Part I
Chapter 2
Numbers and Functions

2.1 Operators

2.1.1 Operators as functions and vice versa

An operator is a function of two parameters that is written between those parameters ('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 something 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 parameters ('`infix`'), associates to the right (hence the '`r`' in `infixr`), and has priority `3`.

Sometimes it is more clear to write an operator before its parameters, 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 3.6) and for function application and cannot be taken by any ordinary operator. The priorities of the operators in the standard modules are as below:

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). 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 parameter 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 7 parentheses are required to override the high priority of function calls: square (3+4). Only selection from an array or record has an 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.

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 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 `!^!` 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 insists, 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 parameters, for instance `plus 3 5`.

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

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

Calling a function with less parameters 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 parameter 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 parameter 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 (annonymous) 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 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 behaviour, 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 errors 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 systems gives the error message

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

2.3 Functions as parameters

2.3.1 Functions on lists

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 a *parameter* of other functions.

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

Functions which take a function as a parameter 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 an other 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:

$\underline{\texttt{twice inc 0}}$
$\rightarrow$ `inc (`$\underline{\texttt{inc 0}}$`)`
$\rightarrow$ `inc (`$\underline{\texttt{0+1}}$`)`
$\rightarrow$ $\underline{\texttt{inc 1}}$
$\rightarrow$ $\underline{\texttt{1+1}}$
$\rightarrow$ `2`

$\underline{\texttt{twice twice inc}}$ `0` // the formal argument `f` is bound to `twice`, and `x` is bound to `inc`.
$\rightarrow$ `twice (twice inc) 0`
$\rightarrow$ `(twice inc) ((twice inc) 0)`
$\rightarrow_*$ `(twice inc) 2` // as in the previous example
$\rightarrow$ `inc (inc 2)`
$\rightarrow_*$ `inc 3`
$\rightarrow_*$ `4`

The parentheses in the type declaration of higher order functions can be necessary to indicate which arguments belong to the function supplied as argument 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.

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:

```
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 parameter 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 called again. In the recursive call, the parameter is shorter (the list xs is shorter than the list [x:xs]); therefore finally the non-recursive part of the function will be used.

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 a parameter to filter. Examples of the use of filter are:

```
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'.

You can design higher order functions by tracking down the similarities in function definitions. 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 &&).

By passing these two variables as parameters, a generally useful higher order function is born:

```
foldr op e []       = e
foldr op e [x:xs]   = op x (foldr op e xs)
```

which only differs from the three special cases that the operator op is written function-like, in front of its two parameters. Given this function, the other three functions can be defined by partially parameterizing the general function:

```
sum     = foldr (+) 0
product = foldr (*) 1
```

```
and      = foldr (&&) True
```

The function `foldr` can be used in many other cases; that is why it is predefined (for efficiency reasons even as a macro!) in the standard environment.

The name of `foldr` can be explained as follows. The value of

```
foldr (+) e [w,x,y,z]
```

equals the value of the expression

```
(w + (x + (y + (z + e))))
```

The function `foldr` 'folds' the list into one value, by inserting between all elements the given operator, starting on the right side with the given initial value. We will treat the use of higher order functions on lists in more detail in chapter I.6.

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 parameters: 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 parameter of the recursive call of `until` is not 'smaller' than the formal parameter. 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 parameter of `>`); the function `until` will keep on counting indefinitely, and it will never return a result.

If the computer does not answer because it is in such an infinite recursion, the running program has to be interrupted by the user.

2.3.3 Function composition

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 which calculates when given the parameter *x* the value of x^2+3x+1'. The lambda notation is often used when passing functions as a parameter to other functions, as in:

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

Lambda notaion 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 that does not deserve a name.

2.3.4 The lambda notation

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`, 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 with the desired two arguments. For this reason it is not allowed to write `(o) g f x = g (f x)`. Without function composition a local function must be used.

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

Not all functions can be composed. 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 also shown in the type of `o`.

Since an infix operator should have exactly two arguments, we have to define the function composition operator `o` using a local function definition. The more intuative 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.

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 a parameter 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` defined in section can be defined:

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

In the standard module `StdFunc` the function composition operator is 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 list have disappeared from sight. Isn't it 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 is divisible by *n* if the remainder of the division by *n* equals zero.
- 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"
```

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.

When you find even this definition of `day` to longwinded you can use the daynumber as list selector:

```
day :: Day Month Year -> String
day d m y
 = ["Sun","Mon","Tues","Wednes","Thurs","Fri","Satur"]!daynumber d m y +++ "day"
```

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 year `0`. The function `daynumber` will be easier by this extrapolation: the `1`st of January of the year `0` would be on a Sunday.

```
daynumber :: Day Month Year -> Int
daynumber d m y =   ( (y-1)*365
                    + (y-1)/4
                    - (y-1)/100
                    + (y-1)/400
                    + sum (take (m-1) (months y))
                    + 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 predefinedconditional function `if` to eliminate the local definition `feb` in months. The definition beomes:

```
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 c t e | c = t
             = e
```

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 a parameter 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`.

```
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 $1\frac{2}{3}$. 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)
```

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` is in both definition of `diff` 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
```

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
y_1	=	$0.5*(y_0+3/y_0)$	=	2
y_2	=	$0.5*(y_1+3/y_1)$	=	1.75
y_3	=	$0.5*(y_2+3/y_2)$	=	1.732142857
y_4	=	$0.5*(y_3+3/y_3)$	=	1.732050810
y_5	=	$0.5*(y_4+3/y_4)$	=	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 :: a a -> Bool | - a
(~=~) 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`. So to this functions it looks like `x` is a constant.

2.5 Exercises

Exercise 2.1 Combining operators

Define the function `odd` using the operators `even`, `+` and `o`.