Mads Tofte, March 1, 1989
Laboratory for Foundations of Computer Science
Department of Computer Science
Edinburgh University

Four Lectures on Standard ML

The following notes give an overview of Standard ML with emphasis placed on the Modules part of the language.

The notes are, to the best of my knowledge, faithful to "The Definition of Standard ML, Version 2"[1], as regards syntax, semantics and terminology. They have been written so as to be independent of any particular implementation. The exercises in the first 3 lectures can be tackled without the use of a machine, although having access to an implementation will no doubt be beneficial. The project in Lecture 4 presupposes access to an implementation of the full language, including modules. (At present, the Edinburgh compiler does not fall into this category; the author used the New Jersey Standard ML compiler.)

Lecture 1 gives an introduction to ML aimed at the reader who is familiar with some programming language but does not know ML. Both the Core Language and the Modules are covered by way of example.

Lecture 2 discusses the use of ML modules in the development of large programs. A useful methodology for programming with functors, signatures and structures is presented.

Lecture 3 gives a fairly detailed account of the static semantics of ML modules, for those who really want to understand the crucial notions of sharing and signature matching.

Lecture 4 presents a one day project intended to give the student an opportunity of modifying a non-trivial piece of software using functors, signatures and structures.

[1] R. Harper, R. Milner and M. Tofte: "The Definition of Standard ML, Version 2", (ECS–LFCS–88–62) Labroatory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh.

1 ML at a Glance

Suppose we were to draw a map of the landscape of programming languages. Where would ML fit in? COBOL and ML could safely be put down far apart. The input/output facilities in COBOL operate on specific kinds of input/output devices, for instance allowing the programmer to declare index sequential files. ML just has the notion of STREAMS, a stream being a sequence of characters, much like streams in UNIX or text files in PASCAL. On the other hand, ML is extremely concise compared to the verbose COBOL and ML is much better suited for structuring data and algorithms than COBOL is.

ML is closer related to PASCAL. Like PASCAL, ML has data types and there is a type checker which checks the validity of programs before they are run. Both PASCAL and ML follow the tradition of ALGOL in that variables can have local scope which is determined statically from the source program. However, PASCAL and ML are radically different in how algorithms are expressed. In PASCAL, as in many other languages, a variable can be updated (using `:=`). Algorithms are often expressed as iterated sequences of statements (using `while` loops, for instance), where the effect of executing one statement is to change the underlying store. In ML, statements are replaced by EXPRESSIONS; the effect of evaluating an expression is to produce a value. Moreover, variables cannot be updated; REFERENCES are special values that can be updated, and as all other values they can be bound to identifiers, but only rarely are the values one binds to variables references. Iteration is expressed using recursive functions instead of loops. In ML, functions are values which can be passed as arguments to functions and returned as results from functions, and ML programmers do this all the time. ML is an example of a FUNCTIONAL language; PASCAL is an example of a PROCEDURAL language.

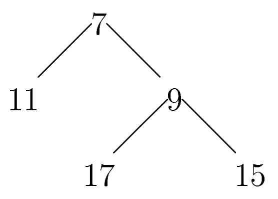
LISP is also sometimes referred to as a functional language. In LISP, programs can be treated as data, so that LISP programs directly can decompose and transform LISP programs. This is harder in ML. On the other hand, the type discipline of ML is extremely helpful in detecting many of the mistakes that pass unnoticed in a LISP program.

Like ADA, ML has language constructs for writing large programs. Roughly speaking, a STRUCTURE in ML corresponds to a PACKAGE in ADA; a SIGNATURE corresponds to a PACKAGE INTERFACE and a FUNCTOR in ML corresponds to a GENERIC PACKAGE in ADA. However, ML admits structures (not just types) as parameters to functors.

1.1 An ML session

An ML session is an interactive dialogue between the ML system and the user. You type a PROGRAM in the form of one or more DECLARATIONS (terminated by semicolon) and the system responds either by accepting the declarations or, in case the program is ill-formed, by printing an error message.

To give a concrete idea about what ML programs look like, we shall work through the following example. Consider the problem of implementing heaps. A HEAP is a binary tree of ITEMS, for example:



For a binary tree to be a heap, it must satisfy that for every item i in the tree, i is less than or equal to all items occurring below i. In the above picture items are integers and the relation "less than or equal" is the normal $\leq$ on integers. The advantage of a heap is that it always gives fast access to a minimal item and that it is easy to insert and delete items from a heap. This has made the heap a popular data structure in a number of very different applications. It was originally conceived under the name "priority queue" as a means of scheduling processes in an operating system; in that case the items are processes and the partial ordering is that process p is less than or equal to process q, if p should be executed no later than q. Heaps are also used in the HEAP SORT algorithm, which is based on the observation that one can sort a list of items by first inserting the items one by one in a heap and then removing them one by one.

1.2 Types and Values

In the following figures we present the ML declarations the author provided in this particular session. The responses from the ML compiler are not shown. For clarity, the actual input has been edited using typewriter font for the reserved words and italics for identifiers, regardless of whether these identifiers are pervasives (e.g. *int*) or declared by the user (e.g. *item*).

```
type item = int;

fun leq(p: item, q: item): bool =
    p <=q;

infix leq;
fun max(p, q) = if p leq q then q else p
and min(p, q) = if p leq q then p else q

datatype tree = L of item
              | N of item * tree * tree;

val t = N(7, L 11, N(9, L 17, L 15));

fun top(L i) = i
|   top(N(i, _, _)) = i;
```

We start out by considering integer heaps only; therefore we first declare the type *item* to be an abbreviation for *int*. Then we declare a function *leq* to be the pervasive `<=` on integers. We then declare that *leq* is to be used as an infix operator, as illustrated in the declaration of the two functions *max* and *min*.

Every binary tree is either a leaf containing an item or it is a node containing an item and two trees (the subtrees). This is expressed by the `datatype` declaration. `datatype` declarations are automatically recursive, i.e. data types can be declared in terms of themselves. This is illustrated by the declaration of *tree*. This data type has two CONSTRUCTORS, L and N. Note that for example `7` is an item, but L APPLIED TO `7`, written L`(7)`, or just L `7`, is of type *tree*. Then the heap from the earlier picture is bound to the value variable t.

Exercise 1 Declare a heap *t'* of the same depth as t containing the integers 78, 34, 5, 12, 15, 28, and 9.

To define a function on trees it will suffice to define its value in the case the argument tree is a node and in the case the tree is a node. The declaration of the function *top* illustrates this. (*top* applied to a tree returns the item at the top of the tree). $(L\ i)$ and $(N(i, \_, \_))$ are examples of PATTERNS. Applying a function (here *top*) to an argument (e.g. t) is done by matching the argument against the patterns till a matching pattern is found. For example *top t* evaluates to 7.

1.3 Recursive Functions

```
fun depth(L _) = 1
|   depth(N(i, l, r)) =
        1 + max(depth l, depth r);

depth t;

fun isHeap(L _):bool = true
|   isHeap(N(i, l, r)) =
        i leq top l andalso
        i leq top r andalso
        isHeap l andalso
        isHeap r
```

The function *depth* maps trees to integers; for instance *depth t* evaluates to 3. As spelled out in the declaration of *depth*, the depth of a leaf is 1 and the depth of any other tree is 1 plus the maximum of the depths of the left and right subtrees. The function *depth* is RECURSIVE, i.e. defined in terms of itself. Another example of a recursive function is the function *isHeap* which when applied to a tree returns the value true if the tree is a heap and false otherwise.

Exercise 2 Write a function *size* which when applied to a tree returns the total number of items in the tree.

Exercise 3 The function *top* returns a minimal item of a heap. Write a recursive function *maxItem* which returns a maximal item.

1.4 Raising Exceptions

One often wants to define a function that cannot return a result for some of its argument values. Suppose, for example, that we wish to define a function *initHeap* which for given integer n returns a heap of depth n. This only makes sense for $n \geq 1$. This can be expressed in ML by raising an EXCEPTION in the case $n < 1$. The effect of evaluating the expression `raise` e, where e is an exception, is to discontinue the current evaluation. Often, the exception will be HANDLED by a `handle` expression (not illustrated by our examples); if no handler catches the exception, it propagates to the top-level where it will be reported as an uncaught exception.

```
val initial = 0
exception InitHeap
fun initHeap n =
    if n<1 then raise InitHeap
    else if n = 1 then L(initial)
    else let val t = initHeap(n - 1)
         in  N(initial, t, t)
         end
```

Notice the `let` *dec* `in` *exp* `end` expression. To evaluate it, one first evaluates *initHeap*(n - 1) and binds the resulting value to t. Then one evaluates the body, $N(initial, t, t)$ using this value for t. Notice that the scope of the declaration of t is the expression

N(*initial*, *t*, *t*); in particular the two occurrences of t in that expression do not refer to N(7, L 11, N(9, L 17, L 15)).

Exercise 4 Define functions *leftSub* and *rightSub* which when applied to a tree returns the left and the right subtree, respectively.

Finally, we shall write a function *replace* which when applied to a pair (i, h), where i is an item and h is a heap, returns a pair (i', h'), where i' is the item at the top of the heap h and h' is a heap obtained from h by inserting i in place of the top of h. We must make sure that the resulting tree really is a heap. Therefore, in the case that i is to be inserted in a node above a subtree with a smaller item, i swops place with the smaller item.

```
fun replace(i, h) = (top h, insert(i, h))
and insert(i, L _) = L(i)
|    insert(i, N(_, l, r))=
          if i leq min(top l, top r)
          then N(i, l, r)
          else if (top l) leq (top r) then
               N(top l, insert(i, l), r)
          else (* top r < min(i, top l) *)
               N(top r, l, insert(i, r));

val (out1, t1) = replace(10, t);

t;

val (out2, t2) = replace(20, t1)
```

The special parenthesis (* and *) delimit comments.

Exercise 5 In the case where one recursively inserts i in the left subtree, how can one be sure that it is valid to put *top l* above r in the tree?

If one types an expression followed by a semicolon (such as t; in the above program) the ML system evaluates the expression and prints the result. In the above example, it will turn out that even after we have "replaced" 7 by 10, t is bound to the original heap. Indeed, this "replacement" in no way affects the value bound to t; it simply results in a new value, which subsequently is bound to *t1*.

Exercise 6 After the last declaration, what values are bound to *out2* and *t2*?

1.5 Structures

The above declarations of heaps and operations on heaps belong together. In ML there is a program unit called a STRUCTURE which encapsulates a sequence of declarations. The following declaration declares a structure *Heap* containing all the declarations (copied from above) encapsulated by `struct` and `end`.