

Introduction to ML

Based on materials by
Vitaly Shmatikov

slide 1

ML

- ◆ General-purpose, non-C-like, non-OO language
 - Related languages: Haskell, Ocaml, F#, ...
- ◆ Combination of Lisp and Algol-like features (1958)
 - Expression-oriented
 - Higher-order functions
 - Abstract data types
 - Module system
 - Exceptions
- ◆ Originally intended for interactive use

slide 2

Why Study ML ?

ML is clean and powerful, and has many traits that language designers consider hallmarks of a good high-level language:

- ◆ Types and type checking
 - ML is a statically typed, strict **functional programming** language.
- ◆ Memory management
 - Static scope and block structure, activation records
 - Higher-order functions
- ◆ Garbage collection

slide 3

History of ML



- ◆ Robin Milner
 - Stanford, U. of Edinburgh, Cambridge
 - 1991 Turing Award
- ◆ Logic for Computable Functions (LCF)
 - One of the first automated theorem provers
- ◆ Meta-Language of the LCF system

slide 4

LCF – Logic of Computable Functions

ML was invented as part of the **University of Edinburgh's LCF project**, led by Robin Milner et al., who were conducting research in constructing **automated theorem provers**. Eventually observed that the "Meta Language" they used for proving theorems was more generally useful as a programming language.

slide 5

Logic for Computable Functions

◆ Dana Scott (1969)

- Formulated a logic for proving properties of typed functional programs

◆ Robin Milner (1972)

- Project to automate logic
- Notation for programs
- Notation for assertions and proofs
- Need to write programs that find proofs
 - Too much work to construct full formal proof by hand
- Make sure proofs are correct

slide 6

What is Standard ML?

SML is a strongly typed, impure, strict, functional language:

- ▶ **Strongly typed:** Every expression in the language has a *type* (`int`, `real`, `bool`, etc.). The compiler rejects a program that does not conform to the type system.
- ▶ **Functional:** Every expression evaluates to a *value*. One kind of value is a *function*. In fact, every function is a value. Like other values, functions can be bound to variables, passed as arguments to function calls, returned as values from function calls, and stored in data structures.

What is Standard ML?

SML is a strongly typed, impure, strict, functional language:

- ▶ **Impure** The evaluation of expressions in SML can incur *side-effects*, e.g., assignment to locations in mutable data structures or I/O.
- ▶ **Strict** The arguments to SML functions are evaluated before the function call is performed. Thus, if one of the arguments loops forever, then so does the entire program — regardless of whether or not the function actually needed the argument. Similarly, all side-effects caused by the evaluation of the argument occur before any side-effects caused by the evaluation of the function body.

Functional Programming (FP)

- A program implements a mapping from input values to output values.
- In imperative programming, this mapping is achieved indirectly by commands that read inputs, manipulate them, and write outputs.
- In FP, it is achieved directly: a program in FP is a function.
- Instead of primitive actions, in FP we have primitive functions. Instead of control constructs, in FP we have function composition rules.
- FP has been used as: (a) a convenient setting for studying concepts such as values and types; (b) a technique for language description; (c) a programming style in its own right (focus of this course).

Features of FP

- FP is simple because of its following feature:

Implicit storage management.

Storage is allocated as necessary by built-in operators on data. Storage that becomes inaccessible is automatically deallocated.

- FP is powerful because of its following feature:

Functions are first-order values.

Functions have the same status as any other values. A function can be the value of an expression, it can be passed as an argument, and it can be put in a data structure.

FP Languages

- Begins with LISP (LISt Processing, McCarthy 58).
- Lisp family: Lisp, MacLisp, Scheme, CommonLisp,...
- Other FP languages: ML, Haskel, Miranda, ...
- Differences between Scheme and ML:
 - ▶ Scheme is weakly typed, ML is strongly typed.
 - ▶ In Scheme, there is no syntactic difference between programs and data: a program can be manipulated just like a data.
 - ▶ We choose ML because it is small.
 - ▶ Mastering one FP language, the others are easy to learn.

SML (Standard Meta Language)

SML (/usr/local/sml/bin/sml) supports:

- *Static Scope*: All identifier references resolved at compile time.
- *Strong Typing*: Every expression has a type which can be determined at compile time. (c.f. C++ is not. e.g. virtual function)
- *Polymorphism*: functions, data types (c.f. template in C++).
- *Abstract Data Types*: type = data + functions (as in OOP).
- *Type-Safe Exception Mechanism*: to handle unusual situations arising at run-time. e.g. division-by-zero.
- *Modules*: an ML program = a set of interdependent modules glued together using *functors*.

Deducing Types

- ML is strongly typed - every expression has a type, and type checking is always done at “compile” time.
- SML interpreter will infer the type of an expression automatically.
- Some simple rules are:
 - ▶ The types of arithmetic operators are built-in, and no coercion is done.
 - ▶ In a conditional expression, the expression itself and the sub-expressions follow the then and else must be of the same type.
 - ▶ The return type of a function is the same as the type of the expression that defines the function.

Moscow ML: History

Moscow ML is a light-weight implementation of *Standard ML (SML)*, a strict functional language widely used in teaching and research.

- It implements the *SML Modules* language and some extensions. Moreover, Moscow ML supports most required parts of the *SML Basis Library*.
- It supports separate compilation and the generation of stand-alone executables.

Moscow ML was created by Sergei Romanenko, Claudio Russo, Niels Kokholm, Ken Friis Larsen and Peter Sestoft in the 90's.

Download Moscow ML

Offline (Recommended)

Available for Windows, Mac and Linux at: <https://mosml.org/>
(current version 2.10.1)

Compiled installer at <http://www.itu.dk/~sestoft/mosml.html>
(Old version)

Online

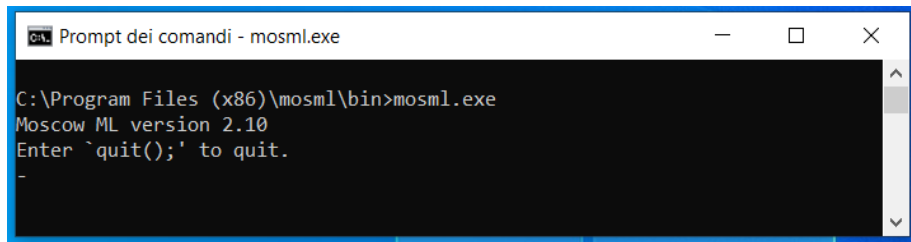
try mosml: A simple web-based environment for experimenting with SML
code at <http://try.mosml.org/>

Or if the above is offline:

Online

SOSML: The Online Interpreter for Standard ML at <https://sosml.org/editor>

Moscow ML: Test



```
Prompt dei comandi - mosml.exe

C:\Program Files (x86)\mosml\bin>mosml.exe
Moscow ML version 2.10
Enter `quit();' to quit.
-
```

Moscow ML

- *Syntax*: how to write correct *expressions*
- *Semantics*: what is the meaning of these expressions
 1. *Type-checking* checks the static environment (types and declarations)
 2. *Evaluation* checks the dynamic environment

The *type-checking* is performed before the program execution

The *evaluation* is performed at runtime

Moscow ML: The interactive shell

Just type your declarations and expressions into the interactive shell and the system will respond. Here is an example session:

```

1 | $ mosml
2 | Moscow ML version 2.10
3 | - 4;
4 | > val it = 4 : int
5 | - "hello_" ^ "_there";
6 | > val it = "hello__there" : string
7 | - val x = 12 and y = ~5;
8 | > val x = 12 : int
9 |     val y = ~5 : int
10 | - x - y * 2;
11 | > val it = 22 : int

```

If you don't specify types, it figures them out for you (as long as the syntax is correct). If you type in an expression directly, it turns it into a declaration binding the result to the special name `it`.

Moscow ML

`it` refers to the value of the last expression

```
1 | 3+4;  
2 | > val it=7 :int
```

Can be called directly

```
1 | it;  
2 | > val it=7 :int
```

Can be used in expressions

```
1 | it+1;  
2 | > val it=8 :int
```


Moscow ML: Shadowing

```
1 | val a = 14;  
2 | val b = 10 * a;  
3 | val a = 3;  
4 | val c = b;
```

What does the dynamic environment look like?

Moscow ML: Compiling

The Moscow ML compiler is called `mosmlc`.

In Windows you can enter:

```
1 | > mosmlc test.sml -o test.exe
```

and in Unix you can enter:

```
1 | $ mosmlc test.sml -o test
```

and you have your executable. To compile and run:

```
1 | > mosmlc test.sml -o test.exe && test
```

or:

```
1 | $ mosmlc test.sml && ./a.out
```

Moscow ML: Declarations

A program is a sequence of declarations. Declarations begin with:

- val, to declare a new value binding
 - fun, to declare a new function
 - type, to introduce a type abbreviation
 - datatype, to define a new type with visible constructors
 - abstype, to define a new type with hidden constructors
 - exception, to define a new exception constructor
 - structure, to define a new structure
 - local, to make a declaration that uses private local declarations
- (but also: signature, functor, infix, infixr, nonfix, open)

5 Basic Types, 3 Composite Types

TYPE	SYMBOL	EXAMPLE	OPERATIONS/TYPES
unit	()	()	—
boolean	bool	true, false	not, andalso, orelse
integer	int	2, 0, 87	\sim , +, -, *, div, mod
real	real	1.3, 3E2	\sim , +, -, *, /
string	string	"hello"	\wedge
tuple	(...)	(1, "yes", 2.5)	int*string*real
list	[...]	[3, 8, 1, 9]	int list
record	{...}	{ID="007", age=51}	{ID:string,age:int}

Basic Types

- **unit** is similar to **void** in C. It is used
 - ▶ whenever an expression has no interesting value.
 - ▶ when a function is to have no arguments.
- The boolean operators **andalso** and **orelse** perform short-circuit evaluations: i.e.
 - E_1 **andalso** $E_2 \Rightarrow$ will NOT evaluate E_2 if E_1 is **false**.
 - E_1 **orelse** $E_2 \Rightarrow$ will NOT evaluate E_2 if E_1 is **true**.
- Negative **int** or **real** values are denoted using the unary operator \sim instead of the usual minus sign.
- Integer division uses **div** and **mod**, and real number division uses $/$.
NO implicit coercion!

Example: int/real

```
- ();  
val it = () : unit
```

```
- 5 + 13;  
val it = 18 : int
```

```
- ~5 + 13;  
val it = 8 : int
```

```
- floor(123.6);  
val it = 123 : int
```

```
- floor(~123.6);  
val it = ~124 : int
```

Moscow ML: Integer

Integer

```
1 | 6;  
2 | > val it=6 : int
```

Negative number

```
1 | ~6;  
2 | > val it=~6 : int
```

Operation

```
1 | 6+~6;  
2 | > val it=0 : int
```

Division

```
1 | 6 div 3;  
2 | > val it=2 : int
```

Moscow ML: More operators

- `div` Integer division
- `/` "Regular" division
- `~` ~~Less than zero~~ Minus
- `round(4.5)` = 4 Integer
- `trunc(4.5)` = 4 Integer
- `ceil(4.5)` = 5 Integer
- `floor(4.5)` = 4 Integer
- `real(6)` Real number

Example: Type Checking in int/real

- 5/6;

```
stdIn:50.2 Error: overloaded variable not defined at type
  symbol: /
  type: int
```

- real(5)/6;

```
stdIn:1.1-49.6 Error: operator and operand don't agree
  operator domain: real * real
  operand:         real * int
  in expression:
    real 5 / 6
```

- real(5)/real(6);

```
val it = 0.83333333333333 : real
```

Example: String

```
- "Hong" ^ " ^"Kong";
val it = "Hong Kong" : string

- size "Hong Kong";
val it = 9 : int

- size "Hong" ^ " ^"Kong";
stdIn:69.1-69.23 Error:
  operator and operand don't agree [tycon mismatch]
  operator domain: string * string
  operand:          int * string
  in expression:    size "Hong" ^ " ^"
```

```
- size("Hong" ^ " ^"Kong");
val it = 9 : int
```

Example: Boolean Expression

- “if <bool-exp> then <then-exp> else <else-exp>” always come together; and its value is that of <then-exp> if <bool-exp> is true, otherwise that of <else-exp>.
- <then-exp> and <else-exp> must match in their types.

```
- if 2=3 then "don't worry" else "be happy";
```

```
val it = "be happy" : string
```

```
- if "don't worry"="be happy" then 1 else 2;
```

```
val it = 2 : int
```

```
- if 2=3 then "don't worry" else 4;
```

```
stdIn:1.1-61.3 Error: types of rules don't agree [literal]
```

```
earlier rule(s): bool -> string
```

```
this rule:      bool -> int
```

```
in rule:        false => 4
```

Composite Type: Tuple

- n -tuple: (e_1, e_2, \dots, e_n) . The n items may be of mixed types.
- 2 n -tuples are equal if their corresponding components are equal.
- Items in a tuple are ordered, and “# k ” selects the k th item.

```
- (4, true, "cat");
  val it = (4,true,"cat") : int * bool * string
- (if 3=8 then "X" else "Y", 9.5/0.5, 5 div 2);
  val it = ("Y",19.0,2) : string * real * int
- (14 mod 3, not false) = (1+1, true);
  val it = true : bool

- #2("for", "your", "info");
  val it = "your" : string
```

List

- Empty list: **nil** or `[]`;
- **nil** : 'a list \Rightarrow a polymorphic object.
- $[e_1, e_2, \dots, e_n]$ is an abbreviation for $e_1 :: e_2 :: \dots :: e_n :: \text{nil}$.
- `::` is the list *constructor* pronounced as “cons”.
- `::` is an infix operator which is right associative.
- `<new-list> = <item> :: <list>`.

$$\begin{aligned}
 1 :: 2 :: 3 :: \text{nil} &= 1 :: (2 :: (3 :: \text{nil})) \\
 &= 1 :: (2 :: [3]) \\
 &= 1 :: [2, 3] \\
 &= [1, 2, 3]
 \end{aligned}$$

- Equality on 2 lists is item-by-item.

List Operators

- cons operator: $:: : 'a \text{ item} * 'a \text{ list} \rightarrow 'a \text{ list}$
- head operator: $hd() : 'a \text{ list} \rightarrow 'a \text{ item}$
- tail operator: $tl() : 'a \text{ list} \rightarrow 'a \text{ list}$
- append operator: $@ : 'a \text{ list} * 'a \text{ list} \rightarrow 'a \text{ list}$

```
- hd([1,2,3,4]);
```

```
val it = 1 : int
```

```
- tl([1,2,3,4]);
```

```
val it = [2,3,4] : int list
```

```
- hd([1,2,3,4])::tl([1,2,3,4]);
```

```
val it = [1,2,3,4] : int list
```

```
- [5,6]@tl([1,2,3,4]);
```

```
val it = [5,6,2,3,4] : int list
```

Functions on Lists

- Function that appends two (arbitrary) lists:

Example

```
- fun app nil l = l
=   | app (h::t) l = h::(app t l);
val app = fn : 'a list -> 'a list -> 'a list
```

⇒ what are the 'a types? **polymorphic type variables**

- And what does it do:

Example

```
- app [1,2,3] [4,5,6];
val it = [1,2,3,4,5,6] : int list
- app ["a","b"] ["c"];
val it = ["a","b","c"] : string list
```

⇒ the arguments must be lists of the same type

Record

- c.f. **struct** in C.
- Syntax: $\{ label_1 = E_1, label_2 = E_2, \dots \}$
- Order does NOT matter since the fields are labelled.
- Tuples are actually short-hands for records.
 $(E_1, E_2, E_3) = \{ 1=E_1, 2=E_2, 3=E_3 \}$

```
- {name="bird", age=5, dead=true};
val it = {age=5,dead=true,name="bird"}
         : {age:int, dead:bool, name:string}
- {name="bird", age=5, dead=true}
  = {age=5, dead=true,name="bird"};
val it = true : bool
```


Moscow ML: Identifiers

Alphanumeric identifiers

a sequence of letters, digits, primes (') and underbars (_) starting with a letter or prime

Symbolic identifiers

any non-empty sequence of the following symbols: ! % & \$ # + - / : < = > ? @ \ ~ ' ^ | *

Reserved words are excluded.

Identifiers

BNF¹ for alphanumeric identifiers:

```

<Id>                ::= <First_Char><Other_Chars>
<First_Char>        ::= [A-Z] | [a-z] | '
<Other_Chars>        ::= <empty> | <Other_Char><Other_Chars>
<Other_Char>         ::= [A-Z] | [a-z] | [0-9] | ['_]
```

BNF for symbolic identifiers:

```

<Id>                ::= <S_Char> | <S_Char><Id>
<S_Char>            ::= [+ - / * < > = ! @ # % ^ ` ~ \ $ ? : ]
```

- '<Other_Char>' are *alpha* variables ONLY used for data types.
- Symbolic identifiers should be used for user-defined operators.

¹Will be covered in the grammar section.

Identifiers: Value Binding

Syntax: **val** *< identifier >* = *< expression >*;

```
- val a_df = 3+2; (* c.f. const int a_df = 3+2; in C++ *)
val a_df = 5 : int
```

```
- val a'a = "Albert" ^ "Einstein";
val a'a = "Albert Einstein" : string
```

```
- val a1b2 = 2;
val a1b2 = 2 : int
```

```
- val +++$$$ = 9*3; (* may hold integral value *)
val +++$$$ = 27 : int
```

```
- +++$$$ + +++$$$; (* Though you don't want to do that *)
val it = 54 : int
```

Declaration: let Statement

let

val <1st-identifier> = < E_1 >;

val <2nd-identifier> = < E_2 >;

...

in

<expression>

end

- The semicolons at the end of each **val** statements is optional.
- c.f. Declaration of local variables in C++

Let Example

```
- let
    val x = 3
    val y = 5
  in
    x*x + 3*y
  end;
val it = 24 : int
```

let: val Example

```
- val z =
  let
    val x = 3
    val y = 5
  in
    x*x + 3*y
  end;
val z = 24 : int
```

- As spaces are immaterial, the statement may as well be written all in one single line as follows:

```
val z = let val x = 3  val y = 5 in x*x + 3*y end;
```

- To avoid too many **val** statements in the **let**-part, one may use tuples to group all identifiers as follows:

```
val z = let val (x, y) = (3, 5) in x*x + 3*y end;
```

Nested let Example

```
- let val x = 3.0  val y = 5.0 in
    let val a = x+y  val b = x-y in
        let val f = a*b*x  val g = a/b/y  in  f/g  end
    end
end;
```

Quiz: What is the output? `val it = 60.0 : real`

$$f/g \Rightarrow (a*b*x)/(a/b/y) \Rightarrow$$

$$((x+y)*(x-y)*x)/((x+y)/(x-y)/y) \Rightarrow$$

$$8.0*(-2.0)*3.0 / 8.0/-2.0/5.0 \Rightarrow -48.0 / -0.8$$

Pattern Matching

- Pattern matching with tuples

- `val (left, right) = ("Einstein", 4);`
`val left = "Einstein" : string`
`val right = 4 : int`

- Pattern matching with lists

- `val x::y = [5,6,7,8];` `(* [5,6,7,8] = 5::[6,7,8] *)`
`val x = 5 : int`
`val y = [6,7,8] : int list`

- Pattern matching with records

- `val {flag=y,count=x} = {count=2,flag=true};`
`val x = 2 : int`
`val y = true : bool`

Pattern Matching: Wildcard Pattern

The wildcard pattern “_” (underscore symbol) may be used for terms that you don't care in pattern matching.

```
- val (left,_) = ("Einstein", 4);  
val left = "Einstein" : string  
  
- val _::a = [1,2,3];  
val a = [2,3] : int list  
  
- val x::_::z = [[1,2],[3,4],[7,9],[0,0]];  
val x = [1,2] : int list  
val z = [[7,9],[0,0]] : int list list
```

Pattern Matching: Bug

Identifiers cannot duplicate in various parts of a pattern.

```
- val (x, x::y) = (3, [3,4,5]);
```

```
stdIn:1.1-287.4 Error: duplicate variable in pattern(s): x
```

```
- val (x, x) = (3,3);
```

```
stdIn:1.1-279.7 Error: duplicate variable in pattern(s): x
```

Functions

- λ -abstractions:

Example

```
- fn x => x+1;  
val it = fn : int -> int
```

- functions can be “declared” and “used”:

Example

```
- val twice = (fn x => 2*x);  
val twice = fn : int -> int  
- twice y;  
val it = 40 : int
```

⇒ what if we wanted a **recursive function**?

Moscow ML: Defining Functions 1/3

Since **functions** are values, you declare them just like any other value. You can take advantage of the fact that each declaration can use previous declarations:

```
1 | val two = 2;
2 | val addSix = fn x => x + two + 4;
3 | val square = fn x:real => x * x;
4 | val magnitude = fn (x,y) => Math.sqrt(square x +
    square y);
5 | val rec fact = fn n => if n = 0 then 1 else n *
    fact (n-1);
```

Functions

- there is a `rec` construction (which almost nobody uses)
- functions are defined “explicitly” using a `fun` declaration:

Example

```
- fun fac n = if (n=0) then 1 else n*(fac (n-1));  
val fac = fn : int -> int
```

- but more commonly using **match patterns**:

Example

```
- fun fac 0 = 1  
=   | fac n = n*(fac (n-1));  
val fac = fn : int -> int  
- fac 10;  
val it = 3628800 : int
```

⇒ match patterns better cover all possible parameter values!

Composite Functions

Given: $f: 'b \rightarrow 'c$ and $g: 'a \rightarrow 'b$.

Define a new function: $h(x) = f \circ g(x) \equiv f(g(x)) : 'a \rightarrow 'c$.

i.e first apply function $g()$ to an input x of $'a$ type, returning a value of $'b$ type, which is then piped into function $f()$ to give the final result of $'c$ type.

```
- fun square x = x*x;      fun twice x = 2*x;
```

```
val square = fn : int -> int
```

```
val twice = fn : int -> int
```

```
- val sq_twice = square o twice; (* Use val NOT fun *)
```

```
val sq_twice = fn : int -> int
```

```
- sq_twice 3;
```

```
val it = 36 : int
```

Functions: It is “fun”

- Syntax: **fun** <identifier> (<parameter-list>) = <expression>;
- Parameter passing method: Call-By-Value.

```
- fun square(x) = x*x;
val square = fn : int -> int
```

```
- fun square x = x*x; (* parentheses are optional *)
val square = fn : int -> int
- square 4;
val it = 16 : int
```

```
- fun first (x,y) = x;
val first = fn : 'a * 'b -> 'a  --> (a', b') first
first (3, "man") => val it = 3 : int
first ("man",3) => val it = "man" : string
```

Type of Functions

Each identifier, variable or function, has a type.

Function : $\langle \text{domain type} \rangle \rightarrow \langle \text{range type} \rangle$

- Argument type may be explicitly specified with $: \langle \text{type} \rangle$. e.g. A function whose input is a **real** number and which returns a **real** number:

```
- fun f_square(x: real) = x*x;
```

```
val f_square = fn : real -> real
```

```
- fun f_square(x):real = x*x;      (* Another way *)
```

- Types can be polymorphic - a dynamic type determined only at run time:

```
first (3, "man") => val it = 3 : int
```

```
first ("man",3) => val it = "man" : string
```


Scope

In functions, identifiers with the same names are resolved using the static lexical scope rule.

```
fun weird(x: real) =
  let val x = x*x
      val x = x*x
  in   x*x*x end;
```

- weird 2.0;

What is the result? 4096.0

```
x*x*x => (x*x)*(x*x)*(x*x) =>
((x*x)*(x*x))*((x*x)*(x*x))*((x*x)*(x*x)) =>
((2.0*2.0)*(2.0*2.0))*((2.0*2.0)*(2.0*2.0))*((2.0*2.0)*(2.0*2.0))
```

More Complex Functions

- Defined with boolean expressions.
 - `fun greater(x,y) = if x > y then x else y;`
 - `fun factorial x = if x = 0`
`= then 1 (* Initial '=' is continuation symbol *)`
`= else x*factorial(x-1);`
- Defined by enumerating **ALL** cases with pattern matching (\Rightarrow more readable).
 - `fun factorial 0 = 1`
`| factorial x = x * factorial(x-1);`

Functions: Bug

When functions are defined by case analysis, SML issues a warning or an error if

- Not all cases are covered.

```
- fun myhead(head::tail) = head;
stdIn:266.1-266.30 Warning: match nonexhaustive
      head :: tail => ...
val myhead = fn : 'a list -> 'a
```

- A case is redundant because of earlier cases.

```
- fun nonsense(_) = 3 | nonsense(0) = 5;
stdIn:275.1-275.47 Error: match redundant
      _ => ...
-->    0 => ...
```

Higher-Order Functions (I)

Functions taking functions as arguments:

```
- fun square x = x*x; fun twice x = 2*x;

- fun apply5 f = f 5;
  val apply5 = fn : (int -> 'a) -> 'a
- apply5 square;    val it = 25 : int

- fun apply f x = f(twice(x));
  val apply = fn : (int -> 'a) -> int -> 'a
- apply square 3;    val it = 36 : int

- fun first x y = x;
  val first = fn : 'a -> 'b -> 'a
- first 2 "hello";   val it = 2 : int
```

Higher-Order Functions (I)(cont'd)

- Function application is left-associative.

Thus, $(\text{first } x \ y) = ((\text{first } x) \ y)$.

- Operator \rightarrow is right-associative.

Thus, $'a \rightarrow 'b \rightarrow 'a = 'a \rightarrow ('b \rightarrow 'a)$.

- i.e. $\text{first}()$ has domain type $= 'a$, range $= 'b \rightarrow 'a$.
- i.e. $\text{first}()$ takes an $'a$ value and returns another function which takes a $'b$ value and returns an $'a$ value.

Moscow ML: Polymorphic types

Definition 2 (Polymorphic types)

A type that contains one or more type variables is called polymorphic
-Graham Hutton, University of Nottingham

```

1  fun ident x=x;
2  > val a'ident=fn : 'a->'a [type variable]
3  fun pair x=(x,x);
4  > val a'pair=fn : 'a->'a * 'a [polymorphic type]
5  fun fst (x,y)=x;
6  > val fst=fn : 'a * 'b -> 'a
7  val foo=pair 4.0;
8  > val foo : real*real
9  fst foo;
10 > val it=4.0 : real

```

Moscow ML: Type inference

Moscow ML type-inference algorithm in action:

```

1  [true, true, false]    (* bool list *)
2  []                     (* 'a list *)
3  fn s => length s * 2    (* 'a list -> int *)
4  fn x => (15.2, x)        (* 'a -> real * 'a *)
5  {x = 3, y = 5}          (* {x : int, y : int} *)
6  fn x => fn y => fn z => (z, x, y)
7  (* 'a -> 'b -> 'c -> 'c * 'a * 'b *)
8  fn (x, y) => x = y;      (* ''a * ''a -> bool *)
9  (fn s => SOME s) []      (* 'a list option *)

```

Types with a double apostrophe prefix are equality types. Equality doesn't work on all types. For example, the type `int->int` will match (unify) with `'a`, **but it will not unify with `"a`**. (More details within a slide)

Higher-Order Functions (II)

Functions returning function:

```
- fun sq_or_twice x = if x > 0 then square else twice;  
val sq_or_twice = fn : int -> int -> int
```

```
- (sq_or_twice 2) 5;  
val it = 25 : int
```

```
- sq_or_twice 2;  
val it = fn : int -> int
```


Functions on List: Examples

- In general, a function on list must deal with the 2 cases:

- ▶ `[]` or `nil`
- ▶ `head::tail`

```
- fun len([]) = 0   |   len(x::tail) = 1 + len(tail);  
- fun sum([]) = 0   |   sum(x::tail) = x + sum(tail);  
- fun mean L = sum L div len L;  
- mean [1,2,3];  
val it = 2 : int
```

```
- fun append([], L2) = L2  
  |   append(x::tail, L2) = x::append(tail, L2);  
- append([3,5], [9,8,7]);  
val it = [3,5,9,8,7] : int list
```

List Function: map

- The built-in **map**() has 2 arguments: a function $f()$ and a list.
- It applies function $f()$ to each element of the list.

```
fun map  $f$  [ ] = [ ]  
|   map  $f$  (head::tail) = ( $f$  head)::(map  $f$  tail);
```

- ▶ Type of list: 'a list
- ▶ Type of f : 'a \rightarrow 'b
- ▶ Type of **map**: ('a \rightarrow 'b) \rightarrow 'a list \rightarrow 'b list

map: Examples

```
- fun odd x = (x mod 2) = 1;
```

```
val odd = fn : int -> bool
```

```
- map odd [1,2,3];
```

```
val it = [true,false,true] : bool list
```

```
- map odd;    What is the result?
```

```
val it = fn : int list -> bool list
```

```
- map;
```

```
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```

List Function: filter

- **filter** applies a boolean test function to each element of a list, removing the element should the test fail.

```
fun filter f [ ] = [ ]
```

```
|   filter f (head::tail) = if (f head)
                             then head::(filter f tail)
                             else (filter f tail);
```

```
- filter;
```

```
val it = fn : ('a -> bool) -> 'a list -> 'a list
```

```
- filter odd;
```

```
val it = fn : int list -> int list
```

```
- filter odd [1,2,3,4,5];
```

```
val it = [1,3,5] : int list
```

List Function: reduce

- **reduce** accumulates a result from a list.

fun reduce f [] v = v

| reduce f (head::tail) v = f (head, reduce f tail v);

- fun add (x,y) = x+y;

- reduce add [1,2,3,4,5] 0;

val it = 15 : int

- reduce;

val it = fn : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b

- reduce add;

val it = fn : int list -> int -> int

- reduce add [1,2,3,4,5];

val it = fn : int -> int

List Function: Example

```
- fun reverse_([], L2) = L2
  |   reverse_(x::tail, L2) = reverse_(tail, x::L2);
- fun reverse L = reverse_(L, []);

- reverse ["D","O","G"];
val it = ["G","O","D"] : string list
```

The same function defined using “let”:

```
fun reverse L =
  let fun rev_([], L2) = L2
        |   rev_(x::tail, L2) = rev_(tail, x::L2)
      in rev_(L, []) end;
```

- *rev*: 'a list \rightarrow 'a list, is SML's built-in operator to do that.

Creating New Infix Operators

Left-associative: **infix** <precedence-level> <operator identifier>.

Right-associative: **infixr** <precedence-level> <operator identifier>.

- If omitted, <precedence-level> is taken as 0 — the lowest level.

PRECEDENCE	OPERATORS	ASSOCIATIVITY	COMMENTS
3	<code>o</code>	—	function composition
	<code>:=</code>	—	assignment
4	<code>=, <, >, <=, >=</code>	left	relational operators
5	<code>::</code>	right	list constructor
	<code>@</code>	right	list concatenation
6	<code>+, -</code>	left	add/subtract
	<code>^</code>	left	string concatenation
7	<code>*, /, div, mod</code>	left	multiply/divide

New Operator (cont'd)

- Create the function
 - `fun **(a,0) = 1 | **(a,b) = a * **(a,b-1);`
 - `val ** = fn : int * int -> int`
- Test it:
 - `**(2,5);`
 - `val it = 32 : int`
- Declare it as a left associative operator:
 - `infix 7 **;`
 - `infix 7 **`
 - `4 + 2**5 - 6;` - `2**3**2;`
 - `val it = 30 : int` `val it = 64 : int`
- Or declare it as a right associative operator:
 - `infixr 7 **;`
 - `infixr 7 **`
 - `2**3**2;`
 - `val it = 512 : int`

Defining New Datatype

Syntax: **datatype** <type-name>

= <1st-constructor> | <2nd-constructor> | ...

- A simple example:

```
datatype Primary_Lights = red | green | blue;
```

```
- red;
```

```
val it = red : Primary_Lights
```

- c.f. enumeration in C++

```
enum Primary_Lights = red, green, blue ;
```

Moscow ML: Defining a Custom Type 1/2

The `type` declaration does not make a new type, just a type abbreviation:

```
1 | type intpair = int * int;  
2 | type person =  
3 | {name: string, birthday: date, weight: real};  
4 | type simplefun = real -> real;  
5 | type text = string;
```

For the IF clause the *type-checker* checks for the guard to be type `BOOL`, and that both branches have the same type

Constructors of Datatype

- More complex objects can be constructed too. e.g.

```
datatype Money = nomoney | fun amount nomoney = 0
  | coin of int           | amount(coin(x)) = x
  | note10 of int         | amount(note10(x)) = 10*x
  | note100 of int        | amount(note100(x)) = 100*x
  | check of string*int;  | amount(check(bank,x)) = x;
```

```
- amount (note100(2));
val it = 200 : int
```

- Money has 5 constructors: nomoney as a constant constructor, coin(int), note10(int), note100(int), and check(string, int).
- Any function on Money should deal with have 5 cases, one for each constructor.

Moscow ML: Defining a Custom Type 2/2

You need to use datatype or abstype to create a new type, distinct from any other. Datatypes are defined with constructors:

```
1  datatype weekday = Monday | Tuesday | Wednesday |
    Thursday | Friday;
2
3  datatype file_descriptor = Stdin | Stdout | Stderr
    | Other of int;
4
5  datatype 'a tree = Empty
6                  | Node of 'a * 'a tree * 'a tree;
7
8  datatype Shape = Circle of {radius: real}
9                | Rectangle of {height: real, width
10                               : real}
11                | Polygon of (real * real) list;
```

Recursive Datatype: Differentiation Example

- datatype expr = constant of int
 - | variable of string
 - | sum of expr * expr
 - | product of expr * expr;
- val zero = constant 0; val one = constant 1;
- fun D x (constant _) = zero
 - | D (variable w) (variable z) = if w = z then one else zero
 - | D x (sum(e1, e2)) = sum(D x e1, D x e2)
 - | D x (product(e1, e2)) =
 - let val term1 = product(D x e1, e2)
 - val term2 = product(e1, D x e2)
 - in sum(term1, term2) end;
- val D = fn : expr -> expr -> expr
- expr has 4 constructors: constant(int), variable(string), sum(expr, expr), product(expr, expr).

Polymorphic Datatype: Binary Tree Example

```
datatype 'a tree =
  empty_tree | leaf of 'a | node of 'a tree*'a tree;
```

- The 'a tree has 3 constructors: empty_tree (constant constructor), leaf('a), and node('a tree, 'a tree).

```
- fun leafcount(empty_tree) = 0
  | leafcount(leaf(x)) = 1
  | leafcount(node(L,R)) = leafcount(L) + leafcount(R);
val leafcount = fn : 'a tree -> int
```

```
- val x = node(node(leaf(1), leaf(2)), leaf(3));
val x = node (node (leaf #,leaf #),leaf 3) : int tree
```

```
- leafcount x;
val it = 3 : int
```

Moscow ML: Case Of

Before seeing how to extract constructs and data values from datatypes

```

1 |      case expression0 of      p1 => expression1
2 |                                     | p2 => expression2
3 |                                     | .....
4 |                                     | pn  => expressionn

```

Each pattern is a constructor name followed by the right number of variables.

For example: *Constructor1* or *Constructor2* *x* or *Constructor3* (*x,y*) or ...

Moscow ML: Datatypes Extract

```

1 | datatype TipoEsempio = PAIO of int * int
2 |                       | TESTO of string
3 |                       | LIBERO

```

```

1 | fun extr (x: TipoEsempio) =
2 |     case x of
3 |         LIBERO => 1
4 |         | TESTO s => 2
5 |         | PAIO (i1,i2) => 3
6 | > val extr = fn : TipoEsempio -> int

```

```

1 | extr(PAIO (1,2));
2 | > val it = 3 : int

```

(see Binary Tree Example File)

Impure FP: Ref-Variables, Assignments

- Reference variable points to a value (c.f. indirect addressing):
val <identifier> = **ref** <expression>.
- Assignment: <identifier> := <expression>
- Dereference: !<identifier>

- val x = ref(2+3);	- val y = ref 9;
val x = ref 5 : int ref	val y = ref 9 : int ref
- x := 9;	- !x = !y;
val it = () : unit	val it = true : bool
- x;	- x = y;
val it = ref 9 : int ref	val it = false : bool
- !x;	
val it = 9 : int	

Value Binding and Environment

- The phrase: “**val** $x = 17$ ” is called a value binding; the variable x is bound to the value 17.
- When an identifier is declared by a value binding, a *new* identifier is “created” — it has nothing whatever to do with any previously declared identifier of the same name.
- Once an identifier is bound to a value, there is no way to change that value.
- Environment: the current set of ordered pairs (identifier, value) that are visible.

Environment: Example

env:

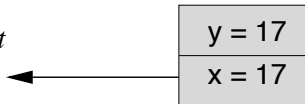
- val x = 17;

val x = 17 : int



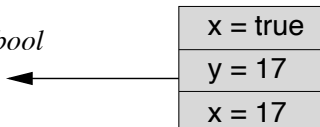
- val y = x;

val y = 17 : int



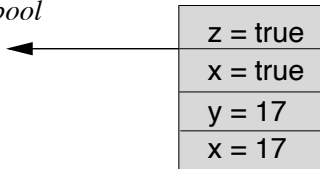
- val x = true;

val x = true : bool



- val z = x;

val z = true : bool



Assignment and Side Effects

- val x = ref 0;

val x = ref 0 : int ref

state: { (x, 0) }

- x := 17;

val it = () : unit

state: { (x, 17) }

- val y = x;

val y = ref 17 : int ref

state: { (x, 17), (y, 17) }

- x := 9;

val it = () : unit

state: { (x, 9), (y, 9) }

- val z = x;

val z = ref 9 : int ref

state: { (x, 9), (y, 9), (z, 9) }

- Notice how the assignment $x := 9$ produces the side-effects such that not only x 's dereferenced value is changed, but also y 's.

Exceptions

See

<http://www.cs.ust.hk/faculty/flin/comp251/ml-exception.txt>.

References

- Standard ML of New Jersey Home Page:
<http://cm.bell-labs.com/cm/cs/what/smlnj/>
- Introduction to ML by R. Harper, postscript file:
<http://www.cs.ust.hk/faculty/flin/comp251/harper.ps>
- Tutorial on running SML at UST:
http://www.cs.ust.hk/faculty/flin/comp251/SML_basics.ps
- SML built-in functions: <http://www.cs.ust.hk/faculty/flin/comp251/ml.functions.html>
- A complete program that sums up numbers in a file:
<http://www.cs.ust.hk/faculty/flin/comp251/sumInts.sml>