

# 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

<code>+</code>	<code>-</code>	<code>*</code>	<code>div</code>	<code>mod</code>	
<code>&lt;</code>	<code>&gt;</code>	<code>&lt;=</code>	<code>&gt;=</code>	<code>=</code>	<code>&lt;&gt;</code>

- Real

Both fractional (`123.456`) and exponent forms (`10e7`) are allowed. Negative signs and exponents use `~` rather than `-` (`~10.0e~12`).

Standard operators include

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>
<code>&lt;</code>	<code>&gt;</code>	<code>&lt;=</code>	<code>&gt;=</code>

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`.

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 relation operators are provided: `<` `>` `<=` `>=` `=` `<>`

- 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 `l1` and `l2` are considered equal if:

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

# List Operators

`hd`  $\equiv$  head of list operator  $\approx$  `car`

`tl`  $\equiv$  tail of list operator  $\approx$  `cdr`

`null`  $\equiv$  null list predicate  $\approx$  `null?`

`@`  $\equiv$  infix list append operator  $\approx$  `append`

# Records

Their general form is

`{name1=val1, name2=val2, ... }`

Field selector names are local to a record.

For example:

`{a=1,b=2};`

`val it = {a=1,b=2} :  
 {a:int, b:int}`

`{a=1,b="xyz"};`

`val it = {a=1,b="xyz"} :  
 {a:int, b:string}`

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

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

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

```
{a=1,b=2}={b=2,a=2-1};
```

```
val it = true : bool
```

#id extracts the field named id from  
a record.

```
#b {a=1,b=2} ;
```

```
val it = 2 : 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:

`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
```