Chapter 3

Recursion

Recursion is a *self referential* style of definition commonly used in both mathematics and computer science. It is a fundamental programming tool, particularly important for manipulating data structures.

This book looks at recursion from several points of view. This chapter introduces the idea by showing how to write recursive functions in computer programs. Chapter 4 introduces induction, the mathematical technique for proving properties about recursive definitions, allowing you to use mathematics to prove the correctness of computer software. Chapter 5 extends induction to handle recursion over trees. Chapter 9 applies the same ideas to mathematics, where recursion is used to define sets inductively. You will find examples of recursion and induction throughout many branches of computer science. In addition to the examples given through these three chapters, we will study a larger case study in Chapter 13, where recursion and induction are applied to the problem of digital circuit design.

The idea in recursion is to break a problem down into two cases: if the problem is 'easy' a direct solution is specified, but if it is 'hard' we proceed through several steps: first the problem is redefined in terms of an easier problem; then we set aside the current problem temporarily and go solve the easier problem; and finally we use the solution to the easier problem to calculate the final result. The idea of splitting a hard problem into easier ones is called the *divide and conquer* strategy, and it is frequently useful in algorithm design.

The factorial function provides a good illustration of the process. Often the factorial function is defined using the clear but informal '...' notation:

$$n! = 1 \times 2 \times \cdots \times n$$

This definition is fine, but the '$\cdots$' notation relies on the reader's understanding to see what it means. An informal definition like this isn't well suited for proving theorems, and it isn't an executable program in most programming languages[1]. A more precise recursive definition of factorial consists of the

[1]Haskell is a very high level programming language, and it actually allows this style:

following pair of equations:

$$\begin{aligned} 0\,! &= 1 \\ (n+1)\,! &= (n+1) \times n\,! \end{aligned}$$

In Haskell, this would be written as:

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

The first equation specifies the easy case, where the argument is 0 and the answer 1 can be supplied directly. The second equation handles the hard case, where the argument is of the form $n + 1$. First the function chooses a slightly easier problem to solve, which is of size n; then it solves for $n!$ by evaluating `factorial n`, and it multiplies the value of $n!$ by $n + 1$ to get the final result.

Recursive definitions consist of a collection of equations that state properties of the function being defined. There are a number of algebraic properties of the factorial function, and one of them is used as the second equation of the recursive definition. In fact, the definition doesn't consist of a set of commands to be obeyed; it consists of a set of true equations describing the salient properties of the function being defined.

Programming languages that allow this style of definition are often called *declarative languages*, because a program consists of a set of declarations of properties. The programmer must find a suitable set of properties to declare, usually via recursive equations, and the programming language implementation then finds a way to solve the equations. The opposite of a declarative language is an *imperative* language, where you give a sequence of commands that, when obeyed, will result in the computation of the result.

3.1 Recursion Over Lists

For recursive functions on lists, the 'easy' case is the empty list `[]` and the 'hard' case is a non-empty list, which can be written in the form `(x:xs)`. A recursive function over lists has the following general form:

```
f ::  [a] ->  type of result
f [] =  result for empty list
f (x:xs) =  result defined using (f xs) and x
```

A simple example is the `length` function, which counts the elements in a list; for example, `length [1,2,3]` is 3.

in Haskell you can define `factorial n  = product [1..n]`. Most programming languages, however, do not allow this, and even Haskell treats the `[1..n]` notation as a high level abbreviation that is executed internally using recursion.

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

An empty list contains no elements, so `length []` returns 0. When the list contains at least one element, the function calculates the length of the rest of the list by evaluating `length xs` and then adds one to get the length of the full list. We can work out the evaluation of `length [1,2,3]` using equational reasoning. This process is similar to solving an algebra problem, and the Haskell program performs essentially the same calculation when it executes. Simplifying an expression by equational reasoning is the right way to 'hand-execute' a functional program. In working through this example, recall that `[1,2,3]` is a shorthand notation for `1:(2:(3:[]))`.

```
length [1,2,3]
= 1 + length [2,3]
= 1 + (1 + length [3])
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 3
```

It is better to think of recursion as a systematic calculation, as above, than to try to imagine low-level subroutine operations inside the computer. Textbooks on imperative programming languages sometimes explain recursion by resorting to the underlying machine language implementation. In a functional language, recursion should be viewed at a high level, as an equational technique, and the low-level details should be left to the compiler.

The function `sum` provides a similar example of recursion. This function adds up the elements of its list; for example, `sum [1,2,3]` returns $1+2+3 = 6$. The type of `sum` reflects the fact that the elements of a list must be numbers if you want to add them up. The type context '`Num a =>`' says that `a` can stand for any type provided that the numeric operations are defined. Thus `sum` could be applied to a list of 32-bit integers, or a list of unbounded integers, or a list of floating point numbers, or a list of complex numbers, and so on.

The definition has the same form as the previous example. We define the sum of an empty list to be 0; this is required to make the recursion work, and it makes sense anyway. It is common to define the base case of functions so that recursions work properly. This also usually gives good algebraic properties to the function. (This is the reason that $0\,!$ is defined to be 1.) For the recursive case, we add the head of the list `x` to the sum of the rest of the elements, computed by the recursive call `sum xs`.

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

The value of `sum [1,2,3]` can be calculated using equational reasoning:

```
sum [1,2,3]
= 1 + sum [2,3]
= 1 + (2 + sum [3])
= 1 + (2 + (3 + sum []))
= 1 + (2 + (3 + 0))
= 6
```

So far, the functions we have written receive a list and return a number. Now we consider functions that return lists. A typical example is the `(++)` function (its name is often pronounced either as 'append' or as 'plus plus'). This function takes two lists and appends them together into one bigger list. For example, `[1,2,3] ++ [9,8,7,6]` returns `[1,2,3,9,8,7,6]`.

Notice that this function has *two* list arguments. The definition uses recursion over the first argument. It's easy to figure out the value of `[] ++ [a,b,c]`; the first list contributes nothing, so the result is simply `[a,b,c]`. This observation provides a base case, which is essential to make recursion work: we can define `[] ++ ys = ys`. For the recursive case we have to consider an expression of the form `(x:xs) ++ ys`. The first element of the result must be the value `x`. Then comes a list consisting of all the elements of `xs`, followed by all the elements of `ys`; that list is simply `xs++ys`. This suggests the following definition:

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

Working out an example is a good way to check your understanding of the definition:

```
[1,2,3] ++ [9,8,7,6]
= 1 : ([2,3] ++ [9,8,7,6])
= 1 : (2 : ([3] ++ [9,8,7,6]))
= 1 : (2 : (3 : ([] ++ [9,8,7,6])))
= 1 : (2 : (3 : [9,8,7,6]))
= 1 : (2 : [3,9,8,7,6])
= 1 : [2,3,9,8,7,6]
= [1,2,3,9,8,7,6]
```

Once we know the structure of the definition—a recursion over `xs`—it is straightforward to work out the equations. The trickiest aspect of writing the definition of `(++)` is deciding over which list to perform the recursion. Notice that the definition just treats `ys` as an ordinary value, and never checks whether it is empty or non-empty. But you can't always assume that if there are several list arguments, the recursion will go over the first one. Some functions perform recursion over the second argument but not the first, and some functions perform a recursion simultaneously over *several* list arguments.

The `zip` function is an example of a function that takes two list arguments and performs a recursion simultaneously over both of them. This function takes two lists, and returns a list of pairs of elements. Within a pair, the first value comes from the first list, and the second value comes from the second list. For example, `zip [1,2,3,4] ['A', '*', 'q', 'x']` returns `[(1,'A'), (2,'*'), (3,'q'), (4,'x')]`. There is a special point to watch out for: the two argument lists might have different lengths. In this case, the result will have the same length as the shorter argument. For example, `zip [1,2,3,4] ['A', '*', 'q']` returns just `[(1,'A'), (2,'*'), (3,'q')]` because there isn't anything to pair up with the 4.

The definition of `zip` must do a recursion over *both* of the argument lists, because the two lists have to stay synchronised with each other. There are two base cases, because it's possible for either of the argument lists to be empty. There is no need to write a third base case `zip [] [] = []`, since the first base case will handle that situation as well.

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

Here is a calculation of the example above. The recursion terminates when the second list becomes empty; the second base case equation defines the result of `zip [4] []` to be `[]` even though the first argument is non-empty.

```
zip [1,2,3,4] ['A', '*', 'q']
= (1,'A') : zip [2,3,4] ['*', 'q']
= (1,'A') : ((2,'*') : zip [3,4] ['q'])
= (1,'A') : ((2,'*') : ((3,'q') : zip [4] []))
= (1,'A') : ((2,'*') : ((3,'q') : []))
= (1,'A') : ((2,'*') : [(3,'q')])
= (1,'A') : [(2,'*'), (3,'q')]
= [(1,'A'), (2,'*'), (3,'q')]
```

The `concat` function takes a list of lists and flattens it into a list of elements. For example, `concat [[1], [2,3], [4,5,6]]` returns one list consisting of all the elements in the argument lists; thus the result is `[1,2,3,4,5,6]`.

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

In working out the example calculation, we just simplify all the applications of `++` directly. Of course each of those applications entails another recursion, which is similar to the examples of `(++)` given above.

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

The base case for a function that builds a list must return a list, and this is often simply `[]`. The recursive case builds a list by attaching a value onto the result returned by the recursive call.

In defining a recursive function `f`, it is important for the recursion to work on a list that is shorter than the original argument to `f`. For example, in the application `sum [1,2,3]`, the recursion calculates `sum [2,3]`, whose argument is one element shorter than the original argument `[1,2,3]`. If a function were defined incorrectly, with a recursion that is bigger than the original problem, then it could just go into an infinite loop.

All of the examples we have seen so far perform a recursion on a list that is one element shorter than the argument. However, provided that the recursion is solving a smaller problem than the original one, the recursive case can be anything—it doesn't necessarily have to work on the tail of the original list. Often a good approach is to try to cut the problem size in half, rather than reducing it by one. This organisation often leads to highly efficient algorithms, so it appears frequently in books on the design and analysis of algorithms. We will look at the `quicksort` function, a good example of this technique.

Quicksort is a fast recursive sorting algorithm. The base case is simple: `quicksort [] = []`. For a non-empty list of the form `(x:xs)`, we will first pick one element called the *splitter*. For convenience that will be `x`, the first element in the list. We will then take all the elements of the rest of the list, `xs`, which are *less than or equal to* the splitter. We will call this list the *small elements* and define it as a list comprehension `[y | y <- xs, y<splitter]`. In a similar way we define the list of *large elements*, which are greater than the splitter, as `[y | y <- xs, y>=splitter]`. Now the complete sorted list consists first of the small elements (in sorted order), followed by the splitter, followed by the large elements (in sorted order).

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (splitter:xs) =
  quicksort [y | y <- xs, y<splitter]
  ++ [splitter]
  ++ quicksort [y | y <- xs, y>=splitter]
```

It is interesting to compare this definition with a conventional one in an imperative language; see any standard book on algorithms.

Exercise 1. Write a recursive function `copy :: [a] -> [a]` that copies its list argument. For example, `copy [2]` $\Rightarrow$ `[2]`.

Exercise 2. Write a function `inverse` that takes a list of pairs and swaps the pair elements. For example,

```
inverse [(1,2),(3,4)] ==> [(2,1),(4,3)]
```

Exercise 3. Write a function

```
merge :: Ord a => [a] -> [a] -> [a]
```

which takes two sorted lists and returns a sorted list containing the elements of each.

Exercise 4. Write `(!!)`, a function that takes a natural number `n` and a list and selects the nth element of the list. List elements are indexed from 0, not 1, and since the type of the incoming number does not prevent it from being out of range, the result should be a `Maybe` type. For example,

```
[1,2,3]!!0 ==> Just 1
[1,2,3]!!2 ==> Just 3
[1,2,3]!!5 ==> Nothing
```

Exercise 5. Write a function `lookup` that takes a value and a list of pairs, and returns the second element of the pair that has the value as its first element. Use a Maybe type to indicate whether the lookup succeeded. For example,

```
lookup 5 [(1,2),(5,3)] ==> Just 3
lookup 6 [(1,2),(5,3)] ==> Nothing
```

Exercise 6. Write a function that counts the number of times an element appears in a list.

Exercise 7. Write a function that takes a value `e` and a list of values `xs` and removes all occurrences of `e` from `xs`.

Exercise 8. Write a function

```
f :: [a] -> [a]
```

that removes alternating elements of its list argument, starting with the first one. For examples, `f [1,2,3,4,5,6,7]` returns `[2,4,6]`.

Exercise 9. Write a function `extract :: [Maybe a] -> [a]` that takes a list of `Maybe` values and returns the elements they contain. For example, `extract [Just 3, Nothing, Just 7] = [3, 7]`.

Exercise 10. Write a function

```
f :: String -> String -> Maybe Int
```

that takes two strings. If the second string appears within the first, it returns the index identifying where it starts. Indexes start from 0. For example,

```
f "abcde" "bc" ==> Just 1
f "abcde" "fg" ==> Nothing
```