Chapter 6

Further ML

In this chapter, we consolidate the previous examples by specifying the basic facilities of ML and the syntax of phrases more precisely, and then go on to treat some additional features such as recursive types. We might start by saying more about interaction with the system.

So far, we have just been typing phrases into ML's toplevel read-eval-print loop and observing the result. However this is not a good method for writing nontrivial programs. Typically, you should write the expressions and declarations in a file. To try things out as you go, they can be inserted in the ML window using 'cut and paste'. This operation can be performed using X-windows and similar systems, or in an editor like Emacs with multiple buffers. However, this becomes laborious and time-consuming for large programs. Instead, you can use ML's `include` function to read in the file directly. For example, if the file `myprog.ml` contains:

```
let pythag x y z =
  x * x + y * y = z * z;;

pythag 3 4 5;;

pythag 5 12 13;;

pythag 1 2 3;;
```

then the toplevel phrase `include "myprog.ml";;` results in:

```
#include "myprog.ml";;
pythag : int -> int -> int -> bool = <fun>
- : bool = true
- : bool = true
- : bool = false
- : unit = ()
```

That is, the ML system responds just as if the phrases had been entered at the top level. The final line is the result of evaluating the `include` expression itself.

In large programs, it is often helpful to include comments. In ML, these are written between the symbols `(*` and `*)`, e.g.

```
(* ------------------------------------------------------ *)
(* This function tests if (x,y,z) is a Pythagorean triple *)
(* ------------------------------------------------------ *)

let pythag x y z =
    x * x + y * y = z * z;;

(*comments*) pythag (*can*) 3 (*go*) 4 (*almost*) 5 (*anywhere*)
(* and (* can (* be (* nested *) quite *) arbitrarily *) *);;
```

6.1 Basic datatypes and operations

ML features several built-in primitive types. From these, composite types may be built using various type constructors. For the moment, we will only use the function space constructor `->` and the Cartesian product constructor `*`, but we will see in due course which others are provided, and how to define new types and type constructors. The primitive types that concern us now are:

- The type `unit`. This is a 1-element type, whose only element is written `()`. Obviously, something of type `unit` conveys no information, so it is commonly used as the return type of imperatively written 'functions' that perform a side-effect, such as `include` above. It is also a convenient argument where the only use of a function type is to delay evaluation.

- The type `bool`. This is a 2-element type of booleans (truth-values) whose elements are written `true` and `false`.

- The type `int`. This contains some finite subset of the positive and negative integers. Typically the permitted range is from -2^{30} (-1073741824) up to $2^{30}-1$ (1073741823).[1] The numerals are written in the usual way, optionally with a negation sign, e.g. `0`, `32`, `-25`.

- The type `string` contains strings (i.e. finite sequences) of characters. They are written and printed between double quotes, e.g. `"hello"`. In order to encode include special characters in strings, C-like escape sequences are used. For example, `\"` is the double quote itself, and `\n` is the newline character.

[1] This is even more limited than expected for machine arithmetic because a bit is stolen for the garbage collector. We will see later how to use an alternative type of integers with unlimited precision.

The above values like `()`, `false`, `7` and `"caml"` are all to be regarded as constants in the lambda calculus sense. There are other constants corresponding to *operations* on the basic types. Some of these may be written as infix operators, for the sake of familiarity. These have a notion of precedence so that expressions are grouped together as one would expect. For example, we write `x + y` rather than `+ x y` and `x < 2 * y + z` rather than `< x (+ (* 2 y) z)`. The logical operator `not` also has a special parsing status, in that the usual left-associativity rule is reversed for it: `not not p` means `not (not p)`. User-defined functions may be granted infix status via the `#infix` directive. For example, here is a definition of a function performing composition of functions:

```
#let successor x = x + 1;;
successor : int -> int = <fun>
#let o f g = fun x -> f(g x);;
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
#let add3 = o successor (o successor successor);;
add3 : int -> int = <fun>
#add3 0;;
- : int = 3
##infix "o";;
#let add3' = successor o successor o successor;;
add3' : int -> int = <fun>
#add3' 0;;
- : int = 3
```

It is not possible to specify the precedence of user-defined infixes, nor to make user-defined non-infix functions right-associative. Note that the implicit operation of 'function application' has a higher precedence than any binary operator, so `successor 1 * 2` parses as `(successor 1) * 2`. If it is desired to use a function with special status as an ordinary constant, simply precede it by `prefix`. For example:

```
#o successor successor;;
Toplevel input:
>o successor successor;;
>^
Syntax error.
#prefix o successor successor;;
- : int -> int = <fun>
#(prefix o) successor successor;;
- : int -> int = <fun>
```

With these questions of concrete syntax out of the way, let us present a systematic list of the operators on the basic types above. The unary operators are:

Operator	Type	Meaning
`-`	`int -> int`	Numeric negation
`not`	`bool -> bool`	Logical negation

and the binary operators, in approximately decreasing order of precedence, are:

Operator	Type	Meaning
`mod`	`int -> int -> int`	Modulus (remainder)
`*`	`int -> int -> int`	Multiplication
`/`	`int -> int -> int`	Truncating division
`+`	`int -> int -> int`	Addition
`-`	`int -> int -> int`	Subtraction
`^`	`string -> string -> string`	String concatenation
`=`	`'a -> 'a -> bool`	Equality
`<>`	`'a -> 'a -> bool`	Inequality
`<`	`'a -> 'a -> bool`	Less than
`<=`	`'a -> 'a -> bool`	Less than or equal
`>`	`'a -> 'a -> bool`	Greater than
`>=`	`'a -> 'a -> bool`	Greater than or equal
`&`	`bool -> bool -> bool`	Boolean 'and'
`or`	`bool -> bool -> bool`	Boolean 'or'

For example, `x > 0 & x < 1` is parsed as `& (> x 0) (< x 1)`. Note that all the comparisons, not just the equality relation, are polymorphic. They not only order integers in the expected way, and strings alphabetically, but all other primitive types and composite types in a fairly natural way. Once again, however, they are not in general allowed to be used on functions.

The two boolean operations `&` and `or` have their own special evaluation strategy, like the conditional expression. In fact, they can be regarded as synonyms for conditional expressions:

$$\begin{aligned} p \mathbin{\&} q &\triangleq \text{if } p \text{ then } q \text{ else false} \\ p \text{ or } q &\triangleq \text{if } p \text{ then true else } q \end{aligned}$$

Thus, the 'and' operation evaluates its first argument, and only if it is true, evaluates its second. Conversely, the 'or' operation evaluates its first argument, and only if it is false evaluates its second.

6.2 Syntax of ML phrases

Expressions in ML can be built up from constants and variables; any identifier that is not currently bound is treated as a variable. Declarations bind names to values of expressions, and declarations can occur locally inside expressions. Thus, the syntax classes of expressions and declarations are mutually recursive. We can represent this by the following BNF grammar.[2]

[2]We neglect many constructs that we won't be concerned with. A few will be introduced later. See the CAML manual for full details.

$$
\begin{array}{rcl}
\mathit{expression} & ::= & \mathit{variable} \\
 & | & \mathit{constant} \\
 & | & \mathit{expression}\ \mathit{expression} \\
 & | & \mathit{expression}\ \mathit{infix}\ \mathit{expression} \\
 & | & \text{not}\ \mathit{expression} \\
 & | & \text{if}\ \mathit{expression}\ \text{then}\ \mathit{expression}\ \text{else}\ \mathit{expression} \\
 & | & \text{fun}\ \mathit{pattern}\ \texttt{->}\ \mathit{expression} \\
 & | & (\mathit{expression}) \\
 & | & \mathit{declaration}\ \text{in}\ \mathit{expression} \\
\mathit{declaration} & ::= & \text{let}\ \mathit{let\_bindings} \\
 & | & \text{let rec}\ \mathit{let\_bindings} \\
\mathit{let\_bindings} & ::= & \mathit{let\_binding} \\
 & | & \mathit{let\_binding}\ \text{and}\ \mathit{let\_bindings} \\
\mathit{let\_binding} & ::= & \mathit{pattern} = \mathit{expression} \\
\mathit{pattern} & ::= & \mathit{variables} \\
\mathit{variables} & ::= & \mathit{variable} \\
 & | & \mathit{variable}\ \mathit{variables}
\end{array}
$$

The syntax class *pattern* will be expanded and explained more thoroughly later on. For the moment, all the cases we are concerned with are either just *variable* or *variable variable* $\cdots$ *variable*. In the first case we simply bind an expression to a name, while the second uses the special syntactic sugar for function declarations, where the arguments are written after the function name to the left of the equals sign. For example, the following is a valid declaration of a function `add4`, which can be used to add 4 to its argument:

```
#let add4 x =
  let y = successor x in
  let z = let w = successor y in
          successor w in
  successor z;;
add4 : int -> int = <fun>
#add4 1;;
- : int = 5
```

It is instructive to unravel this declaration according to the above grammar. A toplevel phrase, terminated by two successive semicolons, may be either an expression or a declaration.

6.3 Further examples

It is easy to define by recursion a function that takes a positive integer n and a function f and returns f^n, i.e. $f \circ \cdots \circ f$ (n times):

```
#let rec funpow n f x =
   if n = 0 then x
   else funpow (n - 1) f (f x);;
funpow : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

In fact, a little thought will show that the function `funpow` takes a machine integer n and returns the Church numeral corresponding to n. Since functions aren't printed, we can't actually look at the lambda expression representing a Church numeral:

```
#funpow 6;;
- : ('_a -> '_a) -> '_a -> '_a = <fun>
```

However it is straightforward to define an inverse function to `funpow` that takes a Church numeral back to a machine integer:

```
#let defrock n = n (fun x -> x + 1) 0;;
defrock : ((int -> int) -> int -> 'a) -> 'a = <fun>
#defrock(funpow 32);;
- : int = 32
```

We can try out some of the arithmetic operations on Church numerals:

```
#let add m n f x = m f (n f x);;
add : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c = <fun>
#let mul m n f x = m (n f) x;;
mul : ('a -> 'b -> 'c) -> ('d -> 'a) -> 'd -> 'b -> 'c = <fun>
#let exp m n f x = n m f x;;
exp : 'a -> ('a -> 'b -> 'c -> 'd) -> 'b -> 'c -> 'd = <fun>
#let test bop x y = defrock (bop (funpow x) (funpow y));;
test :
 ((('a -> 'a) -> 'a -> 'a) ->
  (('b -> 'b) -> 'b -> 'b) -> (int -> int) -> int -> 'c) ->
  int -> int -> 'c = <fun>
#test add 2 10;;
- : int = 12
#test mul 2 10;;
- : int = 20
#test exp 2 10;;
- : int = 1024
```

The above is not a very efficient way of performing arithmetic operations. ML does not have a function for exponentiation, but it is easy to define one by recursion:

```
#let rec exp x n =
  if n = 0 then 1
  else x * exp x (n - 1);;
exp : int -> int -> int = <fun>
```

However this performs n multiplications to calculate x^n. A more efficient way is to exploit the facts that $x^{2n} = (x^n)^2$ and $x^{2n+1} = x(x^n)^2$ as follows:

```
#let square x = x * x;;
square : int -> int = <fun>
#let rec exp x n =
  if n = 0 then 1
  else if n mod 2 = 0 then square(exp x (n / 2))
  else x * square(exp x (n / 2));;
exp : int -> int -> int = <fun>
#infix "exp";;
#2 exp 10;;
- : int = 1024
#2 exp 20;;
- : int = 1048576
```

Another classic operation on natural numbers is to find their greatest common divisor (highest common factor) using Euclid's algorithm:

```
#let rec gcd x y =
     if y = 0 then x else gcd y (x mod y);;
gcd : int -> int -> int = <fun>
#gcd 100 52;;
- : int = 4
#gcd 7 159;;
- : int = 1
#gcd 24 60;;
- : int = 12
```

We have used a notional recursion operator `Rec` to explain recursive definitions. We can actually define such a thing, and use it:

```
#let rec Rec f = f(fun x -> Rec f x);;
Rec : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
#let fact = Rec (fun f n -> if n = 0 then 1 else n * f(n - 1));;
fact : int -> int = <fun>
#fact 3;;
it : int = 6
```

Note, however, that the lambda was essential, otherwise the expression `Rec f` goes into an infinite recursion before it is even applied to its argument:

```
#let rec Rec f = f(Rec f);;
Rec : ('a -> 'a) -> 'a = <fun>
#let fact = Rec (fun f n -> if n = 0 then 1 else n * f(n - 1));;
Uncaught exception: Out_of_memory
```

6.4 Type definitions

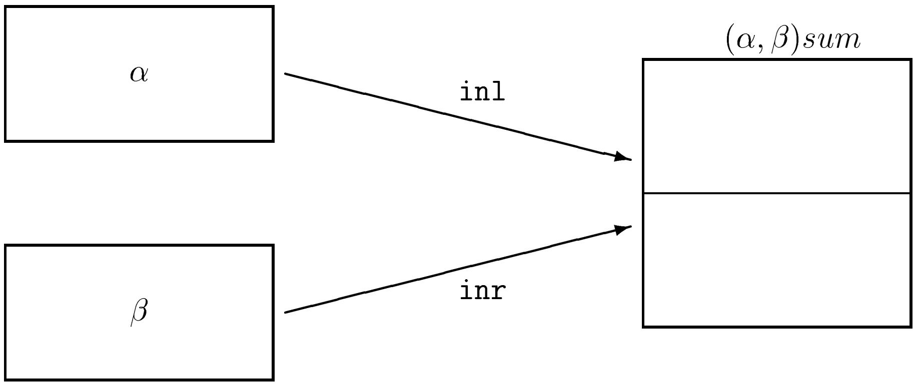
We have promised that ML has facilities for declaring new type constructors, so that composite types can be built up out of existing ones. In fact, ML goes further and allows a composite type to be built up not only out of preexisting types but also from the composite type itself. Such types, naturally enough, are said to be *recursive*. They are declared using the `type` keyword followed by an equation indicating how the new type is built up from existing ones and itself. We will illustrate this by a few examples. The first one is the definition of a *sum* type, intended to correspond to the disjoint union of two existing types.

```
#type ('a,'b)sum = inl of 'a | inr of 'b;;
Type sum defined.
```

Roughly, an object of type `('a,'b)sum` is *either* something of type `'a` *or* something of type `'b`. More formally, however, all these things have different types. The type declaration also declares the so-called *constructors* `inl` and `inr`. These are functions that take objects of the component types and inject them into the new type. Indeed, we can see their types in the ML system and apply them to objects:

```
#inl;;
- : 'a -> ('a, 'b) sum = <fun>
#inr;;
- : 'a -> ('b, 'a) sum = <fun>
#inl 5;;
- : (int, 'a) sum = inl 5
#inr false;;
- : ('a, bool) sum = inr false
```

We can visualize the situation via the following diagram. Given two existing types α and β, the type $(\alpha, \beta)sum$ is composed precisely of separate copies of α and β, and the two constructors map onto the respective copies:



This is similar to a `union` in C, but in ML the copies of the component types are kept apart and one always knows which of these an element of the union belongs to. By contrast, in C the component types are overlapped, and the programmer is responsible for this book-keeping.

6.4.1 Pattern matching

The constructors in such a definition have three very important properties:

- They are exhaustive, i.e. every element of the new type is obtainable either by `inl x` for some `x` or `inr y` for some `y`. That is, the new type contains nothing besides copies of the component types.
- They are injective, i.e. an equality test `inl x = inl y` is true if and only if `x = y`, and similarly for `inr`. That is, the new type contains a faithful copy of each component type without identifying any elements.
- They are distinct, i.e. their ranges are disjoint. More concretely this means in the above example that `inl x = inr y` is false whatever `x` and `y` might be. That is, the copy of each component type is kept apart in the new type.

The second and third properties of constructors justify our using *pattern matching*. This is done by using more general *varstructs* as the arguments in a lambda, e.g.

```
#fun (inl n) -> n > 6
   | (inr b) -> b;;
- : (int, bool) sum -> bool = <fun>
```

This function has the property, naturally enough, that when applied to `inl n` it returns `n > 6` and when applied to `inr b` it returns `b`. It is precisely because of the second and third properties of the constructors that we know this does give a welldefined function. Because the constructors are injective, we can uniquely recover `n` from `inl n` and `b` from `inr b`. Because the constructors are distinct, we know that the two clauses cannot be mutually inconsistent, since no value can correspond to both patterns.

In addition, because the constructors are exhaustive, we know that each value will fall under one pattern or the other, so the function is defined everywhere. Actually, it is permissible to relax this last property by omitting certain patterns, though the ML system then issues a warning:

```
#fun (inr b) -> b;;
Toplevel input:
>fun (inr b) -> b;;
>^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
- : ('a, 'b) sum -> 'b = <fun>
```

If this function is applied to something of the form `inl x`, then it will not work:

```
#let f = fun (inr b) -> b;;
Toplevel input:
>let f = fun (inr b) -> b;;
>        ^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
f : ('a, 'b) sum -> 'b = <fun>
#f (inl 3);;
Uncaught exception: Match_failure ("", 452, 468)
```

Though booleans are built into ML, they are effectively defined by a rather trivial instance of a recursive type, often called an *enumerated type*, where the constructors take no arguments:

```
#type bool = false | true;;
```

Indeed, it is perfectly permissible to define things by matching over the truth values. The following two phrases are completely equivalent:

```
#if 4 < 3 then 1 else 0;;
- : int = 0
#(fun true -> 1 | false -> 0) (4 < 3);;
- : int = 0
```

Pattern matching is, however, not limited to casewise definitions over elements of recursive types, though it is particularly convenient there. For example, we can define a function that tells us whether an integer is zero as follows:

```
#fun 0 -> true | n -> false;;
- : int -> bool = <fun>
#(fun 0 -> true | n -> false) 0;;
- : bool = true
#(fun 0 -> true | n -> false) 1;;
- : bool = false
```

In this case we no longer have mutual exclusivity of patterns, since `0` matches either pattern. The patterns are examined in order, one by one, and the first matching one is used. Note carefully that unless the matches are mutually exclusive, there is no guarantee that each clause holds as a mathematical equation. For example in the above, the function does not return `false` for any `n`, so the second clause is not universally valid.

Note that only *constructors* may be used in the above special way as components of patterns. Ordinary constants will be treated as new variables bound inside the pattern. For example, consider the following:

```
#let true_1 = true;;
true_1 : bool = true
#let false_1 = false;;
false_1 : bool = false
#(fun true_1 -> 1 | false_1 -> 0) (4 < 3);;
Toplevel input:
>(fun true_1 -> 1 | false_1 -> 0) (4 < 3);;
>                   ^^^^^^^
Warning: this matching case is unused.
- : int = 1
```

In general, the unit element `()`, the truth values, the integer numerals, the string constants and the pairing operation (infix comma) have constructor status, as well as other constructors from predefined recursive types. When they occur in a pattern the target value must correspond. All other identifiers match any expression and in the process become bound.

As well as the varstructs in lambda expressions, there are other ways of performing pattern matching. Instead of creating a function via pattern matching and applying it to an expression, one can perform pattern-matching over the expression directly using the following construction:

$$\text{match } \mathit{expression} \text{ with } \mathit{pattern}_1 \texttt{->} E_1 \mid \cdots \mid \mathit{pattern}_n \texttt{->} E_n$$

The simplest alternative of all is to use

$$\text{let } \mathit{pattern} = \mathit{expression}$$

but in this case only a single pattern is allowed.

6.4.2 Recursive types

The previous examples have all been recursive only vacuously, in that we have not defined a type in terms of itself. For a more interesting example, we will declare a type of lists (finite ordered sequences) of elements of type `'a`.

```
#type ('a)list = Nil | Cons of 'a * ('a)list;;
Type list defined.
```

Let us examine the types of the constructors:

```
#Nil;;
- : 'a list = Nil
#Cons;;
- : 'a * 'a list -> 'a list = <fun>
```

The constructor `Nil`, which takes no arguments, simply creates some object of type `('a)list` which is to be thought of as the empty list. The other constructor `Cons` takes an element of type `'a` and an element of the new type `('a)list` and gives another, which we think of as arising from the old list by adding one element to the front of it. For example, we can consider the following:

```
#Nil;;
- : 'a list = Nil
#Cons(1,Nil);;
- : int list = Cons (1, Nil)
#Cons(1,Cons(2,Nil));;
- : int list = Cons (1, Cons (2, Nil))
#Cons(1,Cons(2,Cons(3,Nil)));;
- : int list = Cons (1, Cons (2, Cons (3, Nil)))
```

Because the constructors are distinct and injective, it is easy to see that all these values, which we think of as lists $[]$, $[1]$, $[1;2]$ and $[1;2;3]$, are distinct. Indeed, purely from these properties of the constructors, it follows that arbitrarily long lists of elements may be encoded in the new type. Actually, ML already has a type `list` just like this one defined. The only difference is syntactic: the empty list is written `[]` and the recursive constructor `::`, has infix status. Thus, the above lists are actually written:

```
#[];;
- : 'a list = []
#1::[];;
- : int list = [1]
#1::2::[];;
- : int list = [1; 2]
#1::2::3::[];;
- : int list = [1; 2; 3]
```

The lists are printed in an even more natural notation, and this is also allowed for input. Nevertheless, when the exact expression in terms of constructors is needed, it must be remembered that this is only a surface syntax. For example, we can define functions to take the head and tail of a list, using pattern matching.

```
#let hd (h::t) = h;;
Toplevel input:
>let hd (h::t) = h;;
>    ^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
hd : 'a list -> 'a = <fun>
#let tl (h::t) = t;;
Toplevel input:
>let tl (h::t) = t;;
>    ^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
tl : 'a list -> 'a list = <fun>
```

The compiler warns us that these both fail when applied to the empty list, since there is no pattern to cover it (remember that the constructors are distinct). Let us see them in action:

```
#hd [1;2;3];;
- : int = 1
#tl [1;2;3];;
- : int list = [2; 3]
#hd [];;
Uncaught exception: Match_failure
```

Note that the following is not a correct definition of `hd`. In fact, it constrains the input list to have exactly two elements for matching to succeed, as can be seen by thinking of the version in terms of the constructors:

```
#let hd [x;y] = x;;
Toplevel input:
>let hd [x;y] = x;;
>    ^^^^^^^^^^^^
Warning: this matching is not exhaustive.
hd : 'a list -> 'a = <fun>
#hd [5;6];;
- : int = 5
#hd [5;6;7];;
Uncaught exception: Match_failure
```

Pattern matching can be combined with recursion. For example, here is a function to return the length of a list:

```
#let rec length =
   fun [] -> 0
     | (h::t) -> 1 + length t;;
length : 'a list -> int = <fun>
#length [];;
- : int = 0
#length [5;3;1];;
- : int = 3
```

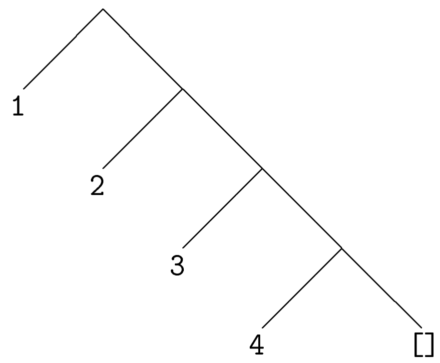
Alternatively, this can be written in terms of our earlier 'destructor' functions `hd` and `tl`:

```
#let rec length l =
   if l = [] then 0
   else 1 + length(tl l);;
```

This latter style of function definition is more usual in many languages, notably LISP, but the direct use of pattern matching is often more elegant.

6.4.3 Tree structures

It is often helpful to visualize the elements of recursive types as tree structures, with the recursive constructors at the branch nodes and the other datatypes at the leaves. The recursiveness merely says that plugging subtrees together gives another tree. In the case of lists the 'trees' are all rather spindly and one-sided, with the list `[1;2;3;4]` being represented as:



It is not difficult to define recursive types which allow more balanced trees, e.g.

```
#type ('a)btree = Leaf of 'a
                | Branch of ('a)btree * ('a)btree;;
```

In general, there can be several different recursive constructors, each with a different number of descendants. This gives a very natural way of representing the *syntax trees* of programming (and other formal) languages. For example, here is a type to represent arithmetical expressions built up from integers by addition and multiplication:

```
#type expression = Integer of int
                 | Sum of expression * expression
                 | Product of expression * expression;;
```

and here is a recursive function to evaluate such expressions:

```
#let rec eval =
   fun (Integer i) -> i
     | (Sum(e1,e2)) -> eval e1 + eval e2
     | (Product(e1,e2)) -> eval e1 * eval e2;;
eval : expression -> int = <fun>
#eval (Product(Sum(Integer 1,Integer 2),Integer 5));;
- : int = 15
```

Such abstract syntax trees are a useful representation which allows all sorts of manipulations. Often the first step programming language compilers and related tools take is to translate the input text into an 'abstract syntax tree' according to the parsing rules. Note that conventions such as precedences and bracketings are not needed once we have reached the level of abstract syntax; the tree structure makes these explicit. We will use such techniques to write ML versions of the formal rules of lambda calculus that we were considering earlier. First, the following type is used to represent lambda terms:

```
#type term = Var of string
           | Const of string
           | Comb of term * term
           | Abs of string * term;;
Type term defined.
```

Note that we make `Abs` take a variable name and a term rather than two terms in order to prohibit the formation of terms that aren't valid. For example, we represent the term $\lambda x\ y.\ y\ (x\ x\ y)$ by:

```
Abs("x",Abs("y",Comb(Var "y",Comb(Comb(Var "x",Var "x"),Var "y"))))
```

Here is the recursive definition of a function `free_in` to decide whether a variable is free in a term:

```
#let rec free_in x =
    fun (Var v) -> x = v
      | (Const c) -> false
      | (Comb(s,t)) -> free_in x s or free_in x t
      | (Abs(v,t)) -> not x = v & free_in x t;;
free_in : string -> term -> bool = <fun>
#free_in "x" (Comb(Var "f",Var "x"));;
- : bool = true
#free_in "x" (Abs("x",Comb(Var "x",Var "y")));;
- : bool = false
```

Similarly, we can define substitution by recursion. First, though, we need a function to rename variables to avoid name clashes. We want the ability to convert an existing variable name to a new one that is not free in some given expression. The renaming is done by adding prime characters.

```
#let rec variant x t =
   if free_in x t then variant (x^"'") t
   else x;;
variant : string -> term -> string = <fun>
#variant "x" (Comb(Var "f",Var "x"));;
- : string = "x'"
#variant "x" (Abs("x",Comb(Var "x",Var "y")));;
- : string = "x"
#variant "x" (Comb(Var "f",Comb(Var "x",Var "x'")));;
- : string = "x''"
```

Now we can define substitution just as we did in abstract terms:

```
#let rec subst u (t,x) =
   match u with
     Var y -> if x = y then t else Var y
   | Const c -> Const c
   | Comb(s1,s2) -> Comb(subst s1 (t,x),subst s2 (t,x))
   | Abs(y,s) -> if x = y then Abs(y,s)
                 else if free_in x s & free_in y t then
                   let z = variant y (Comb(s,t)) in
                   Abs(z,subst (subst s (Var y,z)) (t,x))
                 else Abs(y,subst s (t,x));;
subst : term -> term * string -> term = <fun>
```

Note the very close parallels between the standard mathematical presentation of lambda calculus, as we presented earlier, and the ML version here. All we really need to make the analogy complete is a means of reading and writing the terms in some more readable form, rather than by explicitly invoking the constructors. We will return to this issue later, and see how to add these features.

6.4.4 The subtlety of recursive types

A recursive type may contain nested instances of other type constructors, including the function space constructor. For example, consider the following:

```
#type ('a)embedding = K of ('a)embedding->'a;;
Type embedding defined.
```

If we stop to think about the underlying semantics, this looks disquieting. Consider for example the special case when `'a` is `bool`. We then have an injective function `K:((bool)embedding->bool)->(bool)embedding`. This directly contradicts Cantor's theorem that the set of all subsets of X cannot be injected into X.[3] Hence we need to be more careful with the semantics of types. In fact $\alpha \to \beta$ cannot be interpreted as the full function space, or recursive type constructions like the above are inconsistent. However, since all functions we can actually create are computable, it is reasonable to restrict ourselves to computable functions only. With that restriction, a consistent semantics is possible, although the details are complicated.

The above definition also has interesting consequences for the type system. For example, we can now define the Y combinator, by using K as a kind of type cast. Note that if the Ks are deleted, this is essentially the usual definition of

[3]Proof: consider $C = \{i(s) \mid s \in \wp(X) \text{ and } i(s) \notin s\}$. If $i : \wp(X) \to X$ is injective, we have $i(C) \in C \Leftrightarrow i(C) \notin C$, a contradiction. This is similar to the Russell paradox, and in fact probably inspired it. The analogy is even closer if we consider the equivalent form that there is no surjection $j : X \to \wp(X)$, and prove it by considering $\{s \mid s \notin j(s)\}$.

the Y combinator in untyped lambda calculus. The use of `let` is only used for the sake of efficiency, but we *do* need the η-redex involving `z` in order to prevent looping under ML's evaluation strategy.

```
#let Y h =
  let g (K x) z = h (x (K x)) z in
  g (K g);;
Y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
#let fact = Y (fun f n -> if n = 0 then 1 else n * f(n - 1));;
fact : int -> int = <fun>
#fact 6;;
- : int = 720
```

Thus, recursive types are a powerful addition to the language, and allow us to back off the change of making the recursion operator a primitive.

Exercises

1. What goes wrong in our definition of `subst` if we replace

   `Var y -> if x = y then t else Var y`

   by these separate patterns?

   `Var x -> t | Var y -> Var y`

2. It would be possible, though inefficient, to define natural numbers using the following recursive type:

   ```
   #type num = Zero | Suc of num;;
   ```

   Define functions to perform arithmetic on natural numbers in this form.

3. Use the type definition for the syntax of lambda terms that we gave above. Write a function to convert an arbitrary lambda term into an equivalent in terms of the S, K and I combinators, based on the technique we gave earlier. Represent the combinators by `Const "S"` etc.

4. Extend the syntax of lambda terms to a incorporate Church-style typing. Write functions that check the typing conditions before allowing terms to be formed, but otherwise map down to the primitive constructors.

5. Consider a type of binary trees with strings at the nodes defined by:

   ```
   #type stree = Leaf | Branch of stree * string * stree;;
   ```

Write a function `strings` to return a list of the strings occurring in the tree in left-to-right order, i.e. at each node it appends the list resulting from the left branch, the singleton list at the current node, and the list occurring at the right branch, in that order. Write a function that takes a tree where the corresponding list is alphabetically sorted and a new string, and returns a tree with the new string inserted in the appropriate place.

6. (*) Refine the above functions so that the tree remains almost balanced at each stage, i.e. the maximum depths occurring at the left and right branches differ by at most one.[4]

7. (*) Can you characterize the types of expressions that can be written down in simply typed lambda calculus, without a recursion operator?[5] What is the type of the following recursive function?

```
let rec f i = (i o f) (i o i);;
```

Show how it can be used to write down ML expressions with completely polymorphic type α. Can you write a terminating expression with this property? What about an arbitrary function type $\alpha \to \beta$?

[4]See Adel'son-Vel'skii and Landis (1962) and various algorithm books for more details of these 'AVL trees' and similar methods for balanced trees.

[5]The answer lies in the 'Curry-Howard isomorphism' — see Girard, Lafont, and Taylor (1989) for example.