

## ML

ML: made for symbolic computation (including languages)

- $\mu$ Scheme interpreter < 20% the size of 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!)

Scheme +

pattern matching + static typing + exceptions

## What's New with ML?

Real language means real complexity

Ullman is long, but easily digested

Main areas of novelty:

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

For now, ignore "programming in the large" comes later with **Standard ML**

## Whirlwind tour—Basic values & expressions

Usual infix arithmetic except for "high minus"

```
-3 + 3 = 0
```

String concatenation with infix `hat ^`

True Boolean type `bool (true, false)`

short-circuit `andAlso`, `orElse` (as in C)

Strings "as in C"

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

' is "prime" or "tick mark" (type variables)

Language is completely expression-based like Scheme

```
if a then b else c the same as C a?:b:c
```

Identifiers:

- alphanumeric, `_`, `'` (prime)
- `+/*<>=|@#%&'^- \ | ? :` in any combination  
|----> 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

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

op turns them back again

```
[+, -, *, div]—Error: nonfix identifier required
[op +, op -, op *, op div]
: (int * int -> int) list
```

## 2

## Whirlwind tour—Identifiers

Identifiers:

- alphanumeric, `_`, `'` (prime)
- `+/*<>=|@#%&'^- \ | ? :` in any combination  
|----> 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

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

op turns them back again

```
[+, -, *, div]—Error: nonfix identifier required
[op +, op -, op *, op div]
: (int * int -> int) list
```

## 3

## Whirlwind tour—Environment

Most names are **bound to immutable values** (mutability identified by type)

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  $\lambda$ -term
```

```
fun ident ... like "define" but better
```

In interactive environment **only**, must terminate with semicolon

(Ullman uses bad style in many ways)

## 4

## Whirlwind tour—Structured values & expressions

Unlike Scheme, lists homogeneous

(elements of one type)

```
[1, 2, 3, 4] OK
```

```
["hi", "there"] OK
```

```
["nr", 152] illegal!
```

Tuples are heterogeneous,

but number and types of components are fixed

```
("hi", "there") OK
```

```
("nr", 152) OK, but different type
```

```
("nr", "asst prof", 152, "CS", 148) OK ...
```

List notation is 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)
```

(We'll see there's a better way)

## 5

## 6

## Whirlwind tour—Functions

Function application by juxtaposition:

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

**Application has higher precedence than any infix operation (syntactic gotcha #1)**

ML has  $\lambda$ , 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)
```

Every function takes exactly 1 argument, 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—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\*\*\*

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

Learn to program with exceptions—implementation later

## 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:

```
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 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>"
```

## Whirlwind tour—Types

```
(x1, x2, ..., xn) :  $\tau_1$  *  $\tau_2$  * ... *  $\tau_n$  tuple types
() : unit the empty tuple
ref x :  $\tau$  ref mutable cell
fn x => E :  $\tau_1$  ->  $\tau_2$  function from  $\tau_1$  to  $\tau_2$ 
Types built into initial basis—do not redefine them
datatype bool = true | false
infix 5 ::
datatype 'a list = op :: of 'a * 'a list | nil
datatype 'a option = SOME of 'a | NONE
```

Types used in  $\mu$ Scheme interpreter

```
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 Arity, RuntimeError *)
and lambda = name list * exp
```

Notice mutual recursion

## Whirlwind tour—type examples

```
int
int * real
int * real * int
(int * real) * int
int * (real * int)
int -> int
int * int -> bool
int list
int list list
(int * int) list
```

the (built-in) type of integers  
the type of a pair whose 1st element is integer and whose 2nd is real  
Triples, etc.  
A pair whose first element is a pair. Not a triple.  
Another kind of pair.  
A function from integer to integer.

same as (int\*int)->bool  
List of integers  
List of list of integers  
List of integer-pairs

13

## Tour—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
```

14

## 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:  $\lambda x.F$  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)
```

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
```

15

## 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
```

**Explained in Ullman, §5.3.1**  
or search [google.com](http://google.com) for value polymorphism

16

## Whirlwind tour—more gotchas

**Overloading:**  
can't always tell `op + : int * int -> int`  
from `op + : real * real -> real`  
without context—Moscow ML will assume integers

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

**Equality types**

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

**Syntactic gotchas:**

Put parentheses around anything with |  
case, handle, fn

17

## Whirlwind tour—Running ML

Moscow ML:

```
/usr/local/mosml/bin/mosml
interactive system (remember “;”)
```

use “filename.sml”; for loading files  
Fast compilation, interpreted bytecodes

Standard ML of New Jersey:

```
/usr/local/sml-110.0.3/bin/sml
interactive system
Initial basis about the same
```

Slow compilation, native machine code

Documentation pointed to on reading list  
but Ullman is reasonably accurate  
(see also the “course supplement” to Ullman)

**Function application has higher precedence than any infix operator**

18