ML

another functional programming language

ML

ML: made for symbolic computation (including languages)

- **μScheme interpreter is about 20% the size when written in ML that it is in C**
- **never an unexplained core dump**
  and interpreter rarely halts unexpectedly
- **much savings in error checking**

Theme of the story:

- **functional programming, but**
- **detect errors as early as possible**
  (preferably at compile time!)

ML = Scheme + more familiar syntax +
pattern matching + static typing + exceptions + modules

What's new in ML?

Real language means real complexity

Main areas of novelty:

- **new syntax**
- **programming by pattern matching**
- **static polymorphic type inference**
- **exceptions**

For now, ignore "programming in the large" (modules)

Whirlwind tour: Basic values & expressions

Usual infix arithmetic except for "high minus"

```
~3 + 3 = 0
```

Boolean type `bool` (`true`, `false`)

- short-circuit `andalso`, `orelse` (as in C)

Strings `"as in C"`

String concatenation with infix hat `^`

```
"abc" ^ "xyz"
```

Characters `#"a" #"b" #"c"`

- `'` is "prime" or "tick mark" (type variables - later)

Language is completely expression-based

- like Scheme, but fewer parentheses
- `if a then b else c` is the same as C's `a?b:c`

Whirlwind tour: Identifiers

Identifiers:

- alphanumerics, `_`, `'` (prime)
- `+-/*<>=!@#$%^&'~\|?:` in any combination
  `|:>` is a perfectly good identifier!
- anything beginning with `'` is a *type variable*

Alphanumeric identifiers are case-sensitive

Types and values live in different name spaces

"Fixity": turn ordinary identifiers into infix operators

```
infixr 5 @
```
makes `@` infix, prec 5, assoc R

`op` turns them back again

```
+
```
Error: Ill-formed infix expression

```
op+
    fn :  int * int -> int
```

Whirlwind tour: Environment

Most names are bound to immutable values

(mutability identified by type using keyword `ref`, but we won't use this much)

There is no assignment that changes a binding

Add bindings to top-level environment with

`val` *ident* `=` *exp*	new value
`val rec` *ident* `=` *exp*	recursive λ-term
`fun` *ident* ...	like Scheme's "`define`" but better

In interactive environment only, must terminate with semicolon

Whirlwind tour: Lists and tuples

Unlike Scheme, lists are homogeneous

(elements of one type)

```
[1, 2, 3, 4]                OK
["hi", "there"]             OK
[op+, op-, op*, op div]     OK
["rb", 603]                 illegal!
```

Tuples are heterogeneous,

but number and types of components are fixed

```
("hi", "there")   OK
("rb", 603)       OK, but different type
("rb", "assoc prof", 112, "CS", 603) OK ...
```

More about lists

List notation is an abbreviation (syntactic sugar)

List producer/creator are `::` (cons) and `nil`

`[ ]` abbreviates `nil`

`[x, ...]` abbreviates `x :: [...]`

Example: `[1,2,3]` abbreviates `1 :: 2 :: 3 :: nil`

(`::` associates to the right)

Lots of functions on lists in "initial basis"

(a precise notion of "built in")

`hd`	(car)	`null`	(null?)
`tl`	(cdr)	`length`	(length)

There's also a better way: pattern matching (later)

Whirlwind tour: Functions

Function application by juxtaposition:

```
val L = [1,2,3]
length L
```

ML has λ, spelled "`fn`":

```
val rec length =
  fn L => if null L then 0 else 1 + length (tl L)
```

But most functions use "`fun`"

```
fun length L =
  if null L then 0 else 1 + length (tl L)
```

More about functions

Application has higher precedence than any infix operation

Examples:

```
- fun square x = x * x;
> val square = fn : int -> int
- square 3+4;
> val it = 13 : int
- square (3+4);
> val it = 49 : int
- fun f (a,b,c) = a*b*c;  (* non-curried function *)
> val f = fn : int * int * int -> int
- f (3,4,5);
> val it = 60 : int
- fun g a b c = a*b*c;    (* curried function *)
> val g = fn : int -> int -> int -> int
- g 3 4 5;
> val it = 60 : int
```

Yet more about functions

Every function takes exactly 1 argument, and returns exactly 1 result

For multiple arguments, use tuples!

```
fun factorial n =
  let fun f (i, prod) =
        if i > n then prod else f (i+1, i*prod)
  in  f (1, 1)
  end
```

Note use of "`let` *bindings* `in` *expression* `end`"

Mutual recursion uses `and` (different from `andalso`!)

```
fun a x = ... b (x-1) ...
and b y = ... a (y-1) ...
```

Whirlwind tour: Pattern Matching

"`fun`" more powerful than you thought

```
fun length nil      = 0
  | length (b::t)   = 1 + length t
```

Compiler can *guarantee* you never forget a case!

```
fun factorial 0 = 1
  | factorial n = n * factorial (n-1)
```

This is power! Example: don't need `if` built in

`if e1 then e2 else e3` **becomes**

```
(fn true => e2 | false => e3) e1
```

Read parenthesized expressions from list of characters (as in the μScheme interpreter):

```
fun get (#"("::s) = ⟨read list of exps up to paren⟩
  | get (#")"::s) = ⟨signal encounter with closing paren⟩
  | get (#"'"::s) = ⟨quote next exp⟩
  | get s         = ⟨read an atom⟩
```

More about pattern matching

Convert an S-expression to a string:

```
fun valStr (NIL)     = "( )"
  | valStr (BOOL b)  = if b then "#t" else "#f"
  | valStr (NUM n)   = Int.toString n (*almost*)
  | valStr (SYM v)   = v
  | valStr (PAIR(car, cdr)) = ⟨turn list into string⟩
  | valStr (CLOSURE   _) = "<procedure>"
  | valStr (PRIMITIVE _) = "<procedure>"
```

Yet more about pattern matching

Examples:

```
fun length [ ] = 0
  |   length (_::t) = 1 + length t;

length [10,20,30];

fun reverse [ ] = [ ]
  |   reverse (h::t) = reverse t @ [h];

reverse [10,20,30];

fun mystery [ ] = [ ]
  |   mystery (h::t) = 2*h :: mystery t;

mystery [10,20,30]; (* what does this return? *)
```

Whirlwind tour: Types

$(x_1, x_2, \ldots, x_n) : \tau_1 * \tau_2 * \ldots * \tau_n$	tuple types
`( ) : unit`	the empty tuple
`ref` x `:` τ `ref`	mutable cell
`fn x => E :` τ_1 `->` τ_2	function from τ_1 to τ_2

Types built into initial basis: do not redefine them

```
datatype bool = true | false
infixr 5 ::
datatype 'a list = op :: of 'a * 'a list | nil
datatype 'a option = SOME of 'a | NONE
```

More about types

```
datatype color = Red | Green | Blue;

structure Color = struct
   fun toString Red = "red"
     | toString Green = "green"
     | toString Blue = "blue"
end;

datatype union = I of int | R of real | B of bool | C of color;

structure Union = struct
   fun toString z = case z:union of
            I i => Int.toString i
          | R r => Real.toString r
          | B b => Bool.toString b
          | C c => Color.toString c
end;

Union.toString (C Green);
```

Yet more about types

Types used in μScheme interpreter (Notice the mutual recursion)

```
datatype exp = LITERAL of value
             | VAR     of name
             | SET     of name * exp
             | IFX     of exp * exp * exp
             ...
and    value = NIL
             | BOOL      of bool
             | NUM       of int
             | SYM       of name
             | PAIR      of value * value
             | CLOSURE   of lambda * value ref env
             | PRIMITIVE of primitive
withtype primitive = value list -> value
                         (* raises RuntimeError *)
     and lambda    = name list * exp
```

Whirlwind tour: type constructors

Abstract syntax for types:

ty $\Rightarrow$	TYVAR of string	type variable
$\mid$	TYCON of string * ty list	apply type constructor

Each tycon takes fixed number of arguments.

nullary	`int`, `bool`, `string`, ...
unary	`list`, `option`, ...
binary	`->`
n-ary	tuples (infix `*`)

More about type constructors

Concrete syntax:

ty ⇒ *tyvar*	type variable
| *tycon*	(nullary) type constructor
| *ty tycon*	(unary) type constructor
| `(`*ty*`,` ... `,` *ty*`)` *tycon*	(n-ary) type constructor
| *ty* `*` ... `*` *ty*	tuple type
| *ty* `->` *ty*	arrow (function) type
| `(`*ty*`)`	

tyvar ⇒ *'identifier* `'a, 'b, 'c, ...`

tycon ⇒ *identifier* `list, int, bool, ...`

Whirlwind tour: type examples

`int`	The (built-in) type of integers
`int * real`	The type of a pair whose 1st element is integer and whose 2nd is real
`int * real * int`	Triples, etc.
`(int * real) * int`	A pair whose first element is a pair. Not a triple.
`int * (real * int)`	Another kind of pair.
`int -> int`	A function from integer to integer.
`int * int -> bool`	same as `(int*int)->bool`
`int list`	List of integers
`int list list`	List of list of integers
`(int * int) list`	List of integer-pairs

Example Values and Types

Example values:

```
                          5 : int
                   (5, 6.3) : int * real
                (5, 6.3, 4) : int * real * int
              ((5, 6.3), 4) : (int * real) * int
              (5, (6.3, 4)) : int * (real * int)
                  [5, 6, 7] : int list
[nil, [4,5,6], [6,3], [3]] : int list list
     [(5,6), (7,8), (4,5)] : (int * int) list
```

Whirlwind tour: functions and types

Every function accepts exactly one argument,
returns exactly one result

"Multiple arguments" bundled up into a tuple

Type variables provide polymorphism

Lambda notation: $\lambda x.E$ equivalent to `fn x => E`

```
(fn x => x+1) : int -> int
(fn (f,g) => fn x => f (g x))
   : ('a -> 'b) * ('c -> 'a) -> ('c -> 'b)
```

More about functions and types

Infix operators all instances of type `'a * 'b -> 'c`

```
op > : int * int -> bool
```

Function composition

```
op o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Old friends:

```
length : 'a list -> int
map    : ('a -> 'b) -> ('a list -> 'b list)
curry  : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
id     : 'a -> 'a
foldl  : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldr  : same type as foldl
```

Yet more about functions and types

```
- map (fn x => 2*x) [20,30,40,50];
> val it = [40, 60, 80, 100] : int list

- map op* [(2,3),(4,5),(6,7),(8,9)];
> val it = [6, 20, 42, 72] : int list

- map length [[1,2,3],[4,5,6,7,8],[9,10],[ ]];
> val it = [3, 5, 2, 0] : int list

- foldl op^ "." ["ab","cd","ef"];
> val it = "efcdab." : string

- foldl op:: [ ] [1,2,3,4,5];
> val it = [5, 4, 3, 2, 1] : int list

- foldr op^ "." ["ab","cd","ef"];
> val it = "abcdef." : string

- foldr op:: [ ] [1,2,3,4,5];
> val it = [1, 2, 3, 4, 5] : int list
```

Whirlwind tour: Exceptions

Function application can raise exception

- **instead of normal termination**

Goes directly to handler in calling function

- **If no handler, continue raising exception until caught, or**
- **“uncaught exception” at toplevel**

Scoping rules are weird mix

- **handler in function located statically**
- **function with handler found dynamically**
  - **“unwind the call stack” seeking handler***

More about exceptions

```
- exception Zero;
- exception Neg of int;
- fun f 0 = raise Zero
    |  f x = if x<0 then raise Neg x else x-1;
> val f = fn : int -> int
- fun decrement x = f x
  handle Zero => (print "zero\n"; 0)
     |   Neg x => (print (Int.toString x ^ " negative\n"); x);
> val decrement = fn : int -> int
- decrement 2;
> val it = 1 : int
- decrement 0;
zero
> val it = 0 : int
- decrement ~2;
~2 negative
> val it = ~2 : int
```

Yet more about exceptions

Tremendous power for handling errors

one handler, many places to detect and raise

```
loop (topeval (readtop reader, rho, echo))
handle EOF => finish( )
  | Div       => continue "Division by zero"
  | Overflow  => continue "Arithmetic overflow"
  | RuntimeError msg =>
        continue ("run-time error: " ^ msg)
  | NotFound n => continue (n ^ " not found")
...
```

Pattern matching + exceptions

- **give μScheme interpreter its error-handling power**
  (Source of much code savings)
- $***$**Beats checking error codes at each call in C!**

Learn to program with exceptions

Whirlwind tour: Gotchas

Value polymorphism: A *toplevel* expression with *polymorphic type* cannot be a result of function application

```
- fun rev [ ] = [ ] | rev (h::t) = rev t @ [h];
> val rev = fn : 'a list -> 'a list
- rev [1, 2, 3];
> val it = [3, 2, 1] : int list
- rev [ ];
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier it
> val it = [ ] : 'a list
- val empty = [ ];
> val 'b empty = [ ] : 'b list
```

More gotchas

Overloading:

can't always tell `op + : int * int -> int`

from `op + : real * real -> real`

without context (Moscow ML and Standard ML/NJ assume integer type)

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

Yet more gotchas

Equality types

- if you compare values with `=`, we get `''a`
- "equality type variable": values must "admit equality"
- functions don't admit equality

```
- fun f(w,x,y,z) = if w=x then y else z;
> val (''a, 'b) f = fn : ''a * ''a * 'b * 'b -> 'b
```

Syntactic gotchas:

- Put parentheses around anything with |

```
case, handle, fn
```

Function application has higher precedence than any infix operator

Using reference types

Example:

```
- val x = ref 100;
> val x = ref 100 : int ref
- !x;
> val it = 100 : int
- fun dec( ) = x := !x - 1;
> val dec = fn : unit -> unit
- dec( );
> val it = () : unit
- dec( );
> val it = () : unit
- dec( );
> val it = () : unit
- x;
> val it = ref 97 : int ref
- !x;
> val it = 97 : int
```

Whirlwind tour: Running ML

Moscow ML compiler:

`mosmlc -P full filename.sml` **(* loads full set of libraries *)**

Creates binary executable file `"mosmlout.exe"`

Moscow ML interpreter:

`mosml -P full` **(* for some reason doesn't load "IO" library *)**

Fast compilation, interpreted bytecodes

Interactive system (type ";" at end of each command)

```
load "IO";
use "filename.sml";
runInterpreter(true);
```

Standard ML of New Jersey interpreter:

`sml` **(* autoloads all libraries, but incompatible with our code *)**

Slow compilation, native machine code