• 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 and also (short-circuit and), or else (short-circuit or), not, = and <>.
  A conditional expression, (if boolval then v₁ else v₂) is available.

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

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)

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` ≡ head of list operator ≈ `car`
- `tl` ≡ tail of list operator ≈ `cdr`
- `null` ≡ null list predicate ≈ `null?`
- `@` ≡ infix list append operator ≈ `append`

---

**Records**

Their general form is

```
{name_1=val_1, name_2=val_2, ... }
```

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:

```ml
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:

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

Let bindings are used in expressions:

```ml
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

```ml
(let ( (h (car L) (t (cdr L)) )
      body )
```

Here car and cdr are used as access functions to locate the parts of 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.

```ml
val x = (1,2);
val x = (1,2) : int * int
val (h,t) = x;
val h = 1 : int
val t = 2 : int
val L = [1,2,3];
val L = [1,2,3] : int list
val [v1,v2,v3] = L;
val v1 = 1 : int
val v2 = 2 : int
val v3 = 3 : int
val [1,x,3] = L;
val x = 2 : int
val [1,rest] = L;
(* This is illegal. Why? *)
val yy::rest = L;
val yy = 1 : int
val rest = [2,3] : int list
```
Wildcards

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

```ocaml
val zz::_ = L;
val zz = 1 : int
```

Pattern matching works in records too.

```ocaml
val r = {a=1, b=2};
val r = {a=1, b=2} : {a:int, b:int}
val {a=va, b=vb} = r;
val va = 1 : int
val vb = 2 : int
val {a=wa, _} = r;
val wa = 1 : int
val za = 1 : int
```

Patterns can be nested too.

```ocaml
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

Functions

Functions take a single argument (which can be a tuple).

Function calls are of the form

```
function_name argument;
```

For example

```ocaml
size "xyz";
cos 3.14159;
```

The more conventional form

```ocaml
size("xyz"); or cos(3.14159);
```

is OK (the parentheses around the argument are allowed, but unnecessary).

The form (size "xyz") or (cos 3.14159)

is OK too.

```
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

Patterns can be nested too.

```ocaml
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

```
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

```
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

```
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

```
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

```
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

```
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

```
val x = ((1, 3.0), 5);
val x = ((1, 3.0), 5) : (int * real) * int
val ((1, y), _) = x;
val y = 3.0 : real
```

Note that the call

```
plus(1, 2);
```

passes one argument, the tuple (1, 2) to plus.

The call `dummy();`

passes one argument, the unit value, to dummy.

All parameters are passed by value.
Function Types

The type of a function in ML is denoted as $T_1 \rightarrow T_2$. This says that a parameter of type $T_1$ is mapped to a result of type $T_2$.

The symbol $fn$ denotes a value that is a function.

Thus

```ml
size;
val it = fn : string -> int
not;
val it = fn : bool -> bool
Math.cos;
val it = fn : real -> real
```

(Math is an ML structure—an external library member that contains separately compiled definitions).

User-Defined Functions

The general form is

```ml
fun name arg = expression;
```

ML answers back with the name defined, the fact that it is a function (the $fn$ symbol) and its inferred type.

For example,

```ml
fun twice x = 2*x;
val twice = fn : int -> int
fun twotimes(x) = 2*x;
val twotimes = fn : int -> int
fun fact n = 
  if n=0
  then 1
  else n*fact(n-1);
val fact = fn : int -> int
```

fun plus(x,y):int = x+y;
val plus = fn : int * int -> int

The :int suffix is a type constraint.

It is needed to help ML decide that + is integer plus rather than real plus.