PAGE 5

Also distributed electronically is the *Standard ML of New Jersey Manual* by A. W. Appel, D. B. MacQueen, et al., ATT Bell Laboratories, 1993. This and other material, including the *SML/NJ Library Manual* by E. Gansner and J. H. Reppy and the SML/NJ system itself, is available by anonymous ftp from directory `dist/ml` at host `research.att.com` or from directory `pub/ml` at host `princeton.edu`.

A. W. Appel and D. B. MacQueen, "Standard ML of New Jersey," *International Symposium on Programming Languages, Implementation, and Logic*, pp. 1–13, Springer-Verlag, 1991 is a technical article describing the SML/NJ system.

PAGE 15

This error message is rather mysterious. Obviously, it is saying something about finding a real (i.e., 4.0) when it expected an integer to match the value 3 that followed the `then`. But what's this about "rules"? The explanation lies in

PAGE 34

`fun` <identifier>`(`<parameter list>`) =` <expression>`;`

PAGE 35

$$val\ upper = fn : string \rightarrow string$$

PAGE 42

The form of recursion illustrated in Examples 5.6 and 5.7 is relatively simple. Each call either results in one recursive call with a smaller argument, or we reach the basis case and there is no need for a recursion. Now we shall examine a function where the recursion involves more than one recursive call.

PAGE 47

Another way we could have discovered that `m` and `n` are integers is to look at line (2). We see `m` compared with integer 0, so `m` must be an integer. We also see `n` compared with `m`, and since we already know `m` is an integer, we know the same about `n`. □

PAGE 48

5.4: Consider the factorial function `fact` described in Exercise 5.2(a); if you have not written this function, use the solution given at the end of the book. Describe the changes to the environment that occur as a result of a call to `fact(4)`.

PAGE 50

◇ **Example 6.2.** Let us reconsider the function `reverse` from Example 5.6. There are two patterns for the argument L. If L is empty it matches the pattern `nil`. If L is not empty, it will match the pattern `x::xs`. For instance, if the list has a single element, `x` becomes that element and `xs` gets the value `nil`. A nonempty list cannot match `nil`, and `x::xs` does not match the empty list, because there is no head element to give a value to `x`. (It is not possible to give `x` the value `nil` because `x` is an element, not a list). Thus, the following definition works.

PAGE 79

opens the file named `foo` in the directory in which the ML program is running. To read from the file subsequently, we refer to it by the identifier `infile`, not `foo`. The expression

PAGE 95

◇ **Example 10.4.** To explore further the effect of an `=` or `<>` comparison on the set of permissible types, let us reconsider the two versions of the function `reverse` that we developed in Examples 5.6 and 6.2. These are repeated in Fig. 10.4(a) and (b), respectively. However, here we have used the correct type variable name `''a` (with two quotes) that ML uses to describe the type of the function in Fig. 10.4(a). This type name tells us that any type can be used, provided it is an equality type.

PAGE 109

However, Example 11.8 is somehow unsatisfactory. It is true that we can apply the composition of any two functions to an argument, as long as the types match properly. Yet we cannot address the question of Example 11.7, "what function is the composition of functions $x + 3$ and $y^2 + 2y$?" Function `comp` as we defined it is relatively useless. It is a "shorthand" for $G\big(F(x)\big)$, but it fails to save us keystrokes.

PAGE 114

! **11.14**: Combine the ideas of Exercises 11.9 and 11.11. That is, your function should take a constant g, a function F, and a list $[a_1, \ldots, a_n]$. The desired result is what we get by starting with g and applying F repeatedly, grouping from the left:

PAGE 150

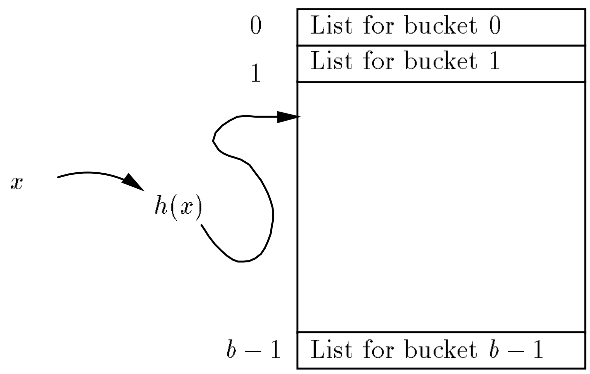
1. The signature can specify a more restrictive type for an identifier than is implicit in the structure definition. Example 14.4 illustrated this capability with the identifiers `create`, `insert`, and `lookup`.

PAGE 177

```
val A = array(n,v);
```

We must specify the type of v if it is not obvious from the value (e.g., if v is `nil`). Further, there are limits on the type that v may have, for technical reasons we shall not discuss.

PAGE 179



Bucket Headers

PAGE 195

In the val-declaration of Fig. 18.1, the identifier `Norm'sRecord` is assigned a value that is a record with three fields. The first field has label `ID` and value 123. The label of the second field is `name`, and the value of the second field is `"Norm dePlume"`. The third field has label `courses` and a value that is a list of three strings. The response repeats the value of the record and indicates its type.[1] ML has discovered from the values of the fields that `ID` is an integer, `name` a string, and `courses` a list of strings. □

PAGE 214

The composition of functions was introduced in Chapter 11, where we showed how is it possible to define a function `comp` that composes functions. Recall that if $F(x)$ is a function that takes an argument of type T_1 and produces a value of type T_2, while $G(y)$ is a function that takes an argument of type T_2 and produces a value of type T_3, then $G\big(F(x)\big)$ is the *composition* of F and G. We compute $G\big(F(x)\big)$ by applying F to an argument x of type T_1, producing a value y of type T_2. Then we apply G to y, and $G(y)$ is the value of type T_3 produced from x by the composition of G and F.

ML provides the lowercase "Oh" as a function composition operator, so it was not necessary for us to define function `comp` explicitly. The composition of F and G is expressed in ML by `G o F`. If F and G are respectively of types $T_1 \to T_2$ and $T_2 \to T_3$, then `G o F` is of type $T_1 \to T_3$. The expression `G o F` can be used anywhere an expression of this type is appropriate. For example, it could be bound as the value of a variable by a val-declaration, or passed as an argument to another function that has a parameter of type $T_1 \to T_3$.

PAGE 221

Note that it is not possible simply to apply `len` to 0, since ML's `fold` is defined so the function G it produces has the list as its first argument. Had the list been the second argument, then we could have defined `length` to be `len 0`. □

PAGE 235

```
fun comb([_,n,m],_) = (
        print(comb1(stringToInt(n), stringToInt(m)));
        print("\n")
    );
val comb = fn : string list * 'a -> unit
```

PAGE 246

(12) `printTree(t);`
((1,2),3)val it = () : unit

PAGE 285

```
fun thousandthPower(x:real) =
        let
            val x = x*x*x*x*x;
            val x = x*x*x*x*x;
            val x = x*x*x*x*x
        in
            x*x*x*x*x*x*x*x
        end;
```

Fig. S3. Solution to Exercise 7.1.

PAGE 296

```
signature SIMPLE = sig
    exception Missing;
    datatype 'label tree =
        Node of 'label * 'label tree list;
    val build : int * int tree list -> int tree;
    val subtree : int * int tree -> int tree
end;
```

PAGE 301

```
structure HashTable =
    MakeHash(StringHT);
```

Fig. S18. Solution to Exercise 16.2.

When we teach students ML in the CS109 course at Stanford, the material covered corresponds closely to what appears in Chapters 1–13. The book can be used as a supplement to a programming language concepts course, in which case Chapters 14 and 15 would surely be included, possibly along with others of the more advanced chapters.

Acknowledgments

I would like to thank Andrew Appel and David MacQueen, both of whom carefully critiqued two iterations of the manuscript. They are tied for the title of "world's greatest referee."

I value a number of important pointers on ML from John Mitchell. Also, Henry Bauer, Richard LeBlanc, Peter Robinson, and Jean Scholtz have my appreciation for their work as referees of the manuscript.

Errata in the first printing were found and pointed out to me by Martin Erwig, Jeffrey Oldham, David Richardson, and Daniel Yankelevich.

Alan Apt, the series editor was a pleasure to work with. The manuscript was ably edited by Sondra Chavez. Errors in the text are mine, however, not hers or the referees'.

Cover Art

Special thanks go to Luca Cardelli, who volunteered to create original art for this book. The result is on the cover.

On-line Access to Code

You can obtain copies of the major pieces of code in this book by anonymous ftp to host `ftp-cs.stanford.edu`. Login with user name `ftp` or `anonymous` and give your name and host as a password. You may then execute

```
cd emlp
```

where you will find the programs from this book.

J. D. U.
Stanford, CA

July, 1994