Programming Languages:

Lecture 10

ML Programming Language

Jinwoo Kim

jwkim@jjay.cuny.edu

Hello World

- The ML prompt is “-”
  - First prompt “-” and secondary prompt “=“
- Expressions typed in are immediately evaluated and usually displayed together with the resulting type
- Expressions are terminated with ";"
- The last result calculated may be referred to as *it*

```
- "Hello World";
val it = "Hello World" : string
```

Hello World (Continued)

- Expression to be evaluated is terminated by a semicolon
  - Expression followed by semicolon yield a response
- The interpreter allows expressions to go over more than one line
  - Where this happens the prompt changes to "="

```
- 3+4;
val it = 7 : int
- 4 + 4 +
= 4;
val it = 12 : int
```

Declaring Constants

- Naming constants and using names in expressions
  - *val* is used for naming constants

```
- val seconds = 60;
val seconds = 60 : int
- val minutes = 60;
val minutes = 60 : int
- val hours = 24;
val hours = 24 : int
- seconds * minutes * hours;
val it = 86400 : int
```

The identifier "it"

- By referring to "*it*", one can use the last value
  - *Any previous value of "it" is lost unless saved*

```
- it div 24;
val it = 3600 : int
- val secinhours = it;
val secinhours = 3600 : int
```

Legal Names: Alphabetic Names

- Alphabetic name
  - *Begins with letter*
  - *Then followed by letters, digits, underscore, or single quotes*
  - *Case sensitive*
  - *Example of “alphabetic names”*
    - ***x***
    - ***UB40***
    - ***h’_h’’_h’’’_456***
    - ***or_any_other_name***

Legal Names: Symbolic Names

- Permitted over following characters
  - *! % & $ # + - * / : < = > ? @ \ ~ ` ^ |*
- *May be as long as you like*
  - ----->
  - *!!?@==>->#!!!???*
- *Should not be one of the ML reserved special syntax*
  - :   _ | =   =>   ->   #
- *Allow whenever an alphabetic name is*

```
- val +-+-+ = 1415;
val +-+-+ = 1415 : int
```

ML’s keywords

- Aboid ML’s keywords when choosing names

- Especially watch out from the short ones:
  - as fn if in of op

- Keywords list

abstype and andalso as case datatype do else end
eqtype exception fn fun functor handle if in include
infix infixr let local nonfix of op open orelse raise rec
sharing sig signature struct structure then type val
while with withtype

Types

- The basic types available are integer, real, string, char, boolean
- From these we can construct objects using tuples, lists, functions and records

Integer Types

- Constants
  - Sequence of digits
    - **1, 123456**
  - ~ for a unary minus sign
    - **~2340**

- Infix operations
  - + - * div mod

- Conventional precedence
  - *(((m * n) * 1) – (m div j)) + j;*
    - ***Parenthesis can be dropped without change of meaning***

Real Types

- Constants
  - Decimal point and E notation
    - **1.123456,   ~1.2E10**

- Infix operations
  - +   -   *   /

- Functions
  - *floor (r)* converts *real* to *int* and *real (i)* converts *int* to *real*
  - *sqrt, sin, cos, tan, exp, ln*
    - ***All of type real -> real***
    - ***All need Math prefix: Math.sqrt, Math.sin***

Strings

- Constants are written in double quotes
  - “ML is the best”;

  val it = “ML is the best” : string

- Special characters
  - \n \t \” \\

- Size returns the number of charaters and ^ is for concatenation
  - “standard” ^ “ ML”

  val it = “standard ML” : string
  - size (it);

  val it = 11 : int

Characters

- Chars are distinguished from strings of length 1 by the # sign
  - “h”

  val it = “h” : string
  - #“h”

  val it = #“h” : char

- Converting between strings and characters using *str* and *sub*
  - String.str(#”h”);

  val it = “h” : string
  - String.sub(“hello”, 0);

  val it = #“h” : char

Boolean

- The two values are
  - true;

val it = true : bool

  - false;

val it = false : bool

Tuples: Cartesian Product Type

- (x1, x2, …, xn)
  - The n-tuple whose components are x1, x2, …, xn
  - The components can be of any type including tuples

- Examples

```
- val a = (1.5, 6.8);
val a = (1.5, 6.8) : real * real
- (1, 6.8);
val it = (1, 6.8) : int * real
- (“str”, 1, true, (#”h”, 1.2));
val it = (“str”, 1, true, (#”h”, 1.2)) : string * int * bool * (char * real)
```

Records

- Enclosed in braces { … }
- Type lists each field as *label : type*
- Records have components (fields) identified by name
  - val me = {name = “tom”, age = 20};

  val me = {age = 20, name = “tom}: {age:int, name:string}
  - The components can be of any type including tuples
- Selecting a field
  - #name(me);

  val it = “tom” : string

Lists

- A list is a finite sequence of elements
  - Elements can appear more than once

- Elements may have any type, but all elements of a list must have same type

```
- [1, 5, 6, 5];
val it = [1, 5, 6, 5] : int list
- [(1, “one”), (2, “two”)];
val it = [(1, “one”), (2, “two”)] : (int * string) list
- [[3.1],[],[5.7,~0.6]];
val it = [[3.1],[],[5.7,~0.6]]:    ???
```

Summary: Types

- The basic types available are integer, real, string, char, boolean
  - From these we can construct objects using tuples, lists, functions and records

- A tuple is a sequence of objects of mixed type (fixed length)
  - (2,"Andrew") : int * string
  - (true,3.5,"x") : bool * real * string
  - ((4,2),(7,3)) : (int * int) * (int * int)

- A list must have identically typed components and may be of any length
  - [1,2,3] : int list
  - ["Andrew","Ben"] : string list
  - [(2,3),(2,2),(9,1)] : (int * int) list
  - [[ ],[1],[1,2]] : int list list

Summary: Types (Contiuned)

- Questions
  - Do objects [1,2] and [1,2,3] have the same type?
  - How about objects (1,2) and (1,2,3)?
- It is important to notice the types of objects and be aware of the restrictions
- While you are learning ML most of your mistakes are likely to get caught by the type checking mechanism

Bindings

- A binding allows us to refer to an item as a symbolic name
  - The key word to create a binding is val
  - The binding becomes part of the environment
  - During a typical ML session you will create bindings thus enriching the global environment and evaluate expressions

```
- val a = 12;
val a = 12 : int
- 15 + a;
val it = 27 : int
```

Binding and Scope

- ML is a *statically scoped* language
  - Identifiers are resolved according to the static structure of the program
  - A *use* of the variable *var* is considered to the *nearest lexically enclosing declaration of var*
  - If we re-declare variable *var*, then subsequent uses of *var* refer to the “most recent” (lexically!) declaration of it
    - ***Any “previous” declarations are temporarily shadowed by the latest one***
  - *Example*
    - ***val x = 2***
    - ***val y = x*x   (* what is the value of y???*)***
    - ***val x = y*x   (* what is the value of x???*)***
    - ***val y = x*y   (* what is the value of y???*)***

Binding and Scope (Continued)

- Now it’s a bit tricky in the presence of function definition
  - *Example*
    - ***val x = 2***
    - ***fun f y = x + y***
    - ***val x = 3***
    - ***val z = f 4   (* what is the value of z???*)***

- *So, what do we learn from this example?*

Binding and Scope (Continued)

- Binding is not assignment
- Scope resolution is *lexical*, not *temporal*
  - *"most recent" declaration of variable always means "nearest lexically enclosing at the point of occurrence"*
- *"Shadowed" bindings are not lost*
  - *The "old" binding for x is still available from previous example (through calls to f), even though a more recent binding has shadowed it*

Another Binding Example

- Unlike most other languages ML allows the left hand side of an assignment to be a structure
  - ML "looks into" the structure and makes the appropriate binding

```
- val (d,e) = (2,"two");
val d = 2 : int
val e = "two" : string
- val [one,two,three] = [1,2,3];
std_in:0.0-0.0 Warning: binding not exhaustive
                one :: two :: three :: nil = ...
val one = 1 : int
val two = 2 : int
val three = 3 : int
```

  - Note that the second series of bindings does succeed despite the dire sounding warning - the meaning of the warning may become clear later

Defining Functions

- A function may be defined using the keyword “fun”
- Function declaration form:

```
fun name (parameters) = body;
```

- Example
  - fun sq (x:int) = x * x;

  val sq = fn : int -> int

Keyword ***fun*** starts the function declaration

sq is the function name

x:int is the formal parameter with type constraint

x*x is the body and it is an expression

The type of function is printed as ***fn***

int -> int is the notation of function type that takes and returns as an integer

Applying a Function

- To execute a function simply give the function name followed by the actual argument
  - sq (3);

  val it = 9 : int

- When a function is called the parameter is evaluated and then passed to the function
  - sq (sq(3));

  val it = 81 : int

- The parentheses around the argument and function definitions are optional
  - fun sq x:int = x * x;

  val sq = fn : int -> int
  - sq 3;

  val it = 9 : int

Arguments and Results

- Any type can be passed/returned !!!
- Every function has one argument and one result
  - val a = (2.0, 4.0);

  *val a = (2.0, 4.0) : real * real*
  - fun lengthvec (x:real, y:real) = sqrt(x*x + y*y);

  *val lengthvec = fn : real * real -> real*

  -- lengthvec a;

  val it = 4.472135954499957 : real
  - fun negvec (x:real, y:real) = (~x, ~y);

  *val negvec = fn : real * real -> real * real*
  - negvec (5.0, 6.0);

  *val it = (~5.0, ~6.0) : real * real*

Simple Example

- What will be the output of following function calls?

```
fun double x = 2*x;

fun inc x = x+1;

fun adda s = s ^ "a";
```

```
double 6;

inc 100;

adda “tub”;
```

$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\}$ what will be outputs?

Expressions and Simple Functions

- ML has a fairly standard set of mathematical and string functions

```
+        integer or real addition
-        integer or real subtraction
*        integer or real multiplication
/        real division
div      integer division           e.g. 27 div 10 is 2
mod      remainder                  e.g. 27 mod 10 is 7
^        string concatenation       e.g. "cub"^"a"
```

- All of the above are infix

Example 1

- Define and test the functions double and triple
  - `fun double x = 2 * x;`
  - `double 3;`

- The function times4 may be defined by applying double twice (function composition)
  - `fun times4  x  =  double ( double x );`

- Use double and triple to define times9 and times6 in a similar way

Example 2

- Functions with more than one input may be defined using "tuples"
  - fun aveI(x,y) = (x+y) div 2;
  - fun aveR(x,y) = (x+y) / 2.0;

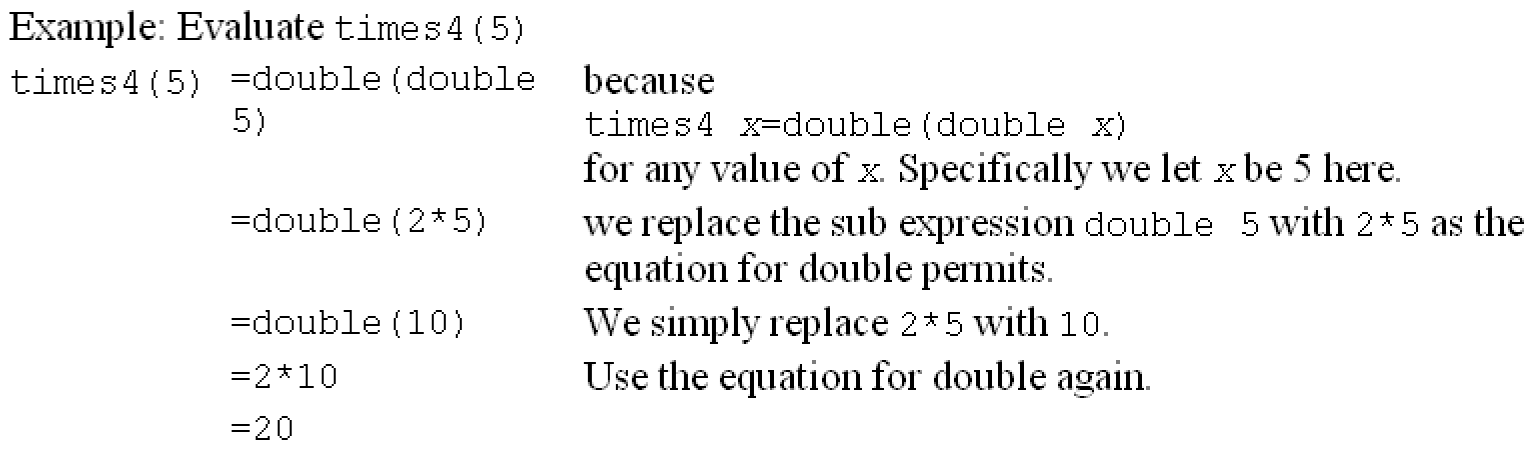
- Notice how ML works out the type of each function for itself. Try...
  - aveI(31 , 35);
  - aveR(3.1 , 3.5);

Example 3

- Declare a function duplicate which accepts a string as input and returns the string concatenated with itself as output
  - duplicate "go" evaluates to "gogo"
- fun duplicate s = s ^ s;
  - duplicate “go”;
- How about this?
  - fun duplicate “s” = “s” ^ “s”;
- Also define quadricate, octicate and hexadecicate

Example 4

- The ML interpreter has a very clear, simple operation
  - The process of interpretation is just that of reduction
  - An expression is entered at the prompt and is reduced according to a simple set of rules.



Example: Evaluate `times4(5)`

`times4(5)`	`=double(double 5)`	because `times4 x=double(double x)` for any value of *x*. Specifically we let *x* be 5 here.
	`=double(2*5)`	we replace the sub expression `double 5` with `2*5` as the equation for double permits.
	`=double(10)`	We simply replace `2*5` with `10`.
	`=2*10`	Use the equation for double again.
	`=20`	

Example 5

- Some pre-defined functions are “size” and “substring”
  - size returns the number of characters in the string
  - substring accepts a string, the start position and length of the substring
    - **note that the first character of the string is number zero**
  - The type or signature of each may be discovered by entering the name of the function alone
  - size : string -> int
  - substring : string * int * int -> string

- Suppose we wish to create the function clip which removes the last character from its input
  - clip "been" = "bee" clip "raven" = "rave"

Example 5 (Continued)

- If s is the input string we need to return the substring starting at 0 of length "one less than the size of the input string"
  - fun clip s = substring(s,0,size s - 1);

- Define the following functions given by example here
  - middle “badge” = “d” , middle “eye” = “y”
  - dtrunc “trouser” = “rouse”, dtrunc “plucky” = “luck”
  - switch “overhang” = “hangover”, switch “selves” = “vessel”
  - dubmid “below” = “bellow”, dubmid “son” = “soon”

Function as Values

- Anonymous functions with *fn* notation
  - fn x => x * x;

  val it = fn : int -> int
  - it (3);

  val it = 9 : int

- The following declarations are identical
  - fun sq x = x * x;
  - val sq = fn x => x * x;

Functions as Parameters

- Functions can be given as parameters to other functions

```
- fun Sigma (f, x, y) =
=      if x <= y then f(x) + Sigma(f, x+1, y)
=                     else 0;
val Sigma =
    fn : (int -> int) * int * int -> int
- Sigma(sq, 1,3);
val it = 14 : int
- Sigma(fn x => x * x, 1, 3);
val it = 14 : int
```

Functions as Return Value

- Functions can also be ***returned*** from other function

```
- fun inttwice (f: (int->int))  =   fn x => f(f(x));
val inttwice = fn : (int -> int) -> int -> int


- inttwice (fn x => x * x);
val it = fn : int -> int
- it(3);
val it = ??? : int
```

Type Inference

- ML deduces the types in expressions

- Type checking in the function

```
- fun facti (n, p) =  if n= 0 then p else facti (n-1, n*p);
```

  – Constant 0 and 1 have type *int*
    – ***There fore n =0 and n-1 involves integers***
    – ***So n has type int***
  – n*p must be integer multiplication, so p has type *int*
  – *facti* returns type *int*, and its argument type is *int* int*

Type Constraints

- Certain functions are overloaded
  - E.g., abs, +, -, ~, *, <
- Type of an overloaded function is determined from context, or is set to *int* by default
- Types can be stated explicitly

```
- fun min(x, y)  =  if x < y then x else y;
val min = fn : int * int -> int
- fun min(x : real, y)  =  if x < y then x else y;
val min = fn : real * real -> real
- fun min(x : string, y)  =  if x < y then x else y;
val min = fn : string * string -> string
- fun min(x, y):real  =  if x < y then x else y;
val min = fn : real * real -> real
```

Polymorphism

- Polymorphism allows us to write generic functions
  - it means that the types need not be fixed

- Consider the function length which returns the length of a list
  - This is a pre-defined function
  - Obviously it does not matter if we are finding the length of a list of integers or strings or anything
  - The type of this function is thus
    - **length : 'a list -> int**
  - the type *variable* 'a can stand for any ML type.

Polymorphic Type Checking

- Weakly typed languages (e.g., Lisp)
  - Give freedom and flexiblity
- Stongly typed languages (e.g., Ada)
  - Give security by restricting the freedom to make mistakes
- Polymorphic type checking in ML
  - Security of strong type checking
  - Great flexibility (like weak type checking)
  - Most type information is deduced automatically
  - An object is polymorphic if it can be regarded as having any kind of type

Polymorphic Function Definitions

- If type inference leaves some types completely unconstrained then the definition is polymorphic
  - A polymorphic type contains a type variable (e.g. ‘a)

```
- fun pairself x  =  (x, x);
val pairself = fn : ‘a -> ‘a * ‘a
- pairself 4.0;
val it = (4.0, 4.0) : real * real
- pairself “NW”;
val it = (“NW”, “NW”) : string * string
- fun pair (x, y)  =  (y, x);
val pair = fn : (‘a * ‘b) -> (‘b * ‘a)
```

Polymorphic Function as Values

- Example

```
- fun twice f  =  fn x => f(f(x));
val twice = fn : ('a -> 'a) -> 'a  -> 'a


-twice (fn x => x * x);
val it = fn: int -> int


- it(2);
val it = 16 : int
```

Lists Operations

- A list in ML is like a linked list in C but without the excruciating complexities of pointers
  - A list is a sequence of items of the same type
  - There are two list constructors, the empty list nil and the cons operator ::
    - **nil constructor is the list containing nothing**
    - **:: operator takes an item on the left and a list on the right to give a list one longer than the original**

```
nil                         []
1::nil                      [1]
2::(1::nil)                 [2,1]
3::(2::(1::nil))            [3,2,1]
```

Lists Operations(Continued)

- cons operator is right associative and so the brackets are not required
  - We can write 3::2::1::nil for [3, 2, 1]

- The operator :: can be used to add a single item to the head of a list

- The operator @ is used to append two lists together
  - It is a common mistake to confuse an item with a list containing a single item
  - E.g. To obtain the list starting with 4 followed by [5,6,7]
    - **we may write 4::[5,6,7] or [4]@[5,6,7]**
    - **however 4@[5,6,7] or [4]::[5,6,7] both break the type rules**

```
::      : 'a * 'a list -> 'a list
nil     : 'a list
```

Curry

- A function of more than one argument may be implemented as a function of a tuple or a "curried" function
  - Consider the function to add two integers using tuples

```
- fun add(x,y)= x+y : int;
val add = fn int * int -> int
```

  - The input to this function is an int*int pair. The Curried version of this function is defined without the brackets or comma:

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

  - The type of this function is int->(int->int)
    - **It is a function which takes an integer and returns a function from an integer to an integer**

```
- add 2 3;
it = 5 : int
```

Pattern Matching

- If we need a function which responds to different input
  - we would use the if _ then _ else structure
  - a case statement in a traditional language
  - In ML however pattern matching is preferred

- E.g. To change a verb from present to past tense we usually add "ed" as a suffix. The function past does this
  - past "clean" = "cleaned"
  - past "polish" = "polished"

- There are irregular verbs which must be treated as special cases such as run -> ran

Pattern Matching (Continued)

```
fun past "run"  = "ran"
|   past "swim" = "swam"
|   past x      = x ^ "ed";
```

- When a function call is evaluated the system attempts to match the input (the actual parameter) with each equation in turn
  - The call past "swim" is matched at the second attempt
  - The final equation has the free variable x as the formal parameter
    - **this will match with any string not caught by the previous equations**

Recursion

- A recursive function is one which calls itself either directly or indirectly
  - Using recursive functions we can achieve the sort of results which would require loops in a traditional language
  - Recursive functions tend to be much shorter and clearer

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

List Processing

- Sum of a list
  - Consider the function sum which adds all the elements of a list
  - sum [2,3,1] = 2 + 3 + 1 = 6

- How can we create function sum?
  - You have to consider two kinds of lists, empty and non-empty list (so you need to use pattern)
  - Now use recursive function definition with two list constructors, :: and nil
    - **consider the value of sum(h::t)**
    - **h is the head of the list - in this case an integer - and t is the tail of the list - i.e. the rest of the list.**

```
fun    sum nil    = 0
|      sum(h::t) = h + sum t;
```

If..then..else

- The expression "if *B* then *S1* else *S2"* tests the boolean expression *B*, it returns the value of *S1* or the value of *S2* depending on the value of *B*
  - Sometimes pattern matching is not convenient (when we wish to compare values for example)
  - E.g.
    - **fun pali s = if explode s = rev(explode s) then s ^ " is a palindrome." else s ^ " is not a palindrome.";**

Pattern Matching and Recursion

- When defining a function over a list we commonly use the two patterns

- However this need not always be the case.
  - Consider the function last, which returns the last element of a list

```
last [4,2,5,1] = 1
last ["sydney","beijeng","manchester"] = "manchester"
```

- The two patterns do not apply in this case
  - Consider the value of last nil
    - **What is the last element of the empty list?**
  - The expression last nil has no sensible value and so we may leave it undefined
  - Instead of having the list of length zero as base case we start at the list of length one
    - **pattern [h], it matches any list containing exactly one item.**

```
fun   last [h]    = h
|     last(h::t)  = last t;
```

Anonymous Function

- A function may be defined without being named

```
fn <parameters> => <expression>
```

  – For Example

```
- fn x => 2*x;
> it = fn : int -> int
- it 14;
> 28 : int
```

  – This can be particularly useful when using higher order functions like map

```
map (fn x=> 2*x) [2,3,4];
```

Map

- The following functions double and increment every item in a list respectively

```
fun   doublist nil = nil
|     doublist(h::t) = 2*h :: (doublist t);
fun   inclist nil = nil
|     inclist(h::t) = (h+1) :: (inclist t);
```

  – Typical executions:

```
doublist [1,2,3,4] = [2,4,6,8]
inclist  [1,2,3,4] = [2,3,4,5]
```

- Plainly we can abstract out of this a function which applies a function over a list (this is Map)

```
fun   map f nil = nil
|     map f (h::t) = (f h)::(map f t);
```

Filter

- filter takes a predicate (a function which returns true or false) and a list
  - It returns the list with only those items for which the predicate s true
  - E.g. Suppose the function even : int -> bool has been defined as

```
fun even n = (n mod 2 = 0);
```

  - then applying filter even over the list [1,2,3,4,5,6] would return only the even values

```
filter even [1,2,3,4,5,6] = [2,4,6]
```

Functional vs. Imperative

- Imperative
  - Using commands to change the state
- Functional
  - Stateless.
  - Using expressions recursively to calculate the result
- Example: Euclid’s algorithm for the Greatest Common Divisor (GCD) of two natural numbers

$$\mathrm{Gcd}(m, n) = \begin{cases} n, & \text{when } m = 0 \\ \mathit{gcd}(\, n \bmod m,\ m), & \textit{otherwise} \end{cases}$$

GCD – C++ vs. ML

- An imperative C++ program

```
int gcd(int m,int n){
    int prevm;
    while (m != 0){
            prevm = m;
             m = n % m;
             n = prevm;
      }
    return n;
}
```

GCD – C++ vs. ML (Continued)

- A functional program in Standard ML

```
fun gcd(m, n) =
        if m=0 then n else gcd(n mod m, m);
```