

Programming Language Concepts and Semantics Part II

Lecture IX Semantics of ML: Introduction to ML basics

In this lecture we will look at basic programming with ML:

- Expressions
- Value declarations
- Functions
- Recursion
- Lists
- Tuples
- Let expressions
- Patterns and Matches
- Polymorphism
- Equality types

Running ML programs

ML provides an **interactive compiler**. You enter expressions followed by semicolon and ML evaluates them returning their **value** and its type:

```
% sml  
Standard ML of New Jersey, Version 110.0.3, January 30, 1998  
- 2+2;  
val it = 4 : int
```

where `it` is a special variable which is set to the value of any expression typed in interactive mode.

Expressions can take up more than one line:

```
- 3.9 -  
= 3.2;  
val it = 0.7 : real
```

So if you get the prompt `'='`, you haven't finished typing your expression.

One can also load programs. For example, the ML program `prog.ml`. can be loaded by typing

```
- use "prog.ml";
```

Any pathname can appear in the string.

You leave the system by typing control-D.

Expressions

Basic expressions are composed of:

- **integer numbers** (1, 55, 79)
- **real numbers** (0.33, 1e3)
- **booleans** (true, false)
- **strings** ("Hello world\n")
- **characters** ("x", "A")

Arithmetic operations (many **overloaded**) are the:

- low precedence **additive operators**: +, -.
- high precedence **multiplicative operators**:
 - *,
 - / (real division),
 - div (integer division which rounds towards $-\infty$) and
 - mod (integer division remainder).
- highest precedence: ~ (unary minus, yeap, wierd!)

Operations on Booleans are:

- orelse (logical or – like ||)
- andalso (logical and – like &&)
- not (logical negation – like !).

As in C, the first two are not **strict**, that is, they don't have to evaluate their second argument.

The usual comparison operators: =, <, >, <=, >= and <> (not equal) return Booleans and are overloaded, i.e., they work for integers, reals (except = and <>), characters or strings.

Common operations on strings are concatenation (^) and size.

Examples

Basically ML is an up-market calculator: it computes the **value of expressions** and infers their **type**.

```
- 1+2*3;  
  val it = 7 : int  
- 81 mod 10;  
  val it = 1 : int  
- ~(7-5);  
  val it = ~2 : int  
- it + 3;  
  val it = 1 : int  
- 2.0/6.0;  
  val it = 0.333333333333 : real  
- 1e~3  
= ;  
  val it = 0.001 : real  
- 1 < 2 orelse 1 = 2;  
  val it = true : bool  
- 1+1 < 3 andalso 1+1 > 1;  
  val it = true : bool  
- not true;  
  val it = false : bool  
- "Hello\tworld";  
  val it "Hello\tworld" x: string
```

However:

- Reals may not be compared with = or <>. Why?
- For characters and strings < means lexicographically precedes etc. (behaves like strcmp).

Value declarations

Alphanumeric identifiers are formed by either a letter or the character `'` followed by zero or more letters, digits, or symbols `'` and `_`.

Most sorts of objects can be **named** using identifiers: The general form is:

```
val <identifier> = <value>
```

```
- val m = 3;
  val m = 3 : int

- val n = 5;
  val n = 5 : int

- m + n * n;
  val it = 28 : int

- it div 4;
  val it = 7 : int
```

You can give the type of a declaration

```
- val pi = 3.14159:real;
  val pi = 3.14159 : real
```

But you don't need to, ML infers it.

Defining functions

The keyword **fun** indicates a function definition. The general form is:

```
fun <identifier> (<parameter list>) = <expression>;
```

```
- val pi = 3.14159;
  val pi = 3.14159 : real
- fun circle_area(r) = pi * r * r;
  val circle_area = fn : real -> real
- val m = circle_area(4.0);
  val m = 50.26544 : real
```

Parentheses are often unnecessary. We can just type

```
- circle_area 2.0;
  val it = 12.56636 : real
- fun circle_area r = pi * r * r;
  val circle_area = fn : real -> real
```

Write a function to compute the circumference of a circle.

If-then-else

It is similar to C's conditional expression

```
.. ? .. : ..
```

For instance, let us assume that `abs.ml` contains

```
(* computes the absolute value *)  
fun abs r =  
  if r > 0 then r  
  else ~r;
```

Note that the `else` must be there. An example session is:

```
- use "abs.ml";  
  [opening abs.ml]  
  val abs = fn : int -> int  
  val it = () : unit  
- abs 3;  
  val it = 3 : int  
- abs ~3;  
  val it = 3 : int
```

How does it know that `r` is `int`?

What if we wanted `r` to be `real`?

Recursive functions

Unlike C programs, `while` or `for` loops are rare in ML programs. Why?

Instead in ML you use recursion.

Write a function to compute the number of digits in a given positive integer:

```
- fun digits n =  
    if n < 10 then 1  
    else 1 + digits (n div 10);  
  val digits = fn : int -> int  
- digits 7634;  
  val it = 4 : int
```

Note that function application has higher precedence than all other operators so

```
1 + digits n div 10
```

means

```
1 + (digits n) div 10
```

What would happen with this code?

Recursive functions (Cont.)

Write a recursive function to compute the factorial function,

```
fac n = n*(n-1)*...*1.
```

Declaring a Function's Type

Because of **overloaded** built-in functions ML may not be able to determine types or determines an unintended type.

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

By default ML prefers ints to reals.

To make square work on reals we must attach `:real` to one of the three occurrences of `x` or to the overall result.

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

The brackets are necessary for precedence.

Values versus variables

Value names are **not** quite the same as variables in imperative languages.

– they cannot be updated to a new value.

A value name identifies an expression which is conceptually evaluated when the expression is first defined.

The name can be reused but this doesn't change the value defined using the previous definition.

The following merely reuses the name `pi` and `circle_area`:

```
- val pi = 3.14159;
  val pi = 3.14159 : real
- fun circle_area(r) = pi * r * r;
  val circle_area = fn : real -> real
- circle_area 2.0;
  val it = 12.56636 : real
- val pi = 0.0;
  val pi = 0.0 : real
- circle_area 2.0;
  val it = 12.56636 : real
- fun circle_area(r) = pi * r * r;
  val circle_area = fn : real -> real
- circle_area 2.0;
  val it = 0.0 : real
```

You can imagine that names and values are stored on a stack, and the most recent value for the name which is visible to the expression, is used.

Type Coercion and Coercion Functions

Type coercion between basic types is **never** automatic.

```
- 3 + 6.4;
```

```
stdIn:25:1-25:8
```

```
Error: operator and operand don't agree [literal]
```

```
operator domain: int * int
```

```
operand:         int * real
```

```
in expression: 3 + 6.4
```

Built-in arithmetic coercion functions are:

- `real: int -> real`

- `floor: real -> int`

Greatest integer no larger than argument

- `ceil: real -> int`

Smallest integer no smaller than argument

- `round: real -> int`

Closest integer, rounding up (abs value) for .5

- `trunc: real -> int`

Drops digits after the decimal point

For example:

```
- real 3 + 6.4;
```

```
val it = 9.4 : real
```

```
- floor 6.5;
```

```
val it = 6 : int
```

```
- ceil 6.5;
```

```
val it = 7 : int
```

```
- round 6.5;
```

```
val it = 7 : int
```

```
- trunc 6.5;
```

```
val it = 6 : int
```

Coercion Functions (Cont.)

Other built-in coercion functions are:

- `ord: char -> int`
Gives integer code (ASCII) for character
- `chr: int -> char`
Gives character coded by integer (ASCII)
- `str: char -> str.`
Gives single character string containing the character

```
- chr 97;  
  val it = #\"a\" : char  
  
- ord #\"b\";  
  val it = 98 : int  
  
- str (chr 97);  
  val it = \"a\" : string
```

Exercise

Write a function `upper` which takes a lowercase character and returns the corresponding uppercase character. For instance, `upper #\"a\"` returns `#\"A\"`

Using this write a function `toupper` which takes any character and returns the corresponding uppercase character if it is not already uppercase, otherwise returning the same character.

Hint:

```
- ord #\"a\" - ord #\"A\";  
  val it = 32 : int
```

Lists

A list is a finite sequence of elements, all of the same type.

If the element type is 'a, the type of the list is 'a list.
'a is called a **type variable**.

```
- [1.0,2.0,3.0];  
  val it = [1.0,2.0,3.0] : real list  
  
- ["ab", "cde"];  
  val it = ["ab","cde"] : string list  
  
- [[1,2], [], [3]];  
  val it = [[1,2],[],[3]] : int list list  
  
- [[], []];  
  val it = [[],[ ]] : 'a list list
```

Lists are constructed from

- [] (the empty list) and
- :: (the list constructor “cons”).

We can also use nil instead of [].

For example, [1,2] is shorthand for 1::(2::[]).

Exercise: What is [[1,2], [], [3]] shorthand for?

Programming with Lists

The key to programming with lists is to reason recursively about the list.

- Either the list is empty and the operation of interest should be straightforward, or
- it is non-empty and can be broken into a first element and a remaining shorter list which is dealt with recursively.

We can write a function to sum the integers in a list of integers:

What if the list l is empty?

- Return 0.

What if the list l has at list one element x?

- Recursively sum the elements in the rest and add x.

The function can thus be defined as:

```
- fun sumList [] = 0  
  = | sumList (x::xs) = x + (sumList xs);  
  val sumList = fn : int list -> int  
  
- sumList [1,3,5];  
  val it = 9 : int
```

Notice the use of **patterns** (a bit like Prolog):

- an empty list will match pattern []
- a non-empty list will match pattern x::xs

Parity of a list

Write functions `odd l` and `even l` which are true if the number of elements in the list `l` is odd or even.

The case for `odd`:

What if the list `l` is empty? Return `false`.

What if the list `l` is `x::xs`?

– Return true if `even xs` is true and vice versa.

The case for `even`:

What if the list `l` is empty? Return `true`.

What if the list `l` is `x::xs`?

– Return true if `odd xs` is true and vice versa.

Since we have **mutual recursion**, we need to use an `and`

```
fun even [] = true
  | even (_::xs) = odd xs
and
  odd [] = false
  | odd (_::xs) = even xs;

- use "evenodddlist.ml";
  [opening evenodddlist.ml]
  val even = fn : 'a list -> bool
  val odd = fn : 'a list -> bool
  val it = () : unit
```

The underscores are **wildcards** which will match anything.

Programming with Lists (Cont.)

Write a function to return the last element in a list.

What if the list `l` contains one element? Return this element.

What if the list `l` contains more than one element?

– Recursively find the last element of the list tail.

What is the actual code?

```
val last = fn : 'a list -> 'a

- last [1,3,5];
  val it = 5 : int
- last ["abc","def"];
  val it = "def" : string
- last [];
  stdIn:25.1-25.8 Warning: type vars not generalized because
  value restriction are instantiated to dummy types (X1,X2)

  uncaught exception nonexhaustive match failure
  raised at: stdIn:22.24
```

List Operations

To convert between strings and lists, use the built-ins `explode` and `implode`:

```
- explode "Bomb";  
  val it = ["#B",#"o",#"m",#"b"] : char list  
- implode it;  
  val it = "Bomb" : string
```

The built-in infix operator `@` appends lists:

```
- [1,2] @ [4,5];  
  val it = [1,2,4,5] : int list
```

The function `hd` returns the first element of a list:

```
- hd [2,3,5];  
  val it = 2 : int
```

The function `tl` returns the rest of the list:

```
- tl [2,3,5];  
  val it = [3,5] : int list
```

We can use `hd`, `tl` and `null` instead of patterns. However, with lists we usually use **pattern matching**.

Reversing a List

Write a function `reverse l` which returns the list `l` in reverse. You can use `@`.

E.g. `reverse [1,2,3]` should return `[3,2,1]`.

What if the list `l` is empty? Return `[]`.

What if the list `l` is `x::xs`?

- reverse `xs` ;
- add `x` to the end of this.

Actually ML has a library function `rev` to reverse lists.

Tuples

One of the most useful types in ML is the **tuple**. This is an ordered collection of values.

For example

```
- ( 1.0, "abc", 2);  
  val it = (1.0,"abc",2) : real * string * int  
- ( [1.0], ("abc", 2));  
  val it = ([1.0],("abc",2)) : real list * (string * int)
```

Elements in the tuple do not need to have the same type
–like a record or struct but with no name.

A 2-tuple is called a **pair**.

Bizarre Tuples:

- A 1-tuple (pi) is not distinguished from pi.
- There is only one 0-tuple, namely (). (Here we can't drop the parentheses!). This has a type inhabited by () and () alone, namely **unit** (similar to C's "void").

```
- ( 1.0 );  
  val it = 1.0 : real  
- ();  
  val it = () : unit
```

Passing Multiple Arguments to a Function

ML functions only take **one** argument. Tuples allow us to pass (and return) more than one argument to a function.

Write a Boolean function `mem (y,l)` which returns true if `y` is a member of the list `l` and false otherwise.

E.g. `mem (2,[1,2,3])` should return true.

We need to use a tuple because `mem` has two arguments.

What if the list `l` is empty? Return false.

What if the list `l` is `x::xs`? Two cases:

- `y` is the first element `x`;
- `y` is an element of the remainder of the list `xs`.

What is the actual ML function?

Insertion Sort

With tuples and lists we are now in a position to do some real programming!

Write a function `insert (x,ys)` which appropriately inserts integer `x` into the sorted list of integers `ys`.

E.g. `insert (2,[1,3])` should return `[1,2,3]`.

Now write a function `insertSort ys` which returns the list `ys` sorted (e.g., `insertSort [2,1,3]` should return `[1,2,3]`), and does so by using the insertion sort algorithm.

Efficient Reverse

In a previous lecture we gave a definition of a function `reverse` that reversed a list. Unfortunately its time complexity is quadratic in the size of the input list.

It is possible to give a better definition of `reverse` which is linear in the size of the input list.

The idea is quite simple—we construct the reversed list as we traverse the original list, and when we have finished the original list we return this.

The ML program to do this is:

```
(* move elements from the first list to the
   second and then return this list *)
fun rev1 ([], ys) = ys
  | rev1 ((x::xs), ys) = rev1 (xs,(x::ys));

fun reverse xs = rev1 (xs, []);
```

For instance consider reversing `[1,2,3]`

Structuring Data with Tuples

Tuples are not only useful for passing more than one argument to a function, they can also be used to structure data.

Imagine that we wish to build a module for 2-D vectors.
We can represent a vector by a tuple (x, y) .

```
- val zerovec = (0.0, 0.0);  
  val zerovec = (0.0, 0.0) : real * real  
- val a = (3.0, 4.0);  
  val a = (3.0, 4.0) : real * real  
- fun positive (x, y) = x > 0.0 andalso y > 0.0;  
  val positive = fn : real * real -> bool  
- positive a;  
  val it = true : bool
```

A function `addvec` to add two vectors needs to take two vectors which are themselves tuples, i.e. tuples of tuples.

```
- fun addvec ((x1, x2), (y1, y2)) =  
  (x1 + y1, x2 + y2) : real * real;  
  val addvec = fn : (real * real) * (real * real) ->  
    real * real  
- addvec (zerovec, a);  
  val it = (3.0,4.0) : real * real
```

Notice how we use **pattern matching** to access the tuple elements.

Structuring Data with Tuples (Cont.)

Remember to always use brackets when passing arguments to a function which expects a tuple:

```
- addvec zerovec a;  
stdIn:24.1-24.17 Error: operator and operand  
don't agree [tycon mismatch]  
operator domain: (real * real) * (real * real)  
operand:          real * real  
in expression:  
  addvec zerovec
```

Let Expressions

Sometimes we need to create some temporary values
– i.e. local variables.

the `let in end` expression allows you to do this. The general form is:

```
let
  val <first variable> = <first expression>;
  val <second variable> = <second expression>;
  ...
  val <last variable> = <last expression>;
in
  <expression>
end
```

Each variable is visible in all subsequent expressions until end.

```
fun circle_area r =
  let
    val pi = 3.14159;
  in
    pi * r * r
  end;

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

Let Expressions (Cont.)

Let expressions are also useful when a function returns a complex data structure, such as a tuple.

```
fun split [] = ([],[])
  | split [a] = ([a],[])
  | split (a::b::cs) =
    let
      val (M,N) = split(cs)
    in
      (a::M, b::N)
    end;

val split = fn : 'a list -> 'a list * 'a list

- split [1,2,3,4,5];
val it = ([1,3,5],[2,4]) : int list * int list
```

Notice the use of pattern matching in the “variable position.”

Split (cont.)

How does `split` really work? Consider `split [1,2,3]`.

Pattern Matching

We have seen simple pattern matching for both lists and tuples. Patterns and pattern matching can be quite complex.

E.g. does `([1,2,3],5)` match `(x::y::zs,w)`?

Try matching the expression/parse trees for

`(::1,::(2,::(3,[])))`, `5` and `(::(X,::(Y,::Zs)),W)`

Restrictions on Patterns

Variables can only occur once:

```
fun eqlist []           = true
  | eqlist [_]         = true
  | eqlist (x::x::xs) = eqlist (x::xs);
```

stdIn:25.6-27.41

Error: duplicate variable in pattern(s): x

Instead:

```
fun eqlist []           = true
  | eqlist [_]         = true
  | eqlist (x::y::xs) = x=y andalso eqlist (y::xs);
```

Patterns should exhaust all possibilities:

```
fun prod [x] = x : int
  | prod (x::xs) = x * (prod xs);
stdIn:20.1-21.32 Warning: match nonexhaustive
  x :: nil => ...
  x :: xs => ...
```

val prod = fn : int list -> int

Indeed, `prod []` will give a run-time error.

Restrictions on Patterns (Cont.)

Cannot do arithmetic in patterns:

```
fun square(0) = 0
  | square(x+1) = 1 + 2*x + square(x);
```

stdIn:1.5-28.39

Error: non-constructor applied to argument in pattern: +

In general, you cannot match function calls, including `+`, only data constructors.

Exercise: Is the following legal?

```
fun eqlist []           = true
  | eqlist [_]         = true
  | eqlist ([x,y]@xs) = x=y andalso eqlist (y::xs);
```

Up to now we have only seen the following data constructors: `,` for tuples, `nil`, `[]`, and `::` for lists, and the constants for integers, reals, bools, characters, and strings.

The rest (such as `+`, `-`, `@`, etc, are functions). Why can't they be matched?

Naming Patterns – As

Sometimes you wish to match the argument to a pattern but also to have a name for the argument for later use. `as` allows you to do this:

```
fun merge([],M) = M
  | merge(L,[]) = L
  | merge(L as x::xs, M as y::ys) =
    if x < y then x::merge(xs,M)
    else y::merge(L,ys);

val merge = fn : int list * int list -> int list
```

Matches

We have seen **pattern matching** in function definitions. More generally ML provides **matches**. They have form

```
<pattern 1> => <expression1> |
<pattern 2> => <expression2> |
...
<pattern n> => <expressionn>
```

The patterns are tried in order. The result is the expression corresponding to the first pattern matched.

Matches

We can write the function `len`

```
fun len [] = 0
  | len (x::xs) = 1 + len xs;
```

as

```
val rec len = fn
  [] => 0
  | (x::xs) => 1 + len xs;
```

The `rec` indicates that the definition is recursive, i.e. the definition of `len` is in terms of `len`.

More generally

```
fun f P1 = E1 | f P2 = E2 | ... | f Pk = Ek;
```

is short for

```
val rec f = fn P1 => E1 | P2 => E2 | ... | Pk => Ek;
```

Notice that `fn P1 => E1 | P2 => E2 | ... | Pk => Ek` is an expression – it is an **anonymous function**.

Case Expressions

Matches are also used in **case** expressions:

```
case <expression> of <match>
```

```
fun len L =
  case L of
    [] => 0
  | (x::xs) => 1 + len xs;
```

Note that

```
if E1 then E2 else E3
```

is actually just short for

```
case E1 of true => E2 | false => E3
```

Print

Because the programming environment for ML is interactive we have not needed input/output. However, in practical applications programs need to read and write data to and from files.

ML provides a built-in print operator that writes a string to standard output.

```
- print "hello world\n";  
hello world  
val it = () : unit
```

Note the type of print: it always returns the void value (). The fact that has the side effect of printing a string is not reflected in its type.

To print values other than strings, we must use library functions to first convert the value into a string.

```
- fun printReal r = print (Real.toString r);  
val printReal = fn : real -> unit  
- printReal 4.0;  
4.0val it = () : unit
```

Statement Lists

It is often useful to execute a sequence of two or more “statements” with side effects such as print expressions.

In ML this is done using

```
(<first expression> ; ... ; <last expression>)
```

The statements are executed sequentially, much like statements in a C statement block.

However in ML everything is an expression, and the value of a statement list is the value of the last expression in the list.

```
fun printListOfInt nil = ()  
  | printListOfInt (x::xs) = (  
    print (Int.toString x);  
    print "\n";  
    printListOfInt (xs)  
  );
```

```
val printListOfInt = fn : int list -> unit  
- printListOfInt [3,4,5];  
3  
4  
5  
val it = () : unit
```

Do not confuse with let expressions. What are the differences?

Polymorphic Functions

Consider the **identity function**:

```
- fun id x = x;  
  val id = fn : 'a -> 'a
```

This function works on all types of arguments: **reals, lists, functions, etc.**

It is **not overloaded**, it is **polymorphic**. Its type

```
'a -> 'a
```

is a **type scheme**, and **'a** is a **type variable**.

Some more examples:

```
fun len [] = 0  
  | len (x::xs) = 1 + len xs;  
val len = fn : 'a list -> int
```

```
fun reverse [] = []  
  | reverse (x::xs) = (reverse xs) @ [x];  
val reverse = fn : 'a list -> 'a list
```

```
fun fst (x, y) = x;  
val fst = fn : 'a * 'b -> 'a
```

```
- fst (("abc", 7), ("def", 6));  
  val it = ("abc",7) : string * int  
- fst (3.0,1);  
  val it = 3.0 : real
```

Polymorphism

Polymorphic type checking is a secure yet flexible type discipline. Most ML programs need not be cluttered with type specifications, as types are deduced automatically.

ML is strongly typed:

“Well-typed programs cannot go wrong.”

Once the type checker has accepted the program, no type errors can occur at run-time.

(Division by zero is not a “type error!”)

A **polymorphic function** can have different types within the same expression:

```
- len [1.0,2.0] + len ["abc","def"]  
  val it = 4 : int
```

Equality Types

There is a slight problem with polymorphism and equality.

```
- fun mem(x, []) = false
  | mem(x, y::ys) = (x = y) orelse mem(x, ys);
  val mem = fn : 'a * 'a list -> bool
```

If 'a is a function type or a real, mem will want to compare.

```
- mem(3, [1,2,3]);
  val it = true : bool
- mem(3.0, [1.0,2.0,3.0]);
  stdIn:140.1-140.23 Error: operator and operand
    don't agree [equality type required]
  operator domain: 'Z * 'Z list
  operand:         real * real list
  in expression:
    mem (3.0,1.0 :: 2.0 :: <exp> :: <exp>)
```

ML will correctly not allow this since function types and reals are not comparable in ML.

Equality Types (Cont.)

The built-in function = is polymorphic 'a * 'a -> bool.

We don't need to write a special function to test list equality:

```
- [2,3,4] = [2,3,4];
  val it = true : bool
```

The types admitting equality testing are called **equality types**. Type variables ranging over these are 'a, 'b, etc.

Equality testing is possible for most types, including tuples and lists made from equality types.

Equality Types (Cont.)

What is the difference between these three functions? Why?

```
fun len1 [] = 0
  | len1 (x::xs) = 1 + len1 xs;
```

```
val len1 = fn : 'a list -> int
```

```
fun len2 x = if x=[] then 0 else 1 + len2(tl x);
```

```
val len2 = fn : ''a list -> int
```

```
fun len3 x = if null x then 0 else 1 + len3(tl x);
```

```
val len3 = fn : 'a list -> int
```