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: 1980

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 b_squared = b * b
        sq_4_a_c = sqrt(4.0 * a * c)
        two_a = 2.0 * a
    in
      ((b_squared + sq_4_a_c)/two_a,
       (b_squared - sq_4_a_c)/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 hd::tl = s;
val hd = "1" : string
val tl = ["2", "3"] : string list
- val hd::_ = s;
val hd = "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,   l) = l
    | append(hd::tl,l) = hd::append(tl,l);
val append = fn : 'a list * 'a list -> 'a list
```

Something is special about these last two functions...