Part III
Chapter 2
Programming styles and paradigms

In the previous chapter we treated several ways to construct programs. In this chapter we show various styles to construct the functions in the modules. In the previous chapter we focused on the organisation and order of events. In this chapter we will focus on the appearance of the functions themselves.

This book is devoted to functional programming languages. In section 2 we will threat the major alternative programming paradigms briefly. We will focus on how the essential aspects of these languages can be mimicked in Clean.

2.1 Programming styles

Also in the concrete writing of functions and program fragments there are several choices to be made. Until now you will have made most of these choices on your intuition (or by copying and adapting examples). In this section a number of possibilities will be shown in order to show the available options. Again there is not a style that is preferable in all situations or a method that will solve each and every problem. Moreover, the shown styles can be mixed.

We will illustrate some programming styles using the n-queens problem. The question is how to place n queens on a n×n chess board without attacking each other. The solutions for n=6 can be drawn as shown in figure 1.

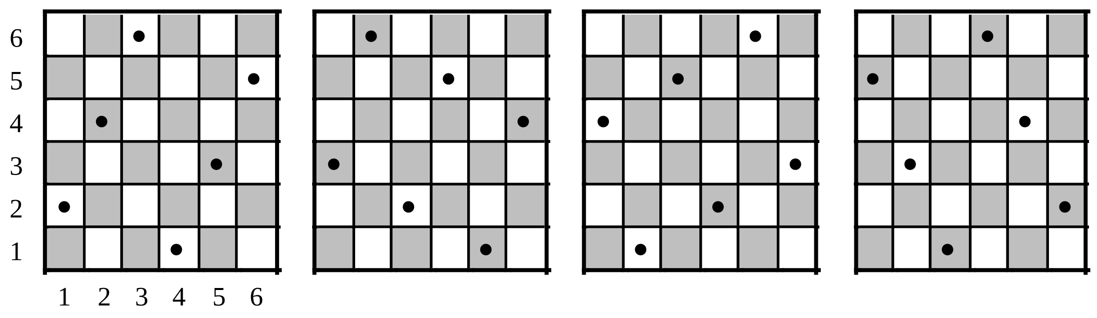

Figure 1 The solutions of the six queens problem.

A constructive algorithm that places queens at the correct places is not known. We have to find correct solutions by trial and error. A naive algorithm generates all possible ways to place the queens on the board and selects the allowed configurations form this list. In such an algorithm the position of a queen will be represented as a tuple of the column number and row number. The first solution depicted above is represented as:

```
[(1,2),(2,4),(3,6),(4,1),(5,3),(6,5)]
```

For simplicity we will indicate rows and columns by integers. The type of the representation of a configuration is `[(Int,Int)]`. Chess players probably prefer to indicate this solution as: a2, b4, c6, d1, e3, f5.

When we think a little before we start to program, we observe that it is useless to consider configurations with two (or more) queens in the same column (or row). This implies that each column should contain exactly one queen for any allowed configuration. This observation can be used to simplify the representation. Instead of recording the configuration as a list of tuples, it can better be stored as a list of row numbers: one for each column. Configurations are represented by elements of type `[Int]`. The first solution depicted above will now be represented as:

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

This new representation of the solution also implies a significant reduction of the search space. All allowed configurations can be found by filtering the safe configurations from the permutations of `[1..m]`. We will use a somewhat more sophisticated algorithm to generate all allowed configurations. We will start with an empty board try to extend the partial solutions step by step to a full solutions as long as it is safe. The iterations for the n=4 case are (unsafe partial solutions for iteration 2 are shown in strike trough):

```
0:  [[]]
1:  [[1],[2],[3],[4]]
2:  [~~[1,1]~~,~~[1,2]~~,[1,3],[1,4],~~[2,1]~~,~~[2,2]~~,~~[2,3]~~,[2,4],[3,1],~~[3,2]~~,~~[3,3]~~,~~[3,4]~~,…]
=   [[1,3],[1,4],[2,4],[3,1],[4,1],[4,2]]
3:  [[1,4,2],[2,4,1],[3,1,4],[4,1,3]]
4:  [[2,4,1,3],[3,1,4,2]]
```

Note that we have not yet written a single line of program code. Our representation and algorithm is still independent of a particular programming language or style. The representation of the datastructure in a functional language is obvious. We will now write down the algorithm in Clean in a number of programming styles.

List comprehensions

The given algorithm can be written very easy using list comprehensions. Since all solutions to generate are independent we use the list of successes method (see also the chapter about parser combinators in part II). The function `qs` has as argument the number of queens that still must be placed. If there are zero queens to place, the only solution is an empty list. In case there are n queens to place, we extend the solutions of n-1 queens by all possible positions for the current column that are safe.

```
queens :: Int -> [[Int]]
queens m = qs m
  where qs 0 = [[]]
        qs n = [p++[q] \\ p <- qs (n-1), q <- pos | safe p q]
        pos = [1..m]
```

To check whether a given (partial) solution is safe or not, we test for each existing column `i` and queen on row `j` in that column if the new queen at position `q` in column `x` is not attacked. The new queen at `(x,q)` is attacked by the queen at position `(i,j)` if they are on the same row (`j==q`), on the same diagonal that goes down to the right (`i+j==x+q`) or the diagonal that goes up to the right (`i-j==x-q`).

```
safe :: [Int] Int -> Bool
safe p q = and [j<>q && i+j <> x+q && i-j <> x-q \\ i <- [1..] & j <- p]
  where x = length p+1
```

Using these functions the solutions for the 6 queens problem can be found by executing:

```
Start = queens 6
```

The result is:

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

As you could have expected, or seen in figure 1, there is much symmetry in this solutions. For n=6 there is in fact only one solution. The other configurations can be obtained by mirroring this solution. This symmetry can be used to improve the algorithm, but this symmetry is not exploited in here.

Explicit recursion

Instead of using list comprehensions to express the algorithm that solves the n-queens problem, we can use explicit recursion. We do not aim to be as close as possible to the implementation using list comprehensions, we just give a clear algorithm. The program begins calling the function `qs` with the number of queens to place. For zero queens the only solution is, again, the empty list of positions. For `n` queens we `extend` the solution for `n-1` queens by one queen in column m-n+1. To extend a partial solution `p` the function `next` tries all possible extensions with one queen. Since we are working with lists of solutions, `extend` continues by extending the other partial solutions `ps`. The function `next` generates a call to `test` for the given partial solution and all possible positions for the next queen. Finally, the function `test` checks whether the position `q` for a queen is a `safe` extension of partial solution `p`. When it is safe, a list containing this solution is constructed, otherwise it yields the empty list of solutions.

```
queens :: Int -> [[Int]]
queens m = qs m
  where qs :: Int -> [[Int]]
        qs 0 = [[]]
        qs n = extend (qs (n-1))

        extend :: [[Int]] -> [[Int]]
        extend [p:ps] = next pos p ++ extend ps
        extend []     = []

        pos = [1..m]

        next :: [Int] [Int] -> [[Int]]
        next [q:r] p = test p q ++ next r p
        next [] p = []

        test :: [Int] Int -> [[Int]]
        test p q | safe p q  = [p++[q]]
                 | otherwise = []
```

To test whether a partial solution `p` can be safely extended by a queen in column `x` at row `q` or not, we check for each column `i` if that queen at row `j` is at the same row or diagonal as the new queen. Without list comprehensions it is more convenient to continue with the next column using the and operator, `&&`, and a recursive call then to generate a list of booleans and use the function `and` to test whether they are all `True`.

```
safe :: [Int] Int -> Bool
safe p q = s q 1 (length p + 1) p
  where s q i x [j:r] = j<>q && i+j <> x+q && i-j <> x-q && s q (i+1) x r
        s q i x []    = True
```

These functions can be used exactly in the same way as the previous implementation.

There are of course many variation possible on this theme. For instance we can incorporate the function `test` in `next` to obtain:

```
        next` :: [Int] [Int] -> [[Int]]
        next` [q:r] p | safe p q  = [p++[q]: next` r p]
                      | otherwise = next` r p
        next` [] p = []
```

Toolbox functions

Since list manipulations are very common in functional programs, a set of powerful list processing functions is predefined in StdList. Most of these functions are introduced in part I and used occasionally in part II. These functions are like the famous Swiss army knife, there is a tool for almost everything, but it takes some experience to recognise this and use the apparatus with success.

Unless you are an experienced tool kit user you will generally have to write a (partially) direct recursive version first. In a next step you recognise the recursion patterns in this program and replace them by applications of the appropriate toolbox functions. In a next step you will probably rearrange the terms using standard transformations for these functions. An example of such a transformation is `map f (map g l) ≡ map (f o g) l`.

We will take the direct recursive program as starting point for the version using toolbox functions. The function `qs` cannot be written using toolbox functions since it is not a list processing function. The function `extend` basically maps `next pos` to each partial solution of n-1 queens. We replace the application of `extend` in the second alternative of `qs` by `map next`. Pattern matching on the argument `pos` will not be needed, so we will generate it later. The function `flatten` is included to transform the list of list of solutions to a list of solutions.

Inside the function `next` we first generate all possible positions for the new queen by `[1..m]`. This term was formerly called `pos`. From this list of extensions we filter those which are a safe extension of the partial solution `p` we have. Using `map (\q -> p++[q])` we extend the partial solution by all safe position for the new queen.

```
queens :: Int -> [[Int]]
queens m = qs m
  where qs :: Int -> [[Int]]
        qs 0 = [[]]
        qs n = flatten (map next (qs (n-1)))

        next :: [Int] -> [[Int]]
        next p = map (\q -> p++[q]) (filter (\q -> safe p q) [1..m])
```

For the function `safe` we have to process all elements in the partial solution. Since the function should yield a Boolean instead of a list we use `foldl`. The function `foldl` applies the same function to all elements of the list, hence it is impossible to count the column number `i` while we process the list. This is solved by turning the list of positions `p` in a column to a list of positions on the board by `zip2 [1..] p`.

```
safe :: [Int] Int -> Bool
safe p q = foldl (s q (length p + 1)) True (zip2 [1..] p)
  where s q x b (i,j) = j<>q && i+j <> x+q && i-j <> x-q && b
```

Since neither `next` nor `safe` uses pattern matching, it is possible to write the second alternative for `qs` in one (very long) expression. We do not recommend this since it decreases readability for human beings.

Continuation passing

A continuation determines the thing to do when the current function is finished. Instead of specifying this in the application of the function, we can pass the continuations as argument. This style of programming is applicable in many situations. It can be used as to increase the efficiency, but is gives you also additional power. One of the roots of the continuation passing style is in (denotational) semantics [e.g. Nielson 92]. It is also used in compiler construction [e.g. Appel 92].

In our example of the queens program in the version using explicit recursion we have the following functions:

```
extend :: [[Int]] -> [[Int]]
extend [p:ps] = next pos p ++ extend ps
extend []     = []

next :: [Int] [Int] -> [[Int]]
next [q:r] p = test p q ++ next r p
next [] p = []

test :: [Int] Int -> [[Int]]
test p q | safe p q  = [p++[q]]
         | otherwise = []
```

The applications of the append operator in the functions `extend` and `next` can easily be replaced by a continuation:

```
extend :: [[Int]] -> [[Int]]
extend [p:ps] = next pos p (extend ps)
extend []     = []

next :: [Int] [Int] [[Int]] -> [[Int]]
next [q:r] p c = test p q (next r p)
next []    p c = c

test :: [Int] Int [[Int]] -> [[Int]]
test p q c | safe p q  = [p++[q]:c]
           | otherwise = c
```

Similar to the program using direct recursion it is possible to combine the functions `next` and `test` to one function `next``.

```
next` :: [Int] [Int] [[Int]] -> [[Int]]
next` [q:r] p c | safe p q  = [p++[q]:next` r p c]
                | otherwise = next` r p c
next` []    p c = c
```

In the taste of many people this will be a more pure implementation of backtracking than the list of successes method used in the previous implementation. As you can see these implementations are equivalent and can be obtained from each other by a straight forward transformation. The implementation using continuations is more efficient since the append operator is omitted. In effect the Clean compiler uses a rather sophisticated transformation algorithm for list comprehensions to achieve exactly this effect [Koopman ??].

In this situation the continuation is a datastructure, in can also be a function that is the be applied to the result of the current expression. Especially when there are multiple results this can be advantageous since it prevents packing and unpacking these results in a datastructure. It is even possible to equip a function with multiple continuations. The continuation to use is determined by the result of the current function.

Characteristics of the programming styles

A number of styles for programming in the small has been shown above. In addition to these styles there are programming methodologies like monads. This is discussed in the next section. This subsection gives some general characteristics of the programming styles shown. In order to say something about the efficiency we list the execution time of the various programs first.

Execution times

In order to compare the efficiency of the various variants of the queens program we compute the number of solutions on a 10×10 board. The initial function evaluated is `Start = length (queens 10)` There appear to be 724 possible configurations. We used a Macintosh Performa 630 and Clean 1.1 for the timings. Each program had a heap of 500K and a stack of 100K. We kept the io time as low as possible by making a program that shows only the number of solutions and switching `Show Garbage Collections` off. The timings are accurate to about 0.02 seconds.

Execution time in seconds of various versions of the `queens` program	execution	garbage collection	io	total
1: list comprehensions	39.66	0.47	0.00	40.13
2: list comprehensions using `safe` from 3	9.75	0.05	0.00	9.80
3: direct recursion	10.78	0.05	0.00	10.83
4: direct recursion using `next``	9.71	0.04	0.00	9.75
5: toolbox functions	67.45	1.80	0.00	69.25
6: toolbox functions using `safe` from 3	17.65	0.16	0.00	17.81
7: continuations	10.20	0.03	0.00	10.23
8: continuations using `next``	9.63	0.02	0.00	9.65

Table 2.1 Timings of various implementations of the queens program.

There is a difference in speed from nearly one order of magnitude between the slowest and fastest program. Using only a tail recursive version of `safe` instead of the very similar `and` over a list of booleans differs a factor 4 in performance! This function checks each possibility and apparently it is much more efficient to use a tail recursive version instead of using an intermediate list of booleans. The reasons for the differences in efficiency is the efficient manipulation of strict arguments of a basic type as unboxed values. This is explained in the next chapter.

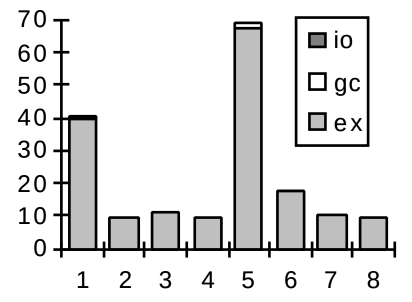

Figure 2.2. Timings of the `queens` programs (data from table 2.1).

These figures show that it is worthwhile to experiment with programming styles and variations of the program when execution time is a factor in the quality of your program. Tips and tricks to accelerate your programs can be found in chapter III.3.

List comprehensions

List can be generated by list comprehensions. This is a very compact and clear notation. The limitations are that only list can be generated and that it is impossible to transfer information from one element to an other one. The efficiency is determined by the cleverness of the compiler. After optimisation of the inner list comprehension, the price to be paid for a clear program is very limited in the given example.

Explicit recursion

Each algorithm can be expressed using explicit recursion. It is possible to control the efficiency by a careful construction of the implementation. In general there are many functions needed and the program will be longer than the version using list comprehensions or toolbox functions. Due to the arbitrary recursion patterns it can be harder to understand the program. The understandability can be improved by carefully choosing meaningful names.

Toolbox functions

For many common recursion patterns over lists there is a toolbox function that implements exactly that pattern. Using a toolbox function in this situation shows these patterns clearly. Since the toolbox functions can also be used to yield other types than lists they are more powerful than list comprehensions. However, it requires extensive training to write and understand programs with toolbox functions. The extensive use of Curried functions and intermediate datastructures by the toolbox functions has a runtime penalty. Due to transformations that can be automated to some extend [Wadler, Koopman], this additional costs can often be limited by a very advanced implementation.

Continuations

Writing programs using explicit recursion and continuations is very efficient. It takes some training to recognise the possibilities to use continuations, but once you are used to it you will appreciate it as a very powerful concept. Due to the availability of the continuation(s) it is more powerful than plain recursion. For instance we can replace the continuation by another expression, or remove duplicated occurrences of the current result from it.

Conclusion

There is not a programming style that is favourable in all situations. When you want to write a program a program quickly, or a program that is as clear as possible list comprehensions are probably a good candidate. When you want to make the use of standard recursion patterns clear, or want to be able to transform a simple implementation to an efficient one you should consider the use of toolbox functions. Explicit recursion can always be used, but can be less clear or compact. Fortunately, these styles can be mixed. This enables you to pick the most suitable style for each and every function.

2.2 Programming paradigms

The functional paradigm that is used to write programs in this book is not the only computational model used for the construction of programs. In this section we discuss the major alternatives to functional programming. We will show to what extend and how the strong point of writing programs using the alternative paradigms can be accomplished in functional programming.

Imperative

Imperative programming shows two striking differences when it is compared to functional programming. The first one is that the order of execution of statements in an imperative program is well defined and clearly visible. Moreover, the result of the program is critically dependent of the evaluation order. Functional programs are referential transparent. This property guarantees that the actual strategy used to compute the value of an expression is irrelevant for the obtained value. In a functional language the reduction strategy plays a less prominent role. The second difference between imperative and functional programming is the implicit state that is available everywhere in an imperative program. Showing the flow of data explicit as is done in a functional program has as advantage that the dependencies are made explicit and clear. When we change our mind and want to extend an existing functional program this explicit passing of data can become an obstacle. When the extension of the program implies that additional information has to be passed around, this can imply that large parts of the program need to be changed to incorporate this change. Since the program state in imperative programs is implicit, it is very easy to extend this state.

State manipulations generally produce a result and change the state. In a functional language this new state should be part of the result of the state manipulation. These two products of the state manipulation are generally packed in one tuple. Passing the various versions of the state around between different state manipulations makes the existence of dependencies very clear. Sometimes the explicit passing of the state makes its existence and the dependencies a little bit to apparent. There are techniques, like the parser combinators shown in part II, that can be used to make the existence of the state less prominent. Passing around the state is completely implicit in parsers constructed with the given parser combinators.

A state that has to be passed around is used by all state manipulation functions. A consequence of this approach is that the state is known in the manipulation functions. This can hinder the flexibility. When the type of the state is modified to reflect for instance the introduction of a new component, all manipulation functions must be changed accordingly. Fortunately there are some programming styles that can be used to make the programs prepared for changes. In part I we proposed to use records and record updates to represent a program state that is prepared for extension. This is still a good advice in many situations, but there can be reasons to use an other representation. The IOstate for instance, is represented by an abstract datatype. Using an abstract datatype has as advantage that all manipulations are done by the supplied interface and the actual implementation is hidden. Changing the type implies generally that (some of) the manipulation functions must be changed, but that their applications remain unchanged. The manipulation functions to change form a well defined set that can be found in one module.

In the functional community there is an other program technique, called monads [Wadler 95], that is heavily used to structure the manipulation of a state. For languages that are not equipped with an uniqueness type system, monads are used to model interaction with the world in a safe and referential transparent way. As long as the state is only manipulated with the operations of the monad, it will be single threaded. Although this is an important application of monads, they deserve some attention as structuring technique outside this application field. Monads combine the manipulation of the state used as abstract datatype with the implicit passing of a state and inventing names for the additional results yielded by the state manipulations. A monad is a member of the type class `monad`. The type class prescribe the availability of some manipulations. State manipulation using only these monad manipulations are independent of the actual instance of the class `monad` and hence well prepared for changes in the state to manipulate.

Monads have their roots in Category theory [Lane 71]. They can be used to model a wide variety of language features, including state, continuations, interaction exceptions and non-termination [Moggi 89, 91]. We will only describe the general notion of a monad and illustrate its possibilities by an application. Actually you have seen a simple form of monads and their main manipulation functions in the IO chapter in part I. The monad is a type `St s a` that describes a function that given a state `s` produces a tuple `(a,s)`. This tuple contains a value of type `a` and a new state.

```
::St s a :== s -> (a,s)
```

In StdEnv (StdFunc) the function `return` and the operator `` `bind` ``[1] are defined to support programming in a monadic style. In a more general setting a monad is a type constructor class of the form:

```
class Monad m
  where Return :: a -> m a
        (`Bind`) infix 0 :: (m a) (a -> m b) -> m b
```

Monads should obey a number of laws. There are many other operators in Clean that should obey laws. For instance the addition operator `+` should have a left and right unit element `zero` and it should be associative. This is:

```
zero + x ≡ x
x + zero ≡ x
(x + y) + z ≡ x + (y + z)
```

A binary operation with left and right unit that is associative (like `+`) is called a *monoid* in category theory. A *monad* differs from a monoid in that the right operand is a function. The laws for monads are similar to the laws for the monoid `+`. There is a sense in which `Return` should be the left and right unit for `` `Bind` ``, and `` `Bind` `` should be associative. The last law requires that `x` does not occur free in `o` since it is not bound by the lambda expression `\x -> n` in the right-hand side of the law. It is bound by `\x` in the left-hand side.

[1]The habit to write the infix operators associated with monads between back quotes is caused by the fact that this is the standard way to denote an infix operator in Haskell and Gofer. These are the favourite programming languages of many monad addicts.

```
Return v `Bind` f ≡ f v
m `Bind` (\x -> Return x) ≡ m
m `Bind` (\x -> (n `Bind` (\y -> o))) ≡ (m `Bind` (\x -> n)) `Bind` (\y -> o)
```

The clean system does not complain when you define an instance of + or the monadic functions that does not obey the given laws. These laws are necessary in category theory, but not in functional programming. Nevertheless you are encouraged to construct your instances such that they obey the laws, many people will expect them to be valid.

A possible application of monads are the combinator parsers from part II. The state was the list of input symbols to process. The function `Return` should behave like `succeed` and the operator `` `Bind` `` like sequential composition: <&=>. It is tempting to define instance of the functions in this class like:

```
Return :: b -> Parser s b
Return b = succeed b

(`Bind`) infix 0 :: (Parser s a) (a -> Parser s b) -> Parser s b
(`Bind`) p f = q
  where q xs = [t \\ (r,v) <- p xs, t <- f v r]
```

The definition for `Return` is as clean has we might expect. The definition for `` `Bind` ``[2] is indeed very similar to the definition of <&=>.

By a reduction sequence using the given function definitions we can show that the functions for the parser monad obey the given laws for monads. For the first law we have:

```
Return b `Bind` f
=   succeed b `Bind` f                      using the definition of Return
=   (\xs -> [(xs,b)]) `Bind` f              using the definition of succeed
=   \xs -> [t \\ (r,v) <- [(xs,b)], t <- f v r] using the definition of `Bind`
=   \xs -> [t \\ t <- f b xs]               the first generator has only one element
=   \xs -> f b xs                           definition of list comprehension
=   f b                                     Currying
```

Verification of the second law goes very similar:

```
m `Bind` (\x -> Return x)
=   \xs -> [t \\ (r,v) <- m xs, t <- (\x -> Return x) v r]
=   \xs -> [t \\ (r,v) <- m xs, t <- Return v r]
=   \xs -> [t \\ (r,v) <- m xs, t <- [(r,v)]]
=   \xs -> [(r,v) \\ (r,v) <- m xs]
=   \xs -> m xs
=   m
```

Finaly, the third law. This one is the most tricky since we have to use the definition for `` `Bind` `` in two directions.

```
m `Bind` (\x -> (n `Bind` (\y -> o)))
=   m `Bind` (\x->\xs->[t\\(r,v)<-n xs, t<-(\y->o) v r]
=   \ys->[u\\(s,w)<-m ys, u<-(\x->\xs->[t\\(r,v)<-n xs, t<-(\y->o) v r]) w s]
=   \ys->[u\\(s,w)<-m ys, u<-(\xs->[t\\(r,v)<-(\x->n) w xs, t<-(\y->o) v r]) s]
=   \ys->[u\\(s,w)<-m ys, u<-[t\\(r,v)<-(\x->n) w s, t<-(\y->o) v r]]
=   \ys->[t\\(s,w)<-m ys, (r,v)<-(\x->n) w s, t<-(\y->o) v r]
=   (\ys->[t\\(r,v)<-(\xs->[u\\(s,w)<-m xs, u<-(\x->n) w s]) ys,t<-(\y->o) v r]
=   (\xs->[u\\(s,w)<-m xs, u<-(\x->n) w s]) `Bind` (\y -> o)
=   (m `Bind` (\x -> n)) `Bind` (\y -> o)
```

Unfortunately the Clean system does not accept these definitions. The Clean system requires that instances of constructor classes, like `Monad`, are algebraic types. Hence, the Clean system reports that `Return` and `` `Bind` `` are illegal instances. There are two ways to work around this. The first solution is to give these functions a slightly different name and to work with them as if they belong to the class `Monad`. The other method to cope with this, is to transform the type parser into an algebraic type. We show the consequences of the second solution and define an algebraic type for parsers. The old type `Parser` is re-

[2]In the literature you will usually find the equivalent, but less elegant, definition:

```
q xs = flatten [f v r \\ (r,v) <- p xs]
```

peated for clarity. In order to distinguish these types we use different names. When you decide to use an algebraic type for parsers it is a good idea to have only the type `PARSER`.

```
:: PARSER a b = Pars ([a] -> [([a],b)])
:: Parser a b :== [a] -> [([a],b)]
```

Since parsers become now a datatype they cannot be applied silently to their input. In order to do this we use the function `pars`.

```
pars :: (PARSER a b) -> Parser a b
pars (Pars p) = p
```

Also the parser combinators must be changed accordingly. When this is done properly the parsers themselves need not to be changed (only the type `Parser` must be changed in `PARSER` in the situation where you have both types). The definition of the most important parser combinators becomes:

```
token  :: [s] -> PARSER s [s] | Eq s
token k = Pars p
   where p xs | k==take n xs = [(drop n xs,k)]
                             = []
         n = length k

succeed :: r -> PARSER s r
succeed v = Pars p where  p xs = [(xs,v)]

fail :: PARSER s r
fail = Pars p where p xs = []

(<&>) infixr 6 :: (PARSER s a) (PARSER s b) -> PARSER s (a,b)
(<&>) p q = Pars r
  where r xs = [(xs2,(v,w)) \\ (xs1,v) <- pars p xs, (xs2,w) <- pars q xs1]

(<|>) infixr 4 :: (PARSER s a) (PARSER s a) -> PARSER s a
(<|>) p1 p2 = Pars p where p xs = pars p1 xs ++ pars p2 xs
```

It looks attractive to avoid the use of the function `pars` by destruction of the algebraic type `PARSER` in the patterns of the operators. Using this the sequential composition operator becomes:

```
(<&>) (Pars p) (Pars q) = Pars r
  where r xs = [(xs2,(v,w)) \\ (xs1,v) <- p xs, (xs2,w) <- q xs1]
```

However, this is *not* equivalent with the version above. The new definition has other strictness properties. Applications of the operator `<*>`, constructed using `<&>`, cause the construction of an infinite expression and are not longer terminating! See chapter III.3.

An advantage of the monad approach is that it does not relay on the concrete representation of parsers. It is possible to define an other type for the parsers, make it part of the type class `Monad` and everything ought to work smoothly. In order to achieve parsers that are really independent of the representation of parser all parser combinators need to be replaced by monad manipulations. A common extension of the class monad is:

```
class MonadZeroPlus m   | Monad m
  where Zero :: m a
        (`Plus`) infixr 4 :: (m a) (m a) -> m a
```

Since `PARSER c` is already member of the class `Monad`, it can be made member of the class `MonadZeroPlus` by defining the functions `Zero` and `` `Plus` `` for it. Using the requirements from category theory the proper definitions for these functions are:

```
instance MonadZeroPlus (PARSER c)
  where Zero :: PARSER c a
        Zero = fail

        (`Plus`) :: (PARSER c a) (PARSER c a) -> PARSER c a
        (`Plus`) p q = p <|> q
```

When the set of monadic parser constructs is extended by a function to recognise items in the input (e.g. the parser `symbol`) it forms a complete basis for constructing parsers. The function `Return` corresponds to `succeed` and `Zero` fits to `fail`. The operator `` `Bind` `` is similar to `<&>` and ``` `Plus` ``` corresponds to `<|>`. Using these elementary components the parser `token` can be constructed as:

```
token :: [s] -> PARSER s [s] | Eq s
token []     = Return []
token [x:xs] = symbol x  `Bind` (\y  ->
               token  xs `Bind` (\ys ->
               Return [y:ys]))
```

Although this definition of the parser combinator `token` is correct, it decreases the performance of the entire parser by about 25%.

Functions constructed using only monad manipulations are independent of the concrete instance of the monad (just as polymorphic functions are independent of the concrete argument type). This has as advantage that one and the same function can be used for various datatypes. For instance, parse functions can be used for an ordinary parser and for a parser that records the file name and line number in order to produce better error messages.

Logic programming

A pure logic language consists of function definitions similar to the definitions in Clean. The important difference is that the function definitions can be used in two directions. In a functional program function definitions can only be used to replace an instance of the left-hand side by the specified body. In a logic it is possible to replace an instance of the right-hand side by a corresponding instance of the left hand side. Usually there are several ways to execute a program using this reduction strategy. The implementation of the logic language contains a built-in backtrack mechanism to try all reduction sequences necessary to achieve the desired result.

Although this built-in backtracking is convenient in several situations it is not needed nor always wanted. We have seen at several places (e.g. n-queens example and parsers) that backtracking can easily be replaced by a list of successes style program. This is easy to use and introduces backtracking only in those situations were it is really needed. When backtracking is needed it must be programmed explicitly. Due to the extensive treatment of the list of successes method in the rest of this book, we don't repeat its treatment here.

Object Oriented

A popular variant of top-down program design is called object oriented (OO) development. The distinctive feature of working object oriented is that the important entities to model are indicated in the design phases and remain present during all subsequent steps. The indicted objects are made more concrete in each subsequent phase. There are special programming languages to support object oriented programming (e.g. C++ and Eiffel). Basically objects in these languages are yet an other variant of abstract datatypes. The field inside an abstract datatype are called attributes and the manipulation functions are entitled methods. When objects are constructed by composition of simple objects there is also a strong relation with bottom-up development. In section ?? we will show how a number of other features of OO-languages can be mimicked in Clean.

Hiding

Inheritance

Object identity

Exercises

1 Use list comprehensions to generate all Pythagorean triangles. This are all triangles with sides of length a, b and c such that $a^2+b^2 = c^2$.

2 Replace this by: find the way to the cheese for a mouse in a maze.

3 Use list comprehensions to write down the simpler algorithm to generate the solutions of the n-queens problem: generate all permutations of `[1..m]` and select the safe ones.

4 Hard The given algorithm uses that there can be only one queen in each column. For the same reason there can be also only one queen in each row. Adapt the program given above to use this fact. Does this improve the performance?

5 Use direct recursion to generate all Pythagorean triangles. This are all triangles with sides of length a, b and c such that $a^2+b^2 = c^2$.

6 Use direct recursion to write down the simpler algorithm to generate the solutions of the n-queens problem: generate all permutations of `[1..m]` and select the safe ones.

7 Use toolbox functions to generate all Pythagorean triangles. This are all triangles with sides of length a, b and c such that $a^2+b^2 = c^2$.

8 Hard The given algorithm uses that there can be only one queen in each column. For the same reason there can be also only one queen in each row. Adapt the program given above to use this fact. Does this improve the performance?

9 Use toolbox functions to write down the simpler algorithm to generate the solutions of the n-queens problem: generate all permutations of `[1..m]` and select the safe ones.

10 Use toolbox functions to combine the two passes over the list `[1..m]` in the function `next` to one pass over the list.

11 Hard The given algorithm uses that there can be only one queen in each column. For the same reason there can be also only one queen in each row. Adapt the program given above to use this fact. Does this improve the performance?

12 Use continuations to generate all Pythagorean triangles. This are all triangles with sides of length a, b and c such that $a^2+b^2 = c^2$.

13 Use continuations to write down the simpler algorithm to generate the solutions of the n-queens problem: generate all permutations of `[1..m]` and select the safe ones.

14 Hard The given algorithm uses that there can be only one queen in each column. For the same reason there can be also only one queen in each row. Adapt the program given above to use this fact. Does this improve the performance?

15 Repeat the timings for the various versions of programs developed in the exercise sequence 1, 4, 7,11 and/or 2, 5, 8, 12. Does these figures agree with the timings shown above.

16 Hard Develop an algorithm for the n-queens problem using symmetry. You are free to choose the programming style that suits you best.

17 Extend the given parser fragments to a complete monadic parser for the language Tiny as described in the parser combinator chapter of part II.

18 Define a suitable monad to represent the state of the Tiny interpreter analogues to the state of the interpreter used in chapter II.5. Redefine the interpreter using the monadic state.