A session with ML

val it = 14 : int
2 + 3 * 4 : int

(After your input with a semicolon)

The prompt is the - sign, and it changes to = this step of the top-level loop; unit is the type.

This tells us how to interpret ML's output: it stands for the value of the expression evaluated in the this step of the top-level loop; unit is the type.

val it = ( ) : unit

Although ML stands for Meta Language, we are

(DEFINING FUNCTIONS)

(Generics)

(Tuples versus records)

(LOCALITY)

(USER-DEFINED DATA TYPES)

(REDUCE)

(FUNCTION COMPOSITION)

(HIGHER-ORDER FUNCTIONS)

(PRECEDENCE OF OPERATIONS)

(TYPING)

(TYPING INFERENCES)

(DEFINING FUNCTIONS)

(POINTS)

CS13125, Functional Programming. Page 113
A function definition. Note that the value of `succ` is the functional expression `fn:int->int`.

```ml
- fun succ x = x + 1;
```

And an application of this function:

```ml
- 3 * succ 4 * succ 5;
```

ML is great with lists (as expected!).

```ml
- fun length( x ) = if null( x ) then 0 else 1 + length( tl( x ) );
```

The type of list elements has not been determined. ML leaves it open, as indicated by the type "placeholder" `'

```ml
val length = fn : 'a list -> int
```

A small test:

```ml
- length( nil );
```

```
val it = 0 : int
```

Yet another form, without parentheses:

```ml
- fun length nil = 0 |
  length ( a::x ) = 1 + length x;
```

A small test:

```ml
- length ( ["a", "bb", "ccc"] );
```

```
val it = 3 : int
```

The same function could be defined as a series of patterns (rather like in Prolog):

```ml
- fun length nil = 0 |
  length ( a::x ) = 1 + length x;
```

A small test:

```ml
- length ( ["a", "bb", "ccc"] );
```

```
val it = 3 : int
```

ML is great with lists (as expected!).

```ml
val it = 90 : int
```

A function definition. Note that the value of `succ` is the functional expression `fn:int->int`.

```ml
- fun succ x = x + 1;
```

And an application of this function:

```ml
- func succ = fn : int -> int
```

```
val it = 1 : int
```

A function definition. Note that the value of `succ` is the functional expression `fn:int->int`.

```ml
- fun succ x = x + 1;
```

And an application of this function:

```ml
- func succ = fn : int -> int
```

```
val it = 1 : int
```
No presentation would be complete without this:

```ml
fun append( x, z ) =  
  if null( x ) then z else  
  hd(x) :: append( tl(x), z );
val append = fn : 'a list * 'a list -> 'a list
(The :: denotes list construction, the same as cons in Scheme.) The arguments are lists of "things", as is the value of the function. The * denotes the cross-product. This will work—there are only strings on the lists.

And a definition with patterns:

```ml
fun append( nil, z ) = z | append( a::y, z ) = a :: append( y, z );
val append = fn : 'a list * 'a list -> 'a list
```

A simple application:

```ml
append( [1, 2, 3, 4], [5, 6] );
val it = [1,2,3,4,5,6] : int list
```

Another application:

```ml
append( ["a", "b"], [3] );
std_in:1.1-1.25 Error: operator and operand don't agree (tycon mismatch)
operator domain: string list * string list
operand:         string list * int list
in expression:
append ("a" :: "b" :: nil,3 :: nil)
That's right: ML requires type agreement! This will work—there are only strings on the lists.
```
By the way, string concatenation is available too:

```
"abcd" ^ "efghijk";
```

val it = "abcdefghijk" : string

More function definitions... This reverses the first list and tucks it onto the second list:

```
fun reverse( nil, z ) = z |
reverse( a::y, z ) = reverse( y, a::z );
```

val reverse = fn :

Will it work?...

```
reverse( [1, 2, 3], [4] );
```

val it = [3, 2, 1, 4] : int list

Whew. Now, how do we reverse a list?

```
fun rev x = reverse( x, nil );
```

val rev = fn :

Does this work?...

```
rev( [1, 2, 3] );
```

val it = [3, 2, 1] : int list

We already did this in Scheme:

```
fun same_neighbours L =
  if null L then false else
  if null (tl L) then false else
  if hd L = hd (tl L) then true
  else same_neighbours (tl L);
```

val same_neighbours =

fun same_neighbours nil = false |
same_neighbours (a::nil) = false |
same_neighbours (a::b::L) =
  if a = b then
    true
  else
    same_neighbours (b::L);
Type inference in ML is very elaborate, and quite powerful. First, what happens when operand types are not specified? Here ML notices that 1 is an integer:

```ml
fun succ x = x + 1;
val succ = fn : int -> int
```

Here, 1.0 is a real number:

```ml
fun succr x = x + 1.0;
val succr = fn : real -> real
```

However, there is nothing to help ML:

```ml
fun sq x = x * x;
```

Here, however, there is nothing to help ML:

```ml
val sq = fn : int -> int
```

A hint is necessary—just one hint will be enough:

```ml
fun sq x: int = x * x;
val sq = fn : int -> int
```

Or any of these:

```
- fun sq x = x * x : int;
val sq = fn : int -> int
- fun sq x =(x: int) * x;
val sq = fn : int -> int
```

Precedence of operations in ML—one example...

```ml
val sq = fn : ( int * ( int * int ) ) => int
```

Try parentheses to evaluate : before length.

```
val tl = 2 : int
val tl = 1 : int
```

Here, however, there is nothing to help ML:

```
val sq = fn : ( real * real ) => real
```

Here, 1.0 is a real number:

```
val sq = fn : ( int * ( int * int ) ) => int
```

Inference:

are not specified? Here ML notices that 1 is an integer in ML is very elaborate, and quite powerful. First, what happens when operand types are not specified? Here ML notices that 1 is an integer:

```ml
fun succ x = x + 1;
val succ = fn : int -> int
```

Here, 1.0 is a real number:

```ml
fun succr x = x + 1.0;
val succr = fn : real -> real
```
Higher-order functions are very similar to the same functions in Scheme. First, the built-in `map`.

```
val it = [1, 9, 25] : int list

val it = [1, 4, 9] : int list
```

Here's how this form is defined:

```
fun map f nil = nil |
map f ( a::y ) = ( f a ) :: map f y;
```

The interpretation of this functional value is a little complicated: `map f` is a function from `'a list` to `'b list`.

Here's an example:

```
fun add x y: int = x + y;
val add = fn : int -> int -> int

val succ2 = add 2;
val succ2 = fn : int -> int

val it = 9 : int
```

Similarly, `map sq` is a function from `int list` to `int list`, and `map length` is a function from `a list list` to `int list`.

```
val squarelist = map sq;
val squarelist = fn : int list -> int list

val it = [25, 49, 121] : int list

val lengths = map length;
val lengths = fn : 'a list list -> int list

val it = [1, 2, 3] : int list
```

Let's explore this situation on a simpler example:

```
fun a_list_to_just_int : int list -> int
  val it = 9 : int
```

(Observe the parentheses-free notation.)
A form of `map` with parentheses is also possible:

```ml
fun mapp( f, nil ) = nil |
  mapp( f, a::y ) = ( f a ) :: mapp( f, y );
```

val mapp = fn : ('a -> 'b) * 'a list -> 'b list

- mapp( sq, [1, 2, 3] );
  val it = [1, 4, 9] : int list

`map` works well with anonymous functions (they correspond to lambda expressions in Scheme):

- map ( fn x => x*x*x ) [2, 3, 4];
  val it = [8, 27, 64] : int list

- val sq = fn x:int => x*x;
  val sq = fn : int -> int

- sq 12;
  val it = 144 : int

By the way, to negate a number use `~`:

- sq ~12;
  val it = 144 : int

Function composition:

```ml
val pow4 = sq o sq;
val pow4 = fn : int -> int

- pow4 4;
  val it = 256 : int
```

Precedences may be confusing—write `(hd o tl)`.

```ml
val lst = [5, 3, 8];
val second = hd o tl;
val second = fn : 'a list -> 'a

- second lst;
  val it = 3 : int
```

By the way, operator precedence is also possible:

```ml
- (hd o tl) [fn x=>x:int, fn x=>x*x:int] 7;
  val it = 49 : int
```

A form of `map` with parentheses is also possible:
Reducers (see the corresponding Scheme notes!):

- \[ \text{fun reduce}\,(f,\, \text{nil},\, v_0) = v_0 \]
- \[ \text{reduce}\,(f,\, (\, a::y\,),\, v_0) = f(a,\, \text{reduce}\,(f,\, y,\, v_0)); \]

\[ \text{val reduce} = \text{fn} : (\, 'a * 'b -> 'b\,) \to 'a \text{ list} \to 'b \to 'b \]

We can use \text{reduce} with anonymous functions:

- \[ \text{reduce}(\text{fn}(x,\, y:\text{int})=x+y,\, [1,\, 2,\, 3,\, 4],\, 0); \]
  \[ \text{val it} = 10 : \text{int} \]
- \[ \text{reduce}(\text{fn}(x,\, y:\text{int})=x*y,\, [1,\, 2,\, 3,\, 4],\, 1); \]
  \[ \text{val it} = 24 : \text{int} \]

The same, more simply, with an operator promoted to a function:

- \[ \text{reduce}(\text{op}\, +,\, [1,\, 2,\, 3,\, 4],\, 0); \]
  \[ \text{val it} = 10 : \text{int} \]
- \[ \text{reduce}(\text{op}\, *,\, [1,\, 2,\, 3,\, 4],\, 1); \]
  \[ \text{val it} = 24 : \text{int} \]

Finally, a parenthesis-free version of \text{reduce}:

- \[ \text{fun reduce}\,(f,\, \text{nil},\, v_0) = v_0 \]
- \[ \text{reduce}\,(f,\, (\, a::y\,),\, v_0) = f(a,\, \text{reduce}\,(f,\, y,\, v_0)); \]

\[ \text{val reduce} = \text{fn} : (\, 'a * 'b -> 'b\,) \to 'a \text{ list} \to 'b \to 'b \]

User-defined data types. An enumerated type:

- \[ \text{datatype colour} = \text{red} | \text{amber} | \text{green}; \]
- \[ \text{con red : colour} \]
- \[ \text{con amber : colour} \]
- \[ \text{con green : colour} \]
- \[ \text{val red} = \text{red} : \text{colour} \]
- \[ \text{val amber} = \text{amber} : \text{colour} \]
- \[ \text{val green} = \text{green} : \text{colour} \]
- \[ \text{length} \, [\text{red},\, \text{green},\, \text{red},\, \text{amber}] ; \]
  \[ \text{val it} = 4 : \text{int} \]

A type with functions as members:

- \[ \text{datatype tree} = \text{nul} | \text{node}\,(\, \text{int} * \text{tree} * \text{tree}\, ) ; \]
- \[ \text{con node} : \text{int} * \text{tree} * \text{tree} \to \text{tree} \]
- \[ \text{con nul} : \text{tree} \]
- \[ \text{node} ; \]
  \[ \text{val it} = \text{fn} : \text{int} * \text{tree} * \text{tree} \to \text{tree} \]

We can use \text{reduce} with anonymous functions:

- \[ \text{q},\, \,(\, a\, \text{list} \to '\text{a}',\, \text{fn}) : \text{reduce} = \text{fn} \]
- \[ \text{reduce}(\, \text{op}\, \text{fn},\, \text{nil},\, v_0) !\]
ML checks completeness of definitions:

```
fun left(node(a, L, R)) = L;
```

Warning: match not exhaustive

```
val left = fn : tree -> tree
```

We can use exceptions to make it complete:

```
extension
fun right(node(a, L, R)) = R |
  right(null) = raise NoRight;
```

```
val right = fn : tree -> tree
```

```
val right nul;
uncaught exception NoRight
```

Inserting into a tree (treated as a BST):

```
fun insert( a, nul ) = node( a, nul, nul ) |
  insert( a, node( b, L, R ) ) =
    if a < b then
      node( b, insert( a, L ), R )
    else if a > b then
      node( b, L, insert( a, R ) )
    else node( b, L, R );
```

```
val insert = fn : int * tree -> tree
```

```
val my_tree = insert(7, insert(3, insert(9,
  insert(4, insert(3, null)))));
```

```
val my_tree = node (3,nul,
  node (4,nul,node #)) : tree
```

```
val right(my_tree);
val it = node (4,nul,
  node (9,node #,nul)) : tree
```

```
val right(right(my_tree));
val it = node (9,
  node (7,nul,nul),nul) : tree
```

```
val inorder( my_tree );
val it = [3,4,7,9] : int list
```

```
fun inorder(nul) = nil |
  inorder(node(a, L, R)) = inorder(L) @ (a::inorder(R));
```

```
val inorder = fn : tree -> int list
```

```
Tree traversal:
```

```
fun reverse : tree => tree
val reverse = fn : tree => tree!
```

```
exception exhaustion;
```

```
val noRight = fn : tree => tree
```

```
val noRight = fn : tree => tree
```

```
val left = fn : tree => tree
```

```
val left = fn : tree => tree
```

```
val reverse : tree => tree
```

```
val reverse : tree => tree
```

```
ML checks completeness of definitions:
```

```
fun reverse : tree => tree
val reverse = fn : tree => tree!
```

```
exception exhaustion;
```

```
val noRight = fn : tree => tree
```

```
val noRight = fn : tree => tree
```

```
val left = fn : tree => tree
```

```
val left = fn : tree => tree
```

```
ML checks completeness of definitions:
```

```
fun reverse : tree => tree
val reverse = fn : tree => tree!
```

```
exception exhaustion;
```

```
val noRight = fn : tree => tree
```

```
val noRight = fn : tree => tree
```

```
val left = fn : tree => tree
```

```
val left = fn : tree => tree
```

```
ML checks completeness of definitions:
```

```
fun reverse : tree => tree
val reverse = fn : tree => tree!
```

```
exception exhaustion;
```

```
val noRight = fn : tree => tree
```

```
val noRight = fn : tree => tree
```

```
val left = fn : tree => tree
```

```
val left = fn : tree => tree
```

```
ML checks completeness of definitions:
```
Locality in ML.

```ml
let val aa = [1,2]
  in tl aa
  end;
val it = [2] : int list

But aa remains undefined:

```ml
let val aa = [1,2]
  in aa @ bb
  end

std_in:2.1-2.2 Error: unbound variable or constructor aa
```

More local objects:

```ml
let val (aa, bb) = ([1,2], [3,4,5])
  in aa @ bb
  end

```

Nesting is also possible:

```ml
let val aa = [1,2]
  in let val bb = [3,4,5]
  in aa @ bb
  end
  end;
val it = [1,2] @ [3,4,5] : int list

The same without nesting:

```ml
let val tt = [1,2,3,4,5] : int list
  in let tt = true : bool
    in let tt = false : bool
    end
    end
    end

val it = false : bool

- anniversary 30;
val it = false : bool

- anniversary 45;
val it = true : bool
```

Local functions:

```ml
local fun divides(x, y) = y mod x = 0
  in fun anniversary age = divides(10, age) or else
    divides(25, age)
    end
  end;
val anniversary = fn : int -> bool

- anniversary 30;
val it = false : bool

- anniversary 45;
val it = true : bool
```

Variables or constructor aa are local to the function周年纪念日，aa在函数外部仍然是未定义的。
Local patterns:

```ml
- fun mirror (p as (x, y)) = (p, (y, x));
val mirror = fn : 'a * 'b -> ('a * 'b) * ('b * 'a)
- mirror (6,17);
val it = ((6,17),(17,6)) : (int * int) * (int * int)

Minimum of a list:

```ml
- local
  fun minl_aux(elt, lst): int =
    if null lst then elt
    else if elt > hd(lst) then
      minl_aux(hd lst, tl lst)
    else minl_aux(elt, tl lst)
  in
  fun minl L =
    if null L then ~1000000000
    else minl_aux(hd L, tl L)
  end;
val minl = fn : int list -> int
- minl [1, 2, 3, 0, 5, 4, ~9, 8];
val it = ~9 : int
```

Tuples versus records in ML.

```ml
Record values in a tuple are not important:

```ml
- val tt = true : bool
  !{ { [3] }, { q = b = a } } = { { [3] }, { q = b = a } }

Record values in a record are important:

```ml
- val tt = true : bool
  !{ 3 = 4, a = 3, b = 4 } = !{ 3 = 4, a = 3, b = 4 }

Records may have elements of mixed types:

```ml
- { a = 3, b = "four" } = { b = "four", a = 3 }
- (3, "four") = (4, 3); val it = false : bool
- { a = 3, b = "four" } = { b = "four", a = 3}; val it = true : bool

Tuples have elements of the same type:

```ml
- (3, "four") = ("four", 3); std_in:3.1-3.29 Error: operator and operand don't agree (tycon mismatch)
- (3, "four") = ("four", 3)
- Records may have elements of mixed types:
- { a = 3, b = "four" } = { b = "four", a = 3 }

Tuples versus records in ML.

```ml
- (3, "four") = (4, 3); val it = false : bool
- (3, "four") = (4, 3)

Local patterns:

```ml
- fun mirror (p as (x, y)) = (p, (y, x));
val mirror = fn : 'a * 'b -> ('a * 'b) * ('b * 'a)
- mirror (6,17);
val it = ((6,17),(17,6)) : (int * int) * (int * int)
- Minimum of a list:

```ml
- local
  fun minl_aux(elt, lst): int =
    if null lst then elt
    else if elt > hd(lst) then
      minl_aux(hd lst, tl lst)
    else minl_aux(elt, tl lst)
  in
  fun minl L =
    if null L then ~1000000000
    else minl_aux(hd L, tl L)
  end;
val minl = fn : int list -> int
- minl [1, 2, 3, 0, 5, 4, ~9, 8];
val it = ~9 : int
- Records may have elements of mixed types:
- { a = 3, b = "four" } = { b = "four", a = 3 }

Tuples versus records in ML.

```ml
- (3, "four") = (4, 3); std_in:3.1-3.29 Error: operator and operand don't agree (tycon mismatch)
- (3, "four") = ("four", 3)
- Records may have elements of mixed types:
- { a = 3, b = "four" } = { b = "four", a = 3 }

Tuples versus records in ML.

```ml
- (3, "four") = (4, 3); val it = false : bool
- (3, "four") = (4, 3)
- Local patterns:

```ml
- fun mirror (p as (x, y)) = (p, (y, x));
val mirror = fn : 'a * 'b -> ('a * 'b) * ('b * 'a)
- mirror (6,17);
val it = ((6,17),(17,6)) : (int * int) * (int * int)
- Minimum of a list:

```ml
- local
  fun minl_aux(elt, lst): int =
    if null lst then elt
    else if elt > hd(lst) then
      minl_aux(hd lst, tl lst)
    else minl_aux(elt, tl lst)
  in
  fun minl L =
    if null L then ~1000000000
    else minl_aux(hd L, tl L)
  end;
val minl = fn : int list -> int
- minl [1, 2, 3, 0, 5, 4, ~9, 8];
val it = ~9 : int
- Records may have elements of mixed types:
- { a = 3, b = "four" } = { b = "four", a = 3 }

Tuples versus records in ML.

```ml
- (3, "four") = (4, 3); std_in:3.1-3.29 Error: operator and operand don't agree (tycon mismatch)
- (3, "four") = ("four", 3)
- Records may have elements of mixed types:
- { a = 3, b = "four" } = { b = "four", a = 3 }

Tuples versus records in ML.

```ml
- (3, "four") = (4, 3); val it = false : bool
- (3, "four") = (4, 3)
- Local patterns:

```ml
- fun mirror (p as (x, y)) = (p, (y, x));
val mirror = fn : 'a * 'b -> ('a * 'b) * ('b * 'a)
- mirror (6,17);
val it = ((6,17),(17,6)) : (int * int) * (int * int)
- Minimum of a list:

```ml
- local
  fun minl_aux(elt, lst): int =
    if null lst then elt
    else if elt > hd(lst) then
      minl_aux(hd lst, tl lst)
    else minl_aux(elt, tl lst)
  in
  fun minl L =
    if null L then ~1000000000
    else minl_aux(hd L, tl L)
  end;
val minl = fn : int list -> int
- minl [1, 2, 3, 0, 5, 4, ~9, 8];
val it = ~9 : int
- Records may have elements of mixed types:
- { a = 3, b = "four" } = { b = "four", a = 3 }

Tuples versus records in ML.

```ml
- (3, "four") = (4, 3); std_in:3.1-3.29 Error: operator and operand don't agree (tycon mismatch)
- (3, "four") = ("four", 3)
- Records may have elements of mixed types:
- { a = 3, b = "four" } = { b = "four", a = 3 }

Tuples versus records in ML.
Generic types in ML.

datatype 'a list =
  null | cons of 'a * ('a list);

datatype 'a list

fun cons : 'a * 'a list -> 'a list
fun null : 'a list

val twoFunc = cons(fn x:int=>x*x*x,
  cons(fn x=>x*x, null));

fun head(cons(x, y)) = x;

val head = fn : 'a list -> 'a

val it = 27 : int