `freadc` can be applied to both unique and non-unique files. In the first case the resulting file is also unique and can, for example, be used for further reading as well as for writing. In the second case the resulting file is also not unique, hence write access is not permitted.

One can also indicate relations between attribute variables appearing in the type specification of a function, by adding so called *coercion statements*. These statements consist of attribute inequalities of the form `u <= v`. The idea is that attribute substitutions are only allowed if the resulting attribute inequalities are valid, i.e. not resulting in an equality of the form

'non-unique □ unique'.

The use of coercion statements is illustrated by the next example in which the uniqueness type of the well-known append operator `++` is shown.

```
(++) infixr 5 :: v:[u:a] w:[u:a] -> x:[u:a], [v w x<=u, w<=x]
```

The first coercion statement expresses uniqueness propagation for lists: if the elements `a` are unique (by choosing `*` for `u`) these statements force `v`, `w` and `x` to be instantiated with `*` as well. Note that `u <= *` iff `u = *`. The latter statement `w<=x` expresses that the spine uniqueness of `append`'s result depends only on the spine attribute `w` of the second argument. This reflects the operational behaviour of `append`, namely, to obtain the result list, the first list

argument is fully copied, after which the final tail pointer is redirected to the second list argument.

```
(++) [x:xs] list = [x: xs ++ list]
(++) _      list = list
```

In CLEAN it is permitted to omit attribute variables and attribute inequalities that arise from propagation properties; those will be added automatically by the type system. As a consequence, the following type specification for ++ is also valid.

```
(++) infixr 5 :: [u:a] w:[u:a] -> x:[u:a], [w<=x]
```

Of course, it is always allowed to use a more specific type (by instantiating type and/or attribute variables). All types given below are valid types for ++.

```
(++) infixr 5 ::  [u:a] x:[u:a]  -> x:[u:a]
(++) infixr 5 :: *[*Int] *[*Int] ->  *[*Int]
(++) infixr 5 ::  [a]    *[a]    ->  *[a]
```

To make types more readable, CLEAN offers the possibility to use *anonymous* attribute variables as a shorthand for attribute variables of which the actual names are not essential. Using anonymous attributes ++ can be typed as follows.

```
(++) infixr 5 :: [.a] w:[.a] -> x:[.a], [w<=v]
```

This is the type derived by the compiler. The type system of CLEAN will substitute real attribute variables for the anonymous ones. Each dot gives rise to a new attribute variable except for the dots attached to type variables: type variables are attributed uniformly in the sense that all occurrences of the same type variable will obtain the same attribute. In the above example this means that all dots are replaced by one and the same (new) attribute variable.

Finally, we can always use an unique list where an ordinary list is expected. So, it is sufficient to specify the following type for append:

```
(++) infixr 5::[.a] u:[.a] -> u:[.a]
```

Apart from strictness annotations this is the type specified for the append operator in the module `StdList` of the standard environment.

4.3.8 Attributed datatypes

First, remember that types of data constructors are not specified by the programmer but derived from their corresponding datatype definition. For example, the (classical) definition of the `List` type

```
:: List a = Cons a (List a) | Nil
```

leads to the following types for its data constructors.

```
Cons :: a (List a) -> List a
Nil  :: List a
```

To be able to create unique instances of datatypes, a programmer does not have change the corresponding datatype definition itself; the type system of CLEAN will automatically generate appropriate uniqueness variants for the (classical) types of all data constructors. Such a uniqueness variant is obtained via a consistent attribution of all types and subtypes appearing in a datatype definition. E.g., for `Cons` this attribution yields the type

```
Cons :: u:a v:(List u:a) -> v:List u:a, [v<=u]
```

Describing the attribution mechanism in all its details is beyond the scope of this book; the procedure can be found in the reference manual and [Barendsen 93]. The main property is that all type variables and all occurrences of the defined type constructor, say `T`, will receive an attribute variable. Again this is done in a uniform way: equal variables will receive equal attributes, and the occurrences of `T` are equally attributed as well. Besides that, attribute variables are added at non-variable positions if they are required by the propagation properties of the corresponding type constructors (see example below). The coercion statements are, as usual, determined by the propagation rule. As an example, consider the following `Tree` definition.


```
:: Tree a = Node a [Tree a]
```

The type of the data constructor `Node` is

```
Node :: u:a w:[v:Tree u:a] -> v:Tree u:a, [v<=u, w<=v]
```

Observe that the uniqueness variable `w` and the coercion statement `[w<=v]` are added since the list type 'propagates' the `v` attribute of its element.

One can also specify that a part of datatype definition, should receive the same attribute as the defined type constructor, say `T`, itself. For this purpose the anonymous (.) attribute variable is reserved, which can be attached to any (sub) type on the right-hand of `T`'s definition. The idea is that the dot attribute denotes the same attribute as the one assigned to the occurrences of `T`. This is particularly useful if one wants to store functions into data structures; see also the next section on higher-order uniqueness typing. For example, the following definition of `Tree`

```
:: Tree2 = Node2 .Int [Tree2]
```

causes the type for the data constructor `Node` to be expanded to

```
Node2 :: u:Int v:[u:Tree2] -> u:Tree2, [v<=u]
```

Unfortunately, the type attribute `v` is not used in the result of the constructor `Node`. Hence, there is no way to store the uniqueness of the arguments of `Node` in its type. So, in contrast with the type `List`, it is not possible to construct unique instances of the type `Tree`. This implies that the function to reverse trees

```
swap (Node a leafs) = Node a [swap leaf \\ leaf <- rev leafs]

rev :: [.a] -> [.a]
rev list = rev_ list []
where  rev_ [x:xs] list = rev_ xs [x:list]
       rev_ []     list = list
```

obtains type

```
swap :: (Tree a) -> Tree a
```

instead of

```
swap :: u:(Tree .a) -> u:(Tree .a)
```

This implies that an unique tree will loose its uniqueness attribute when it is swapped by this function `swap`. Due to the abbreviations introduced above the last type can also be written as:

```
swap :: (Tree .a) -> (Tree .a)
```

When we do need unique instances of type `Tree`, we have to indicate that the list of trees inside a node has the same uniqueness type attribute as the entire node:

```
:: Tree a = Node a .[Tree a]
```

Now the compiler will derive and accept the type that indicates that swapping an unique tree yields an unique tree: `swap :: (Tree .a) -> (Tree .a)`.

When all trees ought to be unique we should define

```
:: *Tree a = Node a [Tree a]
```

The corresponding type of the function `swap` is

```
swap :: *(Tree .a) -> *Tree .a
```

In practice the pre-defined attribution scheme appears to be too restrictive. First of all, it is convenient if it is allowed to indicate that certain parts of a data structure are always unique. Take, for instance, the type `Progstate`, defined in chapter 5 containing the (unique) file system of type `Files`.

```
:: *ProgState = {files :: Files}
```

According to the above attribution mechanism, the `Files` would have been non-unique. To circumvent this problem, one can make `Progstate` polymorphic, that is to say, the definition `Progstate` becomes

```
:: Progstate file_system = {files :: file_system}
```

Then, one replaces all uses of `Progstate` by `Progstate *Files`. This is, of course, a bit laborious, therefore, it is permitted to include `*` attributes in datatype definitions themselves. So, the definition of `Progstate`, is indeed valid. Note that, due to the outwards propagation of the `*` attribute, `Progstate` itself becomes unique. This explains the `*` on the left-hand side of the definition of `Progstate`.

4.3.9 Higher order uniqueness typing

Higher-order functions give rise to partial (often called *curried*) applications (of functions as well as of data constructors), i.e. applications in which the actual number of arguments is less than the arity of the corresponding symbol. If these partial applications contain unique sub-expressions one has to be careful. Consider, for example the following function `fwritec` with type `fwritec :: *File Char -> *File` in the application `(fwritec unifile)` (assuming that `unifile` returns a unique file). Clearly, the type of this application is of the form `o:(Char -> *File)`. The question is: what kind of attribute is `o`? Is it a variable, is it `*`, or is it 'not unique'. Before making a decision, we will illustrate that it is dangerous to allow the above application to be shared. For example, if `(fwritec unifile)` is passed to a function

```
WriteAB write_fun = (write_fun 'a', write_fun 'b')
```

Then the argument of `fwritec` is not longer unique at the moment one of the two write operations takes place. Apparently, the `(fwritec unifile)` expression is *essentially* unique: its reference count should never become greater than 1. To prevent such an essentially unique expression from being copied, the uniqueness type system considers the `->` type constructor in combination with the `*` attribute as special: it is not permitted to discard its uniqueness. Now, the question about the attribute `o` can be answered: it is set to `*`. If `WriteAB` is typed as follows

```
WriteAB :: (Char -> u:File) -> (u:File, u:File)
WriteAB write_fun = (write_fun 'a', write_fun 'b')
```

the expression `WriteAB (fwritec unifile)` is rejected by the type system because it does not allow the actual argument of type `*(Char -> *File)` to be coerced to `(Char -> u:File)`. One can easily see that it is impossible to type `WriteAB` in such a way that the expression becomes type-able. This is exactly what we want for files.

To define data structures containing curried applications it is often convenient to use the (anonymous) dot attribute. Example

```
:: Object a b = { state :: a
                , fun   :: .(b -> a)
                }

new :: *Object *File Char
new = {state = unifile, fun = fwritec unifile}
```

Since `new` contains an essentially unique expression it becomes essentially unique itself. So, the result of `new` can only be coerced to a unique object implying that in any containing `new`, the attribute type requested by the context has to be unique.

Determining the type of a curried application of a function (or data constructor) is somewhat more involved if the type of that function contains attribute variables instead of concrete attributes. Mostly, these variables will result in additional coercion statements. as can be seen in the example below.

```
Prepend :: u:[.a] [.a] -> v:[.a], [u<=v]
Prepend a b = Append b a

PrependList :: u:[.a] -> w:([.] -> v:[.a]), [u<=v, w<=u]
PrependList a = Prepend a
```

Some explanation is in place. The expression `PrependList some_list` yields a function that, when applied to another list, say `other_list`, delivers a new list consisting of the concatenation of `other_list` and `some_list`. Let us call this final result `new_list`. If `new_list` should be unique (i.e. `v` becomes `*`) then, because of the coercion statement `u<=v` the attribute `u` becomes `*` as well. But, if `u = *` then also `w = *`, for, `w<=u`. This implies that (arrow) type of the original expression `PrependList some_list` becomes unique, and hence this expression cannot be shared. The general rule for determining the uniqueness type of curried variants of (function or data) symbols is quite complex, it can be found in the reference manual and [Barendsen 93].

4.3.10 Creating unique objects

In the preceding subsections we showed how unique objects can be manipulated. The question that remains is how to become the initial unique object. All unique objects corresponding with real world entities, like files and windows, are retrieved from the `world`. This is explained in detail in chapter 5.

It is also possible to have unique objects of other datatypes. Especially for arraysit is useful to have unique instances, since only unique arrays can be updated destructively. Such data structures are very useful in cases ultimate efficiency is required.

4.4 Exercises

1. Define an instance for type `Q` of the standard class `Arith`.

   ```
   class Arith a | PlusMin, MultDiv, abs, sign, ~ a
   ```

2. Define complex numbers similar to `Q` and specify an instance of the class `Arith` for this new type.
3. Define an instance of `PlusMin` for lists `[a]` such that, for instance, the addition of two lists takes place element wise (if necessary, the shortest list is extended with zeros to obtain two lists of equal length). So, `[1,2,3] + [4,5] = [5,7,3]`.
4. Why should a `Pipe` object contain at least one function (each `Pipe` object ends with a `Direct` constructor containing the final function to be applied)? One can image a definition of `Pipe` with a kind of `Nil` constructor with no argument as a terminator.

Part I
Chapter 5
Interactive Programs

5.1 Changing files in the World
5.2 Environment Passing Techniques
5.3 Handling Events
5.4 Dialogs
5.5 Windows
5.6 Timers
5.7 A Line Drawing Program
5.8 Exercises

In the previous Chapter we showed how uniqueness typing can be used to manipulate objects like files and windows in a safe manner. In this chapter it is described in more detail how this can be used to do file I/O. Furthermore we explain how interactive window-based programs can be written in a pure functional language like Clean.

Writing window applications is not a simple task. To make life easier a large library has been written in Clean (the Object I/O library) which offers functions and data structures with which interactive applications can be described platform independent on a high level of abstraction. This chapter contains several examples to illustrate the functionality offered by this CLEAN library.

An important advantage of the CLEAN library is that the same program can run without any change on different machines (e.g. Macintosh, Windows '98). On each machine the resulting menus, dialogs and windows adhere to the look and feel which is common on that machine.

Writng window-based interactive applications is certainly not a trivial task. For programmers who want to write real world applications using the CLEAN Object I/O library we highly recommend reading Peter Achten's Object I/O reference manual (for the latest information see www.cs.kun.nl/~clean).

5.1 Changing files in the World

Suppose we want to write a program that copies a complete file. It is easy to define an extension of the examples in the previous Chapter such that not just one or two characters are written but a complete list of characters is copied to a file. Combining this with the function to read characters from a file gives a function to copy a file.

```
CharListWrite :: [Char] *File -> *File
CharListWrite []     f = f
CharListWrite [c:cs] f = CharListWrite cs (fwritec c f)

CharFileCopy :: File *File -> *File
CharFileCopy infile outfile = CharListWrite (CharListRead infile) outfile
```

Reading characters from a file requires a few more lines than writing since for reading not only an environment (the file) has to be passed but also a result has to be yielded. The file from which characters are read is not required to be unique since no destructive update is involved in reading.


```
CharListRead :: File -> [Char]
CharListRead f
    | not readok = []
    | otherwise  = [char : CharListRead filewithchangedreadpointer]
where
    (readok,char,filewithchangedreadpointer) = sfreadc f
```

The read function is lazy. So, character by character the file will be read when the characters are needed for the evaluation (although the actual library implementation of `sfreadc` will probably use some kind of buffering scheme).

This completes the file copy function, but we do not have a file copy *program* yet. What is missing are functions to open and close the files in question and, of course, we have to arrange that the file system is accessible. A copy function that also opens and closes files is given below. It takes the unique file system as argument and yields it as a result. The file system is modelled by the abstract datatype `*Files`.

```
CopyFile :: String String *Files -> *Files
CopyFile inputfname outputfname filesys
    |   readok && writeok && closeok
              = finalfilesystem
    |   not readok  = abort ("Cannot open input file: '"   +++ inputfname  +++ "'")
    |   not writeok = abort ("Cannot open output file: '"  +++ outputfname +++ "'")
    |   not closeok = abort ("Cannot close output file: '" +++ outputfname +++ "'")
where
    (readok,inputfile,touchedfilesys)  = sfopen inputfname FReadText filesys
    (writeok,outputfile,nwfilesys)     = fopen outputfname FWriteText touchedfilesys
    copiedfile                         = CharFileCopy inputfile outputfile
    (closeok,finalfilesystem)          = fclose copiedfile nwfilesys
```

Using nested scopes, the function `CopyFile` can be written a bit more elegant. We do not have to invent names for the various 'versions' of the file system. Note that this version is only syntactically different from the previous function. The various versions of the file system still exist, but all versions have the same name. The scopes of the #-definitions determine which version is used.

```
CopyFile :: String String *Files -> *Files
CopyFile inputfname outputfname files
  # (readok,infile,files)   = sfopen inputfname FReadText files
  | not readok              = abort ("Cannot open input file: '"  +++ inputfname  +++ "'")
  # (writeok,outfile,files) = fopen outputfname FWriteText files
  | not writeok             = abort ("Cannot open output file: '" +++ outputfname +++ "'")
  # copiedfile              = CharFileCopy infile outfile
    (closeok,files)         = fclose copiedfile files
  | not closeok             = abort ("Cannot close output file: '"+++ outputfname +++ "'")
  | otherwise               = files
```

In the definitions the library functions `fopen` and `sfopen` are used to open files. The difference between them is that `fopen` requires the file to be unique and `sfopen` allows sharing of the file. Both functions have argument attributes indicating the way the file is used (`FReadText`, `FWriteText`). Another possible attribute would be `FAppendText`. Similar attributes exist for dealing with files with data.

Accessing the file system itself means accessing the 'outside world' of the program. This is made possible by allowing the `Start` rule to have an abstract parameter `World` which encapsulates the complete status of the machine. Also this world is represented by an abstract datatype.

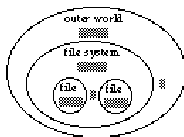


Figure 5.1: The abstract type `World` encapsulating the file system.

Employing unique environment passing, functions have been defined in the library that semantically produce new worlds changing the file system contained in it (`appFiles`). The final result of the CLEAN program is then the world with the changed file system.

```
inputfilename   :== "source.txt"
outputfilename  :== "copy.txt"

Start :: *World -> *World
Start world = appFiles (CopyFile inputfilename outputfilename) world
```

This completes the file copy program. Other ways to read files are line-by-line or megabyte-by-megabyte which may be more appropriate depending on the context. It is certainly more efficient than reading a file character-by-character. The corresponding read-functions are given below.

```
LineListRead :: File -> [String]
LineListRead f
    |   sfend f         = []
    #   (line,filerest) = sfreadline f      // line still includes newline character
    = [line : LineListRead filerest]

MegStringsRead :: File -> [String]
MegStringsRead f
    |   sfend f             = []
    #   (string,filerest)   = sfreads f MegaByte
    =   [string : MegStringsRead filerest]
where
    MegaByte = 1024 * 1024
```

The functions given above are lazy. So, the relevant parts of a file are read only when this is needed for the evaluation of the program.

Sometimes it may be wanted to read a file completely before anything else is done. Below a strict read-function is given which reads in the entire file at once.

```
CharListReadEntireFile :: File -> [Char]
CharListReadEntireFile f
    #   (readok,char,filewithchangedreadpointer) = sfreadc f
    |   not readok  = []
    #!  chars       = CharListReadEntireFile filewithchangedreadpointer
                    = [char : chars]
```

The `#!` construct (a strict let construct) forces evaluation of the defined values independent whether they are being used later or not.

Hello world

A classical and famous exercise in computer science is to create a program that shows the message `hello world` to the user. The simplest CLEAN program that does this is of course:

```
Start = "hello world"
```

The result is displayed on the console (be sure that this option is set in the program environment). A more complicated way to show this message to the user, is by opening the console explicitly. The console is a very simple window. The console is treated just like a file. One can read information from the console and write information to the console by using the read and write functions defined in `StdFile`. By applying read and write function one after another on the console one can achieve an easy synchronization between reading and writing. This is is shown in the program below. To open the "file" console the function `stdio` has to be applied to the world. The console can be closed by using the function `fclose`.


```
module hello1

import StdEnv

Start :: *World -> *World
Start world
    #   (console,world) = stdio world
        console         = fwrites "Hello World" console
        (ok,world)      = fclose console world
    |   not ok          = abort "Cannot close console"
    =   world
```

In this definition we again used the nested scopes of let expressions. One can re-use the names for console and world. Furthermore the order in which actions are written corresponds to the order in which the I/O with the console will happen.

We extend the example a little by reading the name of the user and generating a personal message. Now it becomes clear why the console is used as a single file to do both output and input. Reading the name of the user can only be done after writing the message `"What is your name?"` to the console. The data dependency realized by passing around the unique console automatically establishes the desired synchronization.

```
module hello2

import StdEnv

Start :: *World -> *World
Start world
    #   (console,world) = stdio world
        console         = fwrites "What is your name?\n" console
        (name,console)  = freadline console
        console         = fwrites ("Hello " +++ name) console
        (_,console)     = freadline console
        (ok,world)      = fclose console world
    |   not ok          = abort "Cannot close console"
    |   otherwise       = world
```

In this program we have added a second `readline` action in order to force the program to wait after writing the message to the user.

5.2 Environment Passing Techniques

Consider the following definitions:

```
WriteAB :: *File -> *File
WriteAB file =  fileAB
where
    fileA   = fwritec 'a' file
    fileAB  = fwritec 'b' fileA

WriteAB :: *File -> *File
WriteAB file = fwritec 'b' (fwritec 'a' file)

WriteAB :: *File -> *File
WriteAB file
    #   file = fwritec 'a' file
        file = fwritec 'b' file
    =   file
```

They are equivalent using slightly different styles of programming with environment passing functions.

A disadvantage of the first one is that new names have to be invented: `fileA` and `fileAB`. If such a style is used throughout a larger program one tends to come up with less clear na-

mes such as file1 and file2 (or even file` and file``) which makes it harder to understand what is going on.

The second style avoids this but has as disadvantage that the order of reading the function composition is the reverse of the order in which the function applications will be executed.

The third style uses the nested scope style. It is quite readable but dangerous as well since the same name can be re-used in several definitions. An error is easily made. If the re-use of names is restricted to unique objects like files, the world and the console the type system can detect many kinds of type errors. If also other names are re-used in this style of programming (which is quite similar to an imperative style of programming) typing errors might be introduced which cannot easily be detected by the type system.

Below some other styles of defining the same function are given (for writing characters to a file one of the last styles is preferable since they avoid the disadvantages mentioned above):

```
WriteAB :: (*File -> *File)        // brackets indicate function is defined with arity 0
WriteAB = fwritec 'b' o fwritec 'a'
```

With seq a list of state-transition functions is applied consecutively. The function seq is a standard library function that is defined as follows:

```
seq :: [s->s] s -> s
seq []     x = x
seq [f:fs] x = seq fs (f x)
```

Some alternative definitions of WriteAB using the function seq:

```
WriteAB :: (*File -> *File)
WriteAB = seq [fwritec 'a', fwritec 'b']

WriteAB :: *File -> *File
WriteAB file = seq [fwritec 'a', fwritec 'b'] file

WriteAB :: (*File -> *File)
WriteAB = seq o map fwritec ['ab']
```

A convenient way to write information to a file is by using the overloaded infix operator <<< (see StdFile). It assumes a file on the left-hand-side and a value (of which type an instance of <<< should exist) to write to the file on the right-hand-side. It can be used as follows:

```
WriteAB :: *File -> *File
WriteAB file = file <<< 'a' <<< 'b'

WriteFac :: Int *File -> *File
WriteFac n file = file <<< " The value of fac " <<< n <<< " is " <<< fac n
```

The advantage of using the overloaded <<< operator is that the programmer can define instances for user-defined algebraic data structures.

5.2.1 Nested scope style

Functions of similar type such as the functions above that convert a value to a string to be written to a file can easily be combined by using functions like seq or <<<. But, functions which read information from a file most probably produce results of different type (e.g. freadc reads a character while freads reads a string). It is more complicated to combine such functions. The most direct way to do this is by using the nested scope style.

Assume that one want to read a zip code from a file consisting of an integer value followed by two characters. The following example shows how this can be written in CLEAN.

```
readzipcode :: *File -> ((Int,Char,Char),*File)
readzipcode file
    #   (b1,i, file)   = freadi file
        (b2,c1,file)   = freadc file
        (b3,c2,file)   = freadc file
    |   b1 && b2 && b3 = ((i,c1,c2), file)
    |   otherwise      = abort "readzipcode failure"
```


In the let construct (indicated by #) the results of a read action can be easily specified. The order in which the read actions are performed is controlled by single threadedly passing around the uniquely attributed file parameter.

5.2.2 Monadic style

Thanks to the uniqueness typing of CLEAN it is possible to explicitly pass around destructively objects like files. However, most other functional languages lack such a powerful type system. So, how do they provide the destructive update of an object such as a file without the loss of referential transparency? Well there is neat tric: do not pass around the object explicitly, but pass it around implicitly hidden from the programmer. If the object remains hidden throughout the program it cannot become shared (because no user-defined function explicitly has it as parameter). In this way referential transparency is guaranteed. This approach is taken in the lazy and pure functional language HASKELL. It is called the monadic approach. All updatable objects are hidden in one big state equivalent to the World in CLEAN. In the monadic approach however this state is hidden and never explicitly accessed by the functions that use it. The collection of functions is called a monad Such functions are also called monadic functions. A HASKELL program yields a monadic function to be applied on this hidden monad. Such a function has type (also shown is the type of the IO monad in which the state is hidden):

```
:: Monad_function s a :== s -> (a,s)                  // IO a :== Monad_function World a
```

The main problem to solve is: how can one read values from a file which is hidden in a monad if one does not have access to this hidden monad. The tric is performed by a special operator, called bind, which can combine two monadic functions in a special way.

```
:: St s a :== s -> (a,s)

(`bind`) infix 0 :: (St s a) (a -> (St s b)) -> (St s b)
(`bind`) f_sta a_fstb = stb
where
    stb st =a_fstb a nst
    where
        (a,nst) =   f_sta st

return :: a -> (St s a)
return x =  \s -> (x,s)
```

The bind function ensures that the two monadic functions are applied one after another applied on the monad. And, very clever, the result of the first monadic function is given as additional argument to the second monadic function! In this way the result of the previous monadic function can be inspected by the second one.

In CLEAN one can also use the monadic style of programming (without being forced to actually hide the state monad). The zip-code can be read from a file in a monadic style in the following way:

```
readzipcode :: (*File -> ((Int,Char,Char),*File))      // St *File (Int,Char,Char)
readzipcode
  = freadint  `bind` \(b1,i)->
    freadchar `bind` \(b2,c1->
    freadchar `bind` \(b3,c2)   ->
    if (b1 && b2 && b3) (return (i,c1,c2))
                        (abort "readzipcode failure")
where
    freadint  file = ((b,i),file1) where (b,i,file1) =  freadi file
    freadchar file = ((b,c),file1) where (b,c,file1) =  freadc file
```

This example shows how state-transition functions producing different types of results can be combined in CLEAN in a monadic style of programming.

The advantage of the monadic approach is that no additional type system like uniqueness typing is needed to guarantee safe I/O in a pure functional language. The disadvantage is that, in order to make it impossible for programmers to duplicate “unique” objects, these

objects have to kept hidden from them. Therefore, all objects to be updated are stored into one hidden data structure, the monad, which can only be accessed by the run-time-system only. The monad cannot be accessed directly by the programmer. It can only be accessed and changed indirectly. To do this a programmer has to offer a combination of monadic functions to be applied by the system on the hidden monad. So, the monadic style is less flexible than the solution offered by CLEAN's uniqueness typing system which enables an arbitrary number of updateable objects which can be freely passed around and can be accessed directly. If one wishes one can program in a monadic style in CLEAN too. The monad may but does not have to be hidden from the user. The uniqueness typing can be used to guarantee that the monad is treated correctly.

5.2.3 Tracing program execution

Let's be honest: when programs are getting big a formal proof of the correctness of all functions being used is undoable. In the future one might hope that automatic proof systems become powerful enough to assist the programmer in proving the correctness. However, in reality a very large part of the program might still contain errors despite of a careful program design, the type correctness and careful testing. When an error occurs one has to find out which function is causing the error. In a large program such a function can be difficult to find. So, it might be handy to make a trace of the things happening in your program. Generally this can require a substantial redesign of your program since you have to pass a file or list around in your program in order to accommodate the trace. Fortunately, there is one exception to the environment passing of files. One can always write information to a special file, `stderr` without a need to open this file explicitly. The trace can be realized by writing information to `stderr`.

As example we show how to construct a trace of the simple Fibonacci function:

```
fib n = (if (n<2) 1 (fib (n-1) + fib (n-2))) ---> ("fib ", n)

Start = fib 4
```

This yields the following trace:

```
fib 4
fib 2
fib 0
fib 1
fib 3
fib 1
fib 2
fib 0
fib 1
```

We usually write this trace as:

```
Start
→  fib 4
→* fib 3 + fib 2
→  fib 3 + fib 1 + fib 0
→  fib 3 + fib 1 + 1
→  fib 3 + 1 + 1
→  fib 3 + 2
→* fib 2 + fib 1 + 2
→  fib 2 + 1 + 2
→* fib 1 + fib 0 + 1 + 2
→  fib 1 + 1 + 1 + 2
→  1 + 1 + 1 + 2
→* 5
```

From this trace it is clear that the operator `+` evaluates its second argument first. The trace function ---> is an overloaded infix operator based on the function `trace_n` which is defined in `StdDebug`. It yields it's left-hand-side as result and writes as a side-effect it's right-hand-side as trace to `stderr`.


```
(--->) infix :: a b -> a | toString a
(--->) value message = trace_n message value
```

5.3 Handling Events

The previous sections showed us how to write a program that changes the file system. The file names however were coded directly in the program. Of course one would want to specify such parameters in an interactive way by using a dialog For this purpose, it must not only be possible to fetch the file system out of the world but also the events that are generated by the user of the program (keys typed in, mouse clicks) have to be accessible and response on the screen must be given.

A large library is written in CLEAN that makes it possible to address this event queue and deal with monitor output. Similar to addressing the file system indirectly, the event queue can be changed indirectly as an abstract unique part from the world (using `startMDI`).

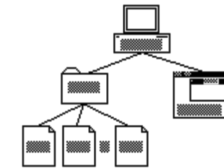


Figure 5.2 The world encapsulating the file system and the event queue.

This figure can be seen as a refinement of figure 5.1 where the dots, …, are replaced by the events. In figure 5.2 we have used icons instead of the names in figure 5.1.

5.3.1 Events

To deal with events the programmer has to create an object (an 'abstract device') of a special algebraic datatype which is predefined in the CLEAN Object I/O library. This data structure not only specifies how the required dialog, window and the like have to look like. The data structure also defines what kind of events are reacted upon (buttons, key presses) and which event-handling function has to be applied when an event is triggered by the user. The data structure can interpreted by predefined library functions that automatically take care of the drawing of objects and handling of events as will be explained hereafter.

It is important to realize that the devices are specified using ordinary data structures and functions in CLEAN. This implies that the definition of the devices can be manipulated just like any other data structure in CLEAN. In this way all kinds of devices can be created dynamically!

The high level way in which dialogs and windows are specified makes it possible to define them in platform independent way: the look and feel of the dialogs and the other devices will depend of the concrete operating system used. In this book we will usually show how the examples look on an Apple Macintosh. But, the CLEAN code can be used unchanged on other platforms where the CLEAN system is available. The resulting application will have the appropriate look and feel that is typical for that platform.

Each event-handing function used in the abstract device specification has a unique parameter that represents the current state of the program (called the `PSt`, pronouce "process state"). This process state itself consists of the local "logical" state and the abstraction (called the `IOSt`, pronounce "I/O state") of the states of the graphical user interface components (also called abstract devices) of the interactive program.

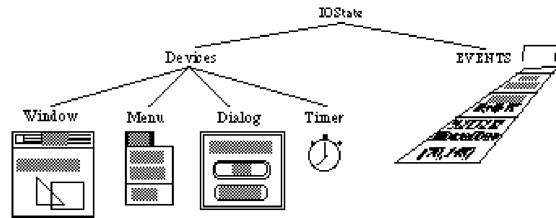


Figure 5.3: The `IOSt` and its components.

The library function `startMDI` takes two initial logical states, initial event-handling functions to perform, optional attributes (that will be ignored for the time being), and the world. The initial event-handling functions create the required graphical user interface components. From that moment on, `startMDI` will deal with all interactions until the function `closeProcess` is called. It then delivers the final world value that will contain among others the leftover events. The two logical states together form the program state which contains all information that should be remembered between events. Usually they are records.

The events handled by `startMDI` are all the inputs a program can get. Typical examples are pressing or releasing of keys at the keyboard, mouse clicks or mouse movements in a window, selection of items from the menu system, and timers. The events are handled in the order of occurrence in the event stream. The function `startMDI` will search in the device definitions in the specified data structure which call-back function (event-handler) is specified to handle this event. This call-back function will be applied to the current process state. The `IOSt` component of this process state contains all device definitions of the program. Using the appropriate library functions these definitions can be changed by the call-back functions. The type of the program state can be chosen freely and will depend on the kind of application programmed. As illustrated above, the initial process state is formed by the two logical arguments of the application of `startMDI`.

The order of events in the event queue determines the behavior of the program. This order of events is the actual input of the program. The object I/O-system looks in the I/O-state of the process state for a handler for the current event. This handler is applied to the current process state. This produces a new process state. This process is repeated until the handler of one of the events indicates that the program should terminate.

So, all call-back functions deliver a new process state. The function `startMDI` will supply this state as an argument to the call-back function corresponding to the next event. This continues until one of the call-back functions applies `closeProcess` to the process state. This closes all currently open devices and restores the unprocessed events back into the world environment that is delivered as a result. Events without an appropriate handler are ignored.

The function `startMDI` is predefined in the CLEAN I/O library. The following simplified version illustrates how event handling is done (*warning*: this is pseudo-code to explain the event handling).


```
startMDI :: !.l !.p !(ProcessInit (PSt .l .p)) ![ProcessAttribute (PSt .l .p)]
                                                                !*World -> *World
startMDI local public initialHandlers attributes world
    #   (events,world)  = openEventQueue world
        ps              = {ls=local,ps=public,io=createIOSt events}
        ps              = seq initialHandlers ps
        {io}            = DoIO ps
    = closeEventQueue io.events world
where
    DoIO ps=:{io}
        |   signalsQuit io  = ps
        #   (newEvent,io)   = getNextEvent io
            (f,io)          = findhandler newEvent io
            ps              = f {ps & io=io}
        =   DoIO ps
```

This shows that first of all, the initial I/O-operations are performed (done by `(seq initialHandlers ps)`). When the initial I/O-operations are finished, the events are processed in the order of appearance in the event queue. The event handler to be used is determined by the current set of devices, which is recorded in the I/O-state component of the process state. The I/O-state is part of the arguments and result of the event handler. This implies that it is possible that event handlers install new call-back functions to process the events in the tail of the queue. A program only has to invoke the function `startMDI` to an appropriate list of initialization actions. This controls the entire behaviour of the program.

Programming for window systems requires some knowledge of the corresponding terminology (menu, pop-up menu, modal dialog, radio button, close box etc.). Such knowledge is assumed to be present with the reader (as a user of such systems). However, when it is felt appropriate, some of this terminology will be explained when it occurs in an example.

5.4 Dialogs

Dialogs are a highly structured form of windows. To offer structured input, dialogs contain controls. Controls are commonly standard predefined graphical user interface elements but they can also be defined by the programmer. Examples of standard controls are text fields (`TextControl`), user input text fields (`EditControl`), and buttons (`ButtonControl`). Controls can be defined by combining existing controls as well. In this way it is easy to make more complicated combinations. A dialog containing the most common controls is depicted in figure 5.4.

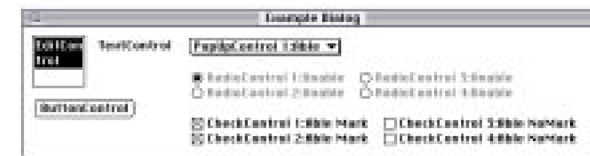


Figure 5.4: An example dialog containing a collection of controls.

Dialogs can be opened in two "mode":

Modeless dialogs: This is the general mode of opening a dialog. A modeless dialog is opened using the function `openDialog`. The dialog will stay open until it is closed by the program using the function `closeWindow`. In the meantime the program can do other things.

Modal dialogs: Sometimes it is necessary to force the user to handle a dialog completely before the program can continue. For this purpose modal dialogs are useful. Modal dialogs are dialogs that cannot be pushed to the background. They are opened using

the function openModalDialog. This function terminates only when its dialog has been closed.

The conventional use of a dialog is that the user is (kindly) requested to fill in some of the edit controls, make the appropriate selections in the pop up, radio, or check controls, and then confirm the action by pressing one of the button controls. As explained earlier, pressing the button control will cause evaluation of the associated event handler. This function will need to know what the user has filled in and what selections have been made. For this purpose it can retrieve a WState which is an abstract datatype containing all information of the dialog. The value of text fields and the selected radio buttons or check boxes can be extracted from this datatype using the appropriate manipulation functions. In this chapter we show a number of examples. A complete definition can be found in StdControl.

5.4.1 A Hello World Dialog

As a first example of a program using dialogs we use another hello world variant. This dialog to be generated contains only a text message to the user and one button. On a Macintosh it looks like:

Figure 5.5: The hello world dialog.

In addition the program contains a single menu as well. This pull-down menu contains only the quit command. The CLEAN code for all this looks as follows:

```
module helloDialog

import StdEnv, StdIO

:: NoState = NoState

Start :: *World -> *World
Start world
= startMDI NoState NoState (opendialog o openmenu) [] world
where
    opendialog ps
        #   (okId,ps) = accPIO openId ps
        =   snd o openDialog NoState (dialog okId)

    dialog okId =   Dialog "Hello Dialog"
                    (   TextControl "Hello World, I am a Clean user!"
                                            [   ControlPos (Center,zero)]
                    :+: ButtonControl "OK"  [   ControlPos (Left,zero)
                                            ,   ControlId okId
                                            ,   ControlFunction quit        ]
                    )                       [   WindowOk okId               ]
    openmenu    =   snd o openMenu undef filemenu
    filemenu    =   Menu "File"
                    (   MenuItem "Quit"     [   MenuShortKey 'Q'
                                            ,   MenuFunction quit           ]
                    )                       []

quit :: (.ls,PSt .l .p) -> (.ls,PSt .l .p)
quit (ls,ps) = (ls,closeProcess ps)
```

Each control is identified by an appropriate data constructor (TextControl, ButtonControl). The last argument of a control is always the list of control attributes.

The arguments of Dialog are its title ("Hello Dialog"), the controls (text and button), and a list of attributes (the identification of the default button). The size of the dialog is determined by the size of its controls, unless overruled by the attributes. The function that opens the dialog, openDialog, has an additional argument, NoState. Every device instance has a local state in which data can be stored that is of interest only for that device instance. The initial value of that local state is given as the first argument of every device open function. Because in this example the dialog is very simple the singleton type constructor NoState is used as an initial value.

The fact that all device instances have a global state and a local state is also visible in the types of the event handlers. Every event handler is a state transition function of type (PSt .l .p) -> (PSt .l .p). Of course, having a local state is of no use if you can't get to it. Every event handler is supplied with the current value of its local state. So the general type of a callback function is actually (.ls,PSt .l .p) -> (.ls,PSt .l .p) with ls the same type as the local state. This is the reason why quit has the given type. Note that in this particular example the following type would also be correct:

```
(NoState,PSt NoState NoState) -> (NoState,PSt NoState NoState)
```

because all the indicated logical and local states have been chosen to be the singleton type NoState. However, there is no need to give quit such a restricted type. In general, it is best to provide a most general (i.e. as polymorphic as possible) type for any function or component so that it can be reused in as many as possible contexts.

The program opens only one menu defined by filemenu. Its definition is an instance of the Menu type constructor. Its arguments are its title ("File"), the menu elements (the "Quit" command), and a list of attributes (empty). The only menu element is an instance of the type constructor MenuItem. Its arguments are its name ("Quit"), and a list of attributes (the keyboard shortcut 'Q', and the same call back function as the button in the dialog. The call back function of the OK-button activates the quit function as well and will quit the program.

Each part of the device definitions can be addressed by its Id (an abstract value identifying the part of the device). Using these Id's dialogs can be closed and opened, windows can be written, text typed in a dialog can be read, menu items can be disabled etcetera, etcetera. However, an id is only needed when we want to perform special actions with the device. In all other situation the value of the id is irrelevant and it can be left out. In this example we need only an id for the OK-button, in order to make it the default button of the dialog (using the WindowOk attribute).

It is a recommended programming style to keep the Id's of devices as much as possible local. The program above creates the Id within the dialog definition. In case one needs a known number of Id's the function openIds is useful. It returns a list of Id's the elements of which can be accessed using a list pattern with a wild-card which is easily extended when more Id's are required.

Do not forget to import the required modules when you write interactive programs. On Unix and Linux systems you should also link the appropriate libraries for window based programs. See the documentation of your CLEAN distribution.

5.4.2 A File Copy Dialog

Now suppose that we want to write an interactive version of the file copying program by showing the following dialog (see Figure 5.6) to the user:

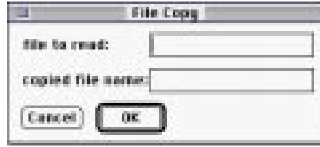


Figure 5.6: The dialog result of the function `CopyFileDialog`.

Also in the case of the file copy program the program state can be `NoState`. The interactive file copying program has a similar structure as the hello world examples above.

```
module copyfile

import StdEnv,StdIO

Start :: *World -> *World
Start world = CopyFileDialogInWorld world

CopyFileDialogInWorld :: *World -> *World
CopyFileDialogInWorld world
    #   initstate = NoState
    = startMDI initstate initstate (openmenu o opendialog) [] world
where
    openmenu = snd o openMenu undef QuitMenu

    opendialog ps
        #   (ids,ps)= openIds 2 ps
        =   snd (openDialog NoState (CopyFileDialog ids!!1 ids!!2 CopyFile) ps)
```

The initialization actions performed by `startMDI` will open the menu (`openmenu`) and the dialog (`opendialog`). The CLEAN I/O library takes care of drawing the specifed menus and dialogs. In the device definitions it also defined what will happen when a menu item is selected, or a dialog button is pressed.

The definition of `QuitMenu` is just an algebraic data structure that is built dynamically using the constructors `Menu` and `MenuItem`. The definition of the type `Menu` in the library shows the different options that are supported by the library.

```
QuitMenu :: Menu MenuItem (PSt .l .p)
QuitMenu =  Menu "File"
            (   MenuItem "Quit" [   MenuShortKey 'Q'
                                ,   MenuFunction QuitFun
                                ]
            )   []
where
    QuitFun (ls,ps) = (ls,closeProcess ps)
```

The appearance of the dialog (see Figure 5.6) is determined by function `CopyFileDialog`. The function to be executed when the `ok` button is pressed, is passed as argument to `CopyFileDialog`. The dialog definition itself consists again of an enumeration of its components.

```
CopyFileDialog dlgId okId copyfun
  = Dialog "File Copy"
    (   TextControl "file to read: "            []
    :+: EditControl defaultinp length nrlines   [   ControlId srcInputId]
    :+: TextControl "copied file name: "        [   ControlPos (Left,zero)  ]
    :+: EditControl defaultin length nrlines    [   ControlId dstInputId
                                                ,   ControlPos (Below srcInputId,zero)]
    :+: ButtonControl "Cancel"                  [   ControlPos (Left,zero)
                                                ,   ControlFunction (cancel dlgId) ]
    :+: ButtonControl "OK"                      [   ControlId okId
                                                ,   ControlFunction (ok dlgId copyfun) ]
    )                                           [   WindowIddlgId
                                                ,   WindowOkokId]
where
    length      = hmm 50.0
```



```
    nrlines     = 1
    defaultin   = ""
    (dlgId,okId,srcInputId,dstInputId) = (ids!!0,ids!!1,ids!!2,ids!!3)

    cancel id (ls,ps)   = (ls,closeWindow id ps)
    ok id fun (ls,ps)
        #   (Just wstate,ps)    = accPIO (getWindow id) ps
            [(_,Just inputfilename),(_,Just outputfilename):_]
                                = getControlTexts [srcInputId,dstInputId] wstate
        #   ps                  = appFiles (fun inputfilename outputfilename) ps
        =   (ls,closeWindow id ps)
```

The dialog definition itself again is an algebraic data structure. Although the definition above looks very static it is very important to realize that each I/O data structure is built dynamically as usual and interpreted by the library functions at run-time to create the proper reflections on the screen. This means that the full power of CLEAN can be used to generate more complicated dialogs with conditional contents. Through the use of constructors and id's to indicate parts of the dialog, the dialog layout can be specified (`Below srcInputId`). The structure contains functions that are called when the corresponding button is selected. These functions can address the contents of the dialog with the library functions `getWindow` (to obtain the entire content) and `getControlTexts` (to obtain the text content of the indicated controls). The example above shows how useful higher-order functions are in these definitions (`ok dlgId copyfun` and `cancel dlgId`).

A disadvantage of the dialog defined above is that it does not enable the user to browse through the file system to search for the files to be copied. Using the functions from the library module `StdFileSelect` such dialogs are created in the way that is standard for the actual machine the program will be running on.

```
import StdFileSelect

FileReadDialog fun (ls,ps)
    #   (maybe_name,ps) = selectInputFile ps
    |   isJust maybe_name   = (ls,fun (fromJust maybe_name) ps)
    |   otherwise           = (ls,ps)

FileWriteDialog fun (ls,ps)
    #   (maybe_name,ps) = selectOutputFile prompt defaultFile ps
    |   isJust maybe_name   = (ls,fun (fromJust maybe_name) ps)
    |   otherwise           = (ls,ps)
where
    prompt          = "Write output as:"
    defaultFile     = "file.copy"
```

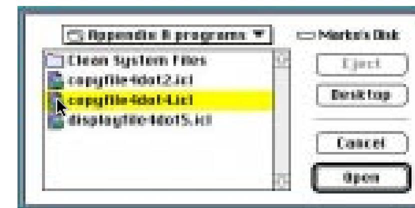


Figure 5.7: A standard `selectInputFile` dialog.

5.4.3 Function Test Dialogs

Suppose you have written a function `GreatFun` and you want to test it with some input values. A way to do this is to use 'console' mode and introduce a start rule with as its right-hand-side a tuple or a list of applications of the function with the different input values.

```
Start = map GreatFun [1..1000]
```

or e.g.

```
Start = (GreatFun 'a', GreatFun 1, GreatFun "GreatFun")
```

Reality learns us that in this static way of testing less variety in testing occurs compared with dynamic interactive testing. For interactive testing, a dialog in which input-values can be typed, will be very helpful.

The previous section has shown how to define a dialog. Here we will define a function that takes a list of functions as argument and produces an interactive program with dialogs with which the functions can be tested. We want this definition to be very general. We use overloading to require that the input values (typed in the dialog as a String) can be converted to the required argument of the test function.

The overloaded test dialog can be used to test a function on a structured argument (a list, a tree, a record, ...) straightforwardly. All that is needed is to write instances of fromString and toString for types or subtypes if they are not already available.

The function test dialogs are organized as a module that should be imported in a module that contains the functions to be tested. The imported module contains a function that generates the appropriate dialogs and overloaded functions to do the conversion from strings for the function arguments and to a string for the function result. The definition module that contains an explanation of its use looks like:

```
definition module funtest

from StdString import String
import StdEnv,StdIO

functionTest :: [(([String]->String),[String],String)] *World -> *World

no_arg    :: y             [String] -> String |  toString y
one_arg   :: (x -> y)      [String] -> String |  fromString x & toString y
two_arg   :: (x y -> z)    [String] -> String |  fromString x & fromString y
                                                 & toString z
three_arg :: (x y z -> w)  [String] -> String |  fromString x & fromString y
                                                 & fromString z & toString w
```

Apply functiontest as an interactive test for functions. The functions to test are given in the list funs (see below). For each function there is a tuple containing:

- the function to be tested, of type [String] -> String,
- a list of initial values, of type [String],
- and a title of the dialog, a String.

This module should be used in programs like:

```
module functiontest
import funtest

Start world = functionTest funs world

double :: Int -> Int
double  x = x + x

plus:: Int Int -> Int
plus x y = x + y

funs = [ (one_arg double       ,["2"]      ,"double")
       , (two_arg plus         ,["2","10"] ,"plus")
       , (no_arg "Hello world" ,[]         ,"Program")
       ]
```

The corresponding implementation is a standard menu and dialog system that calls a function that defines the dialog. The menu contains the standard quit command and a menu to select the desired function. The dialog opens the first function test dialog (if it exists).

```
implementation module funtest
```



```
import StdEnv, StdIO

functionTest :: [(([String]->String),[String],String)] *World -> *World
functionTest [] world = world
functionTest funs world
    #   (ids,world) = openIds (length funs) world
    =   startMDI NoState NoState (initialIO funs ids) [] world

initialIO funs dialogIds = openfilemenu o openfunmenu
where
    openfilemenu    =   snd o openMenu NoState fileMenu
    openfunmenu =   snd o openMenu NoState funMenu
    fileMenu    =   Menu "File"
                    (   MenuItem "Quit" [ MenuShortKey 'Q'
                                        , MenuFunction Quit
                                        ]
                    )   []
    funMenu     =   Menu "Functions"
                    (ListLS
                    [   MenuItem fname  [ MenuFunction (noLS opentest)
                                        : if (c<='9') [MenuShortKey c] []
                                        ]
                    \\  (_,_,fname) <- funs
                    &   opentest    <- opentests
                    &   c           <- ['1'..]
                    ]
                    )   []
    opentests   =   [functiondialog id fun \\ fun <- funs & id <- dialogIds]

    Quit (ls,ps)    =   (ls,closeProcess ps)
```

The function that defines the dialogs (functiondialog) is defined below. It is quite similar to the file copy dialog. First the list of fields for the function arguments are generated. For each argument there is a text control indicating the argument and an edit control to hold the actual argument value.

We complete the dialog by fields for the result and three buttons. The first button closes this dialog. The next button quits the program. The last button is the default button. This button evaluates the function to be tested. The actual arguments are extracted from the edit fields in the dialog. These arguments are passed as a list of strings to the function to be tested. Note that the close function can be used in general for many kinds of dialogs.

```
functiondialog :: Id (([String]->String),[String],String) (PSt .l .p) -> PSt .l .p
functiondialog dlgId (fun,initvals,name) ps
#   (argIds,  ps) = accPIO (openIds arity) ps
    (resultId,ps) = accPIO openId ps
    (evalId,  ps) = accPIO openId ps
=   snd (openDialog o (dialog argIds resultId evalId) ps)
where
    dialog argIds resultId evalId
    =   Dialog name
        (ListLS
        [   TextControl ("arg "+++toString n) [ControlPos (Left,zero)]
        :+: EditControl val width nrlines
            [   ControlId (argIds!!n)
            :   if (n==0) [] [ControlPos (Below (argIds!!(n-1)),zero)]
            ]
        \\  val <- initvals
        &   n   <- [0..]
        ]
        :+: TextControl "result" [ControlPos (Left,zero)]
        :+: EditControl "" width nrlines
            [   ControlId resultId
            :   if (arity==0) [] [ControlPos (Below (argIds!!(arity-1)),zero)]
            ]
        :+: ButtonControl "Close"
              [ ControlFunction (close dlgId)   ]
        :+: ButtonControl "Quit"
              [ ControlFunction Quit    ]
```

```
          :+: ButtonControl "Eval"
              [   ControlId evalId
              ,   ControlFunction (eval dlgId argIds resultId fun)]
          )   [   WindowId dlgId
              ,   WindowOk evalId ]

      arity = length initvals
      eval id argIds resultId fun (ls,ps)
          #   (Just wstate,ps)    = accPIO (getWindow id) ps
              input               = [fromJust arg\\(_,arg)<-getControlTexts argIds wstate]
          =   (ls,appPIO (setWindow dlgId (setControlTexts [(resultId,fun input)])) ps)

  close :: Id (.ls,PSt .l .p) -> (.ls,PSt .l .p)
  close id (ls,ps) = (ls,closeWindow id ps)
```

The arguments of the function to be tested are collected as a list of strings from the dialog. The following functions can be used to do the appropriate type conversions. By using type classes these functions can be very general. Adding similar functions to handle functions with another number of arguments is very simple.

```
  no_arg :: y [String] -> String |    toString y
  no_arg f [] = toString f
  no_arg f l  = "This function should have no arguments instead of "+++toString (length l)

  one_arg :: (x -> y) [String] -> String |    fromString x & toString y
  one_arg f [x]   = toString (f (fromString x))
  one_arg f l     = "This function should have 1 argument instead of "+++toString (length l)

  two_arg :: (x y -> z) [String] -> String    | fromString x & fromString y & toString z
  two_arg f [x,y] = toString (f (fromString x) (fromString y))
  two_arg f l     = "This function should have 2 arguments instead of "+++toString (length l)

  three_arg :: (x y z -> w) [String] -> String
               | fromString x & fromString y & fromString z & toString w
  three_arg f [x,y,z] = toString (f (fromString x) (fromString y) (fromString z))
  three_arg f l = "This function should have 3 arguments instead of "+++toString (length l)
```

We need only the definition of some constants to complete this module.

```
  nrlines :== 2
  width   :== hmm 100.0
```

In order to test the functions `sqrt`, `power`, and `"Hello world"` we write the following program.

```
  module functiontest
  import funtest

  Start world = functionTest funs world

  funs =  [ (one_arg sqrt        ,["2"]      ,"sqrt")
          , (two_arg power       ,["2","10"] ,"power")
          , (no_arg "Hello world" ,[]        ,"Program")
          ]
```

When we execute this program and open all dialogs we obtain an interface as shown in the next figure. This enables the user to test functions interactively.

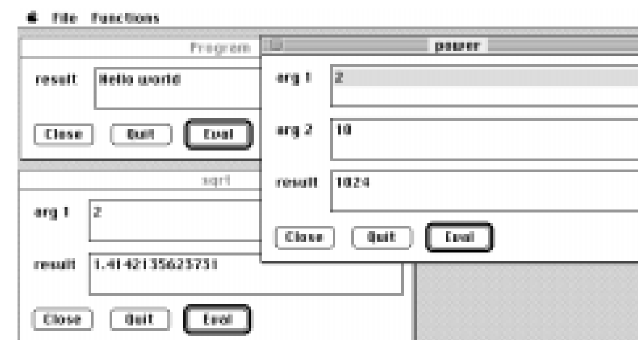


Figure 5.8: An example of the use of the function test dialog system generator.

This completes already the full definition of general dialogs for testing polymorhic functions. Writing test dialogs for user-defined data structures (a list, a tree, a record) is straightforward (in fact: all that is needed is to write instances of `fromString` and `toString`).

The only problem with this program might be that it is to general. When overloaded functions are tested the internal overloading cannot always be solved. By defining a version with a restricted (polymorphic) type instead of the overloaded type this problem is solved in the usual way.

5.4.4 An Input Dialog for a Menu Function

Similarly, an input dialog for a menu function can be defined. A menu function is the argument function of the `MenuFunction` attribute. Its type is the standard process state transition function, `IdFun *(ls,ps)`, augmented with a local menu state. It is the type of a menu function like `open`, `test sin` and `quit` in the previous examples.

In this case we have overloaded the dialog definition itself by requiring `fromString` to be defined on the result of `getControlTexts`.

```
  menufunctiondialog :: Id (([String]->IdFun (PSt .l .p)),[String],String) (PSt .l .p)
                            -> PSt .l .p
  menufunctiondialog dlgId (fun,initvals,name) ps
      #   (argIds,ps) = accPIO (openIds arity) ps
          (evalId,ps) = accPIO openId ps
          (resultId,ps)   = accPIO openId ps
      =   snd (openDialog o (dialog argIds resultId evalId) ps)
  where
      dialog argIds resultId evalId
       =  Dialog name
          (ListLS
              [   TextControl ("arg "+++toString n) [ControlPos (Left,zero)]
              :+: EditControl val width nrlines
                  [   ControlId (argIds!!n)
                  :   if (n==0) [] [ControlPos (Below (argIds!!(n-1)),zero)]
                  ]
              \\  val <- initvals
              &   n   <- [0..]
              ]
          :+: ButtonControl "Close" [ControlFunction (close dlgId)]
          :+: ButtonControl "Quit"  [ControlFunction Quit]
```

```
        :+: ButtonControl "Eval"
            [   ControlId evalId
            ,   ControlFunction (eval dlgId argIds resultId fun)]
        )   [   WindowId dlgId
            ,   WindowOk evalId ]
    where
        arity = length initvals
        eval id argIds resultId fun (ls,ps)
            #   (Just wstate,ps)    = accPIO (getWindow id) ps
                input               = [fromJust arg\\(_,arg)<-getControlTexts argIds wstate]
            =   (ls,fun input ps)
```

This input dialog can be used for all kinds of 'menu' that require a single (structured) input. The result of applying the function `inputdialog` to a name, a width and a menu function is again a menu function incorporating the extra input!

5.4.5 Generic Notices

Notices are simple dialogs that contain a number of text lines, followed by at least one button. The button present the user with a number of options. Choosing an option closes the notice, and the program can continue its operation. Here is its type definition:

```
:: Notice ls ps     = Notice [TextLine] (NoticeButton *(ls,ps)) [NoticeButton *(ls,ps)]
:: NoticeButton ps  = NoticeButton TextLine (IdFun ps)
```

We intend to make notices a new instance of the `Dialogs` type constructor class. So we have to provide implementations for the overloaded functions `openDialog`, `openModalDialog`, and `getDialogType`. We also add a convenience function, `openNotice`, which opens a notice in case one is not interested in a local state.

```
instance Dialogs Notice where
    openDialog :: .ls (Notice .ls (PSt .l .p)) (PSt .l .p)  -> (ErrorReport,PSt .l .p)
    openDialog ls notice ps
        #   (wId, ps) = accPIO openId ps
            (okId,ps) = accPIO openId ps
        =   openDialog ls (noticeToDialog wId okId notice) ps

    openModalDialog :: .ls (Notice .ls (PSt .l .p)) (PSt .l .p) -> (ErrorReport,PSt .l .p)
    openModalDialog ls notice ps
        #   (wId, ps) = accPIO openId ps
            (okId,ps) = accPIO openId ps
        =   openModalDialog ls (noticeToDialog wId okId notice) ps

    getDialogType :: (Notice .ls .ps) -> WindowType
    getDialogType notice = "Notice"

openNotice :: (Notice .ls (PSt .l .p)) (PSt .l .p) -> PSt .l .p
openNotice notice ps = snd (openModalDialog undef notice ps)
```

The function noticeToDialog transforms a Notice into a Dialog. It conveniently uses list comprehensions and compound controls to control the layout. Here is its definition.

```
noticeToDialog :: Id Id (Notice .ls (PSt .l .p))
    -> Dialog  (:+: (CompoundControl (ListLS TextControl))
               (:+: ButtonControl
                    (ListLS ButtonControl)
               )) .ls (PSt .l .p)
noticeToDialog wId okId (Notice texts (NoticeButton text f) buttons)
=   Dialog ""
    (   CompoundControl
        (   ListLS
            [   TextControl text [ControlPos (Left,zero)] \\ text <- texts  ]
          ) [   ControlHMargin 0 0
              , ControlVMargin 0 0
              , ControlItemSpace 3 3
              ]
    :+: ButtonControl text
          [ ControlFunction (noticefun f)
        ,   ControlPos (Right,zero)
        ,   ControlId  okId
            ]
        :+: ListLS
            [   ButtonControl text
                  [ ControlFunction (noticefun f)
                ,   ControlPos (LeftOfPrev,zero)
                ]
            \\  (NoticeButton text f)   <-  buttons
            ]
        )   [   WindowIdwId
            ,   WindowOkokId
            ]
    where
        noticefun f (ls,ps) = f (ls,closeWindow wId ps)
```


We can export this new instance of the `Dialogs` type constructor class in a new module, `notice`.

```
definition module notice

import StdWindow

:: Notice ls ps    = Notice [TextLine] (NoticeButton *(ls,ps)) [NoticeButton *(ls,ps)]
:: NoticeButton ps = NoticeButton TextLine (IdFun ps)

instance Dialogs Notice

openNotice :: (Notice .ls (PSt .l .p)) (PSt .l .p) -> PSt .l .p
```

Given the notice implementation, we can use it in the following examples. They are self-explanatory.

```
import StdEnv, StdIO, notice

//  warning on function to be applied: default Cancel
warnCancel :: [a] .(IdFun (PSt .l .p)) (PSt .l .p) -> PSt .l .p | toString a
warnCancel info fun ps  = openNotice warningdef ps
where
    warningdef = Notice (map toString info) (NoticeButton "Cancel" id)
                                            [NoticeButton "OK" (noLS fun)]

//  warning on function to be applied: default OK
warnOK :: [a] .(IdFun (PSt .l .p)) (PSt .l .p) -> PSt .l .p | toString a
warnOK info fun ps  = openNotice warningdef ps
where
    warningdef = Notice (map toString info) (NoticeButton "OK" (noLS fun))
                                            [NoticeButton "Cancel" id]

//  message to user: continue on OK
inform :: [String] (PSt .l .p) -> PSt .l .p
inform strings ps   = openNotice (Notice strings (NoticeButton "OK" id) []) ps
```

The functions above can be used to inform and warn the user of the program but also to supply information to the programmer about arguments and (sub)structures when a specific function is called. The latter can be very helpful when debugging the program.

Figure 5.9: Some simple applications of the notices defined above.

These general functions to generate notices are used in the example programs below.

5.5 Windows

Programming windows is more elaborate than programming a dialog (a dialog has more structure so the library can deal with most of the work). Consequently, a window must have an *update function* that redraws (part of) the window when required (e.g. when the window is put in front of another window or when it is scrolled). Furthermore, a window usually has a *keyboard function* and a *mouse function* to deal with characters typed in and with mouse actions.

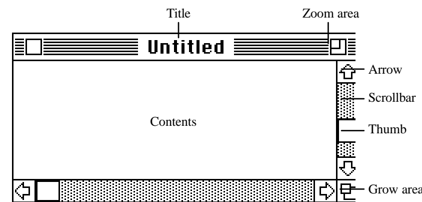


Figure 5.10: Some window terminology.

The contents of the window is composed of pixels. A pixel is a single point of the drawing area. Each pixel has an unique position: a `Point`. A `Point` is a pair of integers:

```
:: Point = {   x   :: !Int
           ,   y   :: !Int
           }

instance == Point
instance +    Point
instance -    Point
instance zero Point
```

The origin, usually the point `zero`, is the left upper corner of the drawing. As a matter of fact it is possible to use coordinates different from `zero` for the left upper corner. The coordinates increase to the right and down. The first integer determines the horizontal position. It is often called x-coordinate. The other integer is usually called the y-coordinate, remember that it increments when you move from the top to the bottom. This is different from what is custom in mathematics!

The window scrolls over this picture area. The object I/O library takes care of scrolling, zooming, and growing of the window. The actions associated with mouse events, keyboard events and with clicking in the close box are determined by the program. Your program always works in the coordinate system of the picture. When your program draws something in the part of the picture area that is not in the current window then the drawing has no visual effect, although it will be evaluated. In order to speed up drawing you can define the drawing functions such that only items inside the current window are shown. This is only worthwhile when drawing happens to be (too) time consuming.

The parts of the window that are currently outside the window, or are hidden beyond some other window, are not remembered by the system. In order to restore these parts of the picture on the screen the system needs to know the content of every window area. This is specified by the `WindowLook` attribute, which has a function argument of type `Look`.

```
:: Look          :== SelectState -> UpdateState -> [DrawFunction]
:: UpdateState  =   {   oldFrame:: !ViewFrame
                    ,   newFrame:: !ViewFrame
                    ,   updArea :: !UpdateArea
                    }
:: ViewFrame    :== Rectangle
:: UpdateArea   :== [ViewFrame]
:: DrawFunction :== *Picture -> *Picture
```


This update function has as arguments the current `SelectState` of the window, and a description of the area to be updated which is defined in a record of type `UpdateState`. This record contains the list of rectangles to be updated (`updArea` field), and the currently visible part of the window (`newFrame` field). In case the update was generated because the size of the window was changed, the previous size of the window is also given (`oldFrame` field). This field is equal to the `newFrame` field in case the window was not resized.

5.5.1 Hello World in a Window

Yet another way to write a hello world program is by putting the message in a window. The following program uses a fixed size, non scrolling, window to hold the message. The message itself is written by the look function. This program can be terminated by the quit function from the file menu, or any of the listed window events.

```
module helloWindow

import StdIO

:: NoState = NoState

Start :: *World -> *World
Start world = startMDI NoState NoState (openwindow o openmenu) [] world
where
    openwindow      =   snd o openWindow NoState window
    window          =   Window "Hello window"
                        NilLS
                        [   WindowKeyboard   filterKey   Able quitFunction
                        ,   WindowMouse      filterMouse Able quitFunction
                        ,   WindowClose      quit
                        ,   WindowViewDomain {corner1=zero,corner2={x=160,y=100}}
                        ,   WindowLook       look
                        ]
    openmenu        =   snd o openMenu NoState file
    file            =   Menu "File"
                        (   MenuItem "Quit" [MenuShortKey 'Q',MenuFunction quit]
                        )   []
    quitFunction ps =   quit ps
    quit (ls,ps)    =   (ls,closeProcess ps)
    look            =   drawAt {x=30,y=30} "Hello World!"
    filterKey   key =   getKeyboardStateKeyState key<>KeyUp
    filterMouse mouse  =   getMouseStateButtonState mouse==ButtonDown
```

This program produces a window as shown in the next figure.

Figure 5.11: The window of the hello world program helloWindow.

This program will quit when the user selects the menu item quit from the file menu, or use the appropriate keyboard shortcut. Other events that cause the program to terminate are: when the close box is selected (this evaluates the `WindowClose` attribute function `quit`); when the mouse button is pressed inside the window (this event is filtered, or guarded, by the function `filterMouse`, and if it evaluates to `True` the mouse function `quitFunction` is evaluated); and when a key has been pressed (this event is filtered, or guarded, by the function `filterKey`, and if it evaluates to `True` the keyboard function `quitFunction` is evaluated). It is very well possible to associate different actions to these events, but for the current program there are no other meaningful actions.

5.5.2 Peano Curves

To show how some simple lines are drawn inside a window we will treat Peano curves. Apart from axioms about numbers, Giuseppe Peano (1858-1932) also studied how you can draw a line to cover a square. A simple way to do this is by drawing lines from left to right and back at regular distances. More interesting curves can be obtained using the following algorithm. The order zero is to do nothing at all. In the first order we start in the left upper quadrant, move to pen to the right, down and to the left. This is the curve Peano 1. Since the net movement of the pen is down, we call this curve `south`. In the second Peano curve we replace each of the lines from Peano 1 with a similar figure. The line to the left is replaced by south, east, north and east. Each of the new lines is only half as long as the lines in the previous order. By repeating this process to the added lines, we obtain the following sequence of Peano curves.

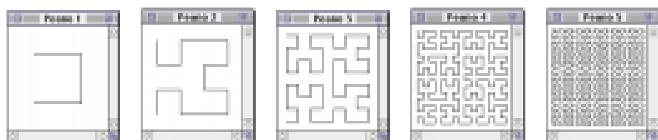


Figure 5.12: Some Peano curves.

In order to write a program that generates figures we have to generate a list of draw functions. From `StdPicture` and `StdIOCommon` we use the following types:

```
:: Picture

class Drawables figure
where
    draw    :: !figure !*Picture -> *Picture
    drawAt  :: !Point   !figure !*Picture -> *Picture

instance Drawables Vector

:: Vector = {vx::!Int,vy::!Int}                    // defined in StdIOCommon
```

In particular we use the function `setPenPos` to move the pen to the argument coordinate, and the `Vector` instance of the overloaded `draw` function to draw a line from the current pen position over the given vector to obtain a new pen position.

The pictures above are generated by four mutually recursive functions. The integer argument, `n`, determines the number of the approximation. The length of the lines, `d`, is determined by the window size and the approximation used. Instead of generating lists of lines in each of the functions and appending these lists we use *continuations*. In general a continuation determines what has to be done when the current function is finished. In this situation the continuation contains the list of lines to be drawn after this pen movement is finished.

```
module peano

peano :: Int -> [*Picture -> *Picture]
peano n = [ setPenPos {x=d/2,y=d/2}
          : south n []
          ]
where
    south 0 c   = c
    south n c   = east (n-1)    [ lineEast
                                : south (n-1)   [ lineSouth
                                                : south (n-1) [lineWest:west (n-1) c]
                                                ]
                                ]
```



```
    east 0 c    = c
    east n c    = south (n-1)   [ lineSouth
                                : east (n-1)[ lineEast
                                            : east (n-1)[lineNorth: north (n-1) c]
                                            ]
                                ]
    north 0 c   = c
    north n c   = west (n-1)    [ lineWest
                                : north (n-1)   [ lineNorth
                                                : north (n-1) [lineEast: east (n-1) c]
                                                ]
                                ]
    west 0 c    = c
    west n c    = north (n-1)   [ lineNorth
                                : west (n-1)[ lineWest
                                            : west (n-1) [lineSouth: south (n-1) c]
                                            ]
                                ]
    lineEast  = draw {vx=  d, vy=  0}
    lineWest  = draw {vx= ~d, vy=  0}
    lineSouth = draw {vx=  0, vy=  d}
    lineNorth = draw {vx=  0, vy= ~d}
    d = windowSize / (2^n)
```

Embedding in a program

We need a window to draw these curves. This is done in a fairly standard way. The window is created by the proper initialization action of `startMDI`, which also opens two menus. There is no need for a logical state. The current order of the Peano curve will be stored implicitly in the look function.

```
import StdEnv, StdIO

::  NoState  = NoState

Start :: *World -> *World
Start world
    = startMDI NoState NoState [openfilemenu,openfiguremenu,openwindow] [] world
```

The menu-system contains two menus. The file menu contains only the menu item quit. The figure menu contains items to generate various Peano curves. This menu is generated by an appropriate list comprehension.

```
openfilemenu = snd o openMenu undef file
file =  Menu "File"
        (   MenuItem "Quit"
            [   MenuShortKey 'Q'
            ,   MenuFunction (noLS closeProcess)
            ]
        )   []
openfiguremenu = snd o openMenu undef fig
fig  =  Menu "Figure"
        (   ListLS
                [   MenuItem (toString i)
                    [   MenuShortKey (toChar (i + toInt '0'))
                    ,   MenuFunction (noLS (changeFigure i))
                    ]
                    \\ i <- [1..8]
                ]
        )   []
```

Changing a figure requires three actions. First, the window title is changed in order to reflect the current Peano curve drawn, using `setWindowTitle`. Then, the window look function is set to the new curve, using `setWindowLook`. Finally, the entire picture domain is erased and the new figure is drawn, using `drawInWindow`.

```
changeFigure peano_nr ps=:{io}
    #   (wId,io)        = getActiveWindow io
    |   isNothing wId   = {ps & io=io}
    #   wId             = fromJust wId
        io              = setWindowTitle wId title io
        io              = setWindowLook wId False (\_ _-> (seq figure)) io
    =   {ps  & io = appWindowPicture wId redraw io}
where
    redraw =[   setPenColour White
            ,   fill        pictDomain
            ,   setPenColour Black
            :   figure
            ]
    figure = peano peano_nr
```

The window is a fairly standard scroll window. The only thing that is a little bit special is the fact that we insert a white margin around the figure. This implies that the left upper corner of the picture domain has not the coordinates `zero`, but `{x = ~margin, y =~ margin}`. Of course it is also possible to use `zero` as left upper corner and to move the figure a little bit. Usually it is more convenient to start the figure in `zero` and to add the white margins separately. The `WindowHScroll` and `WindowVScroll` attributes add a horizontal and vertical scrollbar respectively. Their attribute function is evaluated whenever the user is operating the scrollbars. In that case a new thumb position needs to be calculated. The function scroll gives a generally applicable implementation of such a scroll function. The constructors `Horizontal` and `Vertical` are predefined in the object I/O library.

```
openwindow = snd o openWindow undef window
window
 =  Window
        "Peano"                                 // Window title
        NilLS                                   // No controls
        [   WindowHScroll   (scroll Horizontal)     // Horizontal scroll bar
        ,   WindowVScroll   (scroll Vertical)       // Vertical scroll bar
        ,   WindowViewDomainpictDomain              // Picture domain
        ,   WindowSize      {w=windowSize+2*margin,h=windowSize+2*margin}
                                                // Initial window size
        ,   WindowLook      (\_ _-> seq (peano 1))  // The look function
        ,   WindowResize                        // The window is resizable
        ]
where
    scroll :: Direction ViewFrame SliderState SliderMove -> Int
    scroll direction frame {sliderThumb} move
        =   case move of
                SliderIncSmall  -> sliderThumb+10
                SliderDecSmall  -> sliderThumb-10
                SliderIncLarge  -> sliderThumb+edge
                SliderDecLarge  -> sliderThumb-edge
                SliderThumb x   -> x
    where
        size= rectangleSize frame
        edge= if (direction==Horizontal) size.w size.h
```

We only need to define some constants to make this program complete (the images in figure 5.13 were generated using `windowSize` of 128):

```
windowSize  :== 512
pictDomain  :== {   corner1 =   {x= ~margin,          y= ~margin}
                ,   corner2 =   {x= windowSize+margin,y= windowSize+margin}
                }
margin      :== 4
```

Memory use

Although the program described above works correct, it uses enormous amounts of memory (several megabytes for `peano 8`). This program uses so much memory since Peano curves consists of a very large number of lines and that the program holds this entire list of lines for future reuse.

When we call the number of lines `c`, for complexity, the number of lines in a Peano curve of order `n` is:

```
c 0 = 3
c n = 3 + 4*c (n-1)
```

This is obvious from the structure of the definition of the functions `north`, `east`, `south` and `west`. In the next chapter we will argue that this implies that the number of lines increases exponential with the order of the Peano curve.

```
c 1  = 3
c 2  = 15
c 3  = 63
c 4  = 255
c 5  = 1023
c 6  = 4095
c 7  = 16383
c 8  = 65535
c 9  = 262143
c 10 = 1048575
```

The reason that this huge amount of lines, represented as picture update functions, is remembered becomes clear from the function `changeFigure`. The value `figure` is the list of lines in the Peano curve. It is computed in order to update the window by passing it as argument to `drawInWindow`. Since the same argument is used in the new window look function, the result of the computation is stored for reuse. This is an example of the graph reduction scheme used by CLEAN: an expression is evaluated at most once. Usually this is a benefit, by sharing the result of the reduction, programs become faster.

In this situation you might prefer a small program over a program that consumes megabytes of memory and is only a little bit faster. This can be achieved by making list of lines in the window update function and in the initial window update two separate expressions. Instead of sharing the result of `peano peano_nr`, we substitute it at each reference:

```
changeFigure peano_nr ps=:{io}
    #   (wId,io)        = getActiveWindow io
    |   isNothing wId   = {ps & io=io}
    #   wId             = fromJust wId
    #   io              = setWindowTitle wId title io
    #   io              = setWindowLook wId False (\_ _->peano peano_nr) io
    =   {ps & io = appWindowPicture wId [   setPenColour White
                                        ,   fill        pictDomain
                                        ,   setPenColour Black
                                        :   peano peano_nr
                                        ] io
        }
```

This reduces the memory requirements for this program with about one order of magnitude, without a significant increase of the execution time. The list of lines is recomputed for each update, but a line becomes garbage as soon as it is drawn, this implies that the memory needed to store the line can be reused immediately after drawing.

As a final remark, one might observe that the `changeFigure` function contains a small redundancy. After setting the look of the window, the very same look functions are drawn in the window. This can be combined by passing a `True` Boolean argument to `setWindowLook` and skip the application of `drawInWindow` altogether:

```
changeFigure peano_nr ps=:{io}
    #   (wId,io)        = getActiveWindow io
    |   isNothing wId   = {ps & io=io}
    #   wId             = fromJust wId
        io              = setWindowTitle wId ("peano"+++toString peano_nr) io
        io              = setWindowLook wId False (_ _ -> (seq figure) io
    =   {ps & io = appWindowPicture wId redraw io }
```

```
    where
        redraw = seq [  setPenColour White
                     ,  fill         pictDomain
                     ,  setPenColour Black
                     :  seq figure
                     ]
        figure = peano peano_nr
```

5.5.3 A Window to show Text

Let us take the functions to read in a file and make a program that shows the contents of the file in a window extended with the possibility of selecting (highlighting) a line with the mouse and with the possibility of scrolling using keyboard arrows. This will give us a simple program with a window definition with an update function, a keyboard function and a mouse function.

The overall menu structure is straightforward. The program state contains two fields to indicate whether a line is selected and which line is selected.

```
module displayfileinwindow
import StdEnv, StdIO

::  ProgState = {   select       :: Bool
                ,   selectedline    :: Int
                }
::  NoState  =  NoState

Start world =   startMDI initstate o openmenu []  world
where
    initstate = {   select       =  False
                ,   selectedline =  abort "No line selected"
                }
    openmenu  = snd o openMenu undef menu
    menu      = Menu "File"
                (   MenuItem "Read File..."
                    [   MenuShortKey 'O'
                    ,   MenuFunction (noLS (FileReadDialog (Show LineListRead)))
                    ]
                :+: MenuSeparator   []
                :+: MenuItem "Quit" [MenuShortKey 'Q',MenuFunction Quit]
                )   []

    Quit (ls,ps) = (ls,closeProcess ps)
```

The function `Show` takes a file access function, opens the file, and calls `DisplayInWindow` to display the result in a window.

```
Show readfun name ps
    #   ((readok,file),ps)  = accFiles (open name) ps
    |   not readok          = abort ("Could not open input file '" +++ name +++ "'")
    |   otherwise           = DisplayInWindow (readfun file) ps
where
    open nameps
        #   (readok,file,ps) = sfopen name FreadText ps
        =   ((readok,file)ps)
```

The window definition specifies the usual attributes and passes the text (a list of strings) to the update function as a list of lines (each represented as a list of characters).

```
DisplayInWindow text ps=:{ls}
    #   (font,ps) = accPIO (accScreenPicture getInfoFont) ps
    =   snd (openWindow undef (windowdef font) ps)
where
    windowdef font
    =   Window "Read Result"                    // title
        NilLS                                   // no controls
        [   WindowHScroll (scroll Horizontal)   // horizontal scrollbar
        ,   WindowVScroll (scroll Vertical)     // vertical scrollbar
```



```
        ,   WindowViewDomain{   corner1={x= ~whiteMargin,y=0}
                            ,   corner2={x= maxLineWidth,y=length lines*font.height}
                            }
        ,   WindowSize  {w=640,h=480}               // initial size
        ,   WindowLook  (updatefunction ls lines)   // window look
        ,   WindowKeyboard filterKey Able getkeys   // keyboard handling
        ,   WindowMouse filterMouse Able getmouse   // mouse handling
        ,   WindowResize                            // window is resizable
        ]
    where
        lines = mklines (flatten (map fromString text))
        scroll direction frame {sliderThumb} action
            =   case action of
                    SliderIncSmall  -> sliderThumb+metric
                    SliderDecSmall  -> sliderThumb-metric
                    SliderIncLarge  -> sliderThumb+edge
                    SliderDecLarge  -> sliderThumb-edge
                    SliderThumb x   -> x
        where
            size            = rectangleSize frame
            (edge,metric)   = if (direction==Horizontal)    (size.w,font.width)
                                                            (size.h,font.height)

whiteMargin  = 5
maxLineWidth = 1024
```

The units of scrolling and the size of the domain are defined using the font sizes which are taken from the default font of the application. These values are calculated by the function `getInfoFont` and stored in a record of type `InfoFont`. Observe that the type of `getInfoFont` is overloaded: it can be applied to any environment that belongs to the `FontEnv` type constructor class. These are the `World` (module `StdFont`), `(PSt .l .p)` (module `StdPSt`), and `Picture` (module `StdPicture`).

```
::  InfoFont =  {   font        :: Font
                ,   width    :: Int
                ,   height   :: Int
                ,   up       :: Int
                }

getInfoFont :: *Picture -> (InfoFont,*Picture)
getInfoFont env
    #   (font,env)      = openDefaultFont env
        (metrics,env)   = getFontMetrics font env
    =   (   {   font= font
            ,   width   = metrics.fMaxWidth
            ,   height  = fontLineHeight metrics
            ,   up      = metrics.fAscent+metrics.fLeading
            }
        ,   env
        )
```

The look function of a window is called automatically when (part of) the window must be redrawn. It has the list of domains that must be redrawn as a parameter. Its result is a list of drawing functions that are to be applied on the window.

In this case the look function is defined locally within the definition of the window. This enables to use the defined constants `whiteMargin` and `maxLineWidth` as well as the calculated font information directly. The look function is parameterised with the current program state record and the text lines. In order to keep things relatively simple the complete lines are drawn even when part of them is outside the redraw area (this has no visible effect apart from a very small inefficiency).

```
updatefunction s=:{select,selectedline} textlines _ updstate=:{updArea}
    = seq (flatten (map update updArea))
where
    update domain=:{corner1=c1=:{y=top},corner2=c2=:{y=bot}}
     = [   setPenColour White
        ,   fill {corner1={c1 & x= ~whiteMargin},corner2={c2 & x=maxLineWidth}}
        ,   setPenColour Black
        :   drawlines (tolinenumber top) (tolinenumber (dec bot)) textlines
        ]

    drawlines first last textlines
     = [   setPenPos {x=0,y=(towindowcoordinate first) + font.up}
        :   map drawline (textlines%(first,last))
        ]
        ++  hilite
    where
        hilite
            | select && (selectedline >= first ||  selectedline <= last)
                = hiliteline selectedline
                = []

    drawline xs pict
        #   pict         = draw line pict
        #   (width,pict) = getFontStringWidth font.font line pict
        =   movePenPos {vx= ~width,vy=font.height} pict
    where
        line = toString xs
```

The drawing functions from the library use, of course, window co-ordinates in the window domain while each program usually has its own co-ordinates in (a part of) its state. So, a program will usually contain transformation functions between the different sets of co-ordinates.

In this case the program will have to transform window co-ordinates to line numbers and vice versa.

```
tolinenumber windowcoordinate = windowcoordinate / font.height

towindowcoordinate linenumber = linenumber * font.height // top of the line
```

Using these transformations it is simple to write a function that highlights a line. Highlighting is done by the overloaded functions `hilite` and `hiliteAt` of the type constructor class `Hilites` which has the same signature as the `Drawables` class that we have seen earlier. The instances that can be highlighted are boxes and rectangles only.

```
hiliteline linenr   = [hilite (towindowrectangle linenr)]

towindowrectangle linenumber
    =   { corner1 = {x = ~whiteMargin,  y = winco              }
        , corner2 = {x =  maxLineWidth, y = winco + font.height}
        }
where
    winco = towindowcoordinate linenumber
```

The keyboard handler

The keyboard function is called automatically when a key is hit. It has the keyboard information (is it a keydown?, which key?, is a *meta-key* or *modifier* such as shift, alt/option, command or control down pressed?) as a parameter and of course the process state. Its result is a modified process state. An associated predicate on keyboard information filters the cases in which the function is interested. This function can be used to simplify the function implementation. If you are interested in getting *all* keyboard events, then the expression `(const True)` does the trick. In this case the filter is defined by the function `filterKey`. In this case, the program is only interested in special keys that are down. This is done as follows:

```
filterKey (SpecialKey kcode keyState _) = keyState <> KeyUp
filterKey _                             = False
```


The keyboard function `getkeys` calls `moveWindowViewFrame` which in its turn will cause a call of the window look function again. Calculation of the scrolling vector is straightforward. To calculate the vector in case of page down and page up, the current size of the view frame is needed. This information is obtained by the function `getWindowViewFrame` (in `StdWindow`). The function `rectangleSize` (defined in `StdIOCommon`) returns the `Size` of a `Rectangle`. Recall that `Size` is a record type with the integer fields `w` and `h` representing the width and height respectively.

```
getkeys (SpecialKey kcode _ _) (ls,ps=:{io})
    #   (wid,io)    = getActiveWindow io
        wid         = fromJust wid
        (frame,io)  = getWindowViewFrame wid io
    =   (ls,{ps & io = moveWindowViewFrame wid (v (rectangleSize frame).h) io})
where
    v pagesize
     = case kcode of
        LeftKey     ->  {zero & vx= ~font.width}
        RightKey    ->  {zero & vx=  font.width}
        UpKey       ->  {zero & vy= ~font.height}
        DownKey     ->  {zero & vy=  font.height}
        PgUpKey     ->  {zero & vy= ~pagesize}
        PgDownKey   ->  {zero & vy=  pagesize}
        _           ->  zero
```

The mouse handler

The mouse function is called when a mouse action is performed. It has as its parameter the mouse information (position, no/single/double/triple/long click, modifier keys down) and of course the process state. Its result is a modified process state. An associated predicate on mouse information filters the cases in which the function is interested. This function can be used to simplify the function implementation. If you are interested in getting *all* mouse events, then the expression `(const True)` does the trick. In this case, the program is only interested in double down mouse events. This is done as follows:

```
filterMouse (MouseDown _ _ 2) = True
filterMouse _                 = False
```

The mouse function `filterMouse` changes the selected line, highlights the new selection and de-highlights the old one (by highlighting it again). The mouse function also needs to change the look function of the window because it is parameterised with the current logical state.

```
getmouse (MouseDown {y} _ _) (ls,ps=:{ls=state=:{select,selectedline=oldselection},io})
    #   (wid,io)=   getActiveWindow io
        wid         =   fromJust wid
        io          =   appWindowPicture wid (changeselection oldselection selection) io
=   (ls,    {ps &   ls =   newstate
            ,   io  =   setWindowLook wid False (updatefunction newstate lines) io
        }
    )
where
    selection       =   tolinenumber y
    newstate        =   {state & select = True,selectedline = selection}

    changeselection old new
      | select      =   seq (hiliteline old ++ hiliteline new)
      | otherwise   =   seq (hiliteline new)
```

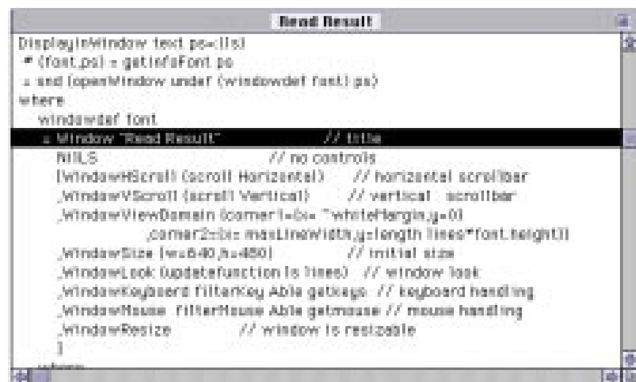


Figure 5.13: A view of the display file program when it has read in its own source.

5.6 Timers

Apart from reacting on user events by defining dialog systems or window systems as devices it is also possible to define timer devices with call-back functions that are called for timer events that are created by the system when a specified time interval has passed.

This can be used to show information on a regular basis or to change it e.g. in a shoot-them-up game. Another way of using a timer is to create some kind of background behaviour such as an autosave facility in an editor that saves the edited file on a regular basis.

Adding timers is very similar to adding dialogs and menus: they can be created in the initialization functions of the `startMDI` function (but of course also at different occasions). Timers can be identified with an Id. They are characterised by a time interval and a call-back function to is executed whenever the timer interval has elapsed. Timers can be enabled and disabled, but to do so its Id must be known. Below a definition is given of a single timer which saves the displayed file in a copy every five minutes.

```
opentimer = snd o openTimer undef timer
timer id
  = Timer TimerInterval                          // the timer interval (in ticks)
    NilLS                                        // timer has no elements
    [   TimerId timerId                          // the Id of the timer
    ,   TimerSelectState Unable                  // timer is initially not working
    ,   TimerFunction timerfunction              // when working, this is its action
    ]
where
    TimerInterval = 300 * ticksPerSecond         // 300 seconds = 5 minutes

timerfunction nrofintervalspassed (ls,ps)       // see inform function of section 5.4
  # ps  =   inform ["Hello"] ps
  = (ls,ps)
```

Such a timer could be used for an autosave function that toggles the menu item title and function changing it from enabling to disabling and vice-versa. This is also a good illustration of the use of local state for the menu item. It keeps a Boolean that states if the auto save option is on. Initially its off. Selecting the menu item should negate the Boolean and change the menu item's title and `SelectState` of the timer accordingly.


```
    …
    {   newLS   =   False                  // local state of menu item, initial value
    ,   newDef  =   MenuItem "Enable AutoSave"
                    [   MenuId autoSaveId
                    ,   MenuShortKey 'S'
                    ,   MenuFunction AutoSave
                    ]
    }
    …

AutoSave :: (Bool,PSt .l .p) -> (Bool,PSt .l .p)
AutoSave (autosave,ps)
    =   (   not autosave
        ,   appListPIO
            [   toggle  timerId
            ,   setMenu menuId [setMenuElementTitles [(autoSaveId,title)]]
            ]   ps
        )
where
    (toggle,title) =if autosave (disableTimer,"Enable AutoSave" )
                                (enableTimer, "Disable AutoSave")
```

Note that many of the program changes required for exercise 5.6 are also required for this auto-save function.

5.7 A Line Drawing Program

In order to show how all pieces introduced above fit together we will show a complete window based program. The program is a simple line drawing tool. It is not intended as a complete drawing program, but to illustrate the structure of such programs. To limit the size of the program we have restricted ourselves to the basic possibilities. As a consequence there are a lot of desirable options of a drawing program that are missing. Adding these features does not require new techniques.

On the side of devices handled, the program is rather complete. It contains, of course, a window to make drawings. It uses the mouse to create and change lines. The drawing can be stored and retrieved from a file. There is a timer to remind the user to save the picture. Dialogs are used for a help function and the standard about dialog. Finally there is a handler for input from the keyboard.

The program is called `Linedraw`. It starts by importing a list of needed modules. The name of the modules indicates their function. These modules contain the type definitions and functions used to manipulate the object I/O-system and all devices. You are encouraged to read the .dcl files whenever appropriate. These files determine the allowed constructs and contain useful comments about the use and semantics of these constructs. The module `notice` contains the notice implementation that was discussed in Section 5.4.5.

```
module Linedraw

import StdEnv,StdId,StdMenu,StdPicture,StdProcess,StdPSt,StdTimer,notice
```

Since this program handles drawing inside a window we will use window co-ordinates. The origin is the left upper corner and a point is indicated by a pair of integers.

The program state

As a first step the important datatypes of the program are defined. The logical state of the program is a record to enable extensions. Since this is a pure line drawing tool, a list of lines is all there is for drawing. Finally, the program state contains the name of the last file used, this is the default when we save the drawing again.

```
:: ProgState    =   {   lines   :: [Line]          // The drawing
                    ,   fname   :: String          // Name of file to store drawing
                    }
```

```
InitProgState  =  {   lines   = []
                  ,   fname   = ""
                  }
```

Drawing lines in the object I/O library is done via points and vectors. Since we are going to store lines anyway, this is a good occasion to introduce a new instance of the `Drawables` type constructor class to draw lines.

```
:: Line =   {   end1:: !Point
            ,   end2:: !Point
            }                              // A line is defined by two endpoints

instance zero Line
where
    zero = { end1 = zero, end2 = zero }

instance Drawables Line
where
    draw     {end1,end2} p = drawLine end1 end2 p
    drawAt _ {end1,end2} p = drawLine end1 end2 p
```

The window device will also contain a local state. In this state information is stored that is used by the mouse when drawing lines. Here the `Maybe` datatype comes in handy (it is defined in the module `StdMaybe`). Only when the mouse is actually tracking, a line is stored in the window state, otherwise it is simply `Nothing`.

```
:: WindowState  =  {   trackline   ::  Maybe Line
                   }
InitWindowState =  {   trackline   =   Nothing
                   }
```

Global structure

We will describe the program top down. This implies that we begin with the `Start` rule. The first part of the program should be fairly standard by now. There are two differences with previous versions. The first difference is that the initialization actions are parameterised with the Ids of the drawing window and timer respectively. All further device definitions are local to the initialization function `initialIO`, and can therefore refer to these Ids. The second difference is that we add a process attribute to the (otherwise empty) fourth argument of `startMDI`. Its purpose is explained below. Let us first have a look at the `Start` rule.

```
Start :: *World -> *World
Start world
    #   (wId,world) = openId world
    #   (tId,world) = openId world
    =   startMDI InitProgState o (initialIO (wId,tId) [] world
where
    initialIO (wId,tId)
            =   openfilemenuo
                openeditmenuo
                openwindow  o
                opentimer   o
                openaboutmenu
    openfilemenu = snd o openMenu NoState file
    file    =   Menu "File"
                (   MenuItem "About"            [ MenuShortKey 'I', MenuFunction openabout ]
                :+: MenuSeparator          []
                :+: MenuItem "Open"        [ MenuShortKey 'O', MenuFunction Open ]
                :+: MenuItem "Save"        [ MenuShortKey 'S', MenuFunction Save ]
                :+: MenuSeparator          []
                :+: MenuItem "Quit"        [ MenuShortKey 'Q', MenuFunction Quit ]
                )                          []
    openeditmenu = snd o openMenu NoState edit
    edit    =   Menu "Edit"
                (   MenuItem "Remove Line"  [ MenuShortKey 'R', MenuFunction Remove ]
                :+: MenuSeparator          []
                :+: MenuItem "Help"        [ MenuShortKey 'H', MenuFunction Help ]
                )                          []
    openwindow = snd o openWindow InitWindowState window
```



```
    window  =   Window "Picture"                                    // Window title
                NilLS                                               // Window has no controls
                [   WindowIdwId                                     // Window Id
                ,   WindowHScroll (scroll Horizontal)               // Horizontal scroll bar
                ,   WindowVScroll (scroll Vertical)                 // Vertical scroll bar
                ,   WindowViewDomain PictDomain                     // View domain of window
                ,   WindowSize InitWindowSize                       // Initial size of window
                ,   WindowLook (\_ ->look [])                       // Window look
                ,   WindowMouse filterMouse Able HandleMouse        // Mouse handling
                ,   WindowKeyboard filterKey Able HandleKey         // Keyboard handling
                ,   WindowClose Quit                                // Closing quits program
                ,   WindowResize                                    // Window is resizable
                ]
    opentimer = snd o openTimer NoState timer
    timer   =   Timer time NilLS
                [   TimerId tId
                ,   TimerFunction remindSave
                ]
    openaboutmenu   = snd o openMenu NoState about
    about   =   Notice  ["A line drawing tool"] (NoticeButton "OK" id)
                [   NoticeButton "Help" Help
                ]
```

The scrolling function `scroll` is identical to the one discussed in Section 5.5.3, except that it uses a constant (`10`) for small scrolling portions and does not rely on a font for this purpose.

During the development of such a program you can begin with a less elaborated user interface. In fact during the development of this example we started without timers. Also the help function and the possibility to remove lines were not present in the first prototypes.

It is convenient to construct programs in an incremental way. We begin with a very simple version of the program and add extensions one by one. The program is compiled and tested before each subsequent addition. For window based programs we can omit a part of the menu structure or use a no-operation for the menu functions: `id` (defined in `StdFunc`) Also the window look function can be a no-operation in the first approximations of your program. The mouse handler and keyboard handler of the window can be omitted in the beginning, or we can again use a no-operation.

As promised, the definition of the `startMDI` rule contains the `ProcessHelp` attribute which is parameterised with a standard process state transition function. The purpose of this attribute is to give the user online information about the program. On the Macintosh it will become the first item in the Apple-system-menu. In this case, the function uses the `openNotice` function introduced in Section 5.4.5. The specified notice looks like:

Figure 5.14: The about dialog of the line drawing tool.

Furthermore, the current `Start` rule determines that there is a timer. This timer will be used to remind the user of the drawing tool to save his work. Initially the timer is disabled. It will be enabled as soon as something is changed to the drawing.

The first menu function that we implement is the function `Quit`. This enables us to leave the test versions of our program in a descent way. Fortunately the implementation is very simple, we simply close the process.

```
Quit :: (.ls,PSt ProgState .p) -> (.ls,PSt ProgState .p)
Quit (ls,ps) = (ls,closeProcess ps)
```

Mouse handling

Next we went immediately to the essential part of the program: mouse handling. This is generally a good strategy: do the hard and crucial parts of the program as soon as possible. The simple details that complete your program can be added later. The difficult part determines most likely the success of the program under construction. There is no excuse to spend time on simple work on a program that still risks to be changed fundamentally.

The first thing that needs to be done with the mouse is drawing lines. A line starts at the point where the mouse button is pushed and ends where the mouse button is released. While the mouse is dragged the line under construction is drawn like a rubber band. This 'abstract specification' already tells us what mouse events we are interested in: a sequence of mouse down, an arbitrary number of mouse drags, and a final mouse up event. This is actually filtered by the mouseFilter function:

```
filterMouse :: MouseState -> Bool
filterMouse (MouseMove _ _) = False
filterMouse _               = True
```

As discussed earlier, the window has a local state, a record of type WindowState. The mouse will store the information it needs to operate properly. The mouse state which is argument of the mouse handler contains the current position of the mouse. We need to remember the start point of the line. The previous end point of the line is needed in order to erase the version of the line drawn. Drawing lines consists of three phases, each phase is adequately defined by one function alternative of HandleMouse, matching on the MouseState alternative constructors MouseDown, MouseDrag, and MouseUp. We discuss them in sequence.

When the mouse button goes down, HandleMouse stores the current mouse position in the window state. The timer is disabled (using timerOff) since we do not want to interfer with the drawing of a line.

```
HandleMouse (MouseDown pos _ _) (window,ps)
= ({window & trackline=Just {end1=pos,end2=pos}},timerOff ps)
```

When the mouse is being dragged, HandleMouse erases the previously tracked line and then draws the new tracked line. The new tracked line is stored in the window state. Proceeding in this way gives the effect of a rubber band. The drawing function xorPicture is used to prevent damage to the existing picture. Drawing any object twice in XorMode restores the original picture. To prevent flickering, redrawing is suppressed in case the mouse was at the same position.

```
HandleMouse (MouseDrag pos _) (window=:{trackline=Just track},ps)
    |   pos == track.end2   = (window,ps)
    #   newtrack            = {track & end2=pos}
    =   (   { window & trackline=Just newtrack }
        ,   appPIO (appWindowPicture wId (appXorPicture (draw track o draw newtrack))) ps
        )
```

When the mouse button goes up the line is completed and tracking has finished. So the window state is reset to Nothing, and the new line is added to the local program state. Because the picture has changed, the timer is switched on again (using timerOn). The look function of the window also needs to be updated. Since the line is already visible, there is no need to refresh the window which is indicated by the False Boolean argument of setWindowLook.

```
HandleMouse (MouseUp pos _) (window=:{trackline=Just track},ps=:{ls=progstate=:{lines}})
    #   newline         = {track & end2=pos}
        newlines        = [newline:lines]
        newprogstate    = {progstate & lines=newlines}
        ps              = {ps & ls=newprogstate}
        ps              = timerOn ps
        ps              = appPIO (setWindowLook wId False (\_ _->look newlines)) ps
    =   ({window & trackline=Nothing},ps)
```

With these functions you can compile your program again and draw some lines. You will soon discover that it is desirable to change the drawing. A very simple way to change the

picture is by removing the last line drawn. This is accomplished by the menu function Remove. If there are lines overlapping with the line to be removed, it is not sufficient to erase that line. This would create holes in the overlapping lines. We simply erase the entire picture and draw all remaining lines again. This time we achieve this by giving setWindowLook a True Boolean argument which redraws the entire window. With some more programming effort the amount of drawing can be reduced, but there is currently no reason to spend this effort. Removing a line changes the picture so we make sure the timer is switched on. If the list of lines is empty, there is no line to be removed. We make the computer beep in order to indicate this error.

```
Remove (ls,ps=:{ls={lines=[]}}) = (ls,appPIO beep ps)
Remove (ls,ps=:{ls=state=:{lines=[_:rest]},io})
    =   (ls,{ ps    & ls={state & lines=rest}
                    , io=setWindowLook wId True (\_ _->look rest) io
            }
        )

look :: [Line] -> (*Picture -> *Picture)
look ls =   seq [   setPenColour White
                ,   fill PictDomain
                ,   setPenColour Black
                :   seq (map draw ls)
                ]
```

An other way to change the picture is by editing an existing line. If the user presses the mouse button with the shift key down very close to one of the ends of the line, that line can be changed. We use very close to the end of a line instead of at the end of a line since it appears to be difficult to position the mouse exactly at the end of the line.

We change the function HandleMouse. First we check whether the shift key is pressed. If it is, we try to find a line end touched by the mouse. If such a line is found, we remove it from the state, and start drawing the line with the removed line as initial version. If no line is touched the program ignores this mouse event. If the shift key is not pressed, we proceed as in the previous version of the function HandleMouse. It is sufficient to add the new alternative before every other alternative of HandleMouse, as in CLEAN alternatives are evaluated in textual order.

The function touch determines whether or not a point is very close to the end of a one of the given lines. Instead of yielding a Boolean, this function uses the type Maybe. In case of success the line touched and the list of all other lines is returned, otherwise Nothing.

```
HandleMouse (MouseDown pos {shiftDown} nrDown) (window,ps=:{ls=state=:{lines}})
    |   shiftDown
    =   case touch pos lines of
         Just (track,ls) -> (   { window & trackline  = Just track }
                            ,   timerOff { ps   & ls = {state & lines=ls} }
                            )
         Nothing         -> HandleMouse (MouseDown pos NoModifiers nrDown) (window,ps)

touch :: Point [Line] -> Maybe (Line,[Line])
touch p [] = Nothing
touch p [l=:{end1=s,end2=e}:r]
    |   closeTo p s = Just ({end1=e,end2=s},r)
    |   closeTo p e = Just (l,r)
    |   otherwise   = case touch p r of
                        Just (t,x)  -> Just (t,[l:x])
                        Nothing     -> Nothing
where closeTo {x=a,y=b} {x,y} = (a-x)^2 + (b-y)^2 <= 10
```

File IO

Next we want to be able to store the drawing in a file and read it back. Each line is represented by its end points. Each of these points consists of two integers. A line is stored as four integers in a data file. We use the dialog from StdFileSelect to determine the name of the output file.

```
Save (ls,ps=:{ls=state=:{fname,lines}})
    #   (maybe_fn,ps)        =   selectOutputFile "Save as" fname ps
    |   isNothing maybe_fn   =   (ls,ps)
    #   fn                   =   fromJust maybe_fn
    #   (ok,file,ps)         =   acccFiles (fopen fn FWriteData) ps
    |   not ok               =   (ls,inform ["Cannot open file"] ps)
    #   file                 =   seq [   fwritei i
                                     \\  {end1,end2} <- lines
                                     ,   i <- [end1.x,end1.y,end2.x,end2.y]
                                     ] file
    #   (ok,ps)              =   accFiles (fclose file) ps
    |   not ok               =   (ls,inform ["Cannot close file"] ps)
    =   (ls,{ps & ls={state & fname=fn}})
```

We reused the `inform` notice, introduced in section 5.4.5, in case that there is something wrong with opening or closing the file.

Opening and reading lines from a file is very similar. We open the file as a unique file in order to allow reuse. In this way the user can read a drawing from a file, change it and save it again in the same file. Each sequence of four integers found in the file is interpreted as a line. We do not do any checks on the format of the file. This implies that almost any data file can be interpreted as a drawing. For such a simple program this is sufficient.

```
Open (ls,ps=:{ls=state})
    #   (maybe_fn,ps)       = selectInputFile ps
    |   isNothing maybe_fn  = (ls,ps)
    #   fn                  = fromJust maybe_fn
    #   (ok,file,ps)        = accFiles (fopen fn FReadData) ps
    |   not ok              = (ls,inform ["Cannot open file"] ps)
    #   (ints,file)         = readInts file
    #   (ok,ps)             = accFiles (fclose file) ps
    |   not ok              = (ls,inform ["Cannot close file"] ps)
    #   lines               = toLines ints
    #   ps                  = appPIO (setWindowLook wId True (\_ _->look lines)) ps
    =   (ls,{ps & ls={state & lines=lines,fname=fn}})
where
    toLines :: [Int] -> [Line]
    toLines [a,b,x,y:r] = [{end1={x=a,y=b},end2={x=x,y=y}}:toLines r]
    toLines _           = []

    readInts :: *File -> ([Int],*File)
    readInts file
        #   (end,file)  = fend file
        |   end         = ([],file)
        #   (ok,i,file) = freadi file
        |   not ok      = ([],file)
        #   (is,file)   = readInts file
        =   ([i:is],file)
```

The keyboard handler

As a next step we add the keyboard handler to the window of our drawing program. The arrow keys scroll the window and the back space key is equivalent to the menu item `Remove`. `KeyUp` events are ignored by the first alternative, this implies that the handler reacts on `KeyDown` and `KeyStillDown` events. This is expressed conveniently by the keyboard filter function `filterKey`:

```
filterKey :: KeyboardState -> Bool
filterKey key = getKeyboardStateKeyState key<>KeyUp
```

The scrolling keys are all special keys. So the scrolling alternative of `HandleKey` is very similar to the keyboard function discussed in Section 5.5.3. Again `getWindowViewFrame` is used to determine the current dimension of the window. Depending on the special key the window view frame is moved, using `moveWindowViewFrame`. The backspace key is an ASCII key, `'\b'`. If it is matched in the second alternative of `HandleKey`, then the `Remove` function is applied.


```
HandleKey (SpecialKey kcode _ _) (ls,ps=:{io})
    #   (frame,io) = getWindowViewFrame wId io
    =   (ls,{ps & io = moveWindowViewFrame wId (v (rectangleSize frame).h) io})
where
    v pagesize
     = case kcode of
        LeftKey     ->  {zero & vx= ~10}
        RightKey    ->  {zero & vx=  10}
        UpKey       ->  {zero & vy= ~10}
        DownKey     ->  {zero & vy=  10}
        PgUpKey     ->  {zero & vy= ~pagesize}
        PgDownKey   ->  {zero & vy=  pagesize}
        _           ->  zero
HandleKey (CharKey char _) state
    |   char=='\b'
        =   Remove state
        =   state
```

The function `Help` just shows a notice containing a few lines of text containing some information about using this program.

```
Help :: (.ls,PSt ProgState .p) -> (.ls,PSt ProgState .p)
Help (ls,ps) = (ls,inform helptext ps)

helptext :==    [ "Hold down shift to move end points of line"
                , "Arrow keys scroll window"
                , "Backspace deletes last line"
                ]
```

Timers

The last device added is the timer. After a predefined amount of time since the first change of the drawing, a notice is shown to the user. This notice reminds the user to save his work. There are two buttons in the notice. The button **Save now** calls the function `Save`. The other button resets the timer. A timer can be reset by first disabling it and then enabling it. This is implemented by the function `timerReset`.

```
timerReset = appListPIO [disableTimer tId,enableTimer tId]
```

The functions `timerOff` and `timerOn` that are supposed to protect the user from displaying this dialog when he is drawing make a naive implementation of these functions using `disableTimer` and `enableTimer` respectively not possible. The reason for this complication is that `enableTimer`, when applied to a disabled timer, resets the latest time stamp of the timer. Because of this, a straightforward approach will always defer the timer whenever the user draws something. What is required is an additional piece of local state that tells the timer if it is not allowed to ask the user to save. We can now profit from the fact that the program state `ProgState` is a record. We change it as follows:

```
:: ProgState    =   {   lines   :: [Line]       // The drawing
                    ,   fname   :: String       // Name of file to store drawing
                    ,   noTimer :: Bool         // The timer should not interfere
                    }
InitProgState   =   {   lines   = []
                    ,   fname   = ""
                    ,   noTimer = False
                    }
```

The function `timerOff` that protects drawing simply sets the new field to `True`.

```
timerOff ps=:{ls=state} = {ps & ls={state & noTimer=True}}
```

If the timer's interval elapses, it checks the `noTimer` flag. If it is not supposed to interfere, it does nothing. Otherwise it happily interrupts the user.

```
RemindSave _ (ls,ps=:{ls=state=:{noTimer}})
    |   noTimer     = (ls,ps)
    |   otherwise   = (ls,timerReset (openNotice notice ps))
where
    notice =    Notice ["Save now?"] (NoticeButton "Later" id)
                [ NoticeButton "Save now" Save ]
```

For the function timerOn there are two cases: *either* the timer did not want to interfere while the noTimer flag was True in which case it can be set to False safely, *or* the timer did want to interfere but was not allowed to. The latter situation is detected because in that case the timer took the bold initiative to set the flag to False. In this case timerOn simply calls the timer function as a delayed action.

```
timerOn ps=:{ls=state=:{noTimer}}
    |   noTimer     = {ps & ls={state & noTimer=False}}
    |   otherwise   = snd (remindSave undef (undef,ps))
```

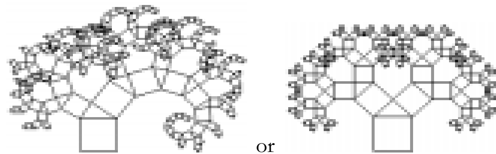
Finally, there are some constants used in the program. The first three constants determines properties of the drawing window. The value time determines the time interval between save reminders.

```
PictDomain      :== {zero & corner2={x=1000,y=1000}}
InitWindowSize  :== {w=500,h=300}
time            :== 5*60*ticksPerSecond    // Time between save reminders
```

This completes our line drawing example. It demonstrates how all parts introduced above can be put together in order to create a complete program. It is tempting to add features to the program in order to make it a better drawing tool. For instance to switch on and off the save reminder and set its time interval. An option to set the thickness would be nice, as well as circles, rectangles, etcetera etcetera. Adding these things requires no new techniques. In order to limit the size of the example we leave it to the user to make these enhancements. Chapter II.4 discusses a more sophisticated drawing tool.

5.8 Exercises

1. Write a program that applies a given transformation function from character lists to character lists on a given file. Structure the program such that the transformation function can be provided as an argument. Test the program with a function that transforms normal characters into capitals and with a function that collects lines, sorts them and concatenates them again to a character list.
2. Combine the FileReadDialog and FileWriteDialog functions into a complete copyfile program which copies files repeatedly as indicated in a dialog by the user.
3. Adapt the program you made for exercise 5.1 such that it transforms files as indicated in a dialog by the user.
4. Write a program that generates one of the following curves in a window:



5. Adapt the display file program such that the user can save the viewed file with a SelectOutputFile dialog. Use (possibly a variant of) the function FileWriteDialog. In order to assure that saving is done instantly instead of lazily the Files component of the ProgState can be made strict by prefixing Files in the type definition of ProgState with an exclamation mark. Add the text as a field in the state record. It may also prove to be useful to add the name of the file and the file itself to this state. In order to allow the user to overwrite the displayed file the program will have to be changed to use fopen for displaying instead of sfopen since a file opened with sfopen can be neither updated nor closed.
6. Adapt the program you made for exercise 5.3 such that it shows the result of a transformation of a file in a window such that the user can browse through it before saving it.

7. Include in the program of exercise 5.6 a menu function opening a dialog with RadioItems such that the user can select the transformation to be applied.
8. Adapt the display file program such that the user can choose with a ScrollingList the font which is used to display the file.
9. Include in the program of exercise 5.7 a timer that scrolls to the next page automatically after a period of time which can be set by the user via an input dialog.
10. Extend an existing program using the function GetCurrentTime and a timer to display the time in hours and minutes every minute. Choose your own way to display the time: in words or as a nice picture using the draw functions from the I/O module deltaPicture.
11. (Large exercise) Extend the display file program with editing capabilities by extending the keyboard and mouse functions. Incorporate the results of exercises 5.6, 5.8 and 5.9 and extend it into your own window-based editor.
12. Change the line drawing program such that only horizontal and vertical lines can be drawn if the shift key is pressed during drawing. The line draw should be the 'best fit' of the line connecting the stating point and the current mouse position.
13. Extend the line drawing program such that the thickness of lines can be chosen from a sub-menu.

Part I
Chapter 6

Efficiency of programs

6.1 Reasoning about efficiency
6.2 Counting reduction steps
6.3 Constant factors
6.4 Exploiting Strictness
6.5 Unboxed values
6.6 The cost of Currying
6.7 Exercises

Until now we haven't bothered much about the efficiency of the programs we have written. We think this is the way it should be. The correctness and clearness is more important than speed. However, sooner or later you will create a program that occurs to be (too) slow. In this section we provide you with the necessary tools to understand the efficiency of your programs.

There are two important aspects of efficiency that deserve some attention. The first aspect is the amount of time needed to execute a given program. The other aspect is the amount of memory space needed to compute the result.

In order to understand the time efficiency of programs we first argue that counting the number of reduction steps is generally a better measure than counting bare seconds. Next we show how we usually work more easily with a proper approximation of the number of reduction steps. Although we give some hints on space efficiency in this chapter, we delay the thorough discussion to part III.

Furthermore, we give some hints how to further improve the efficiency of programs. Lazy evaluation and the use of higher functions can slow down your program. In this Chapter we do not want to advocate that you have to squeeze the last reduction step out of you program. We just want to show that there are some cost associated with certain language constructs and what can be done to reduce these costs when the (lack of) execution speed is a problem.

Your computer is able to do a lot of reduction steps (up to several million) each second. So, usually it is not worthwhile to eliminate all possible reduction steps. Your program should in the first place be correct and solve the given problem. The readability and maintainability of your program is often much more important than the execution speed. Programs that are clear are more likely to be correct and better suited for changes. Too much optimization can be a real burden when you have to understand or change programs. The complexity of the algorithms in your program can be a point of concern.

6.1 Reasoning about efficiency

When you have to measure time complexity of your program you first have to decide which units will be used. Perhaps your first idea is to measure the execution time of the program in seconds. There are two problems with this approach. The first problem is that the execution time is dependent of the actual machine used to execute the program. The

second problem is that the execution time is generally dependent on the input of the program. Also the implementation of the programming language used has generally an important influence. Especially in the situation in which there are several interpreters and compilers involved or implementations from various manufactures.

In order to overcome the first problem we measure the execution time in reduction steps instead of in seconds. Usually it is sufficient to have an approximation of the exact number of reduction steps. The second problem is handled by specifying the number of reduction steps as function of the input of the program. This is often called the *complexity* of the program.

For similar reasons we will use nodes to measure the space complexity of a program or function. The space complexity is also expressed as function of the input. In fact we distinguish the total amount of nodes used during the computation and the maximum number of nodes used at the same moment to hold an (intermediate) expression. Usually we refer to the maximum number of nodes needed at one moment in the computation as the space complexity.

The time complexity of a program (or function) is an approximation of the number of reduction steps needed to execute that program. The space complexity is an approximation of the amount of space needed for the execution. It is more common to consider time complexity than space complexity. When it is clear from the context which complexity is meant we often speak of the complexity.

6.1.1 Upper bounds

We use the *O*-notation to indicate the approximation used in the complexity analysis. The *O*-notation gives an upper bound of the number of reductions steps for sufficient large input values. The expression $O(g)$ is pronounced as *big-oh* of *g*.

This is formally defined as: Let *f* and *g* be functions. The statement $f(n)$ is $O(g(n))$ means that there are positive numbers *c* and *m* such that for all arguments $n \geq m$ we have $|f(n)| \leq c*|g(n)|$. So, $c*|g(n)|$ is an upper bound of $|f(n)|$ for sufficient large arguments.

We usually write $f(n) = O(g(n))$, but this can cause some confusion. The equality is not symmetric; $f(n) = O(g(n))$ does not imply $O(g(n)) = f(n)$. This equality is also not transitive; although $3*n^2 = O(n^2)$ and $7*n^2 = O(n^2)$ this does not imply that $3*n^2 = 7*n^2$. Although this is a strange equality we will use it frequently.

As example we consider the function $f(n) = n^2 + 3*n + 4$. For $n \geq 1$ we have $n^2 + 3*n + 4 \leq n^2 + 3*n*n + 4*n^2 = n^2 + 3*n^2 + 4*n^2 = 8*n^2$. So, $f(n) = O(n^2)$.

Keep in mind that the *O*-notation provides an upper bound. There are many upper bounds for a given function. We have seen that $3*n^2 = O(n^2)$, we can also state that $3*n^2 = O(n^3)$, or $3*n^2 = O(2^n)$.

We can order functions by how fast their value grows. We define $f < g$ as $f = O(g)$ and $g \neq O(f)$. This means that *g* grows faster than *f*. In other words:

$$f < g \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Using this notation we can list the order of the most occurring complexity functions:

$$1 = n^0 < \log(\log n) < \log n < n^b < n = n^1 < n^c < n^{\log n} < c^n < n^n < c^{c^n},$$
where $0 < b < 1 < c$.

In addition we have an ordering within functions of a similar form:

$$n^b < n^c \text{ when } 0 < b < c;$$

$c^n < d^n$ when $1 < c < d$.

In order to give you some idea how fast the number of reduction steps grows for various complexity functions we list some examples in the following table:

function	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$	name
$O(1)$	1	1	1	1	Constant
$O(^{10}\log n)$	1	2	3	4	Logarithmic[1]
$O(\sqrt{n})$	3	10	32	100	
$O(n)$	10	100	1000	10000	Linear
$O(n\ ^{10}\log n)$	10	200	3000	40000	
$O(n^2)$	100	10000	10^6	10^8	Quadratic
$O(n^3)$	1000	10^6	10^9	10^{12}	Cubic
$O(2^n)$	1024	10^{30}	10^{300}	10^{3000}	Exponential

Table 1: **Number of reduction steps as function of the input and complexity.**

Some of these numbers are really huge. Consider that the number of bits on a 1 GigaByte hard disk is 10^{10}, light travels about 10^{16} meters in one year, and the mass of our sun is about $2*10^{30}$ Kg. The number of elementary particles is about 10^{80}. At a speed of one million reduction steps per second a program of 10^{30} reductions takes about $3*10^{16}$ = 30,000,000,000,000,000 years. As a reference, the age of the universe is estimated at $12*10^9$ year. It is currently unlikely that your machine executes a functional program significantly faster than 10^6 or 10^7 reduction steps per second. Using a reduction speed of 10^6 steps per second, 10^9 takes about a quarter of an hour. At this speed 10^{12} reduction steps takes about twelve days, this is probably still more than you want for the average program.

6.1.2 Under bounds

In addition to the approximation of the upper bound of the number of reductions steps needed, we can give an approximation of the under bound of the amount of reductions needed. This is the number of reductions that is needed at least. We use the Ω-notation, pronounced omega notation, for under bounds. The Ω-notation is defined using the O-notation:

$$f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$$

This implies that it is a similar approximation as the upper bound. It is only valid for large arguments and constants are irrelevant.

As an example we consider again the function $f(n) = n^2 + 3*n + 4$. An under bound for those function is the constant. $f(n) = \Omega(1)$. In fact this is an under bound of any function, we cannot expect anything to grow slower than no increase at all. With a little more effort we can show that $f(n) = \Omega(n)$ and even $f(n) = \Omega(n^2)$. This last approximation can be justified as follows: for any $n \geq 0$ we have $f(n) = n^2 + 3*n + 4 > n^2 + 3*n \geq n^2$.

6.1.3 Tight upper bounds

As we have seen upper bounds and under bounds can be very rough approximations. We give hardly any information by saying that a function is $\Omega(1)$ and $O(2^n)$. When the upper bound and under bound are equal we have tight bounds around the function, only the constants in the asymptotic behavior are to be determined. We use the Θ-notation, pronounced *theta notation*, to indicate tight upper bounds:

[1] We used logarithms with base 10 in this table since we use powers of 10 as value for n. A logarithm with base 2 is more common in complexity analysis. This differs only a constant factor (2.3) in the value of the logarithm.

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

For the function $f(n) = n^2 + 3*n + 4$ we have seen $f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$. This makes it obvious that $f(n) = \Theta(n^2)$.

6.2 Counting reduction steps

Now we have developed the tools to express the complexity of functions. Our next task is to calculate the number of reduction steps required by some expression or function to determine the time complexity, or the number of nodes needed to characterize the space complexity. When there are no recursive functions (or operators) involved this is simply a matter of counting. All these functions will be of complexity $\Theta(1)$.

Our running example, $f(n) = n^2 + 3*n + 4$ has time complexity $\Theta(1)$. The value of the function itself grows quadratic, but the amount of steps needed to compute this value is constant: two and three additions. We assume that multiplication and addition is done in a single instruction of your computer, and hence in a single reduction step. The amount of time taken is independent of the value of the operands. The number of nodes needed is also constant: the space complexity is also $\Theta(1)$. This seems obvious, but it isn't necessarily true. A naive implementation of multiplication uses repeated addition. This kind of multiplication is linear in the size of the argument. Even addition becomes linear in the size of the argument when we represent the arguments as Church numbers: a number is either zero, or the successor of a number.

```
:: Nat = Zero | Succ Nat

instance + Nat
where
    (+) Zero     n = n
    (+) (Succ n) m = n + (Succ m)

instance zero Nat where zero = Zero
instance one  Nat where one  = Succ Zero
```

For recursive functions we have to look more carefully at the reduction process. Usually the number of reduction steps can be determined by an inductive reasoning. As example we consider the factorial function `fac`:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

For any non-negative argument this takes `3*n+1` reduction steps (for each recursive call one for `fac`, one for `*` and one for `-`). Hence, the time complexity of this function is Θ(`n`). As a matter of fact also the space complexity is Θ(`n`). The size of the largest intermediate expression `n * (n-1) * … * 2 * 1` is proportional to `n`.

Our second example is the naive Fibonacci function:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Computing `fib n` invokes the computation of `fib (n-1)` and `fib (n-2)`. The computation of `fib (n-1)` on its turn also calls `fib (n-2)`. Within each call of `fib (n-2)` there will be two calls of `fib (n-4)`. In total there will be one call of `fib (n-1)`, two calls of `fib (n-2)`, three calls of `fib (n-3)`, four calls of `fib (n-4)`…. The time (and space) complexity of this function is greater than any power. Hence, `fib n` = $\Theta(2^n)$: the number of reduction steps grows exponentially.

6.2.1 Memorization

It is important to realize that the complexity is a property of the algorithm used, but not necessarily a property of the problem. For our Fibonacci example we can reduce the com-

plexity to O(n) when we manage to reuse the value of `fib (n-m)` when it is needed again. Caching these values is called memorization. A simple approach is to generate a list of Fibonacci numbers. The first two elements will have value `1`, the value of all other numbers can be obtained by adding the previous two.

```
fib2 :: Int -> Int
fib2 n = fibs!!n

fibs :: [Int]
fibs =: [1,1:[f n \\ n <- [2..]]
where
    f n = fibs!!(n-1) + fibs!!(n-2)
```

Now we want to determine the time complexity of `fib2`. It is obvious that selecting element `n` from list `fibs` takes `n` reduction steps. This is O(n). To compute element `n` of the list `fibs` we need to calculate element `n-1` (and hence all previous elements) and to do two list selections each of O(n) and one addition. The calculation of element `n-1` on its turn requires two list selections of O(n-1) and an addition. The total amount of work is `1 + ... +` O(n-1) `+` O(n). This makes the time complexity of `fibs` $O(n^2)$, which is much better than the naive algorithm used above. The space required is proportional to the length of the list of Fibonacci numbers: O(n).

Using an array instead of a list of an list makes an important difference. Array selection is done in constant time and hence the computation of Fibonacci numbers is done in linear time. A drawback of using an array is that we have to indicate the size of the array and hence the maximum Fibonacci number. We will show that we do not need an array to compute Fibonacci numbers in linear time.

It is not useful to make the list of Fibonacci numbers a global constant when only one Fibonacci number is needed in your program. Nevertheless it is essential to achieve the sharing that all numbers used as intermediate values are obtained form one and the same list. This can be achieved by making the `fibs` a local definition of the function `fib2`.

The generation of the list of Fibonacci numbers can be further improved. To compute the next Fibonacci number we need the previous two. Our current algorithm selects these numbers from the list as if this is a random selection. This selection takes O(n) reduction steps. The following function to generate a list of Fibonacci function uses the known order in which the elements are used to compute the next element:

```
fibs2 :: [Int]
fibs2 =: [1,1:[a+b \\ a <- fibs2 & b <- tl fibs2]]
```

When you prefer explicit recursive functions instead of the list comprehension, this algorithm can be written as:

```
fibs3 :: [Int]
fibs3 =: [1, 1: map2 (+) fibs3 (tl fibs3)]

map2 :: (a b->c) [a] [b] -> [c]
map2 f [a:x] [b:y] = [f a b:map2 f x y]
map2 _ _     _     = []
```

The second alternative of `map2` is never used in this application. It is only included to make `map2` a generally useful function.

Computing the next element of the list `fib3` takes only two reductions (`map2` and `+`). This makes this algorithm to compute Fibonacci numbers O(n) in time and space.

The complexity analysis shows that the function `fib4` behaves much better than `fib`. Although the function `fib` is clearer, it is obvious that `fib4` is preferred in programs. List selection using `fibs2` or `fibs3` has the same time complexity, but is only worthwhile when many Fibonacci numbers are used.

When our program needs only a single Fibonacci number there is no need to create a list of these numbers as a CAF. Since only the two previous numbers are used to compute the

next Fibonacci number, there is no need at all to store them in a list. This is used in the following function to compute Fibonacci numbers.

```
fib4 n = f n 1 1
where
    f :: !Int !Int !Int -> Int
    f 0 a b = a
    f n a b = f (n-1) b (a+b)
```

Computing the next Fibonacci number takes three reduction steps. So, this algorithm has a time complexity of O(n). By making the local function `f` strict in all of its arguments we achieved that these arguments are evaluated before `f` is evaluated. This makes the space required for the intermediate expressions a constant. The space complexity of this version of the Fibonacci function is $O(1)$. Using advanced mathematics it is even possible to compute a Fibonacci number in logarithmic time.

6.2.2 Determining the complexity for recursive functions

In the examples above a somewhat ad hoc reasoning is used to determine the complexity of functions. In general it is a convenient to use a function indicating the number of reduction steps (or nodes) involved. Using the definition of the recursive function to analyze it is possible to derive a recursive expression for the complexity function $C(n)$. The complexity can than be settled by an inductive reasoning. The next table lists some possibilties (c and d are arbitrary constants □ 0):

$C(n)$	Complexity
$\frac{1}{2} * C(n\text{-}1)$	$O(1)$
$\frac{1}{2} * C(n\text{-}1) + c*n + d$	$O(n)$
$C(n\text{-}1) + d$	$O(n)$
$C(n\text{-}1) + c*n + d$	$O(n^2)$
$C(n\text{-}1) + c*n^x + d$	$O(n^{x+1})$
$2*C(n\text{-}1) + d$	$O(2^n)$
$2*C(n\text{-}1) + c*n + d$	$O(2^n)$
$C(n/2)$	$O(1)$
$C(n/2) + d$	$O(\log n)$
$C(n/2) + c*n + d$	$O(n)$
$2*C(n/2) + d$	$O(n)$
$2*C(n/2) + c*n + d$	$O(n \log n)$
$4*C(n/2) + c*n^2 + d$	$O(n^2 \log n)$

Table 2: **The complexity for a some recursive relations of the number of reduction steps C(n).**

Although this table is not exhaustive, it is a good starting point to determine the complexity of very many functions.

As example we will show how the given upperbound, $O(\log n)$, of the complexity function C can be verified for $C(n) = C(n/2)+d$. We assume that $C(n/2)$ is $O(\log n/2)$. This implies that there exists positive numbers a and b such that $C(n/2)$ □ $a*\log n/2 + b$ for all n □ 2.

$C(n)$
= $C(n/2)+d$ // Using the recursive equation for $C(n)$
□ $a*\log(n/2) + b + d$ // Using the induction hypothesis
= $a*(\log n - \log 2) + b + d$ // $\log(x/y) = \log x - \log y$
= $a*\log n + b + d - a$ // arithmetic
□ $a*\log n + b$ iff d □ a

We are free to choose positive values a and b. So, we can take a value a such that $a \geq d$ for any d. When we add the fact that $C(0)$ can be done in some finite time, we have proven that $C(n) = O(\log n)$.

It is a good habit to indicate the reason why the step is a proof is valid as a comment. This makes it easy for other people to understand and verify your proof. Even for you as an author of the proof it is very useful. At the moment of writing you have to think why this step is valid, and afterwards it is also for you easier to understand what is going on.

In exactly the same we can show that $C(n) = C(n/2)+d$ implies that $C(n) = O(n)$. For our proof with induction we assume now that $C(n/2) \leq a*n/2 + b$. The goal of our proof is that this implies also that $C(n) \leq a*n + b$.

$C(n)$
$= C(n/2) + d$ // Using the recursive equation for $C(n)$
$\leq a*n/2 + b + d$ // Using the induction hypothesis
$\leq a*n + b + d$ // Since a and n are positive

For the same reasons as above this implies that $C(n) = O(n)$. This is consistent with our claim that we only determine upperbounds and the ordering on functions: $\log n < n$.

When we would postulate that $C(n) = C(n/2)+d$ implies that $C(n) = O(1)$ we have as induction hypothesis $C(n/2) \leq b$.

$C(n)$
$= C(n/2) + d$ // Using the recursive equation for $C(n)$
$\leq b + d$ // Using the induction hypothesis

But $C(n) = O(1)$ this implies that $C(n) \leq b$. This yields a contradiction. For arbitrary d the equation $b + d \leq b$ is **not** valid. $C(n) = C(n/2)+d$ only implies that $C(n) = O(1)$ when $d = 0$. This is a special case in table 2.

As illustration of these rules we return to the complexity of some of our examples. The number of reduction steps of the factorial example above we have $C(n) = C(n-1) + 3$, hence the complexity is $O(n)$. For the naive Fibonacci function `fib` we have $C(n) = C(n-1) + C(n-2) + 4 \leq 2*C(n-1)$, this justifies our claim that this function has complexity $O(2^n)$. The time complexity to compute element `n` of the list `fibs` is $C(n-1)$ to compute the preceding part of the list, plus two list selections of `n` and `n-1` reductions plus two subtractions and one addition. This implies that $C(n) \leq C(n-1) + 2*n + 4$, so the complexity is indeed $O(n^2)$. For `fib3` we have $C(n) = C(n-1) + 3$. This implies that this function is $O(n)$.

6.2.3 Manipulation recursive data structures

When we try to use the same techniques to determine the complexity of the naive `reverse` function we immediately run into problems. This function is defined as:

```
reverse :: [a] -> [a]
reverse []    = []
reverse [a:x] = reverse x ++ [a]
```

The problem is that the value of the argument is largely irrelevant. The length of the list determines the number of needed reduction steps, not the actual value of these elements. For a list of n elements we have $C(n)$ is equal to the amount of work to reverse a list of length n-1 and the amount of work to append `[a]` to the reversed tail of the list. Looking at the definition of the append operator it is obvious that this takes a number of steps proportional to the length of the first list: $O(n)$.

```
(++) infixr 5 :: ![a] [a] -> [a]
(++) [hd:tl] list = [hd:tl ++ list]
(++) nil     list = list
```

For the function `reverse` we have $C(n) \leq C(n-1) + n + 1$. Hence the function `reverse` is $O(n^2)$.

Again the complexity is a property of the algorithm used, not necessarily of property of the problem. It is the application of the append operator that causes the complexity to grow to $O(n^2)$. Using another definition with an additional argument to hold the part of the list reversed up to the current element accomplish an $O(n)$ complexity.

```
reverse :: [a] -> [a]
reverse l = rev l []
where
    rev [a:x] l = rev x [a:l]
    rev []    l = l
```

For this function we have $C(n) = C(n-1) + 1$. This implies that this function is indeed $O(n)$. It is obvious that we cannot reverse a list without processing each element in the list at least once, this is $O(n)$. So, this is also an under bound.

Using such an additional argument to accumulate the result of the function appears to be useful in many situations. This kind of argument is called an *accumulator*. We will show various other applications of an accumulator in this chapter.

Our next example is a FIFO queue, First In First Out. We need functions to create a new queue, to insert and to extract an element in the queue. In Our first approach the queue is modelled as an ordinary list:

```
:: Queue t :== [t]

new :: Queue t
new = []

ins :: t (Queue t) -> (Queue t)
ins e queue = queue++[e]

ext :: (Queue t) -> (t,Queue t)
ext [e:queue] = (e,queue)
ext _         = abort "extracting from empty queue"
```

Due to the FIFO behavior of the queue the program

```
Start = fst (ext (ins 42 (ins 1 new)))
```

yields `1`. Inserting an element in the queue has a complexity proportional to the length of the queue since the append operator has a complexity proportional to the length of the first list. Storing the list to represent the queue in reversed order makes inserting $O(1)$, but makes extracting expensive. We have to select the last element of a list and to remove the last element of that list. This is $O(n)$.

Using a cleaver trick we can insert and extract elements in a FIFO queue in constant time. Consider the following implementation of the queue:

```
:: Queue t = Queue [t] [t]

new :: Queue t
new = Queue [] []

ins :: t (Queue t) -> (Queue t)
ins e (Queue l m) = (Queue l [e:m])

ext :: (Queue t) -> (t,Queue t)
ext (Queue [e:l] m ) = (e,Queue l m)
ext (Queue _     []) = abort "extracting from empty queue"
ext (Queue _     m ) = ext (Queue (reverse m) [])
```

Inserting an element in the queue is done in constant time. We just add an element in front of a list. Extracting is also done in constant time when the first list in the data structure `Queue` is not empty. When the first list in the data structure is exhausted, we reverse the second list. Reversing a list of length `n` is $O(n)$. We have to do this only after `n` inserts. So, on average inserting is also done in constant time! Again, the complexity is a property of the algorithm, not of the problem.

As a matter of fact lazy evaluation makes things a little more complicated. The work to insert an element in the first queue is delayed until its is needed. This implies that it is delayed until we extract an element from the queue. It holds that inserting and extracting that element is proportional to the amount of elements in the queue.

6.2.4 Estimating the average complexity

The analysis of functions that behave differently based on the value of the list elements is somewhat more complicated. In Chapter 3 we introduced the following definition for insertion sort.

```
isort :: [a] -> [a] | Ord a
isort []    = []
isort [a:x] = Insert a (isort x)

Insert :: a [a] -> [a] | Ord a
Insert e [] = [e]
Insert e [x:xs]
    | e<=x      = [e,x : xs]
    | otherwise = [x : Insert e xs]
```

It is obvious that this algorithm requires one reduction step of the function `isort` for each element of the list. The problem is do determine the number of reductions required by `insert`. This heavily depends on the values in the list used as actual argument. When this list happens to be sorted `e<=x` yields always `True`. Only a single reduction step is required for each insertion in this situation. In this best case, the number of reduction steps is proportional to the length of the list, the complexity is $O(n)$.

When the argument list is sorted in the inverse order and all arguments are different the number of reduction steps required by `Insert` is equal to the number of elements that are already sorted. Hence, the number of reduction steps is `n` for the function `isort` and `1 + 2 + … + (n-1) + n` steps for inserting the next element in the sorted sub-list. From mathematics it is known that

$$1 + 2 + \ldots + (n-1) + n = \sum_{1}^{n} i = \frac{n(n-1)}{2}$$

The total number of reduction steps needed is `n + n*(n-1)/2`. This shows that the worst case complexity of the insertion sort algorithm is $O(n^2)$.

Note that we used a different technique to compute the complexity here. We computed the number of reduction steps required directly. It is often hard to compute the number reductions required directly. Using the recursive function $C(n)$, we have $C(n) = C(n-1) + n$. This implies that the complexity determined in this way is also $O(n^2)$.

The amount of reduction steps needed to sort an arbitrary list will be somewhere between the best case and the worst case. On average we expect that a new element will be inserted in the middle of the list of sorted elements. This requires half of the reduction steps of the worst case behavior. This makes the expected number of reduction steps for an average list `n + n*(n-1)/4`. So, in the average case the complexity of insertion sort is also $O(n^2)$.

As a conclusion we can say that the time taken by insertion sort is essentially quadratic in the length of the list. In the exceptional case that the list is (almost) sorted, the time required is (almost) linear. The space required is always proportional to the length of the list, hence the space complexity is $O(n)$.

The quadratic time complexity of the insertion sort algorithm does not imply that sorting is always $O(n^2)$. We will now look at the time complexity of merge sort. In chapter 3 we implemented this sorting algorithm as:


```
msort :: [a] -> [a] | Ord a
msort xs
    | len<=1    = xs
    | otherwise = merge (msort ys) (msort zs)
where
    ys     = take half xs
    zs     = drop half xs
    half   = len / 2
    len    = length xs

merge :: [a] [a] -> [a] | Ord a
merge [] ys = ys
merge xs [] = xs
merge p=:[x:xs] q=:[y:ys]
    | x <= y    = [x : merge xs q]
    | otherwise = [y : merge p ys]
```

Computation of `len` takes `n` stapes, as usual `n` is the length of the list. The computation of `half` is done in a single step. The functions `take` and `drop` are both $O(n)$. Merging two lists takes $O(n)$ steps. So, one call of `merge` takes $O(n)$ steps plus two times the amount of steps required sorting a list of length `n/2`. This is $C(n) = 2*C(n/2) + c*n$. This implies that merge sort is $O(n*\log n)$. The logarithm reflects the fact that a list of length `n` can be split $^2\log n$ times into lists of length greater than one. On average this algorithm has a better complexity than `isort`.

Our next sorting algorithm to study is quick sort It is defined as:

```
qsort :: [a] -> [a] | Ord a
qsort []     = []
qsort [a:xs] = qsort [x \\ x<-xs | x<a] ++ [a] ++ qsort [x \\ x<-xs | x>=a]
```

In order to compute the complexity of this function we have to consider again various cases. Evaluating the list comprehensions takes $O(n)$ steps, where `n` is the length of the list to process. The first `++` takes a number of steps proportional to the length of the list generated by the first list comprehension. The second `++` takes only 2 steps. This is due to the fact that `++` associates to the right.

Now we consider the case that the input list is sorted. The first list comprehension will yield an empty list, which is sorted in a single reduction step. It takes one step to append the rest of the sorted list to this empty list. It requires $O(n)$ steps to reduce both list comprehensions. The length of the list to sort by the second recursive call of `qsort` is `n-1`. For the total amount of reduction steps we have $C(n) = C(n-1)+2*n+3$. This implies that this 'best case' complexity of `qsort` is $O(n^2)$.

In the 'worst case' situation the list is sorted in the opposite order. The situation is very similar to the best case. The difference with the 'best case' is that all elements of the tail of the input end up in the first recursive call of `qsort` and the second list to sort is empty. Now the first `++` takes `n` reduction steps. This implies that $C(n) = C(n-1)+3*n+2$. The time complexity of the 'worst case' is also $O(n^2)$.

For the average case we assume that both list comprehensions yield a list containing half of the input list. The list comprehensions take both $O(n)$ reductions steps and the first `++` will consume `n/2` steps. This indicates that $C(n) = 2*C(n/2)+c*n$. Hence the `qsort` algorithm is in the average case $O(n*\log n)$. Note that the best case of the input for `isort` is not the best case for `qsort`. In contrast, best case for `isort`, is worst case for `qsort`.

The final sorting algorithm discussed in chapter 3 is tree sort

```
tsort :: ([a] -> [a]) | Eq, Ord a
tsort = labels o listToTree

:: Tree a   = Node a (Tree a) (Tree a) | Leaf
```

```
listToTree :: [a] -> Tree a | Ord, Eq a
listToTree []     = Leaf
listToTree [x:xs] = insertTree x (listToTree xs)

insertTree :: a (Tree a) -> Tree a | Ord a
insertTree e Leaf   = Node e Leaf Leaf
insertTree e (Node x le ri)
    | e<=x  = Node x (insertTree e le) ri
    | e>x   = Node x le (insertTree e ri)

labels :: (Tree a) -> [a]
labels Leaf             = []
labels (Node x le ri)   = labels le ++ [x] ++ labels ri
```

One reduction step of `listToTree` is used for each element in the input list. For `insertTree` again three different cases are considered. When the list is random, the tree will become balanced. This implies that $^2\log n$ reduction steps are needed to insert an element in the tree. When the list is sorted, or sorted in reverse order, we obtain a degenerated (list-like) tree. It will take n reduction steps to insert the next element. The number of reduction steps for the function `labels` depends again on the shape of the tree. When all left sub-trees are empty, there are 3 reduction steps needed for every `Node`. This happens when the input list was inversely sorted. Since insertion is $O(n^2)$ in this situation, the complexity of the entire sorting algorithm is $O(n^2)$. When all right sub-trees are empty, the input was sorted, there are $O(n)$ reduction steps needed to append the next element. Since insertion is $O(n^2)$ in this situation, the entire algorithm is $O(n^2)$.

For balanced trees insertion of one element takes $O(^2\log n)$. Construction of the entire tree requires $O(n*\log n)$ steps. Transforming the tree to a list requires transforming two trees of half the size to lists, $2*C(n-1)$, appending the second list to a list of n/2 elements. For the number of reduction steps we have $C(n) = 2*C(n-1)+n/2+d$. Hence the complexity of transforming the tree to a list is $O(n*\log n)$. This implies that tree sort has complexity $O(n*\log n)$.

Based on this analysis it is hard to say which sorting algorithm should be used. When you know that the list is almost sorted you can use `isort` and you should not use `qsort`. When you know that the list is almost sorted in the inverse order you can use `msort` and you should not use `isort` or `qsort`. For a list that is completely random `qsort` and `msort` a good choices. For an arbitrary list `msort` is a good choice. It is a little more expensive than `qsort` or `tsort` for a complete random list, but it behaves better for sorted lists.

6.2.5 Determining upper bounds and under bounds

Above we have determined the upper bounds of the complexity of various algorithms. When we do this carefully the obtained upper bound is also a tight upper bound. It is clear that this tight upper bound is not necessarily an under bound for the problem. Our first algorithm to reverse list is $O(n^2)$, while the second algorithm is $O(n)$. Also for the sorting algorithm we have a similar situation: the `isort` algorithm has complexity $O(n^2)$, while the exists clearly also sorting algorithms of $O(n*\log n)$.

The question arises whether it is possible to reverse a list in an number of reduction steps that is lower than $O(n)$. This is highly unlikely since we cannot imagine an algorithm that reverses a list without at least one reduction step for each element of the list. Reversing is $\Omega(n)$. Since we have an algorithm with complexity $O(n)$, the complexity of the best reversing algorithms will be $\Theta(n)$.

For sorting algorithms we can also determine an under bound. Also for sorting it is not feasible that there exists a sorting algorithm that processes each list element once. Sorting is at least $\Omega(n)$. We have not yet found a sorting algorithm with this complexity for an average list. Now have to decide whether we start designing better sorting algorithms, or to make a better approximation of the under bound of sorting. For sorting a general list it is not fea-

sible that we can determine the desired position of an element by processing it once. The best we can hope for is that we can determine in which half of the list it should be placed. So, a better approximation of the under bound of sorting is $\Omega(n*\log n)$. Since we know at least one sorting algorithm with this complexity, we can conclude that sorting arbitrary lists is $\Theta(n*\log n)$.

Finding upper bounds of the complexity of an algorithm is not very difficult. When the approximations are made carefully, even determining close upper bounds of the algorithm is merely a matter of counting. Finding tight upper bounds of the problem is more complicated, it involves a study of every algorithm feasible.

Lazy evaluation complicates accurate determination of the number of reduction steps severely. We have always assumed that the entire expression should be evaluated. In the examples we have taken care that this is what happens. However, when we select the first element of a sorted list it is clear that the list will not be sorted entirely due to lazy evaluation. Nevertheless a lot of comparisons are needed that prepares for sorting the entire list. The given determination of the complexity remains valid as an upper bound. Determining the under bound, or accurate number of reduction steps, is complicated by lazy evaluation.

6.3 Constant factors

Above we have emphasized the complexity of problems and hence ignored all constant factors involved. This does not imply that constant factors are not important. The opposite is true, when efficiency is important you have to be keen on reduction steps that can be avoided, even if the overhead is just a constant factor. As a matter of fact, even a precise count of the reduction steps is not the final word, not every reduction steps in CLEAN takes an equal amount of time. So, some experiments can be very useful when the highest speed is desired. See part III for additional information and hints.

The fact that `msort` is $O(n*\log n)$ and sorting is $\Theta(n*\log n)$ does not imply that `msort` is the best sorting algorithm possible. The complexity indicates that for large lists the increase of the time required is proportional to $n*\log n$. For the actual execution time constant factors are important. When we look again at the function `msort`.

```
msort :: [a] -> [a] | Ord a
msort xs
    | len<=1    = xs
    | otherwise = merge (msort ys) (msort zs)
where
    ys   = take half xs
    zs   = drop half xs
    half = len / 2
    len  = length xs
```

It is clear that it can be improved with a constant factor. The functions `take` and `drop` both process the list `xs` until element `half`. This can be combined in a single function `split`:

```
msort2 :: ![a] -> [a] | Ord a
msort2 xs
    | len<=1    = xs
    | otherwise = merge (msort2 ys) (msort2 zs)
where
    (ys,zs) = split (len/2) xs
    len     = length xs

    split :: !Int [a] -> ([a],[a])
    split 0 xs     = ([],xs)
    split n [x:xs] = ([x:xs`],xs``)
    where
        (xs`,xs``) = split (n-1) xs
```

Further analysis shows that there is no real reason to compute the length of the list `xs`. This takes n steps. It is only necessary to split this list into two parts of equal length. This can be done by selecting the odd and even elements. Since we do not want to compute the length

of the list to be sorted, the termination rules should also be changed. This is done in the function `msort3` and the accompanying function `split2`.

```
msort3 :: ![a] -> [a] | Ord a
msort3 []  = []
msort3 [x] = [x]
msort3 xs  = merge (msort3 ys) (msort3 zs)
where
    (ys,zs) = split2 xs

    split2 :: ![a] -> ([a],[a])
    split2 [x,y:r] = ([x:xs],[y:ys]) where (xs,ys) = split2 r
    split2 l       = (l,[])
```

Using accumulators we can avoid the construction of tuples to construct the parts of the list `xs`. In the function `msort4` we call the split function with empty accumulators.

```
msort4 :: ![a] -> [a] | Ord a
msort4 []  = []
msort4 [x] = [x]
msort4 xs  = merge (msort4 ys) (msort4 zs)
where
    (ys,zs) = split3 xs [] []

    split3 :: [a] [a] [a] -> ([a],[a])
    split3 [x,y:r] xs ys = split3 r [x:xs] [y:ys]
    split3 [x]     xs ys = (xs,[x:ys])
    split3 l       xs ys = (xs,ys)
```

Another approach to avoid the computation of the length of the list to be sorted in each recursive call is to determine the length once and pass the actual length as argument to the actual sorting function. Since the supplied length and the actual length should be identical this approach is a little bit more error prone.

A similar technique can be used in the quick sort algorithm. Currently there are two list comprehensions used to split the input list. Using an additional function it is possible to do this in one pass of the input list. This is the topic of one of the exercises.

6.3.1 Generating a pseudo random list

In order to investigate whether the reduced amount of reduction steps yields a more efficient algorithm we need to run the programs and measure the execution times. Since some of the sorting programs are sensitive to the order of elements in the input list we want to apply the sorting functions to a list of random numbers

Due to the referential transparency random number generation is somewhat tricky in functional programs. When one random number is needed we can use for instance the seconds from the clock. However, when we fill a list of numbers in this way the numbers will be strongly correlated. The solution is to use *pseudo-random numbers*. Given a *seed* the next number can be generated by the *linear congruetial method*.

```
nextRan :: Int -> Int
nextRan s = (multiplier*s + increment) mod modulus
```

The constants `multiplier`, `increment` and `modulus` are suitable large numbers. In the examples below we will use the values:

```
multiplier  :== 26183
increment   :== 29303
modulus     :== 65536   //  this is 2^16
```

A sequence of these pseudo random numbers can be generated by:

```
ranList :: [Int]
ranList = iterate nextRan seed
```

The `seed` can be obtained for instance form the clock or be some constant. To compare the sorting functions we will use the constant `42`, this has the advantage that each sorting function has the same input.

The only problem with these numbers is the possibility of cycles. When `nextRan n = n`, or `nextRan (nextRan n) = n`, or `nextRan (nextRan ( ..n)..) = n` for some `n`, there will be a cycle in the generated list of random numbers as soon as this `n` occurs once. When the constants are well chosen there are no troubles with cycles. In fact `nextRan` is an ordinary referential transparent function. It will always yield the same value when the same seed is used. This implies that `ranList` will start a cycle as soon as an element occurs for the second time. It is often desirable that the same number can occur twice in the list of random numbers used without introducing a cycle. This can be achieved by scaling the random numbers.

When we needed random numbers in another range we can scale the numbers in `ranList`. A simple approach is:

```
scaledRans :: Int Int -> [Int]
scaledRans min max = [i mod (max-min+1) + min \\ i <- ranList]
```

When you need random numbers with a nice equal distribution over the whole range this function is only appropriate when `modulus` is a multiple of the range. Consider the situation where `modulus` is about 1.5*range. Numbers in the first part of the range will occur twice as much as the other numbers. When `modulus` is much larger than the range, the ratio 2/1 changes to $n+1/n$. For large n this is a good approximation of 1. The problem can be avoided when by using a slightly more complex function to scale the random numbers.

```
scaledRans :: Int Int -> [Int]
scaledRans min max = [i \\ i <- [i/denominator + min \\ i <- ranList] | i<=max]
where denominator = modulus / (max-min+1)
```

The distribution of these numbers is very well equal.

When only the flatness of the distribution count you can also use the list obtained by `flatten (repeat [min..max])`. It is of course far from random.

6.3.2 Measurements

In order to achieve a reasonable execution time[1] we sort a list of 1000 number 100 times and a list of 4000 elements 100 times. We prefer to sort a shorter list 100 times instead of a long list once in order to reduce the overhead of generating the input list and to reduce the amount of heap space necessary. By selecting one element of the list we prevent that a lot of time is spent on outputting the list. We select the last element in order to be sure that the entire list is sorted. This is achieved by the following program. At the position *sort* we substitute the name of the function to investigate. The `let!` construct is used to force evaluation.

```
Start = ntimes 100

ntimes n = let! e = last (sort rlist) in
           case n of
              0 -> e
              n -> ntimes (n-1)

rlist =: take 1000 ranList
```

The results are listed in the following table. As a reference we included also quick sort, tree sort and insertion sort. The functions `qsort2` and `tsort2` are variants of `qsort` and `tsort` introduced below. The result for the function `I` shows that the time to select the last element and to generate the list to process can be neglected. The function `msort2`` is the function `msort2` where `split` is replaced by the function `splitAt` from the standard environment.

[1]We used a Macintosh Performa 630 for the experiments. The application gets 1 MB of heap space and does use stack checks.

function	1000 elements			4000 elements			ratio		
	execution	gc	total	execution	gc	total	execution	gc	total
msort	9.23	0.56	9.80	44.8	13.85	58.66	4.85	24.73	5.98
msort2	6.23	0.31	6.55	30.63	8.00	38.63	4.92	25.81	5.90
msort2`	10.71	0.76	11.48	51.51	19.81	71.33	4.81	22.51	6.21
msort3	10.11	0.88	11.00	49.26	20.70	69.69	4.87	23.52	6.33
msort4	6.06	0.30	6.36	30.51	11.35	41.86	5.03	23.65	6.58
qsort	8.21	0.48	8.70	39.91	11.30	51.21	4.86	23.54	5.88
qsort2	6.51	0.30	6.81	31.81	6.83	38.65	4.89	22.77	5.67
tsort	9.28	0.53	9.81	38.11	9.70	47.81	4.11	18.30	4.87
tsort2	7.50	0.41	7.91	30.98	7.46	38.45	4.13	18.20	4.86
isort	105.08	3.85	108.93	1809.93	288.68	2098.61	17.22	74.98	19.27
I	0.11	0.00	0.11	0.46	0.00	0.46	4.18	-	4.18

Table 3: **Execution-, garbage collection-, and total time in seconds of various sorting algorithms.**

It is clear that any algorithm with complexity O(n log n) is much better on this size of input lists than isort with complexity $O(n^2)$. Although there is some difference between the various variants of the merge sort algorithm, it is hard to predict which one is the best. For instance, the difference between msort2` and msort2 is caused by an optimization for tuples that does not work across module boundaries. You can't explain this difference based on the function definitions. Hence, it is hard to predict this.

The complexity theory predicts that the ratio between the execution speed for the programs with complexity O(n log n) is 4.80, for an algorithm of $O(n^2)$ this ratio is 16. These numbers correspond pretty well with the measured ratios. Only the time needed by the tree sorts grows slower than expected. This indicates that the used lists are not large enough to neglect initialization effects.

You can see also that the amount of garbage collection time required grows much faster than the execution time. The amount of garbage collection needed is determined by the number of nodes used during execution and the number of nodes that can be regained during garbage collection. For a large list, less memory can be regained during garbage collection, and hence more garbage collections are needed. This increases the total garbage collection time faster as you might expect based on the amount of nodes needed. To reduce the amount of garbage collection time the programs should be equipped with more memory.

For the user of the program only the total execution time matters. This takes both the pure reduction time as well as the time consumed by garbage collections into account. The total execution time is dependent of the amount of heap space used by the program.

Another thing that can be seen from this table is that it is possible to optimize programs by exploring the efficiency of some educated guesses. However, when you use a function with the right complexity it is only worthwhile to use a more complicated algorithm when the execution speed is a of prime interest. The difference in speed between the various sorting algorithms is limited. We recommend to use one of the merge sort functions to sort list of an unknown shape. Quick sort and tree sort behave very well for random list, but for sorted list they are $O(n^2)$. This implies that the execution time will be much longer.

6.3.3 Other ways to speed up programs

Another way to speed up programs is by exploiting *sharing*. In the Fibonacci example above we saw that this can even change the complexity of algorithms. In the program used to measure the execution time of sorting functions we shared the generation of the list to be sorted. Reusing this lists of saves a constant factor for this program.

There are many more ways to speed up programs. We will very briefly mention two other possibilities. The first way to speed up a program is by executing all reduction steps that does not depend on the input of the program before the program is executed. This is called *partial evaluation* [Jones 95]. A way to achieve this effect is by using macro's whenever possible. More sophisticated techniques also look at applications of functions. A simple example illustrates the intention.

```
power :: Int Int -> Int
power 0 x = 1
power n x = x*power (n-1) x

square :: Int -> Int
square x = power 2 x
```

Part of the reduction steps needed to evaluate an application of square does not depend on the value x. By using partial evaluation it is possible to transform the function square to

```
square :: Int -> Int
square x = x*x*1
```

Using the mathematical law x*1 = x, it is even possible to achieve:

```
square :: Int -> Int
square x = x*x
```

The key idea of partial evaluation is to execute rewrite steps that does not depend on the arguments of a function before the program is actually executed. The partially evaluated program will be faster since the number of reduction steps needed to execute is lower.

The next technique to increase the efficiency of programs to combine the effects of several functions to one function. This is called *function fusion*. We will illustrate this with the function qsort as example. This function was defined as:

```
qsort :: [a] -> [a] | Ord a
qsort []     = []
qsort [a:xs] = qsort [x \\ x<-xs | x<a] ++ [a] ++ qsort [x \\ x<-xs | x>=a]
```

At first glance there is only one function involved here. A closer look shows that also the operator ++ is to be considered as a function application. As a matter of fact also the list comprehensions can be seen as function applications. The compiler will transform them to ordinary function applications. We will restrict ourselves here to the function qsort and the operator ++. When qsort has sorted some list of elements smaller than the first element, the operator ++ scans that list in order to append the other elements of the list. During the scanning of the list all cons nodes will be copied. This is clearly a waste of work.

Copying cons nodes can be avoided by using a *continuation* argument in the sorting function. The continuation determines what must be done when this function is finished. In this example what must be appended to the list when the list to be sorted is exhausted. Initially there is nothing to be appended to the sorted list. We use an additional function, qs, that handles sorting with continuations.

```
qsort2 :: [a] -> [a] | Ord a
qsort2 l = qs l []
```

The continuation of sorting all elements smaller than the first element is the sorted list containing all other elements. Its continuation is the continuation supplied as argument to qs. When the list to be sorted is empty we continue with the continuation c.

```
qs :: [a] [a] ->[a] | Ord a
qs []     c = c
qs [a:xs] c = qs [x \\ x<-xs | x<a] [a:qs [x \\ x<-xs | x>=a] c]
```

The trace of the reduction of a small example clarifies the behaviour of these functions:

```
qsort2 [1,2,1]
→  qs [1,2,1] []
→  qs [] [1:qs [2,1] []]
→  [1:qs [2,1] []]
→  [1:qs [1] [2:qs [] []]]
→  [1:qs [] [1:qs [] [2:qs [] []]]]
→  [1:[1:qs [] [2:qs [] []]]]
→  [1:[1:[2:qs [] []]]]
→  [1:[1:[2:[]]]]
=  [1,1,2]
```

```
qsort [1,2,1]
→  qsort []++[1]++qsort [2,1]
→  []++[1]++qsort [2,1]
→  [1]++qsort [2,1]
→  [1:[]++qsort [2,1]]
→  [1:qsort [2,1]]
→  [1:qsort [1]++[2]++qsort []]
→  [1:qsort []++[1]++qsort []++
                          [2]++qsort []]
→  [1:[]++[1]++qsort []++[2]++qsort []]
→  [1:[1]++qsort []++[2]++qsort []]
→  [1:1:[]++qsort []++[2]++qsort []]
→  [1:1:qsort []++[2]++qsort []]
→  [1:1:[]++[2]++qsort []]
→  [1:1:[2]++qsort []]
→  [1:1:2:[]++qsort []]
→  [1:1:2:qsort []]
→  [1:1:2:[]]
=  [1,1,2]
```

Figure 6.1: **Traces showing the advantage of using a continuation in the quick sort algorithm.**

It is obvious that the version of quick sort using continuations requires a smaller number of reduction steps. This explains why it is 25% faster.

In the same spirit we can replace the operator `++` in the tree sort function, `tsort`, by a continuation.

```
tsort2 :: ([a] -> [a]) | Eq, Ord a
tsort2 = swap labels2 [] o listToTree

labels2 :: (Tree a) [a] -> [a]
labels2 Leaf            c = c
labels2 (Node x le ri)  c = labels2 le [x: labels2 c ri]
```

As shown in table 3 above, this function using continuations is indeed (about 20%) more efficient.

6.4 Exploiting Strictness

As explained in previous chapters, a function argument is strict when its value is needed always in the evaluation of a function call. Usually an expression is not evaluated until its value is needed. This implies that expressions causing non-terminating reductions or errors and expressions yielding infinite data structures can be used as function argument. Problems do not arise until the value of such expressions is really needed.

The price we have to pay for lazy evaluation is a little overhead. The graph representing an expression is constructed during a rewrite step. When the value of this expression is needed the nodes in the graph are inspected and later on the root is updated by the result of the rewrite process. This update is necessary since the node may be shared (occurring at several places in the expression). By updating the node in the graph re-computation of its value is prevented.

When the value of a node is know to be needed it is slightly more efficient to compute its value right away and store the result directly. The sub-expressions which are known to be needed anyway, are called *strict*. For these expressions there is no reason to store the expression and to delay its computation until it is needed. The CLEAN compiler uses this to evaluate strict expressions at the moment they are constructed. This does not change the number of reduction steps. It only makes the reduction steps faster.

The CLEAN compiler uses basically the following rules to decide whether an expression is strict:

1) The root of the right-hand side is a strict expression. When a function is evaluated this is done since its value is needed. This implies that also the value of its reduct will be needed. This is repeated until the root of the right hand side cannot be reduced anymore.
2) Strict arguments of functions occurring in a strict context are strict expressions. The function is known to be needed since it occurs in a strict context. In addition it is known that the value of the strict arguments is needed when the result of the function is needed.

These rules are recursively applied to determine as much strict sub-expressions as possible. This implies that the CLEAN compiler can generate more efficient programs when strictness of function arguments is known. Generally strictness is an undecidable property. We do not make all arguments strict in order to be able to exploit the advantages of lazy evaluation. Fortunately, any safe approximation of strictness helps to speed up programs. The CLEAN compiler itself is able to approximate the strictness of function arguments. The compiler uses a sophisticated algorithm base on abstract interpretation [Plasmeijer 94]. A simpler algorithm to determine strictness uses the following rules:

1) Any function is strict in the first pattern of the first alternative. The corresponding expression should be evaluated in order to determine whether this alternative is applicable. This explains why the append operator, `++`, is strict in its first argument.

```
(++) infixr 5 :: ![x] [x] -> [x]
(++) [hd:tl] list = [hd:tl ++ list]
(++) nil     list = list
```

   Since it is generally not known how much of the generated list is needed, the append operator is not strict in its second argument.
2) A function is strict in the arguments that are needed in all of its alternatives. This explains why the function `add` is strict in both of its arguments and `mul` is only strict in its first argument. In the standard environment both `+` and `*` are defined to be strict in both arguments.

```
mul :: !Int Int -> Int;
mul 0 y = 0
mul x y = x*y

add :: !Int !Int -> Int;
add 0 y = y
add x y = x+y
```

You can increase the amount of strictness in your programs by adding strictness information to function arguments in the type definition of functions. Sub-expressions that are known to be strict, but which do not correspond to function arguments can be evaluated strict by defining them as strict local definitions using `#!` or `let!`.

6.5 Unboxed values

Objects that are not stored inside a node in the heap are called *unboxed* values. These unboxed values are handled very efficiently by the CLEAN system. In this situation the CLEAN system is able to avoid the general graph transformations prescribed in the semantics. It is the responsibility of the compiler to use unboxed values and to do the conversion with nodes in the heap whenever appropriate. Strict arguments of a basic type are handled as unboxed value in CLEAN. Although the compiler takes care of this, we can use this to speed up our programs by using strict arguments of a basic type whenever appropriate.

We illustrate the effects using the familiar function length. A naive definition of `Length` is:

```
Length :: ![x] -> Int
Length [a:x] = 1 + Length x
Length []    = 0
```

A trace shows the behavior of this function:

```
Length [7,8,9]
→  1 + Length [8,9]
→  1 + 1 + Length [9]
→  1 + 1 + 1 + Length []
→  1 + 1 + 1 + 0
→  1 + 1 + 1
→  1 + 2
→  3
```

The CLEAN system builds an intermediate expression of the form `1 + 1 + … + 0` of a size proportional to the length of the list. Since the addition is known to be strict in both arguments, the expression is constructed on the stacks rather then in the heap. Nevertheless it takes time and space.

Construction of the intermediate expression can be avoided using an accumulator: a counter indicating the length of the list processed until now.

```
LengthA :: ![x] -> Int
LengthA l = L 0 l
where
    L :: Int [x] -> Int
    L n [a:x] = L (n+1) x
    L n []    = n
```

The expression `Length [7,8,9]` is reduced as:

```
LengthA [7,8,9]
→  L 0 [7,8,9]
→  L (0+1) [8,9]
→  L ((0+1)+1) [9]
→  L (((0+1)+1)+1) []
→  ((0+1)+1)+1
→  (1+1)+1
→  2+1
→  3
```

The problem with this definition is that the expression used as accumulator grows during the processing of the list. Evaluation of the accumulator is delayed until the entire list is processed. This can be avoided by making the accumulator strict.

```
LengthSA :: ![x] -> Int
LengthSA l = L 0 l
where
    L :: !Int [x] -> Int
    L n [a:x] = L (n+1) x
    L n []    = n
```

In fact the CLEAN system itself is able to detect that this accumulator is strict. When you don't switch strictness analysis off, the CLEAN system will transform `LengthA` to `LengthSA`. The trace becomes:

```
LengthSA [7,8,9]
→  L 0 [7,8,9]
→  L (0+1) [8,9]
→  L 1 [8,9]
→  L (1+1) [9]
→  L 2 [9]
→  L (2+1) []
→  L 3 []
→  3
```

Since the accumulator is a strict argument of a basic type, the CLEAN system avoids the construction of data structures in the heap. An unboxed integer will be used instead of the nodes in the heap. In table 4 we list the run time of some programs to illustrate the effect of strictness. We used a Power Macintosh 7600 to compute 1000 times the length of a list of 10000 elements. The application had 400 KB of heap. The difference between the programs is the function used to determine the length of the list.

function	execution	gc	total
Length	5.65	0.01	5.66
LengthA	20.76	86.45	107.21
LengthSA	2.70	0.0	2.70

Table 4: **Runtime in seconds of a program to determine the length of a list.**

Adding a lazy accumulator has a severe runtime penalty. This is caused by that fact that all computations are now done in a lazy context. The intermediate expression `1+1+…+0` is constructed in the heap. Adding the appropriate strictness information makes the function to compute the length of a list twice as efficient as the naive definition. Adding this strictness information improves the efficiency of the computation of the length of a list using an accumulator by a factor of 40. The overloaded version of this function defined in the standard environment does use the efficient algorithm with a strict accumulator.

Adding a strictness annotation can increase the efficiency of the manipulation of basic types significantly. You might even consider adding strictness annotations to arguments that are not strict in order to increase the efficiency. This is only save when you know that the corresponding expression will terminate.

As example we consider the function to replace a list of items by a list of their indices:

```
indices :: [x] -> [Int]
indices l = i 0 l
where   i :: Int [x] -> [Int]
        i n []    = []
        i n [a:x] = [n: i (n+1) x]
```

The local function `i` is not strict in its first argument. When the list of items is empty the argument `n` is not used. Nevertheless, the efficiency of the function `indices` can be doubled (for a list of length 1000) when this argument is made strict by adding an annotation. The cost of this single superfluous addition is outweighed by the more efficient way to handle this argument.

We have seen another example in the function `fib4` to compute Fibonacci numbers in linear time:

```
fib4 n = f n 1 1
where   f :: !Int !Int !Int -> Int
        f 0 a b = a
        f n a b = f (n-1) b (a+b)
```

Making `f` strict in its last argument does cause that one addition is done too much (in the last iteration of `f` the last argument will not be used), but makes the computation of `fib4 45` twelve times as efficient. When `f` evaluates all its arguments lazy, the Fibonacci function slows down by another factor of two.

6.6 The cost of Currying

All functions and constructors can be used as Curried in CLEAN. Although you are encouraged to do this whenever appropriate, there are some runtime costs associated with Currying. When speed becomes an issue it may be worthwhile to consider the elimination of some heavily used Curried functions from you program.

The cost of Currying are caused by the fact that it is not possible to detect at compile time which function is applied and whether it has the right number of arguments. This implies that this should be done at runtime. Moreover certain optimizations cannot be applied for Curried functions. For instance, it is not possible to use unboxed values for strict arguments of basic types. The CLEAN system does not know which function will be applied. Hence, it cannot be determined which arguments will be strict. This causes some additional loss of efficiency compared with a simple application of the function.

To illustrate this effect we consider the function sum to compute the sum of a list of integers. The naive definition is:

```
Sum :: ![Int] -> Int
Sum [a:x] = a + Sum x
Sum []    = 0
```

Using the appropriate fold function this can be written as Foldr (+) 0. Where Foldr is defined as:

```
Foldr :: (a b -> b) b ![a] -> b
Foldr op r [a:x] = op a (Foldr op r x)
Foldr op r []    = r
```

In the function Sum the addition is treated as an ordinary function. It is strict in both arguments and the arguments are of the basic type Int. In the function Foldr the addition is a curried function. This implies that the strictness information cannot be used and the execution will be slower. Moreover it must be checked whether op is a function, or an expression like I (+) that yields a function. Also the number of arguments needed by the function should be checked. Instead of the ordinary addition there can be a weird function like \n -> (+) n, this function takes one of the arguments and yield a function that takes the second argument. Even when these things does not occur, the implementation must cope with the possibility that it is there at runtime. For an ordinary function application, it can be detected at compile time whether there is an ordinary function application.

The function foldr from the standard environment eliminates these drawbacks by using a macro:

```
foldr op r l :== foldr r l
where
    foldr r []    = r
    foldr r [a:x]   = op a (foldr r x)
```

By using this macro, a tailor made foldr is created for each and every application of foldr in the text of your CLEAN program. In this tailor made version the operator can usually be treated as an ordinary function. This implies that the ordinary optimizations will be applied.

As argued above, it is better to equip the function to sum the elements of a list with an accumulator.

```
SumA :: ![Int] -> Int
SumA l = S 0 l
where
    S :: !Int ![Int] -> Int
    S n [a:x] = S (n+a) x
    S n []    = n
```

The accumulator argument n of the function SumA is usually not considered to be strict. Its value will never be used when SumA is applied to an infinite list. However, the function SumA will never yield a result in this situation.

The same recursion pattern is obtained by the expression Foldl (+) 0. This fold function can be defined as:

```
Foldl :: (b a -> b) !b ![a] -> b
Foldl op r [a:x] = Foldl op (op r a) x
Foldl op r []    = r
```

The second argument of this function is made strict exactly for the same reason as in SumA. In StdEnv also this function is defined using a macro to avoid the cost of Currying:

```
foldl op r l :== foldl r l
where
    foldl r []      = r
    foldl r [a:x]   = foldl (op r a) x
```

We will compare the run time of programs computing 1000 times the sum of the list [1..1000] in order to see the effects on the efficiency.

function	execution	gc	total
Sum	6.00	0.00	6.00
Foldr (+) 0	30.13	7.53	37.66
foldr (+) 0	6.01	0.00	6.01
SumA	2.51	0.00	2.51
Foldl (+) 0	19.66	2.55	22.21
foldl (+) 0	2.68	0.01	2.68

Table 5: **Runtime in seconds of a program to determine the costs of Currying.**

The table shows the impact of omitting all strictness information, also the strictness analyser of the CLEAN system is switched off. The only remaining strictness information is the strictness of the operator + from StdEnv.

function	execution	gc	total
Sum	6.03	0.00	6.03
Foldr (+) 0	29.13	7.75	37.41
foldr (+) 0	6.13	0.00	6.13
SumA	16.13	3.70	19.83
Foldl (+) 0	37.86	7.16	45.03
foldl (+) 0	15.61	3.50	19.11

Table 6: **Runtime in seconds of a program to determine the costs of Currying without strictness.**

From these tables we can conclude that there are indeed quite substantial costs involved by using Curried functions. However, we used a Curried function manipulating strict arguments of a basic type here. The main efficiency effect is caused by the loose of the possibilities to treat the arguments as unboxed values. For functions manipulating ordinary datatypes the cost of Currying are much smaller. When we use the predefined folds from StdEnv there is no significant overhead in using Curried functions due to the macro's in the definition of these functions.

6.6.1 Folding to the right or to the left

Many functions can be written as a fold to the left, foldl, or a fold to the right, foldr. As we have seen in section 1, there are differences in the efficiency. For functions like sum it is more efficient to use foldl. The argument e behaves as an accumulator.

A function like reverse can be written using foldl and using foldr:

```
reversel l = foldl (\r x -> [x:r]) [] l

reverser l = foldr (\x r -> r++[x]) [] l
```

Difference in efficiency depends on the length of the list used as argument. The function reverser requires a number of reduction steps proportional to the square of the length of the list. For reversel the number of reduction steps is proportional to the length of the list. For a list of some hundreds of elements the difference in speed is about two orders of magnitude!

Can we conclude from these example that it is always better to use foldl? No, life is not that easy. As a counter example we consider the following definitions:

```
el = foldl (&&) True (repeat 100 False)
er = foldr (&&) True (repeat 100 False)
```

When we evaluate el, the accumulator will become False after inspection of the first Boolean in the list. When you consider the behaviour of && it is clear that the result of the entire expression will be False. Nevertheless, you program will apply the operator && to all other Booleans in the list.

However, we can avoid this by using foldr. This is illustrated by the following trace:

```
foldr (&&) True (repeat 100 False)
→  foldr (&&) True [False: repeat (100-1) False]
→  (&&) False (foldr (&&) True (repeat (100-1) False))
→  False
```

That does make a difference! As a rule of thumb you should use foldl for operators that are strict in both arguments. For operators that are only strict in their first argument foldr is a better choice. For functions like reverse there is not a single operator that can be used with foldl and foldr. In this situation the choice should be determined by the complexity of the function given as argument to the fold. The function `\r x -> [x:r]` in foldl requires a single reduction step. While the function `\x r -> r++[x]` from foldr takes a number of reduction steps proportional to the length of r. Hence foldl is much better in this example than foldr. However, in a map or a filter the function foldr is much better than foldl. Hence, you have to be careful for every use of fold.

It requires some practice to be able to write functions using higher-order list manipulations like fold, map and filter. It takes some additional training to appreciate this kind of definitions. The advantage of using these functions is that it can make the recursive structure of the list processing clear. The drawbacks are the experience needed as a human to read and write these definitions and some computational overhead.

6.7 Exercises

1. In the function qsort there are two list comprehensions used to split the input list. It is possible to split the input list in one pass of the input list similar to msort4. Using an additional function it is possible to do this in one pass of the input list. Determine whether this increases the efficiency of the quick sort function.
2. To achieve the best of both worlds in the quick sort function you can combine splitting the input in one pass and continuation passing. Determine whether the combination of these optimizations does increase the efficiency.
3. When the elements of a list can have the same value it may be worthwhile to split the input list of the quick sort function in three parts: one part less than the first element, the second part equal to the first element and finally all elements greater than the first element. Implement this function and determine whether it increases the speed of sorting the random list. We can increase the amount of duplicates by appending the same list a number of times.
4. Determine and compare the runtime of the sorting functions msort4, qsort4 (from the previous exercise), tsort2 and isort for non-random lists. Use a sorted list and its reversed version as input of the sorting functions to determine execution times. Determine which sorting algorithm is the best.
5. Investigate whether it is useful to pass the length of the list to sort as additional argument to merge sort. This length needs to be computed only once by counting the elements. In recursive calls it can be computed in a single reduction step. We can give the function split a single argument as accumulator. Does this increase the efficiency?
6. Determine the time complexity of the functions qsort2 and tsort2.
7. Study the behavior of sorting functions for sorted lists and inversely sorted lists.

Index

Index to keywords and functions of part I.

-

- 60, 84, 87, 129
-> 14, 29
--> 115

!

!! 42

#

#-definition 103, 110

&

&& 170

*****

* 60, 84, 100
*/ 12

/

/ 60, 84
/* 12
// 12

:

: 39

****

\ 29
\\ 48

{

{!Int} 62
{#Char} 84
{#Int} 62
{Int} 61

+

+ 20, 56, 60, 84, 87, 129, 152
++ 44, 104, 155, 164, 166

<

< 43, 84
<- 48, 49
<<< 113
<= 43
<> 43

=

=: 19
== 43, 84, 129

>

> 43
>= 43

A

about dialog 142
abstract datatype 70
abstract interpretation 166
accumulator 161, 167, 168, 169
actual argument 4
actual parameter 4
ad-hoc polymorphism 83
after 30
algebraic datatype 64, 65, 70
annotation 16
anonymous attribute variables 105
append 155
approximation 35
argument 4
array 25, 61, 108
array comprehension 62
association 24
associativity 24
attribute variable 104
average 56
average complexity 157

B

beep 144
bind 114
binomial coefficient 3
black hole 79
Bool 37
Boole 6
Boolean 6
Boolean value 6
bottom-up 34

C

caching 153
CAF 19, 153
call-back function 117
case 8
case distinction 72, 73
changing a figure 132
Church number 152
class 16
class declaration 84
class instance 16, 17, 84
class variable 84
CLEAN compiler 2, 17, 165
CLEAN system 80
clearness 149
CloseEvents 116
closefile 111
closeProcess 117
co-domain 12
coerced 101
coercion 107
coercion statement 104
comment 12
complexity 150, 154
composition 30
comprehension 48, 62
concatenating 43
conditional 33
cons 39
console 111
constant 4, 19, 151
constant applicative form 19
constant factor 160
constructor 19, 64, 105
continuation 131, 164
control 118
coordinate 129
correct 71
correctness 149
cubic 151
curried 107
curried function 26
Curry 26
Currying 27, 168
Curve 92
cycle in spine 79

D

data abstraction 92
data constructor 64
data path 103
data structure 37, 65, 155
datatype 64
day 31
default instance 88
definition module 20
denominator 60, 86
derivative function 35
derived class member 88
destructive update 99, 109
dialog 116, 118, 127
dialog definition 122
direct proof 72
domain 12, 30
dot-dot expression 39
Drawable 92
drawing 140
drawing lines 131
drop 41, 160
dropWhile 45

E

eager evaluation 52
efficiency 149
empty list 38
encapsulation 70
enumeration 38
enumeration type 64
environment passing 112
environment passing. 99
Eq 88
equal 72
Eratosthenes 55
error 77
  cycle in spine 79
  heap full 79
  illegal instruction 80
  index out of range 78
  stack overflow 80
  syntax 12
  type 13, 17, 113
  uniqueness 101
essentially unique 107
eureka definition 76
event 116
event handling 117
event-handler 117
execution time 149
existential type variable 91
existentially quantified type variables 91
exponential 151
exported function 70

F

factorial 3, 152
False 6
FAppendText 110
fclose 112
Fibonacci 73, 152, 168
Fibonacci number 153
FIFO queue 156

file 98, 109, 113, 144
file system 110
Files 110
filter 28, 44
first-order function 27
flatten 44
Floating-point 5
fold 169, 170
folding 45
foldl 45, 170
Foldl 169
foldr 45, 169, 170
Foldr 169
font 136
fopen 110
formal argument 4
formal parameter 4
FReadText 110
from 50
fromInt 87
fst 55
function application 25
function composition 30
function equality 72
function fusion 164
functions on functions 6
funny character 19
fwritec 98, 107
FWriteText 110

G

garbage 134
garbage collection 79, 163
gcd 60
generator 48
GigaByte 151
graph reduction 96, 134
greatest common divisor 60
guard 8

H

Hamming number 79
HandleMouse 144
handler 117
HASKELL 1
hd 40
head 40
heap 79, 167
heap full 79
heap full error 80
hello world 111, 119, 130
higher-order function 1, 27, 107
higher-order type 89
highlighting 135
Horizontal 133

I

Id 120
if 33
illegal instruction 80
implementation module 20
index out of range 78
indirect sharing 103
induction 73
infinite data structure 77
infix 23
infix operator 30
infixl 23
infixr 23
inform 128
inherited 88
init 40
insertion sort 47, 157
instance 16, 17, 84
instance declaration 84
Integer 5
interface 91
Internal overloading 88
interval notation 39
IO-state 117
isMember 42
isort 47
iterate 29, 53
iteration 28

K

key modifier 137
keyboard 137, 145
keyboard function 129

L

lambda expression 29
lambda notation 29, 30
last 40
layout rule 12
lazy evaluation 51, 96, 165
leap year 33
left upper corner 129
length 42
let 11
let! construct 111
levels of priority 24
lexicographical ordering 43
line 140
Line 92
linear 151
linear congruetial method 161
Linux 120
LISP 1
list 3, 6, 14, 28, 37, 64, 65, 67
List 105
list comprehension 48, 62, 132
local state 120
locally defined 10
Logarithmic 151
look 134
Look 129
lowercase 19

M

macro 19, 169
map 28, 44, 46
members of a class 86
memorization 153
memory use 133
MenuDef 120
menu-element 120
menu-system 120, 132
merge 47
merge sort 48, 157
meta-key 137
MIRANDA 1
ML 1
modal dialog 118
modeless dialog 118
module 2, 20
monad 114
mouse 144
mouse click 116
mouse event 138
mouse function 129, 138
mouse handler 138
mouse handling 143

N

names of function 4
nested generator 49
nested scope 102, 113
nil 39
normal form 4, 72
notice 127, 142, 145
numerator 60, 86

O

o 30
object 91
offered type 101
one 56, 84, 152
O-notation 150
open file 110
openfile 111
OpenNotice 128
operator 23, 24, 25, 30
order of evaluation 98
origin 129
Oval 92
overloaded 16
overloading 60
overloading 83

P

parallel combination of generators 49
partial application 107
partial evaluation 164
partial function 34, 78
partial parameterization 26
partially parameterized function 26
pattern 8
Peano curve 131
Peano number 152
Picture 92
picture area 129
Pipe 95
pixel 129
PlusMin 86
Point 92, 129
polymorphic 15
polymorphic type 15
predicate 6
prefix 23
prime 31, 54
priority 23
private access 101
program synthesis 76
proof 71
propagation of uniqueness 103
propagation rule 103
pseudo-random 161

Q

Q 59, 86
qsort 164
quadratic 151
queue 156
quick sort 50, 158, 164
Quit 121
quotient 31

R

random number 161
range 12, 30
ratio 59
rational number 59, 86
real 5
record 25, 58
Rectangle 92
recursion 10, 73
recursive data structure 155
recursive function 9
redex 4
redex 96
reduct 98
reduction 4, 96
reduction step 4, 150, 152
reference count 98
reference count 98
referential transparency 96
referential transparency 98
rem 31
remainder 31
repeat 52
reserved symbol 5
result type 101
reverse 42, 75, 155, 170

S

Schönfinkel 26
scope 10, 11, 12, 29, 32, 49, 83, 102, 110, 113
scroll 136, 142
search tree 67
selection 25
  by index 62
`seq` 113
`seq` 95
set 48
`setWindowLook` 134
`sfopen` 110
sharing 103, 134, 163
side-effect 71
side-effects 96
signature 84
singleton list 38
snd 56
sort 50
sorting 47, 69, 157
space complexity 150
speed up program 163
spine of a list 38
splitAt 56, 162
`sqrt` 35
square root 35
stack 70, 80, 167
Stack 70
stack overflow 80
standard environment 2
standard library 2
`Start` 2, 4
`startMDI` 117
`StartMDI` 117
state transition function 120
static type system 12
StdDebug 81
`StdEnv` 2, 20, 24, 84, 169
`stderr` 81, 115
`StdFileSelect` 122, 144
`StdInt` 20
`stdio` 112
strict 52, 165
strictness 16, 166
string 14, 63, 84
strongly typed 13
strongly typed 83
subclass 85
`Succ` 152
synonym 19
synonym type 71
syntax 8
synthesis 76

T

tail 40
`take` 41, 160
`takeWhile` 45
termination 77
test 122
test function 125
timer 139, 142, 146
tl 40
top down 141
top-down 34
`toString` 87
total function 34
trace 80, 81, 115
tree 65, 67, 105
tree sort 69, 158, 165
`True` 6
truth value 6
tsort 69, 165
tuple 55
type 1, 12, 17
type constructor 89
type constructor class 89
type context 85
type declaration
  advantage of 14
type errors 113
type synonym 19, 20, 84
type variable 15
types of data constructors 105

U

unboxed value 166
unfold 76
unification 13
uniform substitution 96
unique 99
unique object 103
unique type 100
uniqueness 17, 114
uniqueness property 100
universally quantified 91
Unix 120
`until` 28
update function 129
update of an array
  destructive 63
upper bound 150, 151
uppercase 19

V

`Vertical` 133

W

warn 128
`where` 10, 29
window 129
window look 134
`WindowLook` 129
`World` 110
write to file 109, 111, 113

X

x-coordinate 129

Y

y-coordinate 129

Z

`zero` 56, 84, 87, 88, 129, 141, 152
`Zero` 152
zip 57

Θ

Θ-notation 151

Ω

Ω-notation 151

λ

λ 29