Part I
Chapter 3
Data Structures

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

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

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

3.1 Lists

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

3.1.1 Structure of a list

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

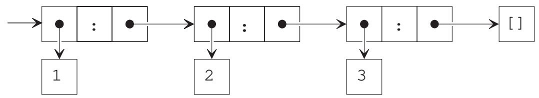

Figure 3.1 Pictorial representation of the list `[1,2,3]`.

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

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

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

There are several ways to construct a list: enumeration, construction using `:` and numeric intervals.

Enumeration

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Construction using :

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

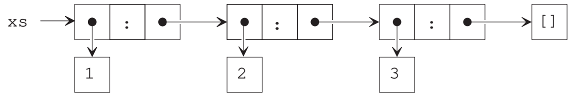

Figure 3.2 Pictorial representation of the list defined as `xs = [1,2,3]`.

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

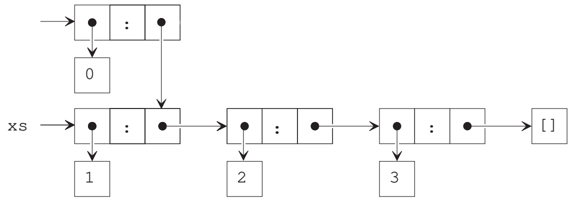

Figure 3.3 Pictorial representation of the list `[0,xs] where xs = [1,2,3]`.

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

Enumerable intervals

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

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

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


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

When the upper bound of a dot-dot expression is not known, it can be omitted. The list generated will be extended as far as necessary. See also section 3.2. Some examples are:

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

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

3.1.2 Functions on lists

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

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

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

Comparing and ordering lists

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

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

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

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

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

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

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

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

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

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

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

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

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

Joining lists

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

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

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

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

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

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

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

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

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

Selecting parts of lists

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

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



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

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

```
hd of []
```

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

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

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

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

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

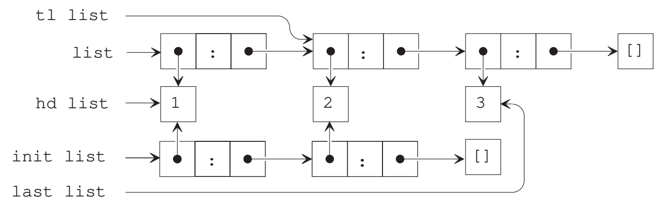

Figure 3.4 Pictorial representation of the list `list = [1,2,3]`, and the result of applying the functions `hd`, `tl`, `init` and `last` to this list.

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

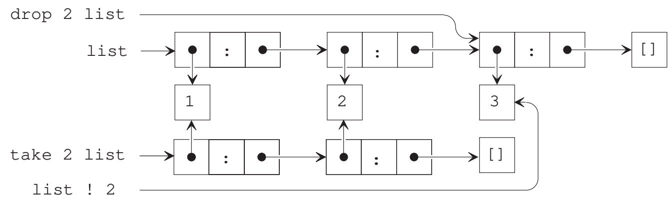

Figure 3.5 Pictorial representation of the list `list = [1,2,3]`, and the result of applying the functions `drop 2`, `take 2` and `! 2` to this list.

These functions are defined as follows:

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

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

Whenever a list is too short as much elements as possible are taken or left out respectively. This follows from the first line in the definitions: if you give the function an empty list, the result is always an empty list, whatever the number is. If these lines were left out of the definitions, then `take` and `drop` would be undefined for lists that are too short. Also with respect to the number of elements to take or drop these functions are foolproof[1]: all negative numbers are treated as 0.

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

```
(!) infixl 9 :: [a] Int -> a
(!) list n
    | n == 0    = hd list
    | otherwise = tl list ! dec n
```

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

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

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

Reversing lists

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

```
reverse []      = []
reverse [x:xs]  = reverse xs ++ [x]
```

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

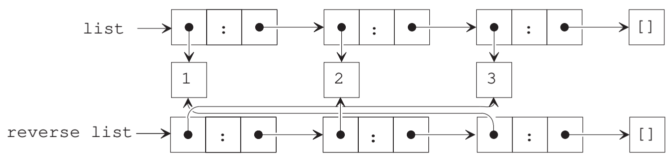

Figure 3.6 Pictorial representation of the list `list = [1,2,3]`, and the effect of applying the function `reverse` to this list.

Properties of lists

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

[1]In this respect the functions shown here differ from the function provided in the StdEnv of Clean 1.2.


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

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

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

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

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

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

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

3.1.3 Higher order functions on lists

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

map and filter

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

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

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

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

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

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

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

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

takewhile and dropwhile

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

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

Compare this definition to that of `filter`.

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

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

3.1.4 Sorting lists

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

A list manipulation that cannot be written in this manner is the sorting (putting the elements in ascending order). The elements should be completely shuffled in order to accomplish sorting. In general this cannot be done by traversing the list once in a recursive function.

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

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

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

Insertion sort

Suppose a sorted list is given. Then a new element can be inserted in the right place using the following function[2]:

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

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

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

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

The function `Insert` can be used to sort a list that is not already sorted. Suppose `[a,b,c,d]` has to be sorted. You can sort this list by takeing an empty list (which is

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

sorted) and insert the elements of the list to be sorted one by one. The effect of applying the sorting function `isort` to our ezample list tshould be:

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

Therefore one possible sorting algorithm is:

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

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

Merge sort

Another sorting algorithm makes use of the possibility to merge two sorted lists into one. This is what the function `merge` does[3]:

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

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

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

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

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

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

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

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

3.1.5 List comprehensions

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

[3]This function definition is included in `StdList`.

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

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

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

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

Thus, the example above has the same value as

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

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

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

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

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

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

evaluates to the list

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

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

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

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

evaluates to the list

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

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

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

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

which evaluates to

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


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

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

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

which is less intuitive.

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

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

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

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

Quick sort

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

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

3.2 Infinite lists

3.2.1 Enumerating all numbers

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

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

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

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

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

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

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

The result will be the list

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

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

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

the result of which is the shorter list

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

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

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

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

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

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

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

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

However, be careful not to write:

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

This is equivalent to

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

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

3.2.2 Lazy evaluation

The evaluation method (the way expressions are calculated) of Clean is called *lazy evaluation*. With lazy evaluation an expression (or part of it) is only computed when it is certain that its value is *really* needed for the result. The opposite of lazy evaluation is *strict evaluation*, also called *eager evaluation*. With eager evaluation, before computing the a function's result, first all actual parameters of the function are evaluated.

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

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

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

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

The lazy behaviour of the operator `==` is caused by its definition. The recursive line from the definition in subsection 3.2.1 reads:

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

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

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

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

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

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

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

3.2.3 Functions generating infinite lists

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

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

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

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

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

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

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

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

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

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

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

In the next subsection an examples is discussed in which `iterate` is used to solve a practical problem: generating the list of all prime numbers.

3.2.4 The list of all prime numbers

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

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

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

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

```
[2,3,4,5,6,7,8,9,10,11,…
```

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

```
[3,5,7,9,11,13,15,17,19,21,…
```

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

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

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

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


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

```
iterate crossout [2..]
where
    crossout [x:xs] = filter (not o multiple x) xs
    multiple x y    = divisible y x
```

The number `y` is a multiple of `x` if `y` is divisible by `x`. The function `divisible` was defined in section 2.4.1 as: `divisible t n = t rem n == 0`. As the starting value is a infinite list. the result of this is an *infinite list of infinite lists.* That super list looks like this:

```
[[2,3,4,5,6,7,8,9,10,11,12,13,14,…
,[3,5,7,9,11,13,15,17,19,21,23,25,27,…
,[5,7,11,13,17,19,23,25,29,31,35,37,41,…
,[7,11,13,17,19,23,29,31,37,41,43,47,49,…
,[11,13,17,19,23,29,31,37,41,43,47,51,53,…
,…
```

You can never see this thing as a whole; if you try to evaluate it, you will only see the beginning of the first list. But you need the complete list to be visible: the desired prime numbers are the first elements of the lists. So the prime numbers can be determined by taking the `head` of each list:

```
primenums :: [Int]
primenums = map head (iterate crossout [2..])
where
    crossout [x:xs] = filter (not o (multiple x)) xs
```

Thanks to lazy evaluation only that part of each list is calculated that is needed for the desired part of the answer. If you want to know the next prime, more elements of every list are calculated as far as necessary.

Often it is hard (as in this example) to imagine what is computed at what moment. But there is no need: while programming you can just pretend infinite lists really exist; the evaluation order is automatically optimized by lazy evaluation.

This algorithm to compute prime numbers is called the sieve of Eratosthenes. Eratosthenes was a greek mathematician born in Cyrene who lived 276-196 BC. His algorithm can be expressed slightly more elegant using list comprehensions:

```
primes :: [Int]
primes = sieve [2..]

sieve :: [Int] -> [Int]
sieve [prime:rest] = [prime: sieve [i \\ i <- rest | i mod prime <> 0]]
```

3.3 Tuples

All elements in a list have to be of the same type, e.g. it is not possible to store both an integer and a string in one and the same list. Sometimes one needs to group information of different types together. A tuple can be used for this.

A *tuple* consists of a fixed number of values that are grouped together (see figure 3.7). The values may be of different types (although that is not obligatory).

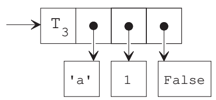

Figure 3.7 Pictorial representation of the tuple `('a',1,False)` of type `(Char,Int,Bool)`.

Tuples are denoted by round parentheses around the elements. Examples of tuples are:

`(1,'a')` a tuple with as elements the integer `1` and the character `'a'`;

`("foo",True,2)`	a tuple with three elements: the string `foo`, the Boolean `True` and the number `2`;
`([1,2],sqrt)`	a tuple with two elements: the list of integers `[1,2]` and the function from real to real `sqrt`;
`(1,(2,3))`	a tuple with two elements: the number `1` and a tuple containing the numbers `2` and `3`.

The tuple of each combination types is a distinct type. The order in which the components appear is important, too. The type of tuples is written by enumerating the types of the elements between parentheses. The four expressions above can be types as follows:

```
(1,'a')           :: (Int,Char)
("foo",True,2)    :: (String,Bool,Int)
([1,2],sqrt)      :: ([Int],Real->Real)
(1,(2,3))         :: (Int,(Int,Int))
```

A tuple with two elements is called a 2-tuple or a *pair*. Tuples with three elements are called 3-tuples etc. There are no 1-tuples: the expression `(7)` is just an integer; for it is allowed to put parentheses around every expression.

The standard library provides some functions that operate on tuples. These are good examples of how to define functions on tuples: by pattern matching.

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y
```

These functions are all polymorphic, but of course it is possible to write your own functions that only work for a specific type of tuple:

```
f :: (Int,Char) -> [Char]
f (n,c) = intChars n ++ [c]
```

Tuples come in handy for functions with multiple results. Functions can have several arguments. However, functions have only a single result. Functions with more than one result are only possible by `wrapping' these results up in some structure, e.g. a tuple. Then the tuple as a whole is the only result.

An example of a function with two results is `splitAt` which is defined in the standard environment. This function delivers the results of `take` and `drop` at the same time. Therefore the function could be defined as follows:

```
splitAt :: Int [a] -> ([a],[a])
splitAt n xs = (take n xs,drop n xs)
```

However, the work of both functions can be done simultaneously. That is why in the standard library `splitAt` is defined as:

```
splitAt :: Int [a] -> ([a],[a])
splitAt 0 xs        = ([] ,xs)
splitAt n []        = ([] ,[])
splitAt n [x:xs]    = ([x:ys],zs)
where
    (ys,zs) = splitAt (n-1) xs
```

The result of the recursive call of `splitAt` can be inspected by writing down a `right-hand side pattern match', which is called a *selector*:

```
splitAt n [x:xs]    = ([x:ys],zs)
where
    (ys,zs) = splitAt (n-1) xs
```

The tuple elements thus obtained can be used in other expressions, in this case to define the result of the function `splitAt`.

The call `splitAt 2 ['clean']` gives the `2`-tuple `(['cl'],['ean'])`. In the definition (at the recursive call) you can see how you can use such a result tuple: by exposing it to a pattern match (here `(ys,zs)`).

Another example is a function which calculates the average of a list, say a list of reals. In this case one can use the predefined functions `sum` and `length`: `average = sum / toReal length`. Again this has the disadvantage that one walks through the list twice. It is much more efficient to use one function `sumlength` which just walks through the list once to calculate both the sum of the elements (of type `Real`) as well as the total number of elements in the list (of type `Int`) at the same time. The function `sumlength` therefore returns one tuple with both results stored in it:

```
average :: [Real] -> Real
average list = mysum / toReal mylength
where
    (mysum,mylength) = sumlength list 0.0 0

    sumlength :: [Real] Real Int -> (Real,Int)
    sumlength [x:xs] sum length = sumlength xs (sum+x) (length+1)
    sumlength []     sum length = (sum,length)
```

Using type classes this function can be made slightly more general:

```
average :: [t] -> t | +, zero, one t
average list = mysum / mylength
where
    (mysum,mylength) = sumlength list zero zero

    sumlength :: [t] t t -> (t,t)   | +, one t
    sumlength [x:xs] sum length = sumlength xs (sum+x) (length+one)
    sumlength []     sum length = (sum,length)
```

3.3.1 Tuples and lists

Tuples can of course appear as elements of a list. A list of two-tuples can be used e.g. for searching (dictionaries, telephone directories etc.). The search function can be easily written using patterns; for the list a `non-empty list with as a first element a 2-tuple' is used.

```
search :: [(a,b)] a -> b | == a
search [(x,y):ts] s
    | x == s    = y
    | otherwise = search ts s
```

The function is polymorphic, so that it works on lists of 2-tuples of arbitrary type. However, the elements should be comparable, which is why the functions `search` is overloaded since `==` is overloaded as well. The element to be searched is intentionally defined as the second parameter, so that the function `search` can easily be partially parameterized with a specific search list, for example:

```
telephoneNr = search telephoneDirectory
translation = search dictionary
```

where `telephoneDirectory` and `dictionary` can be separately defined as constants.

Another function in which 2-tuples play a role is the `zip` function. This function is defined in the standard environment. It has two lists as parameters that are chained together element-wise in the result. For example: `zip [1,2,3] ['abc']` results in the list `[(1,'a'),(2,'b'),(3,'c')]`. If the two lists are not of equal length, the shortest determines the size of the result. The definition is rather straightforward:

```
zip :: [a] [b] -> [(a,b)]
zip []     ys     = []
zip xs     []     = []
zip [x:xs] [y:ys] = [(x,y) : zip xs ys]
```

The function is polymorphic and can thus be used on lists with elements of arbitrary type. The name *zip* reflects the fact that the lists are so to speak `zipped'. The functions `zip` can more compactly be defined using a list comprehension:

```
zip :: [a] [b] -> [(a,b)]
zip as bs = [(a,b) \\ a <- as & b <- bs]
```

If two values of the same type are to be grouped, you can use a list. Sometimes a tuple is more appropriate. A point in the plane, for example, is described by two `Real` numbers.

Such a point can be represented by a list or a 2-tuple. In both cases it possible to define functions working on points, e.g. `distance to the origin'. The function `distanceL` is the list version, `distanceT` the tuple version:

```
distanceL :: [Real] -> Real
distanceL [x,y] = sqrt (x*x+y*y)

distanceT :: (Real,Real) -> Real
distanceT (x,y) = sqrt (x*x+y*y)
```

As long as the function is called correctly, there is no difference. But it could happen that due to a typing error or a logical error the function is called with three coordinates. In the case of `distanceT` this mistake is detected during the analysis of the program: a tuple with three numbers is of another type than a tuple with two numbers. However, using `distanceL` the program is well-typed. Only when the function is really used, it becomes evident that `distanceL` is undefined for a list of three elements. Here the use of tuples instead of lists helps to detect errors.

3.4 Records

Often one would like to group information of possibly different type on a more structural way simply because the information belongs together. Information in a person database may consist for example of a string (name), a Boolean (male), three integers (date of birth) and a Boolean again (Clean user). If one wants to use such a kind of record, one first has to declare its type in a *type definition*, e.g.:

```
:: Person   =  {   name        :: String
               ,   birthdate   :: (Int,Int,Int)
               ,   cleanuser   :: Bool
               }
```

Type definitions in Clean always start with a `::` at the beginning of a line. With this particular type definition a new type is declared, called a *record type*. A record is a kind of tuple. The record elements can be of different type, just like in tuples. However, in a record type, a name (the *field name*) is used to refer to a record element (see also figure 3.8). This fieldname must be used to identify the corresponding record element.

Figure 3.8 Pictorial representation of a record `Person`.

Once the type is defined, a record of that type can be created, e.g. in the following way:

```
SomePerson :: Person
SomePerson  =  {   name        = "Rinus"
               ,   birthdate   = (10,26,1952)
               ,   cleanuser   = True
               }
```

Each of the record elements (identified by `fieldname =`) must get a value of the type as indicated in the record type declaration. The order in which the fields are specified is irrelevant, but all fields of the record have to get a value.

An important difference between a tuple and a record is that a tuple field always has to be selected by its position, as in:

```
fst :: (a,b) -> a
fst (x,y) = x
```

while a record field is selected by field name. For instance, one can define:


```
:: Two a b  =  {   first   :: a
               ,   second  :: b
               }
```

This is a polymorphic record named `Two` with two record elements. One element is named `first` and is of type `a`, the other is named `second` and is of type `b`. A function which selects the first element can be defined as:

```
fstR :: (Two a b) -> a
fstR {first} = first
```

The example illustrates how the pattern matching mechanism can be used to select one or more record elements. The nice thing about this feature is that one only needs to name the fields one is interested in.

```
IsCleanUser :: Person -> String
IsCleanUser {cleanuser = True}  = "Yes"
IsCleanUser _                   = "No"
```

There is a special selection operator, '`.`', to select an element from a record. It expects a expression yielding a record and a field name to select an expression of that record. For instance:

```
GetPersonName :: Person -> String
GetPersonName person = person.name
```

Finally, there is a special language construct which enables you to create a new record given another existing record of the same type. Consider:

```
ChangePersonName :: Person String -> Person
ChangePersonName person newname = {person & name = newname}
```

The new record is created by making a copy of the old record `person`. The constents of the fields of the new record will be exactly the same as the constents of the fields of the old record, with exception of the field `name` which will contain the new name `newname`. The operator `&` is called the *functional update operator*. Do not confuse it with a destructive update (assignment) as is present in imperative languages (like in C, C++, Pascal). Nothing is changed, a complete new record is made which will be identical to the old one with exception of the specified new field values. The old record itself remains unchanged.

The Clean system determines the type of a record from the filed names used. When there are several records with the used field names determining the type fails. The user should explicitly specify the type of the record inside the record. It is not sufficient to that the type of the record can be deduced from the type of the function. It is always allowed to indicate the type of a record explicitly.

```
AnotherPerson :: Person
AnotherPerson = {   Person
                |   name        = "Pieter"
                ,   birthdate   = (7,3,1957)
                ,   cleanuser   = True
                }
```

The records in Clean make it possible to define functions which are less sensible for changes. For instance, assume that one has defined:

```
:: Point    =   {   x :: Real
                ,   y :: Real
                }
MovePoint :: Point (Real,Real) -> Point
MovePoint p (dx,dy) = {p & x = p.x + dx, y = p.y + dy}
```

Now, lets assume that in a later state of the development of the program one would like to add more information to the record `Point`, say

```
:: Point    =   {   x :: Real
                ,   y :: Real
                ,   c :: Color
                }
```

where `Color` is some other type. This change in the definition of the record `Point` has no consequences for the definition of `MovePoint`. If `Point` would be a tuple, one would have to change the definition of `MovePoint` as well, because `Point` would change from a 2-tuple to a 3-tuple which has consequences for the pattern match as well as for the construction of a new point.

So, for clarity and ease of programming we strongly recommend the use of records instead of tuples. Only use tuples for functions which return multiple results.

3.4.1 Rational numbers

An application in which records can be used is an implementation of the *Rational numbers*. The rational numbers form the mathematical set *Q*, numbers that can be written as a *ratio*. It is not possible to use `Real` numbers for calculations with ratios: the calculations must be *exact*, such that the outcome of $\frac{1}{2}+\frac{1}{3}$ is the fraction $\frac{5}{6}$ and not the `Real 0.833333`.

Fractions can be represented by a numerator and a denominator, which are both integer numbers. So the following type definition is obvious:

```
:: Q =  { num :: Int
        , den :: Int
        }
```

Next a number of frequently used fractions get a special name:

```
QZero       = {num = 0,den = 1}
QOne        = {num = 1,den = 1}
QTwo        = {num = 2,den = 1}
QHalf       = {num = 1,den = 2}
QThird      = {num = 1,den = 3}
QQuarter    = {num = 1,den = 4}
```

We want to write some functions that perform the most important arithmetical operations on fractions: multiplication, division, addition and subtraction. Instead of introducing new names for these functions we use the overloading meganism (introduced in section 1.5.5 and explained in more detail in section 4.1) in order to use the obvious operator symbols: `*`, `/`, `+`, `-`.

The problem is that one value can be represented by different fractions. For example, a half can be represented by `{num=1,den=2}`, but also by `{num=2,den=4}` and `{num=17,den=34}`. Therefore the outcome of two times a quarter might `differ' from `half'. To solve this problem a function `simplify` is needed that can simplify a fraction. By applying this function after every operation on fractions, fractions will always be represented in the same way. The result of two times a quarter can then be safely compared to a half: the result is `True`.

The function `simplify` divides the numerator and the denominator by their *greatest common divisor*. The greatest common divisor (`gcd`) of two numbers is the greatest number by which both numbers are divisible. For negative numbers we want a negative nominator. When the denominator is zero the fraction is not defined. The definition of `simplify` reads as follows:

```
simplify :: Q -> Q
simplify {num=n,den=d}
    | d == 0    = abort "denominator of Q is 0!"
    | d < 0     = {num = ~n / g, den = ~d / g}
    | otherwise = {num =  n / g, den =  d / g}
where
    g = gcd n d
```

A simple definition of `gcd x y` determines the greatest divisor of `x` that also divides `y` using `divisors` and `divisible` from subsection 2.4.1.

```
gcd :: Int Int -> Int
gcd x y = last (filter (divisible (abs y)) (divisors (abs x)))
```

(In the standard library a more efficient version of `gcd` is defined:


```
gcd :: Int Int -> Int
gcd x y = gcd' (abs x) (abs y)
where
    gcd' x 0 = x
    gcd' x y = gcd' y (x mod y)
```

This algorithm is based on the fact that if `x` and `y` are divisible by `d` then so is `x mod y` (`=x-(x/y)*y)`).

Using `simplify` we are now in the position to define the mathematical operations. Due to the number of places where a record of type `Q` must be created and simplified it is convenient to introduce an additional function mk`Q`.

```
mkQ :: x x -> Q | toInt x
mkQ n d = simplify {num = toInt n, den = toInt d}
```

To multiply two fractions, the numerators and denominators must be multiplied ($\frac{2}{3}*\frac{5}{4}=\frac{10}{12}$). Then the result can be simplified (to $\frac{5}{6}$):

```
instance * Q
where (*) q1 q2 = mkQ (q1.num*q2.num) (q1.den*q2.den)
```

Dividing by a number is the same as multiplying by the inverse:

```
instance / Q
where (/) q1 q2 = mkQ (q1.num*q2.den) (q1.den*q2.num)
```

Before you can add two fractions, their denominators must be made the same first. ($\frac{1}{4}+\frac{3}{10}=\frac{10}{40}+\frac{12}{40}=\frac{22}{40}$). The product of the denominator can serve as the common denominator. Then the numerators must be multiplied by the denominator of the other fraction, after which they can be added. Finally the result must be simplified (to $\frac{11}{20}$).

```
instance + Q
where (+) q1 q2 = mkQ (q1.num * q2.den + q1.den * q2.num) (q1.den * q2.den)
instance - Q
where (-) q1 q2 = mkQ (q1.num * q2.den - q1.den * q2.num) (q1.den * q2.den)
```

The result of computations with fractions is displayed as a record. If this is not nice enough, you can define a function `toString`:

```
instance toString Q
where toString q
        | sq.den==1 = toString sq.num
        | otherwise = toString sq.num +++ "/" +++ toString sq.den
      where
        sq = simplify q
```

3.5 Arrays

An *array* is a predefined data structure which is used mainly for reasons of efficiency. With a list an array has in common that *all* its elements have to be of the *same* type. With a tuple/record-like data structure an array has in common that it contains a fixed number of elements. The elements of an array are numbered. This number, called the *index*, is used to identify an array element, like field names are used to identify record elements. An array index is an integer number between 0 and the number of array elements - 1.

Arrays are notated using curly braces. For instance,

```
MyArray :: {Int}
MyArray = {1,3,5,7,9}
```

is an array of integers (see figure 3.9). It's type is indicated by `{Int}`, to be read as 'array of `Int`'.

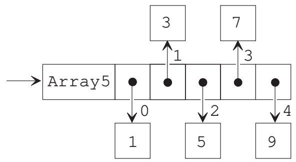

Figure 3.9 Pictorial representation of an array with 5 elements `{1,3,5,7,9}`.

Compare this with a list of integers:

```
MyList :: [Int]
MyList = [1,3,5,7,9]
```

One can use the operator `!` to select the i^{th} element from a list (see subsection 3.1.2): For instance

```
MyList!2
```

will yield the value `5`. To select the i^{th} element from array `a` one writes `a.[i]`. So,

```
MyArray.[2]
```

will also yield the value `5`. Besides the small difference in notation there is big difference in the efficiency between an array selection and a list selection. To select the i^{th} element from a list, one has to recursively walk through the spine of the list until the i^{th} list element is found (see the definition of `!` in subsection 3.1.2). This takes i steps. The i^{th} element of an array can be found directly in *one* step because all the references to the elements are stored in the array box itself (see figure 3.9). Selection can therefore be done very efficiently regardless which element is selected in *constant time.*

The big disadvantage of selection is that it is possible to use an index out of the index range (i.e. index < 0 or index $\geq n$, where n is the number of list/array elements). Such an index error generally cannot be detected at compile-time, such that this will give rise to a run-time error. So, selection both on arrays as on lists is a very dangerous operation because it is a partial function and one easily makes mistakes in the calculation of an index. Selection is the main operation on arrays. The construction of lists is such that selection can generally be avoided. Instead one can without danger recursively traverse the spine of a list until the desired element is found. Hitting on the empty list a special action can be taken. Lists can furthermore easily be extended while an array is fixed sized. Lists are therefore more flexible and less error prone. Unless ultimate efficiency is demanded, the use of lists above arrays is recommended.

But, arrays can be very useful if time and space consumption is becoming very critical, e.g. when one uses a huge and fixed number of elements which are frequently selected and updated in a more or less random order.

3.5.1 Array comprehensions

To increase readability, Clean offers array comprehensions in the same spirit as list comprehension's. For instance, if `ArrayA` is an array and `ListA` a list, then

```
NewArray = {elem \\ elem <- ListA}
```

will create an new array with the same (amount of) elements as in `ListA`. Conversion the other way around is easy as well:

```
NewList = [elem \\ elem <-: ArrayA]
```

Notice that the `<-` symbol is used to draw elements from a list while the `<-:` symbol is used to draw elements from an array.

Also a map-like function on an array can be defined in a straightforward manner:

```
MapArray f a = {f e \\ e <-: a}
```


3.5.2 Lazy, strict and unboxed arrays

To obtain optimal efficiency in time and space, several kinds of arrays can be defined in Clean. By default an array is a *lazy array* (say, of type `{Int}`), i.e. an array consists of a contiguous block of memory containing references to the array elements (see figure 3.9). The same representation is chosen if a *strict array* is used (prefix the element type with a `!`, e.g. `{!Int}`). Strict arrays have to property that its elements will always be evaluated whenever the array is used. For elements of basic type only (`Int`, `Real`, `Char`, `Bool`} an *unboxed array* can be defined (prefix the element type with a `#`, e.g. `{#Int}`). So, by explicitly specifying the type of the array upon creation one can indicate which representation one wants: the default one (lazy), or the strict or the unboxed version of the array.

Unboxed arrays are more efficient than lazy or strict arrays because the array elements themselves are stored in the array box. No references to other boxes have to be regarded. For instance, the following array

```
MyUnboxedArray :: {#Int}
MyUnboxedArray = {1,3,5,7,9}
```

is an unboxed array (due to its type specification) of integers. Compare its representation in figure 3.10 with the default representation given in figure 3.9.

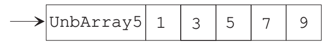

Figure 3.10 Pictorial representation of an unboxed array with 5 elements `{1,3,5,7,9}`.

Lazy, strict and unboxed arrays are regarded by the Clean compiler as objects of *different* types. This means for instance that a function which is expecting an *unboxed* array of `Char` cannot be applied to a *lazy* array of `Char` or the other way around. However, most predefined operations on arrays (like array selection) are overloaded such that they can be used on lazy, strict as well as on unboxed arrays.

A *string* is equivalent with an unboxed array of character `{#Char}`. A type synonym is defined in module `StdString`. For programming convenience, there is special syntax to denote strings. For instance, the string denotation

```
"abc"
```

is equivalent with the unboxed array `{'a','b','c'}`. Compare this with `['abc']` which is equivalent with the list of characters `['a','b','c']`.

3.5.3 Array updates

It is also possible to update an array, using the same notation as for records (see subsection 3.4). In principle a new array is constructed out of existing one. One has to indicate for which index the new array differs from the old one. Assume that `Array5` is an integer array with 5 elements. Then an array with elements `{1,3,5,7,9}` can be created as follows:

```
{Array5 & [0]=1,[1]=3,[2]=5,[3]=7,[4]=9}
```

As with record updating, the order in which the array elements are specified is irrelevant. So, the following definition

```
{Array5 & [1]=3,[0]=1,[3]=7,[4]=9,[2]=5}
```

is also fine.

One can even combine array updates with array comprehension's. So the next two expressions will also yield the array `{1,3,5,7,9}` as result.

```
{Array5 & [i]=2*i+1 \\ i <- [0..4]}
{Array5 & [i]=elem \\ elem <- [1,3..9] & i <- [0..4]}
```

As said before, arrays are mainly important to achieve optimal efficiency. That is why updates of arrays are in Clean only defined on unique arrays, such that the update can always be done *destructively* ! This is semantically sound because the original unique array is known not to be used anymore. Uniqueness is explained in more detail in chapter 4 and 5.

3.5.4 Array patterns

Array elements can be selected in the patterns of a function. This is similar to the selection of the fields of a record. This is illustrated by the following functions.

```
CopyFirst :: Int *{#a} -> {#a} | ArrayElem a
CopyFirst j   a=:{[0]=a0} = {a & [j] = a0}

CopyElem :: Int Int *{#a} -> {#a} | ArrayElem a
CopyElem  i j a=:{[i]=ai} = {a & [j] = ai}

CopyCond :: Int Int *{#a} -> {#a} | ArrayElem, ==, zero a
CopyCond  i j a=:{[i]=ai, [j]=aj} | a.[0]==zero = {a & [j] = ai}
                                                = {a & [i] = aj}
```

It is **not** (yet) allowed to use constants in these patterns. The selection of elements specified in the pattern is done before the right hand side of the rule is constructed. This explains why the given examples are allowed. When the `CopyElem` function is written as

```
CopyElem2 :: Int Int *{#a} -> {#a} | ArrayElem a
CopyElem2  i j a = {a & [j] = a.[i]}
```

it will be rejected by the Clean system. An array can only be updated when it is unique. The reference to the old array, `a.[i]`, in the array update spoils the uniqness properties of the array. Without selection in the pattern this function should be written with a `let!` construct:

```
CopyElem3 :: Int Int *{#a} -> {#a} | ArrayElem a
CopyElem3  i j a = let! ai = a.[i] in {a & [j] = ai}
```

The graphs specified in the strict let part are evaluated before the expression after the keyword `in` is evaluated. This implied that the element `ai` is selected from the array before the array is updated.

3.6 Algebraic datatypes

We have seen several `built-in' ways to structure information: lists, tuples, records and arrays. In some cases these data structures are not appropriate to represent the information. Therefore it has been made possible to define a new, so-called *algebraic datatype* yourself.

An algebraic datatype is a type that defines the way elements can be constructed. In fact, all built-in types are predefined algebraic datatypes. A `list', for instance, is an algebraic type. Lists can be constructed in two ways:

- as the empty list;
- by prepending an element to an existing list using the `[x:xs]` notation.

In the case distinction in definitions of functions that operate on lists these two way construction methods reappear, for example:

```
length :: [t] -> Int
length []       = 0
length [x:xs]   = 1 + length xs
```

By defining the function for both cases, the function is totally defined.

If you want to use a new data structure in Clean, you have to define its type in an algebraic data type definition. For instance:

```
:: Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

is an algebraic data type definition introducing a new type, named `Day`. It moreover introduces seven new constants that have this type `Day` (`Mon`,`Tue`,`Wed`,`Thu`,`Fri`,`Sat`,`Sun`). These constants are called *data constructors*. Once defined in an algebraic type, the data constructors can be used in function definitions. They can appear in expressions to

construct new objects of the specified algebraic type. They can appear in patterns, for instance to discriminate between objects of the same algebraic type.

```
IsWeekend Sat = True
IsWeekend Sun = True
IsWeekend _   = False
```

A data constructor can only belong to *one* algebraic data type definition. As a consequence, the Clean system can directly tell the type of each data constructor. So, `Mon :: Day`, `Tue :: Day`, and so on. And therefore, the type of `IsWeekend` is:

```
IsWeekend :: Day -> Bool
```

The algebraic type `Day` is called an *enumeration type*: the type definition just enumerates all possible values. In chapter 2 we used integers to represent the days of the week. This has both advantages and disadvantages:

- An advantage of the algebraic data type is that well chosen names avoids confusion. When you use integers you have to decide and remember whether the week starts on Sunday or on Monday. Moreover, there is the question whether the first day of the week has number `0` or number `1`.
- An other advantage of the algebraic type is that the type checker is able to verify type correctness. A function that expects or delivers an element of type `Day` will always use one of the listed values. When you use integers, the compiler is only able to verify that an integer is used at each spot a `Day` is expected. It is still possible to use the value `42` where a `Day` is expected. In addition using algebraic can prevent confusion between enumerated types. When we use this definition of `Day` and a similar definition of `Month` it is not possible to interchange the arguments of `daynumber` by accident without making a type error.
- An advantage of using integers to represent days is that the definition of operations like addition, comparison and equality can be reused. In chapter 2 we saw how pleasant this is. For an algebraic type all the needed operations should be defined.

The balance between advantages and disadvantages for the application at hand determines whether it is better to use an algebraic enumeration type or to use integers as encoding (Booleans can be used for types with two values). Unless there are strong reasons to use something else we recommend generally to use an algebraic data type. In the next section we show that it is possible to equip the constructors with arguments and to define recursive types. This is far beyond the possibilities of an encoding of types in integers.

As usual, it is possible to combine algebraic types with orther types in various ways. For example:

```
:: Employee = { name        :: String
              , gender      :: Gender
              , birthdate   :: Date
              , cleanuser   :: Bool
              }
:: Date     = { day         :: Int
              , month       :: Int
              , year        :: Int
              }
:: Gender   = Male | Female
```

These types can be used in functions like:

```
WeekDayOfBirth :: Employee -> Day
WeekDayOfBirth {birthdate=:{day,month,year}}
 = [Sun, Mon, Tue, Wed, Thu, Fri, Sat] ! daynumber day month year
```

Where we use the function `daynumber` as defined in chapter 2. An example of a function generating a value of the type `Employee` is:

```
AnEmployee :: Employee
AnEmployee = { name      = "Pieter"
             , gender    = Male
             , birthdate = {year = 1957, month = 7, day = 3}
```

```
              , cleanuser = True
              }
```

3.6.1 Tree definitions

All data structures can be defined using an algebraic data type definition. In this way one can define data structures with certain properties. For instance, a list is a very flexible data structure. But, it also has a disadvantage. It is a linear structure; as more elements are prepended, the chain (spine) becomes longer (see figure 3.3). Sometimes such a linear structure is not appropriate and a *tree structure* would be better. A (binary) tree can be defined as:

```
:: Tree a   = Node a (Tree a) (Tree a)
            | Leaf
```

You can pronounce this definition as follows. `A tree with elements of type `a` (shortly, a tree of `a`) can be built in two ways: (1) by applying the constant `Node` to three parameters (one of type `a` and two of type tree of `a`), or (2) by using the constant `Leaf`.' `Node` and `Leaf` are two new constants. `Node` is a data constructor of arity three (`Node :: a (Tree a) (Tree a) -> (Tree a)`), `Leaf` is a data constructor of arity zero (`Leaf :: Tree a`). The algebraic type definition also states that the new type `Tree` is polymorphic.

You can construct trees by using the data constructors in an expression (this tree is also drawn in the figure 3.11).

```
Node 4  (Node 2 (Node 1 Leaf Leaf)
                (Node 3 Leaf Leaf)
        )
        (Node 6 (Node 5 Leaf Leaf)
                (Node 7 Leaf Leaf)
        )
```

You don't have to distribute it nicely over the lines; the following is also allowed:

```
Node 4(Node 2(Node 1 Leaf Leaf)(Node 3 Leaf Leaf))
      (Node 6(Node 5 Leaf Leaf)(Node 7 Leaf Leaf))
```

However, the layout of the first expression is clearer.

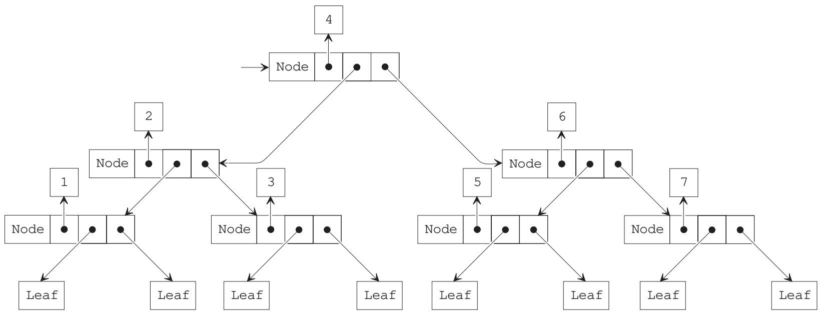

Figure 3.11 Pictorial representation of a tree.

Not every instance of the type tree needs to be as symetrical as the tree show above. This is illustrated by the following example.

```
Node 7  (Node 3 (Node 5 Leaf Leaf)
                Leaf
        )
        Leaf
```

An algebraic data type definition can be seen as the specification of a grammar in which is specified what legal data objects are of a specific type. If you don't construct a data

structure as specified in the algebraic data type definition, a type error is generated at compile time.

Functions on a tree can be defined by making a pattern destinction for every data constructor. The next function, for example, computes the number of `Node` constructions in a tree:

```
sizeT :: Tree a -> Int
sizeT Leaf          = 0
sizeT (Node x p q)  = 1 + sizeT p + sizeT q
```

Compare this function to the function `length` on lists.

There are many more types of trees possible. A few examples:

- Trees in which the information is stored in the leaves (instead of in the nodes as in `Tree`):
```
:: Tree2 a      = Node2 (Tree2 a) (Tree2 a)
                | Leaf2 a
```
- Trees in which information of type `a` is stored in the nodes and information of type `b` in the leaves:
```
:: Tree3 a b    = Node3 a (Tree3 a b) (Tree3 a b)
                | Leaf3 b
```
- Trees that split in three branches at every node instead of two:
```
:: Tree4 a      = Node4 a (Tree4 a) (Tree4 a) (Tree4 a)
                | Leaf4
```
- Trees in which the number of branches in a node is variable:
```
:: Tree5 a      = Node5 a [Tree5 a]
```
  In this tree you don't need a separate constructor for a `leaf', because you can use a node with no outward branches.
- Trees in which every node only has one outward branch:
```
:: Tree6 a      = Node6 a (Tree6 a) | Leaf6
```
  A `tree' of this type is essentially a list: it has a linear structure.
- Trees with different kinds of nodes:
```
:: Tree7 a b    = Node7a Int a (Tree7 a b) (Tree7 a b)
                | Node7b Char (Tree7 a b)
                | Leaf7a b
                | Leaf7b Int
```

3.6.2 Search trees

A good example of a situation in which trees perform better than lists is searching (the presence of) an element in a large collection. You can use a *search tree* for this purpose.

In subsection 3.1.2 a function `isMember` was defined that delivered `True` if an element was present in a list. Whether this function is defined using the standard functions `map` and `or`

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

or directly with recursion

```
isMember e []       = False
isMember e [x:xs]   = x==e || isMember e xs
```

doesn't affect the efficiency that much. In both cases all elements of the list are inspected one by one. As soon as the element is found, the function immediately results in `True` (thanks to lazy evaluation), but if the element is not there the function has to examine all elements to reach that conclusion.

It is somewhat more convenient if the function can assume the list is sorted, i.e. the elements are in increasing order. The search can then be stopped when it has `passed' the wanted element. As a consequence the elements must not only be comparable (class `Eq`), but also orderable (class `Ord`):

```
elem' :: a -> [a] -> Bool | Eq, Ord a
elem' e []      = False
elem' e [x:xs]  = e == x || (e > x && elem' e xs)
```

A much larger improvement can be achieved if the elements are not stored in a list, but in *search tree*. A search tree is a kind of `sorted tree'. It is a tree built following the definition of `Tree` from the previous paragraph:

```
:: Tree a   = Node a (Tree a) (Tree a)
            | Leaf
```

At every node an element is stored and two (smaller) trees: a `left' subtree and a `right' subtree (see figure 3.11). Furthermore, in a search tree it is required that all values in the left subtree are *smaller* or equal tho the value in the node and all values in the right subtree *greater*. The values in the example tree in the figure are chosen so that it is in fact a search tree.

In a search tree the search for an element is very simple. If the value you are looking for is equal to the stored value in an node, you are done. If it is smaller you have to continue searching in the left subtree (the right subtree contains larger values). The other way around, if the value is larger you should look in the right subtree. Thus the function `elemTree` reads as follows:

```
elemTree :: a (Tree a) -> Bool | Eq, Ord a
elemTree e Leaf = False
elemTree e (Node x le ri)
      | e==x  = True
      | e<x   = elemTree e le
      | e>x   = elemTree e ri
```

If the tree is well-balanced, i.e. it doesn't show big holes, the number of elements that has to be searched roughly halves at each step. And the demanded element is found quickly: a collection of thousand elements only has to be halved ten times and a collection of a million elements twenty times. Compare that to the half million steps `isMember` costs on average on a collection of a million elements.

In general you can say the complete search of a collection of `n` elements costs `n` steps with `isMember`, but only ^{2}log n steps with `elemTree`.

Search trees are handy when a large collection has to be searched many times. Also e.g. `search` from subsection 3.3.1 can achieve enormous speed gains by using search trees.

Structure of a search tree

The form of a search tree for a certain collection can be determined `by hand'. Then the search tree can be typed in as one big expression with a lot of data constructors. However, that is an annoying task that can easily be automated.

Like the function `insert` adds an element to a sorted list (see subsection 3.1.4), the function `insertTree` adds an element to a search tree. The result will again be a search tree, i.e. the element will be inserted in the right place:

```
insertTree :: a (Tree a) -> Tree a | Ord a
insertTree e Leaf   = Node e Leaf Leaf
insertTree e (Node x le ri)
      | e<=x  = Node x (insertTree e le) ri
      | e>x   = Node x le (insertTree e ri)
```

If the element is added to a `Leaf` (an `empty' tree), a small tree is built from `e` and two empty trees. Otherwise, the tree is not empty and contains a stored value `x`. This value is used to decide whether `e` should be inserted in the left or the right subtree. When the tree will only be used to decide whether an element occurs in the tree there is no need to store duplicates. It is straight forward to change the function `insertTree` accordingly:

```
insertTree :: a (Tree a) -> Tree a | Ord, Eq a
insertTree e Leaf   = Node e Leaf Leaf
insertTree e node=:(Node x le ri)
      | e<x   = Node x (insertTree e le) ri
      | e==x  = node
      | e>x   = Node x le (insertTree e ri)
```


By using the function `insertTree` repeatedly, all elements of a list can be put in a search tree:

```
listToTree :: [a] -> Tree a | Ord, Eq a
listToTree []     = Leaf
listToTree [x:xs] = insertTree x (listToTree xs)
```

The experienced functional programmer will recognise the pattern of recursion and replace it by an application of the function `foldr`:

```
listToTree :: ([a] -> Tree a) | Ord, Eq a
listToTree = foldr insertTree Leaf
```

Compare this function to `isort` in subsection 3.1.4.

A disadvantage of using `listToTree` is that the resulting search tree is not always well-balanced. This problem is not so obvious when information is added in random order. If, however, the list which is made into a tree is already sorted, the search tree `grows cooked'. For example, when running the program

```
Start = listToTree [1..7]
```

the output will be

```
Node 7 (Node 6 (Node 5 (Node 4 (Node 3 (Node 2 (Node 1 Leaf Leaf) Leaf) Leaf)
Leaf) Leaf) Leaf) Leaf
```

Although this is a search tree (every value is between values in the left and right subtree) the structure is almost linear. Therefore logarithmic search times are not possible in this tree. A better (not `linear') tree with the same values would be:

```
Node 4  (Node 2 (Node 1 Leaf Leaf)
                (Node 3 Leaf Leaf)
        )
        (Node 6 (Node 5 Leaf Leaf)
                (Node 7 Leaf Leaf)
        )
```

3.6.3 Sorting using search trees

The functions that are developed above can be used in a new sorting algorithm. For that one extra function is necessary: a function that puts the elements of a search tree in a list preserving the ordering. This function is defined as follows:

```
labels :: (Tree a) -> [a]
labels Leaf             = []
labels (Node x le ri)   = labels le ++ [x] ++ labels ri
```

The name of the function is inspired by the habit to call the value stored in a node the *label* of that node.

In contrast with `insertTree` this function performs a recursive call to the left *and* the right subtree. In this manner every element of the tree is inspected. As the value `x` is inserted in the right place, the result is a sorted list (provided that the parameter is a search tree).

An arbitrary list can be sorted by transforming it into a search tree with `listToTree` and than summing up the elements in the right order with `labels`:

```
tsort :: ([a] -> [a]) | Eq, Ord a
tsort = labels o listToTree
```

3.6.4 Deleting from search trees

A search tree can be used as a database. Apart from the operations enumerate, insert and build, which are already written, a function for deleting elements comes in handy. This function somewhat resembles the function `insertTree`; depending on the stored value the function is called recursively on its left or right subtree.

```
deleteTree :: a (Tree a) -> (Tree a) | Eq, Ord a
deleteTree e Leaf   = Leaf
```

```
deleteTree e (Node x le ri)
    | e<x   = Node x (deleteTree e le) ri
    | e==x  = join le ri
    | e>x   = Node x le (deleteTree e ri)
```

If, however, the value is found in the tree (the case `e==x`) it can't be left out just like that without leaving a `hole'. That is why a function `join` that joins two search trees is necessary . This function takes the largest element from the left subtree as a new node. If the left subtree is empty, joining is of course no problem:

```
join :: (Tree a) (Tree a) -> (Tree a)
join Leaf b2 = b2
join b1   b2 = Node x b1' b2
where
    (x,b1') = largest b1
```

The function `largest`, apart from giving the largest element of a tree, also gives the tree that results after deleting that largest element. These two results are combined in a tuple. The largest element can be found by choosing the right subtree over and over again:

```
largest :: (Tree a) -> (a,(Tree a))
largest (Node x b1 Leaf) = (x,b1)
largest (Node x b1 b2)   = (y,Node x b1 b2')
where
    (y,b2') = largest b2
```

As the function `largest` is only called from `join` it doesn't have to be defined for a `Leaf`-tree. It is only applied on non-empty trees, because the empty tree is already treated separately in `join`.

3.7 Abstract datatypes

In subsection 1.6 we have explained the module structure of Clean. By default a function only has a meaning inside the implementation module it is defined in. If you want to use a function in another module as well, the type of that function has to be repeated in the corresponding definition module. Now, if you want to export a type, you simply repeat the type declaration in the definition module. For instance, the type `Day` of subsection 3.4.1 is exported by repeating its complete definition

```
definition module day

:: Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

in the definition module.

For software engineering reasons it often much better only to export the *name* of a type but not its concrete definition (the right-hand side of the type definition). In Clean this is done by specifying only the left-hand side of a type in the definition module while the concrete definition (the right-hand side of the type definition) is hidden in the implementation module, e.g.

```
definition module day

:: Day
```

So, Clean's module structure can be used to hide the actual definition of a type. The actual definition of the type can be an algebraic data type, a record type or a synonym type (giving a new name to an existing type).

A type of which the actually definition is hidden is called an *abstract data type*. The advantage of an abstract data type is that, since its concrete structure remains invisible for the outside world, an object of abstract type can only be created and manipulated with help of functions that are exported by the module as well. The outside world can only pass objects of abstract type around or store them in some data structure. They cannot create such an abstract object nor change its contents. The exported functions are the only means with which the abstract data can be created and manipulated.

Modules exporting an abstract data type provide a kind of data encapsulation known from the object oriented style of programming. The exported functions can be seen as the methods to manipulate the abstract objects.

The most well-known example of an abstract data type is a stack. It can be defined as:

```
definition module stack

:: Stack a

Empty   ::   (Stack a)
isEmpty ::   (Stack a) -> Bool
Top     ::   (Stack a) -> a
Push    :: a (Stack a) -> Stack a
Pop     ::   (Stack a) -> Stack a
```

It defines an abstract data type (object) of type 'Stack of anything'. `Empty` is a function (method) which creates an empty stack. The other functions can be used to push an item of type `a` on top of a given stack yielding a stack (`Push`), to remove the top element from a given stack (`Pop`), to retrieve the top element from a given stack (`Top`), and to check whether a given stack is empty or not (`isEmpty`).

In the corresponding implementation module one has to think of a convenient way to represent a stack, given the functions (methods) on stacks one has to provide. A stack can very well be implemented by using a list. No new type is needed. Therefore, a stack can be defined by using a synonym type.

```
implementation module stack

::Stack a :== [a]

Empty :: (Stack a)
Empty = []

isEmpty :: (Stack a) -> Bool
isEmpty []  = True
isEmpty s   = False

Top :: (Stack a) -> a
Top [e:s] = e

Push :: a (Stack a) -> Stack a
Push e s = [e:s]

Pop :: (Stack a) -> Stack a
Pop [e:s] = s
```

3.8 Run-time errors

The static type system of Clean prevents run-time type errors. The compiler ensures that it is impossible to apply a function to arguments of the wrong type. This prevents a lot of errors during program execution. Nevertheless, the compiler is not able to detect all possible errors. In this section we discuss some of the errors that can still occur.

3.8.1 Non-termination

It is perfectly possible to write programs in Clean that can run forever. Sometimes this is the intention of the programmer, in other situations this is considered an error. A very simple example of a program that will not terminate is:

```
Start :: String
Start = f 42

f :: t -> u
f x = f x
```

There is nothing wrong with the type of this program, but it will never produce a result. Programs that yields an infinite data structure are other examples of programs that does not terminate:

```
Start :: [Int]
Start = ones where ones = [1:ones]
```

As we have seen, there are programs manipulating infinite data structures that do terminate. In general it is undecidable whether a given program will terminate or not. So, the compiler is not able to warn you that your program does not terminate.

In large programs it may be pretty complicated to detect the reason why a program does not terminate. When a critical observation of your program does not indicate the error you should isolate the error by breaking your program into pieces that are tested individually. You can prevent a lot these problems by making it a habit to inspect the termination properties of each function you have written immediately after you have written it down. As we have seen there are many valid programs that use infinite data structures. For instance the first 20 prime numbers are computed by (see section 3.2.5):

```
Start :: [Int]
Start = take 20 primes
```

3.8.2 Partial functions

Many of the functions that you write are partial functions: the result of the function is only defined for some of the arguments allowed by the type system. Some examples are:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

depth :: (Tree a) -> Int
depth (Node _ l r) = max (depth l) (depth r)
```

The result of the function `fac` is only defined for integers greater or equal to `0`. For negative arguments the function does not terminate. The function `depth` is only defined for trees that are not a single leaf. There is no rule alternative applicable to the expression `depth Leaf`. Whenever Clean tries to evaluate this expression an appropriate error message is generated:

```
Run time error, rule 'depth' in module 'test' does not match
```

The Clean compiler analyses functions during compilation. Whenever the compiler cannot decide that there is always a rule alternative applicable it generates an appropriate warning:

```
Warning [test.icl,35,depth]: function may fail
```

Termination problems can be prevented by generating an error message or making sure that none of the alternatives is applicable:

```
fac1 :: Int -> Int
fac1 0 = 1
fac1 n | n>0 = n * fac1 (n-1)
             = abort "fac1 has an negative argument"

fac2 :: Int -> Int
fac2 0 = 1
fac2 n | n>0 = n * fac2 (n-1)
```

When called with a negative argument these function will produce the following error messages:

```
fac1 has an negative argument
Run time error, rule 'fac2' in module 'test' does not match
```

Although the error is easy to detect in this way it might be a problem to detect the reason why this expression was generated. You should make it a habit to consider what will happen when the function is called with 'wrong' arguments. With respect to detecting problems the functions `fac1` and `fac2` are considered better than `fac`. When you are

worried about the additional runtime taken by the additional test you might consider doing the test for appropriate arguments once and for all:

```
fac3 :: Int -> Int
fac3 n | n >= 0 = f n
    where   f 0 = 1
            f n = n * f(n-1)
```

In general it is worse to receive a wrong answer than receiving no answer at all. When you obtain no result you are at least aware of the fact that there is a problem. So, ***do not write***:

```
fac4 n | n < 1 = 0
               = n * fac4 (n-1)
```

3.8.3 Cyclic dependencies

When your program uses its own results before they can be computed you have by a nasty error known as *cycle in spine* or *black hole*. The origin of the name of this error is found a possible implementation of functional languages, see part III. This kind errors can be very hard to detect.

We will illustrate this kind of error by a program that generates a sorted list of all numbers of the form 2^n3^m. Computing these numbers is known as the Hamming problem. We will generate Hamming numbers by observing that a new Hamming number can be computed by multiplying and existing number by 2 or 3. Since we generate an infinite list of these numbers we cannot use on ordinary sorting function to sort hamming numbers. We sort these numbers by an adapted version of the function `merge`.

```
ham :: [Int]
ham =: merge [n*2 \\ n <- ham] [n*3 \\ n <- ham]
    where   merge  l=:[a:x] m=:[b:y]
                   | a<b       =  [a:merge x m]
                   | a==b      =  merge l y
                   | otherwise =  [b: merge l y]

Start::[Int]
Start = take 100 Ham
```

Here it is no problem that the function `merge` is only defined for non-empty lists, it will only be used to merge infinite lists. Execution of this program yields:

```
Run Time Warning: cycle in spine detected
```

The source of the error is that the program is not able to generate a first Hamming number. When we know this and observe that 1 is the first hamming number ($1 = 2^0 3^0$), it is easy to give a correct version of this function:

```
ham :: [Int]
ham =: [1:merge [n*2 \\ n <- ham] [n*3 \\ n <- ham]]
    where   merge  l=:[a:x] m=:[b:y]
                   | a<b       =  [a:merge x m]
                   | a==b      =  merge l y
                   | otherwise =  [b: merge l y]
```

When we do not use the computed Hamming numbers to generate new Hamming numbers, but compute these numbers again as in:

```
ham :: [Int]
ham = merge [n*2 \\ n <- ham] [n*3 \\ n <- ham]
    where   merge  l=:[a:x] m=:[b:y]
                   | a<b       =  [a:merge x m]
                   | a==b      =  merge l y
                   | otherwise =  [b: merge l y]
```

we obtain a '`heap full`' message instead of the '`cycle in spine`'. For each occurrence of `ham` the expression is evaluated again. Since none of these functions is able to generate a first element, an infinite expression will be generated. The heap will always be too small to hold this expression.

3.8.4 Insufficient memory

The heap is a piece of memory used by the Clean program to evaluate expressions. When this memory is exhausted the program tries to regain memory by removing parts of the expression that are not needed anymore. This process is called *garbage collection*. When it is not possible to find sufficient unused memory during garbage collection the program is aborted and the error message 'heap full' is displayed. The size of the heap used by programs written in Clean can be determined in the Clean system. When you program displays the 'heap full' message you can try it again after you have increased the heap size. As shown in the previous paragraph it is also possible that a programming error causes this error message. No matter how large the heap is, the program will never behave as intended. In large programs it can be pretty tuff to locate the source of this kind of error.

Apart from the heap, a program written in Clean uses some stacks. These stacks are used to maintain information on function arguments and parts of the expression currently under evaluation. Also these stacks can be too small. What happens when such a *stack overflow* occurs depends on the platform you are using and the options set in the Clean system. When you choose 'Check Stacks' the program should notice that the stack space is exhausted and abort the program with an appropriate message. Stack checks cause a slight run-time overhead. Hence, people often switch stack checks off. Without these checks the stack can 'grow' within memory used for other purposes. The information that was kept there is spoiled. This can give error like 'illegal instruction'.

Whether an erroneous program causes a heap full message or a stack overflow can depend on very small details. The following program will cause a 'heap full' error:

```
Start :: String
Start = f 42

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

We can understand this by writing the first lines of a trace:

```
Start
→  f 42
→  f (f 42)
→  f (f (f 42))
→  …
```

It is clear that this expression will grow without bound. Hence execution will always cause a heap full error.

When we add a strictness annotation is added to the function `f`, the argument of `f` will be evaluated before the application of `f` itself is evaluated (see part III).

```
Start :: String
Start = f 42

f :: !t -> u
f x = f (f x)
```

The trace looks very similar:

```
Start
→  f 42
→  f (f 42)
→  f (f (f 42))
→  …
```

In order to keep track of the function and its argument under evaluation some stack space is used. Now it depends on the relative size of the stack and the memory which is the first to be exhausted. Clean has a built in strictness analyzer that approximates the strictness properties of functions. A very small and semantically irrelevant change may change the derived strictness properties and hence cause the difference between a 'heap full' or 'stack overflow' error.

3.9 Exercises

Exercise 3.1

Equality on tuples can be defined as:

```
(==) (a,b) (c,d) = a == c && b == d
```

Although the equality operator is also applied in the right-hand side expression this function is actually not recursive.

What is the difference between this operator definition and the recursive definition of equality for lists in Section 3.1.2?

Exercise 3.2

Define the function `flatten` (see Section 3.1.2) in terms of `foldr` and `++`.

Exercise 3.3

Write a list comprehension for generating all permutations of some input list.

Exercise 3.4

Describe the effect on the evaluation order of swapping `x==y` and `xs==ys` in the definition of `==` in Section 3.2.2.

Exercise 3.5

Extend the set of operators on rational numbers with == and <.

Exercise 3.6

Discuss how you can guarantee that rational numbers used in ordinary programs are always 'simplified'.

Exercise 3.7

Define an appropriate data type for AVL-trees and define functions for balancing, searching, inserting and deleting elements in such trees.