Fun with functions: higher-order functions

Brigitte Pientka

September 28, 2007

In this note, we will cover a very powerful programming paradigm: Higher-order functions. Higher-order functions are one of the most important mechanisms in the development of modular, well-structured, and reusable programs. They allow us to write very short and compact programs, by abstracting over common functionality. This principle of abstraction is in fact a very important software engineering principle. Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code it is generally beneficial to combine them into one by abstracting out the varying part. The use of higher-order functions allows you to easily abstract out varying parts.

Since abstracting over varying parts to allow code reuse is such an important principle in programming, many languages support it in various disguises.

Step 1: Passing functions as arguments

Functions form the powerful building blocks that allow developers to break code down into simple, more easily managed steps, as well as let programmers break programs into reusable parts. As we have seen early on in the course, functions are values, just as numbers and booleans are values.

But if they are values, can we write functions which take other functions as arguments? In other words, can we write a function `f:int * (int -> int) -> int`? Would this be useful? The answer is YES! It is in fact extremely useful!

Consider the following implementation of the function which computes $\sum_{k=a}^{k=b} k$.

```
fun sumInts(a,b) = if (a > b) then 0 else a + sumInts(a+1,b)
```

Similarly, we can compute the function $\sum_{k=a}^{k=b} k^2$ or $\sum_{k=a}^{k=b} k^3$ or $\sum_{k=a}^{k=b} 3^k$

```
fun sumSquare(a,b) = if (a > b) then 0 else square(a) + sumSquare(a+1,b)
fun sumCubes(a,b) = if (a > b) then 0 else cubes(a) + sumCubes(a+1,b)
fun sumExp (a, b) = if (a > b) then 0 else exp(2, a) + sumExp(a+1, b)
```

All these functions look very similar, and the code is almost the same. But obviously, the sum depends on what function we are summing over! It is natural to ask, if we can write a generic sum function where we only give a lower bound `a`, and upper bound `b`, and a function `f` which describes what needs to be done in each iteration. The answer is, YES, we can! Here is how it works:

```
(* sum: (int -> int) * int * int -> int *)
fun sum(f,a,b) =
    if (a > b) then 0
    else (f a) + sum(f,a+1,b)
```

We can then easily obtain the previous functions as follows:

```
fun sumInts'(a,b) = sum(fn x => x, a, b)
fun sumCubes'(a,b) = sum(cube, a, b)
fun sumSquare'(a,b) = sum(square, a, b)
fun sumExp'(a,b) = sum(fn x => exp(2,x), a, b)
```

Note that we can create our own functions using `fn x => e`, where `e` is the body of the function and `x` is the input argument, and pass them as arguments to the function `sum`. Or we can pass the name of a previously defined function like `square` to the function `sum`. In general, this means we can pass functions as arguments to other functions.

What about if we want to sum up all the odd numbers between a and b?

```
  (* sumOdd: int -> int -> int *)
fun sumOdd (a, b) =
  let
    fun sum' (a,b) =
      if a > b then 0
      else a + sum'(a+2, b)
  in
    if (a mod 2) = 1 then
      (* a was odd *)
      sum'(a,b)
    else
      (* a was even *)
      sum'(a+1, b)
  end
```

This seems to suggest we can generalize the previous `sum` function and abstract over the increment function.

```
fun sum' (f, a, b, inc) =
  if (a > b) then 0
   else (f a) + sum'(f,inc(a),b, inc)
```

Isn't writing products instead of sums similar? – The answer is also YES. Consider the following code for computing the product of a function $f(k)$ for k between a and b.

```
  (* product : (int -> int) * int * int * (int -> int) -> int *)
  fun product(f, a, b, inc) =
  if (a > b) then 1
  else (f a) * product(f, inc(a), b, inc)
(* Using product to define factorial. *)
fun fact n = product(fn x => x, 1, n, fn x => (x + 1))
```

The main difference is two-folded: First, we need to multiply the results, and second we need to return 1 as a result in the base case, instead of 0.

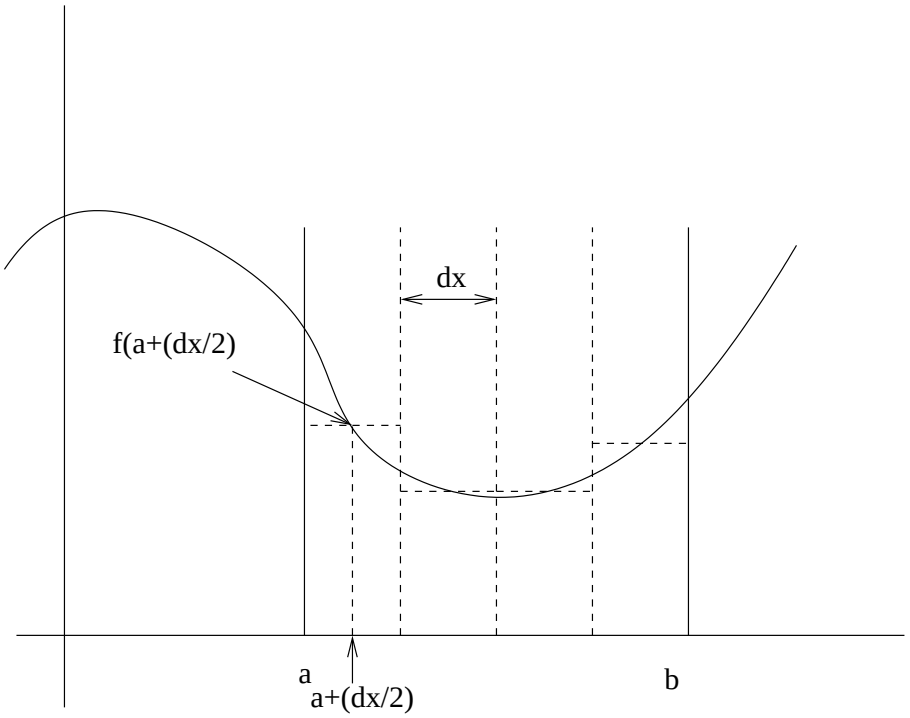
So how could abstract over addition and multiplication to generalize the function `product` and `sum` and define a function `series` which in addition to a function `f:int -> int`, a lower bound `a:int`, an upper bound `b:int`, increment function `inc:int -> int` and an accumulator for the result `r:int` also takes a function `comb:int * int -> int` to combine the results. Our goal is to write this function tail-recursive and by instantiating `comb` with addition we obtain the function `sum` and by instantiating it with multiplication we obtain the function `prod`.

```
  fun series(comb,f,a,b,inc,r) =
  let
    fun series' (a, r) =
      if (a > b) then r
      else
        series' (inc(a), comb(r, f(a)))
  in
    series'(a, r)
  end
fun sumSeries (f,a,b,inc) = series(fn (x,y) => x + y, f, a, b, inc, 0)
fun prodSeries (f,a,b,inc) = series(fn (x,y) => x * y, f, a, b, inc, 1)
```

Ok, we will stop here. Abstraction and higher-order functions are very powerful mechanisms in writing reusable programs.

Example 1: Integral

Let us consider here one familiar problem, namely approximating an integral. The integral of a function $f(x)$ is the area between the curve $y = f(x)$ and the x-axis in the interval $[a, b]$. We can use the rectangle method to approximate the integral of $f(x)$ in the interval$[a, b]$, made by summing up a series of small rectangles using midpoint approximation.



To approximate the area between a and b, we compute the sum of rectangles, where the width of the rectangle is dx. The height is determined by the value of the function f. For example, the area of the first rectangle is $dx * f(a + (dx/2))$.

Then we can approximate the area between a and b by the following series, where $l = a + (\Delta x)/2$ is our starting point.

$$\begin{aligned}\int_a^b f(x)\,dx &\approx f(l) * dx + f(l + dx) * dx + f(l + dx + dx) * dx + \ldots \\ &= dx * (f(l) + f(l + dx) + f(l + 2 * dx) + f(l + 3 * dx) \ldots)\end{aligned}$$

Assuming we have an iterative sum function `sumR` for reals, we can now compute the approximation of an integral as follows:

```
fun integral(f,lo,hi,dx) =
   dx * sumR(f,(lo + (dx / 2.0)), hi, fn x => (x + dx))
```

This is very short and elegant, and directly matches our mathematical theory.

Combining higher-order functions with recursive data-types

We can also combine higher-order functions with recursive data-types. One of the most common uses of higher-order functions is the following: We want to apply a function `f` to all elements of a list.

```
(* map: ('a -> 'b) -> 'a list -> 'b list *)
fun map f nil = nil
  | map f (h::t) = (f h)::(map f t)
```

We can also filter out all the elements in a list which fulfill a certain predicate `p:'a -> bool`.

```
(* filter: ('a -> bool) -> 'a list -> 'a list *)
  fun filter p (nil) = nil
  | filter p (x::l)  =
      if p(x) then x::filter p l
      else filter p l;
```

Step 2: Returning functions as results

In this section, we focus on returning functions as results. We will first show some simple examples demonstrating how we can transform functions into other functions. This means that higher-order functions may act as function generators, because they allow functions to be returned as the result from other functions.

Functions are very powerful. In fact, using a calculus of functions, the lambda-calculus, we can cleanly define what a computable function is. The lambda calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to the Turing machine formalism and was originally conceived by Alonzo Church.

The question of whether two lambda calculus expressions are equivalent cannot be solved by a general algorithm, and this was the first question, even before the halting problem, for which undecidability could be proved. The lambda calculus is also the foundation for many programming languages in particular functional programming.

Example 2: Currying and uncurrying

Currying has its origins in the mathematical study of functions. It was observed by Frege in 1893 that it suffices to restrict attention to functions of a single argument. For example, for any two parameter function $f(x, y)$, there is a one parameter function f' such that $f'(x)$ is a function that can be applied to y to give $(f'(x))(y) = f(x, y)$.

This idea can be easily implemented using higher-order functions. We will show how we can translate a function `f : 'a * 'b -> 'c` which expects a tuple `x:'a * 'b` into a function `f' : 'a -> 'b -> 'c` to which we can pass first the argument of type `'a` and then the second argument of type `'b`. The technique was named by Christopher Strachey after logician Haskell Curry, and in fact any function can be curried or uncurried (the reverse process).

```
(* curry : (('a * 'b) -> 'c) -> ('a -> 'b -> 'c) *)
fun curry f = (fn x => fn y => f (x,y))

(* uncurry: ('a -> 'b -> 'c) -> (('a * 'b) -> 'c) *)
fun uncurry f = (fn (y,x) => f y x)
```

To create functions we use the nameless function definition `fn x => e` where `x` is the input and `e` denotes the body of the function. Recall that the following two function definitions are equivalent:

```
val id = fn x => x

fun id x = x
```

Another silly function we can write is a function which swaps its arguments.

```
(* swap : ('a * 'b -> 'c) -> ('b * 'a -> 'c) *)
fun swap f = (fn (x,y) => f (y,x))
```

Example 3: Derivative

A bit more interesting is to implement the derivative of a function f. The derivative of a function f can be computed as follows:

$$\frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

We can approximating the result of the derivative by choosing ϵ to be some small number. Then given a function `f:real -> real` and a small number `dx`, we can compute a function `f'` which describes the derivative of `f`.

```
fun derive (f, dx) = (fn x => (f (x + dx) - f(x)) / dx)
```

Example 4: Partial evaluation and staged computation

Staged computation refers to explicit or implicit division of a task into stages. It is a standard technique from algorithm design that has found its way to programming languages and environments. Examples are partial evaluation which refers to the global specialization of a program based on a division of the input into static (early) and dynamic (late) data, and run-time code generation which refers to the dynamic generation of optimized code based on run-time values of inputs.

For example, given the following generic function,

```
(* funkyPlus : int -> int -> int *)
fun funkyPlus x y = x * x + y
```

we can first pass it `3`. This will generate the partially evaluated function `fn y => 3 * 3 + y`.

```
(* plus3 : int -> int *)
val plus3 = (funkyPlus 3)
```

We only partially evaluate the `funkyPlus` function by passing only one of the arguments to it. This yields again a function! We can see that templates, which occur in many programming languages are in fact nothing more than higher-order functions.

How would be generate a function which fixes `y`, but awaits still the input `x`?

```
val plus3' = (fn x => funkyPlus x 3)
```

The idea of partial evaluation is quite powerful to achieve code which can be configured during run-time and code which can achieve considerable efficiency gain.

Consider we have first defined a horrible computation:

```
  (* val horriblecomputation : int -> int *)
fun horriblecomputation(x:int):int =
    let fun ackermann(0:int, n:int):int = n+1
          | ackermann(m, 0) = ackermann(m-1, 1)
          | ackermann(m, n) = ackermann(m-1, ackermann(m, n-1))
        val y = Int.abs(x) mod 3 + 2
        fun count(0) = ackermann(y, 4)
          | count(n) = count(n-1)+0*ackermann(y,4)
        val large = 1000
    in
        ackermann(y, 1)*ackermann(y, 2)*ackermann(y, 3)*count(large)
    end;
```

Next, we define a function `f1`, which calls this horrible function.

```
(* val f1 : int * int -> int     *)
fun f1 (x:int, y:int) : int =
    let
       val z = horriblecomputation(x)
    in
       z + y
    end;
```

If we execute `f1` on a few instantiations, where the first instantiation for `x` does not change, then we have to execute this horrible computation each time.

```
val result1 = f1(10, 5);
val result2 = f1(10, 2);
val result3 = f1(10, 18);
```

Would it help to write a curried version?

```
(* val f2 : int -> int -> int *)
fun f2 (x:int) (y:int) : int =
let
  val z = horriblecomputation(x)
in
  z + y
end;
```

Can't we then compute the following function `f2'`:

```
(* val f2' : int -> int *)
val f2' = f2 10;
```

What will happen? – Well, let's see....The function `f2` is equivalent to the following :

```
val f2 =
 fn x => fn y =>
    let
      val z = horriblecomputation(x)
    in
      z + y
    end;
```

If we apply it to 10, then according to our operational semantics, this will evaluate to

```
  fn y =>
    let
      val z = horriblecomputation(10)
    in
      z + y
    end;
```

We have achieved nothing, since the horrible computation is hidden within a function, and we never evaluate inside functions! As a consequence we will compute the horrible computation every time we call the function `f2'`. But this is exactly what we wanted to avoid!

How can we generate a function which will only compute the horrible computation once and capture this result so we can re-use it for different inputs of `y`? – The idea is that we need to factor out the horrible computation. When given an input `x`, we compute the horrible computation yielding `z`, and then create a function which awaits still the input `y` and adds `y` and `z`.

```
fun f3 (x:int) : int -> int =
  let
     val z = horriblecomputation(x)
  in
     (fn y => z + y)
  end;
```

If we now create a function `f3'` s.t.

```
(* val f3' : int -> int *)
val f3' = f3 10;
```

we have done the horrible computation once, and created a closure which stores its result. Hence, when we use `f3'`, we do not need to re-evaluate the horrible computation.

This idea of staging computation is based on the observation that a partial evaluator can convert a two-input program into a staged program that accepts one input and produces another program that accepts the other input and calculates the final answer. The point being that the first stage may be run ahead of time (e.g. at compile time) while the second specialized stage should run faster than the original program. Many current compilers for functional programming employ partial evaluation and staged computation techniques to generate efficient code. It is worth pointing out that to understand this optimization, we really need to understand the operational semantics of how programs are executed.

Continuations

Generally speaking, a continuation is a representation of the execution state of a program (for example, a call stack) at a certain point in time. Many languages have constructs that allow a programmer to save the current execution state into an object, and then restore the state from this object at a later point in time (thereby resuming its execution). One can distinguish between *first-class continuations* (which are not available in Standard ML, although the SML NJ implementation provides them?), and *functions as continuations*. In these notes we only talk about the latter. Thus we do not refer to an extension of the language, but a particular programming technique based on higher-order functions.

A tail-recursive append function

Let us consider the function `append: 'a list * 'a list -> 'a list` that appends two lists.

```
fun append([], k) = k
  | append(h::t, k) = h::append(t,k)
```

This function is not tail-recursive, since it applies the list constructor `::` to the result of the recursive call `append(t,k)`. Nonetheless, this function is quite efficient and the definition above is fully satisfactory from a pragmatic point of view.

But one may still ask, if there is a way to write an append function in tail-recursive form. The answer is "yes", although it will be less efficient than the direct version above. In fact, it is a deep property of ML that *every* function can be rewritten in tail-recursive form!

Suppose we have a function `f:'a -> 'b` and would like to rewrite it as a function `f'` in tail-recursive form. The basic idea is to give f' an additional argument called a *continuation*, which represents the computation that should be done on the result of `f` In the base case, instead of returning a result, we call the continuation. This means that `f'` should have the type

```
f': 'a -> ('b -> 'c) -> 'c
```

In the recursive case, we add whatever computation should be done on the result to the continuation. So a continuation is like a functional accumulator! Or to put it differently, it will be a stack of functions.

When we use this function to compute `f` we give it the *initial continuation* which is often the identity function, indicating no further computation is done on the result.

Applying this basic idea to append yields the following:

```
(* app_tr: 'a list * 'a list * ('a list -> 'b) -> 'b *)
fun app_tr([], k, cont) = cont k
  | app_tr(h::l, k, cont) = app_tr(l, k, fn r => cont (h::r))
```

The first line implements the idea that instead of returning the result `k` we apply the continuation to `k`. The second line implements the idea that instead of constructing

```
x::append(l,k)
```

we call `app_tr` recursively on `l` and `k`, adding the task of prepending `x` to the argument `r` of the continuation `cont`. To illustrate how the functional accumulator is built during the recursive calls, consider the following sample computation.

```
    app_tr([1,2], [3,4], fn r => r)
⟹  app_tr([2], [3,4], fn r1 => (fn r => r) (1::r1))
⟹  app_tr([], [3,4], fn r2 => (fn r1 => (fn r => r) (1::r1)) (2::r2))   (*)
⟹  (fn r2 => (fn r1 => (fn r => r) (1::r1)) (2::r2)) [3,4]
⟹  (fn r1 => (fn r => r) (1::r1)) (2::[3,4]))
⟹  (fn r => r) (1::2::[3,4])
⟹  (1::2::[3,4])
```

This example illustrate how we build up a stack of functions as an accumulator, which keeps track of the computation we still need to do. Although the function is now tail-recursive, we have not saved anything, since we build up and then call the continuation. The benefit of continuations is not necessarily in gaining efficiency, but they provide us with a direct handle on future computation.

Using continuations as function generators

Let us consider again the following simple program to compute the exponent.

```
(* pow k n = n^k *)
fun pow 0 n = 1
  | pow k n = n * pow (k-1) n
```

While we wrote this program in curried form, when we partially evaluate it with `k = 2` we will not have generated a function "square". What we get back is the following:

```
fn n => n * pow 1 n
```

The recursive call of `pow` is shielded by the function abstraction (closure). One interesting question is how we can write a version of this power function which will act as a generator. When given `2` it will produce the function "square", when given `3` it produce the function "cube", etc. So more generally, when given an integer `k` it will produce a function which computes $\underbrace{n * \ldots * n}_{k} * 1$.

The simplest solution, is to factor out the recursive call before we build the closure.

```
fun powG 0 = (fn n => 1)
  | powG k =
  let
    val cont = powG (k-1)
  in
   fn n => n * (cont n)
  end
```

This allows us to compute a power generation function. To illustrate, let us briefly consider what happens when we execute `powG 2`. Let us first start with `powG 1`

```
    powG 2)
⇒   let val c = powG 0 in (fn n => n * c n) end
⇒   let val c = (fn n => 1) in (fn n => n * c n) end
⇒   (fn n => n * (fn => 1) n)

    powG 2)
⇒   let val c = powG 1 in (fn n => n * c n) end
⇒   let val c = (fn n => n * (fn => 1) n) in (fn n => n * c n) end
⇒   (fn n2 => n2 * (fn n1 => n1 * (fn n0 => 1) n1) n2)
```

Is this final result really the square function? – Yes, it is. Intuitively, this function is equivalent to

```
fn n2 => n2 * n2 * 1
```

Intuitively, all the function applications can be reduced as follows:

```
            (fn n2 => n2 * (fn n1 => n1 * (fn n0 => 1) n1) n2)
reduces to  (fn n2 => n2 * n2 (fn n0 => 1) n2)
reduces to  (fn n2 => n2 * n2 * 1)
```

So yes, we will have generated a version of the square function. However, the program is not tail-recursive. We will solve this problem using continuations. We can think of the continuation as a higher-order accumulator. The basic idea is to give the power function an additional argument called a *continuation*, which represents the computation that should be done on `n`. In the base case where `k = 0`, we will return the continuation. In the recursive case, we built up a new continuation (= function) describing what computation should be done on `n`! Applying this idea to write a power generator we get:

```
(* powGen: int -> (int -> int) -> (int -> int)  *)
fun powGen 0 cont = cont
  | powGen k cont = powGen (k-1) (fn n => n * (cont n))
```

How do we initialize the continuation to kick off the computation? It will be `fn n0 => 1`. It is the generator for `k = 0`. Now we can consider what will happen when we ask to generate the square function by executing `powGen 2 (fn n0 => 1)`

```
    powGen 2 (fn n0 => 1)
⇒   powGen 1 (fn n1 => n1 * (fn n0 => 1) n1)
⇒   powGen 0 (fn n2 => n2 * (fn n1 => n1 * (fn n0 => 1) n1) n2)
⇒   (fn n2 => n2 * (fn n1 => n1 * (fn n0 => 1) n1) n2)
```

We note the function is directly built in as the second argument of the function `powGen`.

Control with continuations

In the case of appending two lists and even in the power generation function, continuations can be applied, but they do not necessarily help us in writing simpler or more efficient code. However, in functions with more complex control flow, they can often be used to advantage. Here we will discuss two examples.

Finding element in a tree

The first example demonstrates a continuation-style implementation for finding an element `d` in a binary tree `T` which satisfies some property `p`. If such an element `d` exists, then we return `SOME(d)`, otherwise we return `NONE`. This problem can be solved straightforwardly as follows:

```
datatype 'a tree =
Empty | Node of 'a tree * 'a * 'a tree


(* Finding an element d in the tree T s.t. p(d) is true *)
(* find: ('a -> bool) * 'a tree -> 'a option *)
fun find(p, Empty) = NONE
  | find(p, Node(L, d, R)) =
   if  (p d) then SOME(d)
   else
      (case find(p, L) of
         NONE => find(p, R)
       | SOME(d') => SOME(d'))
```

However, this solution is slightly unsatisfactory, since in every recursion we check, if we have found an element in the left tree (see recursive call `find(p,L)`). If we actually did find an element, we pass this information `SOME(d')` back and must propagate this information up. This seems unnecessary, because once we found an element `d` which satisfies the property `p`, we would like to simply return. Rewriting this function using continuations we can in fact achieve this.

```
(* find_tr: ('a -> bool) * 'a tree * (unit -> 'a option) -> 'a option *)
fun find_tr(p, Empty, cont) = cont ()
  | find_tr(p, Node(L, d, R), cont) =
  if (p d) then SOME(d)                                          (* 1 *)
    else find_tr(p, L, fn () => find_tr(p, R, cont))
fun find'(p, t) =
  find_tr(p, t, fn () => NONE)                                  (* 2 *)
```

The continuation `cont` keeps track of the work we still need to do, i.e. it keeps track of the tree we still need to traverse. However, if we found an element `d` which satisfies the property `p` (see line `(* 1 *)`) we simply return the result `SOME(d)` and throw away the continuation. The continuation plays the role of a *failure continuation*, since it keeps track of what to do when we have not found an element `d` which satisfies `p`. Similarly, we often want to keep track what to do upon success. In this case, the continuation is often called *success continuation*. The next example is such an example.

Regular expression matcher

Next, we discuss the implementation of a simple regular expression matcher. Regular expression matching is a very useful technique for describing commonly occurring patterns. For example, the unix shell provides a mechanism for describing a collection of files by patterns as `*.sml` or `hw[1-3].sml`. The emacs editor provides an even richer language for regular expressions.

Typically, the patterns can be

- Singleton : matching a specific character
- Alternation: choice between two patterns
- Concatenation: succession of patterns
- Iteration : indefinite repetition of patterns

Note that regular expressions provide no concept of nesting of one pattern inside another. For this we require a richer formalism, namely context-free language. We can describe regular expressions inductively:

$$\text{regular expression} r \quad ::= \quad a \mid r_1\ r_2 \mid 0 \mid r_1 + r_2 \mid 1 \mid r^*$$

where a represents a single character from an alphabet. We say a string s matches a regular expression r, iff s is in the set of terms described by r. So for example:

- $a(p^*)l(e + y)$ would match *apple* or *apply*.
- $g(1 + r)(e + a)y$ would match *grey*, *gray* or *gay*.
- $g(1 + o)^*(gle)$ would match *google*, *gogle*, *goooogle* or *ggle*.

In general, we can describe a simple algorithm for a regular expression matcher as follows:

- s never matches 0
- s matches 1 iff s is empty.
- s matches a iff $s = a$.
- s matches $r_1 + r_2$ iff either s matches r_1 or r_2
- s matches $r_1\ r_2$ iff $s = s_1\ s_2$ where s_1 matches r_1 and s_2 matches r_2.
- s matches r^* iff either s is empty or $s = s_1\ s_2$ where s_1 matches r and s_2 matches r^*.

To implement regular expression matcher, we first need to define our regular expressions. We can do this straightforwardly by the following datatype definition:

```
datatype regexp =
Char of char | Times of regexp * regexp | One | Zero |
Plus of regexp * regexp | Star of regexp
```

Next, we will write a function `acc` which takes a regular expression, a character list, and a *continuation*, and yields a boolean value. Informally, the continuation determines how to proceed once an initial segment of the given character list has been determined to match the given regular expression. The remaining character list is passed to the continuation to compute the final result.

```
fun acc (Char(c)) [] k = false
| acc (Char(c)) (c1::s) k = (c = c1) andalso (k s)
| acc (Times(r1, r2)) s k =
  acc r1 s (fn s' => acc r2 s' k)
| acc (One) s k = k s
| acc (Plus(r1, r2)) s k =
  acc r1 s k orelse acc r2 s k
| acc (Zero) s k = false
| acc (Star(r)) s k =
  (k s) orelse acc r s (fn s' => not(s = s')  andalso acc (Star(r)) s' k)
```

What is the initial continuation with which we should call `acc`? We stop when the remaining character string is empty, i.e. we have exhausted our input string. Hence, the initial continuation must test this, i.e. `(fn l => l = [])`.

```
(* accept : regexp * string -> bool *)
fun accept r s = acc r (String.explode s) (fn l => l = []) ;
```

One final remark: out top-level matcher `accept` takes in as input a regular expression describing the pattern `r` and a string `s` for which we must decide whether it is accepted by the pattern `r`. To easily process the string sequentially, we "explode" the string `s` into a list of characters.