

Introduction to ML

Allyn Dimock

U. Mass. Lowell

Material in these notes is largely adapted from a handout by Norman Ramsey, some additional material from Jeffrey Ullman. All material contained herein should be considered as copyright by Norman Ramsey or Jeffrey Ullman.

Introduction to ML – p.1/3:

- **ML: made for symbolic computation (including languages)**
 - μ 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**

Introduction to ML – p.2/3:

What's New with ML?

- **Real language means real complexity**
 - Core language: Pucella in 25 pages, Harper in 130 pages, Ullman 250 pages.
 - Long but not hard.
- **Main areas of novelty:**
 - new syntax
 - programming by pattern matching
 - exceptions
 - static polymorphic type inference
- For now, ignore "programming in the large". Be a consumer of library routines, but not writing modules (yet).

Introduction to ML – p.3/3:

Starting ML

```

~dimock/public_html/courses/languages/software/bin/mosml
$ mosml -P full
- load "IO";
> val it = () : unit
- ~3 + 5;
> val it = 2 : int
- ^D

~dimock/languages/ml/sml-nj/bin/sml
$ sml
...
- ~3 + 5;
val it = 2 : int
- ^D

```

mosmlc, mlton: compilers, not interactive.

Introduction to ML – p.4/3:

Whirlwind tour–Basic values&expressions

- Usual infix arithmetic except for "high minus"
`~3 + 3 = 0`
- `int`, `real`, `string`, `char`, `unit`, `(bool)`
- 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`

Introduction to ML – p.5/3:

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

Introduction to ML – p.6/3:

Whirlwind tour–Identifiers

- "Fixity"–turn ordinary identifiers into infix operators
`infix 5 @`
makes `@` infix, precedence 5, associating to the right
- `op` turns them back again
`[+, -, *, div]`
–Error: nonfix identifier required
`[op +, op -, op *, op div]`
`: (int * int -> int) list`

Introduction to ML – p.7/3:

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 the environment with
`val ident = exp` new value
`val rec ident = exp` recursive λ -term
`fun ident ...` like "define" but better
- In interactive environment **only**, must terminate bindings with semicolon. Considered bad style elsewhere.

Introduction to ML – p.8/3:

Whirlwind tour–Structured values

- Unlike Scheme, lists homogeneous (elements of one type)
[1, 2, 3, 4] OK
["hi", "there"] OK
["high", 5] illegal!
- Tuples are heterogeneous, but number and types of components are fixed
("hi", "there") OK
("high", 5) OK, but different type
(1, (2, "two"), 3) OK ...

Introduction to ML – p.9/3:

Whirlwind tour–Structured values

- 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)

Introduction to ML – p.10/3:

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 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)

Introduction to ML – p.11/3:

Whirlwind tour–Functions

- Every function takes exactly 1 argument, returns exactly 1 result – for multiple arguments, use tuples!

fun factorial n =
 let fun f (i, prod) =
 if 1 > 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) ...

Introduction to ML – p.12/3:

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)
```

Introduction to ML – p.13/3:

Whirlwind tour–Types

$(x_1, x_2, \dots, x_n) : \tau_1 * \tau_2 * \dots * \tau_n$	tuple types
$() : \text{unit}$	the empty tuple
$\text{ref } x : \tau \text{ ref}$	mutable cell
$\text{fn } x \Rightarrow E : \tau_1 \rightarrow \tau_2$	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 | :
datatype 'a option = SOME of 'a | NONE
```

Introduction to ML – p.15/3:

Whirlwind Tour–Pattern Matching

- Patterns give expressive 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 par>
  | get (#'"'::s) = <quote next exp>
  | get s          = <read an atom>
```

Introduction to ML – p.14/3:

Whirlwind tour–type examples

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

Introduction to ML – p.16/3:

Tour-Example Values and Types Example

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
```

Introduction to ML – p.17/3:

Whirlwind tour-type constructors

records: general form of tuples

```
- {name="Dimock", office=215, courses=[91.531, 91.540]};
val it = {courses=[91.531,91.540],name="Dimock",office=215
  : {courses:real list, name:string, office:int}
- #office (it)
val it = 215 : int
- {1=42, 2=3.14};
val it = (42,3.14) : int * real
- {};
val it = () : unit
```

Introduction to ML – p.19/3:

Whirlwind tour-type constructors

- **records: general form of tuples**
- **vector immutable fixed-length sequences**
- **array mutable fixed-length sequences**
- **ref mutable locations `a := !a + 1`**
- **roll-your-own algebraic datatypes (like `bool`, `list`)**

Introduction to ML – p.18/3:

Whirlwind tour-type constructors

roll-your-own algebraic datatypes (like `bool`, `list`)

```
- datatype 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
- Node(5, Leaf, Node(3, Leaf, Leaf));
val it = Node (5,Leaf,Node (3,Leaf,Leaf)) : int tree
```

Introduction to ML – p.20/3:

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`
Scheme (`lambda (x) E`)

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

Introduction to ML – p.21/3:

More pattern matching

```
fun name pattern1 = expression1
  | name pattern2 = expression2
  ...
```

```
fn pattern1 => expression1
  | pattern2 => expression2
  ...
```

```
case expression of
  pattern1 => expression1
  | pattern2 => expression2
  ...
```

```
val pattern = expression
```

Introduction to ML – p.23/3:

Whirlwind tour—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
```

Introduction to ML – p.22/3:

More pattern matching

Patterns:

<code>x</code>	variable	<code>fn x => [x]</code>
<code>(y,z)</code>	tuple	<code>val (a,b) = (Math.sin(x), 1.0 /</code>
<code>E(v)</code>	datatype constructor and argument	
<code>n</code>	constant	<code>fun fact 0 = 1</code>
<code>_</code>	wildcard	<code>fun hd (x::_) = x</code>

Record patterns: `{name=x,office=y,...}`

Nested patterns: `E(x,y) x::_ (a,(b,_))`

Naming parts: `Is as (h::t)`

Abbreviated record pattern `{name,office,...}`

Introduction to ML – p.24/3:

Whirlwind tour–Datatypes

• Types used in ML implementation of μ 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
```

Introduction to ML – p.25/3:

Notice mutual recursion

Whirlwind tour–Exceptions

- Any expression can raise an exception: for instance a function application
- Goes to most recently defined handler. If handler has no case for this exception, continue through older handlers until caught, or "uncaught exception" at toplevel
- Pattern matching + exceptions
 - give μ Scheme interpreter its error-handling power (Source of much code savings)
 - Beats checking error codes at each call in C!

Introduction to ML – p.27/3:

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>"
```

Introduction to ML – p.26/3:

Exceptions – dynamic generativity

```
fun f1 n =
  let exception bar
  in
    (if n = 0 then raise bar else f1 (n - 1))
    handle x => (print "caught\n"; raise bar)
  end
(* f1 3 prints caught 4 times times versus f2 3 once *)
fun f2 n =
  let exception bar
  in
    (if n = 0 then raise bar else f2 (n - 1))
    handle bar => (print "caught\n"; raise bar)
  end
```

Introduction to ML – p.28/3:

Exceptions other languages

- Handler for expression in ML, vs make extent of handler obvious (try block)
- Dynamic generativity peculiar to ML: rewrite last slide to ADA and both would print “caught” same number (four) times.
- Another case: Modula3: can’t define exception in loop.
- finally in Java: have to roll your own in ML.

try ... catch ... finally in Java

Exceptions are categorized by type with subtyping.
Can catch all subtypes of an exception.

Introduction to ML – p.29/3:

Simple Scheme Exception, no value

```
(define top-exception-handler (lambda ()(error "unhandled")))

(define (throw) (top-exception-handler))

(define-syntax try
  (syntax-rules ()
    ((try catch-clause body ...)
     (let* ((result #f)
            (old-handler top-exception-handler)
            (success (call/cc (lambda (cont)
                               (set! top-exception-handler
                                     (lambda () (cont #f)))
                               (set! result (begin body ...))
                               #t))))
      (set! top-exception-handler old-handler)
      (if success result (catch-clause))))))
```

If code in body evaluates (throw) then break out setting success to #f, causing (catch-clause) to be evaluated.

Introduction to ML – p.31/3:

Exceptions other languages

Scheme has no built-in exception mechanism! Roll your own.

```
(call/cc (lambda (k) CODE))
```

Executes CODE except that k is bound to a function that if called, immediately terminates the current evaluation and uses the value passed to k as the value of the whole (call/cc (lambda (k) CODE)) expression.

So basic use of call/cc is as break passing a value.

Fancier: in CODE you can treat k as just another function. Can store it, use it later to break to come other code.

Introduction to ML – p.30/3:

Whirlwind tour–ML 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")
...
```

- Learn to program with exceptions – implementation later

Introduction to ML – p.32/3:

Type system

- **Type ascription**

```
- fun id (x) = x;  
val id = fn : 'a -> 'a  
- fun id (x : int) = x;  
val id = fn : int -> int  
- fun fl (r : real) : int = floor (r);  
val fl = fn : real -> int
```

- **Types as documentation: book's technique.**

```
- val _ = id : int -> int;  
- val _ = op id : int -> int;  
- id 3.14;  
val it = 3.14 : real
```

- **Type inference: subject of 2-3 hour lecture!**