A First Look at ML

ML

- Meta Language
- One of the more popular functional languages (which, admittedly, isn't saying much)
- Edinburgh, 1974, Robin Milner's group
- There are a number of dialects
- We are using Standard ML, but we will just call it ML from now on

```
Standard ML of New Jersey
- 1+2*3;
val it = 7 : int
- 1+2*3
= ;
val it = 7 : int
```

Type an expression after **-** prompt; ML replies with value and type

After the expression put a **;**. (The **;** is not part of the expression.)

If you forget, the next prompt will be **=**, meaning that ML expects more input. (You can then type the **;** it needs.)

Variable **it** is a special variable that is bound to the value of the expression you type

Outline

- Constants
- Operators
- Defining Variables
- Tuples and Lists
- Defining Functions
- ML Types and Type Annotations

```
- 1234;
val it = 1234 : int
- 123.4;
val it = 123.4 : real
```

Integer constants: standard decimal , but use tilde for unary negation (like **`~1`**)

Real constants: standard decimal notation

Note the type names: **`int`**, **`real`**

```
- true;
val it = true : bool
- false;
val it = false : bool
```

Boolean constants **`true`** and **`false`**

ML is case-sensitive: use **`true`**, not **`True`** or **`TRUE`**

Note type name: **`bool`**

```
- "fred";
val it = "fred" : string
- "H";
val it = "H" : string
- #"H";
val it = #"H" : char
```

String constants: text inside double quotes

Can use C-style escapes: **`\n`**, **`\t`**, **`\\`**, **`\"`**, etc.

Character constants: put **`#`** before a 1-character string

Note type names: **`string`** and **`char`**

Outline

- Constants
- Operators
- Defining Variables
- Tuples and Lists
- Defining Functions
- ML Types and Type Annotations

```
- ~ 1 + 2 - 3 * 4 div 5 mod 6;
val it = ~1 : int
- ~ 1.0 + 2.0 - 3.0 * 4.0 / 5.0;
val it = ~1.4 : real
```

Standard operators for integers, using **~** for unary negation and **-** for binary subtraction

Same operators for reals, but use **/** for division

Left associative, precedence is {**+**,**-**} < {*****,**/**,**div**,**mod**} < {**~**}.

```
- "bibity" ^ "bobity" ^ "boo";
val it = "bibitybobityboo" : string
- 2 < 3;
val it = true : bool
- 1.0 <= 1.0;
val it = true : bool
- #"d" > #"c";
val it = true : bool
- "abce" >= "abd";
val it = false : bool
```

String concatenation: **^** operator

Ordering comparisons: **<**, **>**, **<=**, **>=**, apply to **`string`**, **`char`**, **`int`** and **`real`**

Order on strings and characters is lexicographic

```
- 1 = 2;
val it = false : bool
- true <> false;
val it = true : bool
- 1.3 = 1.3;
Error: operator and operand don't agree
                [equality type required]
  operator domain: ''Z * ''Z
  operand:           real * real
  in expression:
    1.3 = 1.3
```

Equality comparisons: **=** and **<>**

Most types are equality testable: these are *equality types*

Type **`real`** is not an equality type

```
- 1 < 2 orelse 3 > 4;
val it = true : bool
- 1 < 2 andalso not (3 < 4);
val it = false : bool
```

Boolean operators: **`andalso`**, **`orelse`**, **`not`**. (And we can also use **`=`** for equivalence and **`<>`** for exclusive or.)

Precedence so far: {**`orelse`**} < {**`andalso`**} < {**`=`**,**`<>`**,**`<`**,**`>`**,**`<=`**,**`>=`**} < {**`+`**,**`-`**,**`^`**} < {**`*`**,**`/`**,**`div`**,**`mod`**} < {**`~`**,**`not`**}

```
- true orelse 1 div 0 = 0;
val it = true : bool
```

Note: **`andalso`** and **`orelse`** are short-circuiting operators: if the first operand of **`orelse`** is true, the second is not evaluated; likewise if the first operand of **`andalso`** is false

Technically, they are not ML operators, but keywords

All true ML operators evaluate all operands

```
- if 1 < 2 then #"x" else #"y";
val it = #"x" : char
- if 1 > 2 then 34 else 56;
val it = 56 : int
- (if 1 < 2 then 34 else 56) + 1;
val it = 35 : int
```

Conditional expression (not statement) using **`if`** … **`then`** … **`else`** …

Similar to C's ternary operator: **`(1<2) ? 'x' : 'y'`**

Value of the expression is the value of the **`then`** part, if the test part is true, or the value of the **`else`** part otherwise

There is no **`if`** … **`then`** construct

Practice

What is the value and ML type for each of these expressions?

```
1 * 2 + 3 * 4
"abc" ^ "def"
if (1 < 2) then 3.0 else 4.0
1 < 2 orelse (1 div 0) = 0
```

What is wrong with each of these expressions?

```
10 / 5
#"a" = #"b" or 1 = 2
1.0 = 1.0
if (1<2) then 3
```

```
- 1 * 2;
val it = 2 : int
- 1.0 * 2.0;
val it = 2.0 : real
- 1.0 * 2;
Error: operator and operand don't agree
[literal]
  operator domain: real * real
  operand:         real * int
  in expression:
    1.0 * 2
```

The ***** operator, and others like **+** and **<**, are *overloaded* to have one meaning on pairs of integers, and another on pairs of reals

ML does not perform implicit type conversion

```
- real(123);
val it = 123.0 : real
- floor(3.6);
val it = 3 : int
- floor 3.6;
val it = 3 : int
- str #"a";
val it = "a" : string
```

Builtin conversion functions: **`real`** (**`int`** to **`real`**), **`floor`** (**`real`** to **`int`**), **`ceil`** (**`real`** to **`int`**), **`round`** (**`real`** to **`int`**), **`trunc`** (**`real`** to **`int`**), **`ord`** (**`char`** to **`int`**), **`chr`** (**`int`** to **`char`**), **`str`** (**`char`** to **`string`**)

You apply a function to an argument in ML just by putting the function next to the argument. Parentheses around the argument are rarely necessary, and the usual ML style is to omit them

Function Associativity

- Function application is left-associative
- So `f a b` means `(f a) b`, which means:
  - first apply `f` to the single argument `a`;
  - then take the value `f` returns, which should be another function;
  - then apply that function to `b`
- More on how this can be useful later
- For now, just watch out for it

```
- square 2+1;
val it = 5 : int
- square (2+1);
val it = 9 : int
```

Function application has higher precedence than any operator

Be careful!

Practice

What if anything is wrong with each of these expressions?

```
trunc 5
ord "a"
if 0 then 1 else 2
if true then 1 else 2.0
chr(trunc(97.0))
chr(trunc 97.0)
chr trunc 97.0
```

Outline

- Constants
- Operators
- Defining Variables
- Tuples and Lists
- Defining Functions
- ML Types and Type Annotations

```
- val x = 1+2*3;
val x = 7 : int
- x;
val it = 7 : int
- val y = if x = 7 then 1.0 else 2.0;
val y = 1.0 : real
```

Define a new variable and bind it to a value using **`val`**.

Variable names should consist of a letter, followed by zero or more letters, digits, and/or underscores.

```
- val fred = 23;
val fred = 23 : int
- fred;
val it = 23 : int
- val fred = true;
val fred = true : bool
- fred;
val it = true : bool
```

You can define a new variable with the same name as an old one, even using a different type. (This is not particularly useful.)

This is *not the same as assignment*. It defines a new variable but does not change the old one. Any part of the program that was using the first definition of **`fred`**, still is after the second definition is made.

Practice

Suppose we make these ML declarations:

```
val a = "123";
val b = "456";
val c = a ^ b ^ "789";
val a = 3 + 4;
```

Then what is the value and type of each of these expressions?

```
a
b
c
```

The Inside Story

- In interactive mode, ML wants the input to be a sequence of declarations
- If you type just an expression *exp* instead of a declaration, ML treats it as if you had typed:

```
val it = exp;
```

Garbage Collection

- Sometimes the ML interpreter will print a line like this, for no apparent reason:

```
GC #0.0.0.0.1.3:   (0 ms)
```

- This is what ML says when it is performing a “garbage collection”: reclaiming pieces of memory that are no longer being used
- Depending on your installation, you may or may not see these messages
- We’ll see much more about garbage collection when we look at Java
- For now, you can ignore these messages

Outline

- Constants
- Operators
- Defining Variables
- Tuples and Lists
- Defining Functions
- ML Types and Type Annotations

```
- val barney = (1+2, 3.0*4.0, "brown");
val barney = (3,12.0,"brown") : int * real * string
- val point1 = ("red", (300,200));
val point1 = ("red",(300,200)) : string * (int *
int)
- #2 barney;
val it = 12.0 : real
- #1 (#2 point1);
val it = 300 : int
```

Use parentheses to form tuples

Tuples can contain other tuples

A tuple is like a record with no field names

To get i'th element of a tuple x, use **`#i x`**

```
- (1, 2);
val it = (1,2) : int * int
- (1);
val it = 1 : int
- #1 (1, 2);
val it = 1 : int
- #1 (1);
Error: operator and operand don't agree [literal]
  operator domain: {1:'Y; 'Z}
  operand:         int
  in expression:
    (fn {1=1,...} => 1) 1
```

There is no such thing as a tuple of one

Tuple Type Constructor

- ML gives the type of a tuple using `*` as a type constructor
- For example, **`int * bool`** is the type of pairs (x,y) where x is an **`int`** and y is a **`bool`**
- Note that parentheses have structural significance here: **`int * (int * bool)`** is not the same as **`(int * int) * bool`**, and neither is the same as **`int * int * bool`**

```
- [1,2,3];
val it = [1,2,3] : int list
- [1.0,2.0];
val it = [1.0,2.0] : real list
- [true];
val it = [true] : bool list
- [(1,2),(1,3)];
val it = [(1,2),(1,3)] : (int * int) list
- [[1,2,3],[1,2]];
val it = [[1,2,3],[1,2]] : int list list
```

Use square brackets to make lists

Unlike tuples, all elements of a list must be the same type

```
- [];
val it = [] : 'a list
- nil;
val it = [] : 'a list
```

Empty list is **`[]`** or **`nil`**

Note the odd type of the empty list: **`'a list`**

Any variable name beginning with an apostrophe is a *type variable;* it stands for a type that is unknown

`'a list` means *a list of elements, type unknown*

The `null` test

```
- null [];
val it = true : bool
- null [1,2,3];
val it = false : bool
```

- **`null`** tests whether a given list is empty
- You could also use an equality test, as in **`x = []`**
- However, **`null x`** is preferred; we will see why in a moment

List Type Constructor

- ML gives the type of lists using **`list`** as a type constructor
- For example, **`int list`** is the type of lists of things, each of which is of type **`int`**
- A list is not a tuple

```
- [1,2,3]@[4,5,6];
val it = [1,2,3,4,5,6] : int list
```

The @ operator concatenates lists

Operands are two lists of the same type

Note: **`1@[2,3,4]`** is wrong: either use **`[1]@[2,3,4]`** or **`1::[2,3,4]`**

```
- val x = #"c"::[];
val x = [#"c"] : char list
- val y = #"b"::x;
val y = [#"b",#"c"] : char list
- val z = #"a"::y;
val z = [#"a",#"b",#"c"] : char list
```

List-builder (*cons*) operator is `::`

It takes an element of any type, and a list of elements of that same type, and produces a new list by putting the new element on the front of the old list

```
- val z = 1::2::3::[];
val z = [1,2,3] : int list
- hd z;
val it = 1 : int
- tl z;
val it = [2,3] : int list
- tl(tl z);
val it = [3] : int list
- tl(tl(tl z));
val it = [] : int list
```

The `::` operator is right-associative

The **`hd`** function gets the head of a list: the first element

The **`tl`** function gets the tail of a list: the whole list after the first element

```
- explode "hello";
val it = [#"h",#"e",#"l",#"l",#"o"] : char list
- implode [#"h",#"i"];
val it = "hi" : string
```

The **`explode`** function converts a string to a list of characters, and the **`implode`** function does the reverse

Practice

What are the values of these expressions?

```
#2(3,4,5)
hd(1::2::nil)
hd(tl(#2([1,2],[3,4])));
```

What is wrong with the following expressions?

```
1@2
hd(tl(tl [1,2]))
[1]::[2,3]
```

Outline

- Constants
- Operators
- Defining Variables
- Tuples and Lists
- Defining Functions
- ML Types and Type Annotations

```
- fun firstChar s = hd (explode s);
val firstChar = fn : string -> char
- firstChar "abc";
val it = #"a" : char
```

Define a new function and bind it to a variable using **`fun`**

Here **`fn`** means a function, the thing itself, considered separately from any name we've given it. The value of **`firstChar`** is a function whose type is **`string -> char`**

It is rarely necessary to declare any types, since ML infers them. ML can tell that **`s`** must be a **`string`**, since we used **`explode`** on it, and it can tell that the function result must be a **`char`**, since it is the **`hd`** of a **`char list`**

Function Definition Syntax

<*fun-def*> **::=**
fun <*function-name*> <*parameter*> **=** <*expression*> **;**

- <*function-name*> can be any legal ML name
- The simplest <*parameter*> is just a single variable name: the formal parameter of the function
- The <*expression*> is any ML expression; its value is the value the function returns
- This is a subset of ML function definition syntax; more in Chapter 7

Function Type Constructor

- ML gives the type of functions using `->` as a type constructor
- For example, **`int -> real`** is the type of a function that takes an **`int`** parameter (the *domain type*) and produces a **`real`** result (the *range type*)

```
- fun quot(a,b) = a div b;
val quot = fn : int * int -> int
- quot (6,2);
val it = 3 : int
- val pair = (6,2);
val pair = (6,2) : int * int
- quot pair;
val it = 3 : int
```

All ML functions take exactly one parameter

To pass more than one thing, you can pass a tuple

```
- fun fact n =
=   if n = 0 then 1
=   else n * fact(n-1);
val fact = fn : int -> int
- fact 5;
val it = 120 : int
```

Recursive factorial function

```
- fun listsum x =
=   if null x then 0
=   else hd x + listsum(tl x);
val listsum = fn : int list -> int
- listsum [1,2,3,4,5];
val it = 15 : int
```

Recursive function to add up the elements of an **`int list`**

A common pattern: base case for **`null x`**, recursive call on **`tl x`**

```
- fun length x =
=   if null x then 0
=   else 1 + length (tl x);
val length = fn : 'a list -> int
- length [true,false,true];
val it = 3 : int
- length [4.0,3.0,2.0,1.0];
val it = 4 : int
```

Recursive function to compute the length of a list

(This is predefined in ML, so you don't need this definition.)

Note type: this works on any type of list. It is *polymorphic*.

```
- fun badlength x =
=   if x=[] then 0
=   else 1 + badlength (tl x);
val badlength = fn : ''a list -> int
- badlength [true,false,true];
val it = 3 : int
- badlength [4.0,3.0,2.0,1.0];
Error: operator and operand don't agree
  [equality type required]
```

Same as previous example, but with **`x=[]`** instead of **`null x`**

Type variables that begin with two apostrophes, like **`''a`**, are restricted to equality types. ML insists on that restriction because we compared **`x`** for equality with the empty list.

That's why you should use **`null x`** instead of **`x=[]`**. It avoids unnecessary type restrictions.

```
- fun reverse L =
=   if null L then nil
=   else reverse(tl L) @ [hd L];
val reverse = fn : 'a list -> 'a list
- reverse [1,2,3];
val it = [3,2,1] : int list
```

Recursive function to reverse a list

That pattern again

Outline

- Constants
- Operators
- Defining Variables
- Tuples and Lists
- Defining Functions
- ML Types and Type Annotations

ML Types So Far

- So far we have the primitive ML types **`int`**, **`real`**, **`bool`**, **`char`**, and **`string`**
- Also we have three type constructors:
  - Tuple types using **`*`**
  - List types using **`list`**
  - Function types using **`->`**

Combining Constructors

- When combining constructors, **`list`** has higher precedence than **`*`**, and **`->`** has lower precedence
  - **`int * bool list`** same as **`int * (bool list)`**
  - **`int * bool list -> real`** same as **`(int * (bool list)) -> real`**
- Use parentheses as necessary for clarity

```
- fun prod(a,b) = a * b;
val prod = fn : int * int -> int
```

Why **`int`**, rather than **`real`**?

ML's *default type* for **`*`** (and **`+`**, and **`–`**) is
`int * int -> int`

You can give an explicit *type annotation* to get **`real`** instead…

```
- fun prod(a:real,b:real):real = a*b;
val prod = fn : real * real -> real
```

Type annotation is a colon followed by a type

Can appear after any variable or expression

These are all equivalent:

```
fun prod(a,b):real = a * b;
fun prod(a:real,b) = a * b;
fun prod(a,b:real) = a * b;
fun prod(a,b) = (a:real) * b;
fun prod(a,b) = a * b:real;
fun prod(a,b) = (a*b):real;
fun prod((a,b):real * real) = a*b;
```

Summary

- Constants and primitive types: **`int`**, **`real`**, **`bool`**, **`char`**, **`string`**
- Operators: **`~`**, **`+`**, **`-`**, **`*`**, **`div`**, **`mod`**, **`/`**, **`^`**, **`::`**, **`@`**, **`<`**, **`>`**, **`<=`**, **`>=`**, **`=`**, **`<>`**, **`not`**, **`andalso`**, **`orelse`**
- Conditional expression
- Function application
- Predefined functions: **`real`**, **`floor`**, **`ceil`**, **`round`**, **`trunc`**, **`ord`**, **`chr`**, **`str`**, **`hd`**, **`tl`**, **`explode`**, **`implode`**, and **`null`**

Summary, Continued

- Defining new variable bindings using **`val`**
- Tuple construction using **`(x,y,…,z)`** and selection using **`#n`**
- List construction using **`[x,y,…,z]`**
- Type constructors **`*`**, **`list`**, and **`->`**
- Function declaration using **`fun`**, including tuple arguments, polymorphic functions, and recursion
- Type annotations