ML—Meta Language

SML is Standard ML, a popular ML variant.
ML is a functional language that is designed to be efficient and type-safe. It demonstrates that a functional language need not use Scheme's odd syntax and need not bear the overhead of dynamic typing.
SML's features and innovations include:
1. Strong, compile-time typing.
2. Automatic type inference rather than user-supplied type declarations.
3. Polymorphism, including “type variables.”

4. Pattern-directed Programming
   fun len([]) = 0 
   | len(a::b) = 1+len(b);
5. Exceptions
6. First-class functions
7. Abstract Data Types
   coin of int | bill of int | check of string*real;
   val dime = coin(10);

A good ML reference is
“Elements of ML Programming,”
by Jeffrey Ullman
(Prentice Hall, 1998)

SML is Interactive

You enter a definition or expression, and SML returns a result with an inferred type.
The command
   use "file name";
loads a set of ML definitions from a file.
For example (SML responses are in blue):
   21;
   val it = 21 : int
   (2 div 3);
   val it = 0 : int
   true;
   val it = true : bool
   "xyz";
   val it = "xyz" : string
Basic SML Predefined Types

• Unit
  Its only value is (). Type unit is similar to void in C; it is used where a type is needed, but no "real" type is appropriate. For example, a call to a write function may return unit as its result.

• Integer
  Constants are sequences of digits. Negative values are prefixed with a ~ rather than a - (is a binary subtraction operator). For example, ~123 is negative 123. Standard operators include + - * div mod < > <= >= = <>

For example, real(3) returns 3.0, floor(3.1) returns 3, ceiling(3.3) returns 4, round(~3.6) returns ~4, trunc(3.9) returns 3.

Mixed mode expressions, like 1 + 2.5 aren't allowed; you must do explicit conversion, like real(1) + 2.5

• Strings
  Strings are delimited by double quotes. Newlines are \n, tabs are \t, and \" and \\ escape double quotes and backslashes. E.g. "Bye now\n". The ^ operator is concatenation.
  "abc" ^ "def" = "abcdef"
  The usual relational operators are provided: < > <= >= = <>

• Real
  Both fractional (123.456) and exponent forms (10e7) are allowed. Negative signs and exponents use ~ rather than -(~10.0e~12).
  Standard operators include + - * / < > <= >=
  Note that = and <> aren't allowed! (Why?)
  Conversion routines include real(int) to convert an int to a real, floor(real) to take the floor of a real, ceil(real) to take the ceiling of a real.
  round(real) to round a real, trunc(real) to truncate a real.

• Characters
  Single characters are delimited by double quotes and prefixed by a #. For example, "#a" or "#\t". A character is not a string of length one. The str function may be used to convert a character into a string. Thus str(#"a") = "a"

• Boolean
  Constants are true and false. Operators include andalso (short-circuit and), orelse (short-circuit or), not, = and <>.
  A conditional expression, (if boolval v1 else v2) is available.
**Tuples**

A tuple type, composed of two or more values of any type is available.
Tuples are delimited by parentheses, and values are separated by commas.
Examples include:

```
(1,2);
val it = (1,2) : int * int
("xyz",1=2);
val it = ("xyz",false) : string * bool
(1,3.0,false);
val it = (1,3.0,false) : int * real * bool
(1,2,(3,4));
val it = (1,2,(3,4)) : int * int * (int * int)
```

Equality is checked componentwise:

```
(1,2) = (0+1,1+1);
val it = true : bool
(1,2,3) = (1,2) causes a compile-time type error (tuples must be of the same length and have corresponding types to be compared).
```

#i selects the i-th component of a tuple (counting from 1). Hence

```
#2(1,2,3);
val it = 2 : int
```

**Lists**

Lists are required to have a single element type for all their elements; their length is unbounded.
Lists are delimited by [ and ] and elements are separated by commas.
Thus [1,2,3] is an integer list.
The empty (or null) list is [] or nil.
The cons operator is ::
Hence [1,2,3] ≡ 1::2::3::[]
Lists are automatically typed by ML:

```
[1,2];
val it = [1,2] : int list
```

**Cons**

Cons is an infix operator represented as ::
The left operand of :: is any value of type T
The right operand of :: is any list of type T list.
The result of :: is a list of type T list.
Hence :: is polymorphic.
[] is the empty list. It has a type 'a list. The symbol 'a, read as “alpha” or “tic a” is a type variable.
Thus [] is a polymorphic constant.
**List Equality**

Two lists may be compared for equality if they are of the same type. Lists \( L_1 \) and \( L_2 \) are considered equal if:

1. They have the same number of elements
2. Corresponding members of the two lists are equal.

**List Operators**

- \( \text{hd} \) \( \equiv \) head of list operator \( \equiv \text{car} \)
- \( \text{tl} \) \( \equiv \) tail of list operator \( \equiv \text{cdr} \)
- \( \text{null} \) \( \equiv \) null list predicate \( \equiv \text{null?} \)
- \( @ \) \( \equiv \) infix list append operator \( \equiv \text{append} \)

**Records**

Their general form is

\[
\{ \text{name}_1=\text{val}_1, \text{name}_2=\text{val}_2, \ldots \}
\]

Field selector names are local to a record.

For example:

\[
\{a=1,b=2\};
\]

\[
\text{val it} = \{a=1,b=2\} :
\{a: \text{int}, b: \text{int}\}
\]

\[
\{a=1,b="xyz"\};
\]

\[
\text{val it} = \{a=1,b="xyz"\} :
\{a: \text{int}, b: \text{string}\}
\]

\[
\{a=1.0,b=[c=[1,2]]\};
\]

\[
\text{val it} = \{a=1.0,b=[c=[1,2]]\} :
\{a: \text{real}, b: [c: \text{int list}]\}
\]

The order of fields is irrelevant; equality is tested using field names.

\[
\{a=1,b=2\}=(b=2,a=2-1);
\]

\[
\text{val it} = \text{true} : \text{bool}
\]

\#id extracts the field named id from a record.

\[
\#b \{a=1,b=2\} ;
\]

\[
\text{val it} = 2 : \text{int}
\]

**Identifiers**

There are two forms:

- Alphanumeric (excluding reserved words)
  
  Any sequence of letters, digits, single quotes and underscores; must begin with a letter or single quote.
  
  Case is significant. Identifiers that begin with a single quote are type variables.
  
  Examples include:
  
  \[
  \text{abc a10 'polar sum_of_20}
  \]

- Symbolic
  
  Any sequence (except predefined operators) of
  
  ! % & + - / ; < = > ? @ \ ^ #
  
  Usually used for user-defined operators.
  
  Examples include: ++ <=> !=
Comments

Of form

(* text *)

May cross line boundaries.

Declaration of Values

The basic form is

val id = expression;

This defines id to be bound to expression; ML answers with the name and value defined and the inferred type.

For example

val x = 10*10;
val x = 100 : int

Redefinition of an identifier is OK, but this is redefinition not assignment;

Thus

val x = 100;
val x = (x=100);

is fine; there is no type error even though the first x is an integer and then it is a boolean.

val x = 100 : int
val x = true : bool

Examples

val x = 1;
val x = 1 : int
val z = (x,x,x);
val z = (1,1,1) : int * int * int
val L = [z,z];
val L = [(1,1,1),(1,1,1)] : (int * int * int) list
val r = {a=L};
val r = {a:((1,1,1),(1,1,1))} : {a:(int * int * int) list}

After rebinding, the “nearest” (most recent) binding is used.

The and symbol (not boolean and) is used for simultaneous binding:

val x = 10;
val x = 10 : int
val x = true and y = x;
val x = true : bool
val y = 10 : int

Local definitions are temporary value definitions:

local
    val x = 10
in
    val u = x*x;
end;
val u = 100 : int

Let bindings are used in expressions:

let
    val x = 10
in
    5*x
end;
val it = 50 : int
Patterns

Scheme (and most other languages) use access or decomposition functions to access the components of a structured object. Thus we might write

\[
\text{let ( (h (car L) (t (cdr L))) ) body }
\]

Here \texttt{car} and \texttt{cdr} are used as access functions to locate the parts of \texttt{L} we want to access.

In ML we can access components of lists (or tuples, or records) directly by using patterns. The context in which the identifier appears tells us the part of the structure it references.

\[
\begin{align*}
\text{val x} & = (1,2); \\
\text{val x} & = (1,2) : \text{int * int} \\
\text{val (h,t)} & = x; \\
\text{val h} & = 1 : \text{int} \\
\text{val t} & = 2 : \text{int} \\
\text{val L} & = [1,2,3]; \\
\text{val L} & = [1,2,3] : \text{int list} \\
\text{val [v1,v2,v3]} & = L; \\
\text{val v1} & = 1 : \text{int} \\
\text{val v2} & = 2 : \text{int} \\
\text{val v3} & = 3 : \text{int} \\
\text{val [1,x,3]} & = L; \\
\text{val x} & = 2 : \text{int} \\
\text{val [1,rest]} & = L; \\
\text{val yy::rest} & = L; \\
\text{val yy} & = 2 : \text{int} \\
\text{val rest} & = [2,3] : \text{int list}
\end{align*}
\]

Wildcards

An underscore (_\texttt{)} may be used as a “wildcard” or “don’t care” symbol. It matches part of a structure without defining a new binding.

\[
\begin{align*}
\text{val zz::_} & = L; \\
\text{val zz} & = 1 : \text{int}
\end{align*}
\]

Pattern matching works in records too.

\[
\begin{align*}
\text{val r} & = \{a=1,b=2\}; \\
\text{val r} & = \{a=1,b=2\} : \{\text{a:int, b:int}\} \\
\text{val \{a=va,b=vb\}} & = r; \\
\text{val va} & = 1 : \text{int} \\
\text{val vb} & = 2 : \text{int} \\
\text{val \{a=va,b=_\}=r;} \\
\text{val wa} & = 1 : \text{int} \\
\text{val \{a=za, ...\}=r;} \\
\text{val za} & = 1 : \text{int}
\end{align*}
\]

Patterns can be nested too.

\[
\begin{align*}
\text{val x} & = ((1,3.0),5); \\
\text{val x} & = ((1,3.0),5) : (\text{int * real}) * \text{int} \\
\text{val ((1,y),_)} & = x; \\
\text{val y} & = 3.0 : \text{real}
\end{align*}
\]