Part I
Chapter 1
Introduction to Functional Programming

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 how a result can be computed, given some parameters. 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 parameter. 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 where this idea has been exploited was Lisp, developed in the early 50s 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 parameter 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, it can be checked by a compiler that whether it actually makes sense to apply a given function to a particular parameter. For example, a program where the square root of a list is taken, can be rejected by a 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 parameters of other functions, functions are `values' in some sense. Therefore, functions have a type, too. 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, the Clean language will be used. Compared to the other languages mentioned above, the implementation of Clean provides easy access to functions manipulating the user interfaces. Also, the type system is enriched, making it possible for implementations to improve the efficiency of program execution.

1.2 The Clean compiler

1.2.1 The `Start' expression

In a functional programming language like Clean you can define functions. The functions are meant to be used in an expression, of which the value must be computed.

The Clean compiler is a program which 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. The expression to be evaluated has a special name, it should be named `Start`. For example:

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

When this `Start` expression is evaluated, the result '`11`' of the evaluation 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, called *definition 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

Start = 5+2*3
```

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

```
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 parameter of a function may be 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 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
```

Large amounts 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. This list is the result of the function `reverse`.

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 a parameter 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 afterwards, as is the case with the predefined functions in the standard environment, in the `Start` expression. Definitions of functions are 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:

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

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

Start = over 10 3
```

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

1.3 Standard functions

1.3.1 Names of functions and operators

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 the Clean standard environment, a large number of standard functions is defined. 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`.

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 might be defined.

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

```
default definition case class code export from if implementation import in
infix infixl infixr instance 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 Numeric functions

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 program

```
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 parameters 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 Boolean functions

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). 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 also), 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 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 Functions on functions

In the functions discussed so far, the parameters were numbers, Booleans or lists. However, the parameter of a function can be a function itself too! An example of that is the function `map`, which takes two parameters: a function and a list. The function `map` applies the parameter 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 Function definitions

1.4.1 Definition by combination

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

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

Functions can also have more than one parameter:

```
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 which lack parameters 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 parameters (if there are any)
- the symbol `=`
- an expression, in which the parameters may be used, together with functions (either functions imported form another module or defined elsewhere in the module).

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 parameters the result is calculated differently than for positive parameters. 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, that 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 was defined to be `True` in the `StdBool` module.

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 parameters;
- 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

The parameters of a function in a function definition, like `x` and `y` at the left hand side of the =-symbol in

```
f x y = x * y
```

are called *formal parameters* of the function. For using it, the function is supplied with *actual parameters*. For example, in the call `f 17 (1+g 6)`, the actual parameter corresponding to `x` is `17`, and the actual parameter corresponding to `y` is `1+g 6`. Thus, actual parameters are *expressions*. Until now, we used only variable names as formal parameters. In most programming languages, formal parameters may only be variables. But in Clean, there are other possibilities: a formal parameter may also be a *pattern*.

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

```
f [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 parameters. For example, the `sqrt` function is not defined for negative numbers, and the `/` operator is not defined for `0` as its second parameter. These functions are called *partial functions*.

You can define functions with different patterns as formal parameter:

```
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 parameter `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.

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 definition in which this is the case is called a *recursive definition* (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. Recursive functions are generally used with two restrictions:

- for a base case there is a non-recursive definition;
- the actual parameter of the recursive call is *closer to the base case* (e.g., numerically smaller, or a shorter list) than the formal parameter 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 parameter in the recursive call (`n-1`) is, as required, smaller than `n`.

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 this case, the recursive call must have a shorter list as parameter, and there should be a non-recursive definition for the empty list. In the previous subsection we promised to define the function `sum` for lists of arbitrary length. A recursive function of `sum` is:

```
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
```

1.4.5 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 parameter).

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. 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.

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

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 `==`. 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 functions 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 that operate on numbers may not be applied to Boolean values, neither to lists. Functions that 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.

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. Other languages only check the types at during the execution of the generated application when function is are actually applied, not during compilation. In those languages you can never be sure that at run time a type error will pop up.

On the other hand, 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.

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 parameters 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 ($\rightarrow$) 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. It does so when type checking a program. However, it is also allowed to annotate a function definition with the type of the function. A typed function definition:

```
sum :: [Int] -> Int
sum []  = 0
sum [x:xs]  = x + sum xs
```

Although such a *type declaration* is strictly speaking superfluous, it has two advantages:

- the compiler checks whether the function indeed has the intended type;
- 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 be adjacent to the function definition.

1.5.3 Polymorphism

For some functions on lists the 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 parameter, 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 parameter 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 parameter 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 of more parameters

Functions with more parameters have a type, too. All the types of the parameters 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 a parameters an 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 parameters 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
```

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` parameters 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, `incre-ment` 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 is indicated by the `!`-symbol.
- *Uniqueness* attributes indicate whether parameters 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 parameter.

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

```
isEven          :: !Int -> Bool             // True if parameter is even
spaces          :: !Int -> .[Char]          // Make list of n spaces
(++) infixr 0   :: ![.a] u:[.a] -> u:[.a]   // Concatenate two lists
```

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

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. Consider the following function definition:

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

From the first alternative it is clear the type 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`. 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
```

The first argument, `x`, should be and `Int`, since it is compared to the integer `0` in the pattern of the first function alternative. The argument `y` is compared to `x` in the guard. The ==-operator has type x x -> Bool, hence the type of `y` should also be `Int`. This tells us that the result of the function has type `int`. The last body enforces that argument `z` should have also type `Int`. 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, no of these types is know. It is sufficient that the type of these arguments is member of the type class `a`. 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`. 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. Fortunately, the Clean system itself is able to determine the type of functions. So, you can write programs without specifying the type of your functions. The rules given above enables you to understand the type error that the compiler might spot. 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'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 pattern matching 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 different 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 Exercises