ML

Same core concepts as Scheme:

- strongly typed
- expression-oriented, largely side-effect-free
- functional
- list-oriented, garbage-collected heap-based
- highly regular and expressive

Extensions:

- statically typed
  - polymorphic type system
  - automatic type inference
- pattern matching
- exceptions
- modules

Designed as a **M**eta **L**anguage for automatic theorem proving system in mid 70's by Milner et al.

Standard ML: mid 80's

SML'97: 1997

Basic datatypes

- unit (like void)

```
- ();
() : unit
```

- bool, int, real, string

```
- 7 > 5;
true : bool

- ~ 24 + 1; (* ~ is negate function *)
~23 : int

- 3.14159;
3.141519 : real

- "hi there"
"hi there" : string
```


Lists

- variable # of elements, but homogeneous element type
- `[...]` constructor notation, like `list`
- `::` and `nil` constructor notation, like `cons` and `()`
  - `::` is infix and left associative

```
- [3,4,5];
[3,4,5] : int list

- 3 :: 4 :: nil;
[3,4] : int list

- (3 :: nil) :: (4 :: 5 :: nil) :: nil;
[[3],[4,5]]: int list list

- [3, "hi there"];
Error: operator and operand don't agree
 operator domain: int * int list
 operand:         int * string list
 in expression:
    (3 : int) :: "hi there" :: nil
```


Tuples and records

- allow heterogeneous element types, but fixed # of elements

Tuples (records with positional components)

```
- ("wow",6);
("wow",6) : string * int

- (1.0, 2, true, "s");
(1.0, 2, true, "s") : real*int*bool*string
```

Records (tuples with unordered, named components)

```
- {name="J", age=1};
{name="J",age=1} : {age:int, name:string}

- {name="J", age=1}={age=3 mod 2,name="J"};
true : bool
```

Bindings

Global variables:

```
- val x = 23;
val x = 17 : int
```

Local variables for declarations:

```
- local
    val x = 10
    val q = x
  in
    val u = 230 * x
    val z = (true, 22 + q)
  end;
val u = 2300 : int
val z = (true,32) : bool * int
- x;
23 : int
```

Local variables for expressions:

```
- let
    val x = 5
    val q = x + 1
  in x * q end;
30 : int
```


Functions

`fun`: named functions

```
- fun succ x = x + 1;
val succ = fn : int -> int
- succ 8;
9 : int

- fun fact n =
    if n <= 1 then 1 else n * fact (n - 1);
val fact = fn : int -> int
```

`fn`: anonymous lambdas

```
- fn x => x + 1;
fn : int -> int
- (fn x => x + 1) 8;
9 : int
```

`fun f x = ...` is sugar for `val x = fn x => ...`

```
- val succ = fn x => x + 1
val succ = fn : int -> int
- succ 8;
9 : int
```


Function types

Types of functions written `Input -> Output`

- only one argument, one result

To pass multiple arguments, use tuples

```
- fun plus(x, y) = x + y;
val plus = fn : int*int -> int
- fun add_to_all(val,lst) =
    if null lst then lst
    else (hd lst + val)::
           add_to_all(val, tl lst);
val add_to_all = fn : int*int list -> int list
```

Precedence rules for expressions:
juxtaposition, then prefix, then infix

Precedence rules for types:
`list` suffix, then `*`, then `->`

To return multiple results, use tuples

```
- fun polar(theta) = (sin theta, cos theta);
val polar = fn : real -> real*real
```


Another example

```
- fun quad(a,b,c) =
    let val root = sqrt(b*b - 4.0*a*c)
        val two_a = 2.0*a
    in
      ((~b + root)/two_a, (~b - root)/two_a)
    end;
val quad = fn : real*real*real -> real*real
```

Pattern matching

Means of decomposing compound values

Reuse constructor syntax to take values apart

```
- val x =(false,17);

- val (a,b) = x;
val a = false : bool
val b = 17 : int
- val (false,c) = x;
val c = 17 : int
- val (true,w) = x;
exception: nonexhaustive binding failure

- val s = ["1", "2", "3"];

- val x::xs = s;
val x = "1" : string
val xs = ["2", "3"] : string list
- val x::_ = s;
val x = "1" : string
```


Pattern matching with functions

Means of defining function behavior by cases:

```
- fun fib 0 = 0
    | fib 1 = 1
    | fib n = fib(n-1) + fib(n-2);
val fib = fn : int -> int

- fun is_empty nil   = true
    | is_empty(_::_) = false;
val is_empty = fn : 'a list -> bool

- fun append(nil,   lst) = lst
    | append(x::xs, lst) =
        x :: append(xs, lst);
val append = fn : 'a list * 'a list -> 'a list
```

Something is special about these last two functions...

Polymorphic functions

Some functions are general

- can be used on arguments of different types

E.g.

- `length`
- `append`
- `is_empty`
- `first_of_pair`

In Scheme, dynamic typing allows these to be defined easily

How to define such functions in a statically-typed language?

- can't in C (or have to use casts)
- in ML: allow functions to have **polymorphic types**

Polymorphic types

A polymorphic type contains one or more **type variables**

E.g.

```
'a list
'a * 'b
(('a * 'b) list * 'a) -> 'b
```

To make a regular type, replace type variables with regular types

- each occurrence of a type variable must be replaced with the same type

When calling a polymorphic function, caller knows real type

- replace type variable with regular type

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

`length([3,4,5])`
replaces `'a` with `int`

`length([(3.0,"hi"), (2.0,"there")])`
replaces `'a` with `int * string`

Examples

```
- fun map(f, nil)    = nil
    | map(f, x::xs) = f x :: map(f, xs);
val map = fn : ('a -> 'b) * 'a list -> 'b list


val square = fn : int -> int

- map square [3,4,5];  (* 'a = int, 'b = int *)
[9,16,25] : int list


val length = fn : 'a list -> int

- map length [[3,4], [5,6,7], []]
    (* length: 'a → int;
       map: 'a → int list, 'b → int *)
[2,3,0] : int list
```


Currying

Simple definition:

```
- fun map f =
    fn lst =>
      if null lst then nil
      else f (hd lst) :: (map f) (tl lst);
val map = fn : ('a->'b) -> 'a list -> 'b list
```

Simple application:

```
- (map square) [3,4,5];
[9,16,25] : int list
```

Exploiting left-to-right associativity of “juxtaposition” fn calling:

```
- map square [3,4,5];
[9,16,25] : int list
```

Pattern-matching-based definition:

```
- fun map f =
    fn nil => nil
     | x::xs => f x :: map f xs;
val map = fn : ('a->'b) -> 'a list -> 'b list
```


Clean syntactic sugar for currying

Allow multiple formal argument patterns ⇒ curried function

```
- fun map f  nil     = nil
    | map f (x::xs) = f x :: map f xs;
val map = fn : ('a->'b) -> 'a list -> 'b list

- map square [3,4,5];
[9,16,25] : int list

- val squarer = map square;
val squarer = fn : 'int list -> 'int list

- map (map square) [[3,4,5], [6,7,8], [1,2]];
[[9,16,25], [36,49,64], [1,4]] : int list
```

Curried is the normal way to define functions

- syntactically cleaner
- semantically more flexible

Polymorphism versus overloading

Polymorphic function:
same function usable for many different argument types

```
- fun append  nil     lst = lst
    | append (x::xs) lst = x::append xs lst;
val append = fn : 'a list -> 'a list -> 'a list
```

Overloaded function:
several different functions but given same name

Resolve overloading to particular function based on:

- static argument types (in ML)
- dynamic argument classes (in object-oriented languages)

```
- 3 + 4;
7 : int
- 3.0 + 4.5;
7.5 : real
- op +;  (* which +? default to int version *)
fn : int*int -> int
- (op +):real*real->real;
fn : real*real -> real
```

Equality types

The = built-in function is polymorphic over all types that "admit equality"

- any type except those containing functions or reals

Use `''a`, `''b`, etc. to stand for these equality types

```
- fun member x  nil    = false
    | member x (y::ys) =
        x = y orelse member x ys;
val member = fn : ''a -> ''a list -> bool

- member 3 [4,5,6,3,4,5];
true : bool
- member 4.5 [3.4,5.6,4.5];
... type error ...
- member ("hi",3,square) [("there",6,double)];
... type error ...
```


Type inference

System infers types of expressions automatically, as follows:

- assign each declared variable a fresh type variable
  - result of function is implicit variable
  - share argument and result type variables across function cases
  - each reference to a let-bound polymorphic identifier (~ named function) gets separate type variables
- each subexpression in construct places constraints on types of its operands
- solve constraints

Constraint solving can partially refine types, e.g.:

- `'a` ⇒ `'b list`
- `'a` ⇒ `''b`

If overloaded operator is unresolved after constraint solving, default to `int` version

Overconstrained (unsatisfiable constraints) ⇒ type error

Underconstrained ⇒
OK for let-bound functions (gives polymorphic function),
type error ("nongeneralizable type variable") otherwise

Properties of ML's type system

Hindley-Milner type system/inference

- universal parametric polymorphism, no constrained polymorphism (other than equality types)
- only let-bound polymorphism
  - cannot pass polymorphic value that's polymorphic inside callee

```
- fun id x = x;
val id = fn : 'a -> 'a
- fun g f = (f 3, f "hi");
(* type error in ML, but in SuperML++: *)
val g = fn : (∀'a.'a->'a) -> int*string
(* not the similar ML type:
val g = fn : ∀'a.(('a->'a) -> int*string) *)
- g id;
(* I wish *)
(3, "hi") : int * string
```

Type inference yields **principal type** for expression

- single most general type that can be inferred

Worst-case complexity of type inference: exponential time

Average case complexity: linear time

Concrete user-defined datatypes

Users can define their own (polymorphic) data structures

Simple example: ML's version of enumerated types

```
datatype sign = Positive | Zero | Negative;
```

Introduces constants

- can be used in patterns

```
- fun signum(x) =
    if x > 0 then Positive
    else if x = 0 then Zero
    else Negative;
val signum = fn : int -> sign

- fun signum_value(Positive) = 1
    | signum_value(Zero)     = 0
    | signum_value(Negative) = ~1;
val signum_value = fn : sign -> int
```

Variant records/tagged unions in ML

Each component of a datatype declaration can have info

- constructors act as functions to create values with that tag
- can be used in patterns to take apart values of a tag

Example:

```
- datatype Sexpr =
    Nil |
    Integer of int |
    Symbol of string |
    Cons of Sexpr * Sexpr;
datatype Sexpr

- Nil;
Nil : Sexpr
- Integer;
Integer : int -> Sexpr
- Symbol;
Symbol : string -> Sexpr
- Cons;
Cons : Sexpr * Sexpr -> Sexpr
```


Using datatypes

```
(* '(3 hi) *)
- val lst =
    Cons(Integer(3), Cons(Symbol("hi"), Nil));
Cons(Integer 3, Cons(Symbol "hi", Nil)) : Sexpr

- fun car  Nil        = Nil
    | car (Cons(x,_)) = x;
val car = fn : Sexpr -> Sexpr
- fun cdr  Nil         = Nil
    | cdr (Cons(_,xs)) = xs;
val cdr = fn : Sexpr -> Sexpr

- cdr lst;
Cons(Symbol "hi", Nil) : Sexpr

- car lst = Integer 3;
(* Sexpr is an equality type;
   = defined automatically *)
true : bool
```


Recursive user-defined data types

Example: binary search trees of integers

```
- datatype int_tree =
    Empty |
    Node of int * int_tree * int_tree;

- fun insert x  Empty = Node(x,Empty,Empty)
    | insert x (n as Node(y,t1,t2)) =
        if x = y then n
        else if x < y then
              Node(y, insert x t1, t2)
             else
              Node(y, t1, insert x t2);
val insert = fn : int -> int_tree -> int_tree

- fun member x  Empty          = false
    | member x (Node(y,t1,t2)) =
        if x = y then true
        else if x < y then member x t1
                      else member x t2;
val member = fn : int -> int_tree -> bool
```

How to write a polymorphic version, over = and <?

`int_tree` is an equality type. Does = do the right thing?

Polymorphic user-defined data types

Polymorphic binary trees, passing = and < functions as args:

```
- datatype 'a tr =
    Empty |
    Node of 'a * 'a tr * 'a tr;
- fun ins eq lt x  Empty = Node(x,Empty,Empty)
    | ins eq lt x (n as Node(y,t1,t2)) =
        if eq(x,y) then n
        else if lt(x,y) then
              Node(y, ins eq lt x t1, t2)
             else
              Node(y, t1, ins eq lt x t2);
val ins = fn :
  ('a*'a->bool)->('a*'a->bool)-> 'a -> 'a tr
   -> 'a tr
- fun memb eq lt x  Empty          = false
    | memb eq lt x (Node(y,t1,t2)) =
        if eq(x,y) then true
        else if lt(x,y) then memb eq lt x t1
                        else memb eq lt x t2;
val memb = fn :
  ('a*'b->bool)->('a*'b->bool)-> 'a -> 'b tr
   -> bool
```

A wrapper

Avoid clients passing `eq` and `lt` procedures by
creating a wrapper data structure storing procedures

```
- datatype 'a tree =
    Tree of {tree: 'a tr,
             eq: 'a * 'a -> bool,
             lt: 'a * 'a -> bool};
- fun make_tree eq_fn lt_fn =
    Tree {tree=Empty, eq=eq_fn, lt=lt_fn};
val make_tree = fn :
  ('a*'a->bool) -> ('a*'a->bool) -> 'a tree

- fun insert x (Tree {tree=tr,
                      eq=eq_fn, lt=lt_fn}) =
    Tree {tree=ins eq_fn lt_fn x tr,
          eq=eq_fn, lt=lt_fn};
val insert = fn : 'a -> 'a tree -> 'a tree

- fun member x (Tree {tree=tr,
                      eq=eq_fn, lt=lt_fn}) =
    memb eq_fn lt_fn x tr;
val member = fn : 'a -> 'a tree -> bool
```


A problem

In Scheme,
can use "distinguished values" to handle exceptional cases

```
(define (find pred lst)
  (cond ((null? lst) #f)
        ((pred (car lst)) (car lst))
        (else (find fn (cdr lst)))))
(find is-positive? '(-3 3 5)) → 3
(find is-positive? '(-3 -5))  → #f
```

In ML, doesn't work well

```
- fun find pred nil    = false
    | find pred (x::xs) =
        if pred x then x else find pred xs;
val find = fn: (bool->bool) -> bool list -> bool

- find is_positive [~3,3,5];
... type error ...
```


Exceptions

Declare exceptions that can be returned from functions
(without affecting normal return type)
and handled in calling functions

```
- exception NotFound;
exception NotFound
- fun find pred nil    = raise NotFound
    | find pred (x::xs) =
        if pred x then x else find pred xs;
val find = fn : ('a->bool) -> 'a list -> 'a

- find is_positive [~3,3,5];
3 : int

- find is_positive [~3,~5];
uncaught exception NotFound
```


Handling exceptions

Add handler clause to expressions to handle (some) exceptions
raised in that expression

- must return same type as handled expression

Syntax:

```
expr handle exn_name => expr
        { | exn_name => expr }
        [ | _ => expr ]
```

```
- (find is_positive [~3,~5])
     handle NotFound => 0
0 : int
```

Exceptions with arguments

Can have exceptions with arguments

```
- exception IOError of int;
exception IOError of int;

- (... raise IOError(-3) ...)
     handle IOError(code) => code ...
```


An application

```
- datatype (''Key,'Value) table =
   Table of (''Key * 'Value) list;

- exception NotFound;
- fun lookup key (Table(assocs)) =
   let fun lookup_helper key nil            =
             raise NotFound;
         | lookup_helper key ((k,v)::asscs) =
             if key = k then v
             else lookup_helper key asscs
   in lookup_helper key assocs end;
val lookup = fn :
  ''key -> (''key,'value) table -> 'value

- fun store key value (Table(assocs)) =
   Table((key,value)::assocs);
val store = fn :
  ''key -> 'value -> (''key,'value) table
   -> (''key,'value) table
```


References

Allow side-effects through explicit reference values:

- `ref      : 'a → 'a ref`
- `!        : 'a ref → 'a`
- `(op :=) : 'a ref * 'a → unit`

```
- val v = ref 0;
val v = ref 0 : int ref
- v := !v + 1;
() : unit
- !v;
1 : int
```

Must say which things are mutable, unlike `set!` et al.

References to polymorphic values?

```
- fun I x = x;
val I = fn : 'a → 'a
- val fp = ref I;
(* error in real SML; pretend it's not *)
val fp = ref fn : ('a → 'a) ref
- (!fp true, !fp 5);
(true, 5) : bool * int
- fp := not;
```

hmmmm...

```
- !fp 5
```

CRASH!!!

Cannot allow references to polymorphic values

- exception arguments similarly cannot be polymorphic

In general, only polymorphic *values* can be bound in `val`/`fun` bindings, not polymorphic *expressions*

- "non-generalizable type variable" error otherwise
- SML'90 had "weakly polymorphic types" instead