Declarations and Expressions

An ML program is essentially a collection of **declarations** and **expressions**. (We can think of the "main" program being some particular declaration or function of interest, which makes use of the other declarations.)

Declarations bind **identifiers** to **values**, e.g.,

```
val a = b + 10
```

or to **functions**, e.g.,

```
fun f x = x + a + 10
```

Both values and function bodies are specified as **expressions**.

Every expression and identifier has a **type**. We can (but almost never need to) associate types explicitly with expressions or identifiers using the notation : *type*.

The main kinds of expressions are as follows:

- Constants, e.g.,

```
1:int  3.0:real  "abc":string   true:bool
```

- Identifiers ("variables").

```
x:int  yyyz:bool
underscores_allowed:int  CaseMatters:real
```

Evaluation

ML programs execute by **evaluating** expressions into values.

Since ML is an "eager" language, an expression is evaluated whenever:

- it is bound to an identifier in a declaration; or
- it is specified as an argument to a function or operator application; or
- it is specified as an argument to a data constructor; or
- it is subjected to a conditional or case operation; or
- it is being returned as the value of a function.

Where there's a choice (e.g., in evaluating the arguments to a pair expression), SML always evaluated left-to-right.

The only places an expression **isn't** evaluated "immediately" are when:

- it appears as the body of a function (it is evaluated only when the function is applied); or
- it appears as one arm of a conditional or case expression (it is evaluated only if that arm is selected).

What does it mean to evaluate an expression?

Expressions (continued)

- Constructor applications

```
(1,true) : int * bool     ("a"::"b"::nil) : string list
```

- Function and operator applications

```
(f 2):real      (g(2)):bool      (x + 2):int
```

- Let-bindings

```
(let x : int = 3 + y in x * x + 2 end) : int
```

- Conditionals and case expressions

```
if (x < 0):bool then ~x else x

case (c:int list) of
  h::t => h
| nil => 0
```

- Anonymous functions

```
(fn x => x + 3):int->int
```

Let-bindings

Declarations can appear:

- at top-level, where they become globally visible in all subsequent top-level entries; or
- in `let` expressions.

The purpose of `let` is to restrict the **scope** of a declaration to a limited part of the program.

Often used in functions just like "local variables" in other languages:

```
fun quadroots(a,b,c) =
           let val d = sqrt (b*b - 4.0*a*c)
           in ((~b + d)/(2.0 * a),(~b-d)/(2.0 * a))
           end
```

But a `let` expression can appear **anywhere** an expression can:

```
(let val x = 1 + 2 in x - 3 end)
        + (let val y = 3 + 4 in y + 1 end)
```

A `let` expression is always evaluated by evaluating the RHS of the declaration to a value, binding that value to the declared identifier and then evaluating the body of the expression.

Declaration Lists

A sequence of declarations following the `let` keyword is evaluated in order, just as if it were a nested sequence of `lets`, e.g.:

```
let val a = 10
    val b = a + 1
in a + b
end
```

is just like:

```
let val a = 10
in let val b = a + 1
   in a + b
   end
end
```

When we begin to consider recursion, things will get a little more complicated.

Cases and Conditionals

A **case** expression allows a value of a constructed type (pairs, lists, etc.) to be analyzed into its components.

Each **rule** of a case is specified by giving a **pattern** and an expression to be evaluated if that **pattern** matches the data value being "cased over". The pattern specifies:

- Which data constructor was used to construct the value
- Identifier names to be bound to the subcomponents of the value

```
fun headplus1 (c:int list) =
  case c of
    x::y => x + 1
  | nil => 1
```

The first rule matches list values constructed with `::` (i.e., non-empty ones), and binds the head element of the list to `x` and the remainder of the list to `y`. These variables can then be used on the right-hand side of the matching rule (only).

The second rule matches list values constructed with `nil` (i.e., empty ones); since such values have no subcomponents, there are no variables in the pattern.

We'll see much more sophisticated patterns later.

Scope in the Interactive System

Declarations entered at the read-eval-print loop are semantically like nested `let` declarations, where the scope of the declaration extends indefinitely (until the end of the interactive session). E.g.,

```
val a = 10;
fun f x = x + a + 1;
val a = 20;
f 2;
...
```

is equivalent to

```
let val a = 10
in let fun f x = x + a + 1
   in let val a = 20
      in let val it = f 2
         in ...
         end
      end
    end
end
```

Or, more compactly:

```
let val a = 10
    fun f x = x + a + 1
    val a = 20
    val it = f 2
in ...
end
```

Conditionals; Derived Forms

Conditional expressions analyze a boolean-valued expression and, depending on the outcome, evaluate one of two sub-expressions:

```
if a < 7 andalso b > 4
       then a + b
       else 84
```

Note that **both** `then` and `else` expressions must always be specified in order that the `if` expression is given a value as a whole.

In fact, the `if` expression is really just a (syntactic) shorthand for a `case` expression over the boolean type. E.g.,

```
case (a < 7 andalso b > 4) of
        true => a + b
      | false => 84
```

This is an example of a **derived form**: a piece of source-language syntax that is defined (in the language reference manual) by macro-expansion into **core** language syntax.

You shouldn't need to worry about whether some syntax is core or derived. Unfortunately, you sometimes do, because compiler error messages are reported in terms of the core syntax translation.

Conjunction and Disjunction

The boolean operators `andalso` (**not** `and` !) and `orelse` are also derived forms for case expressions, e.g.,

e_1 `andalso` e_2
$\equiv$ `if` e_1 `then` e_2 `else false`
$\equiv$
`case` e_1 `of`
  `true =>` e_2
`| false => false`

This definition makes it clear that `andalso` is a "short-circuiting" operator; e_2 is evaluated only if e_1 is known to be true; so is `orelse`.

We'll shortly see that most operators in ML are just like functions. Why can't `andalso` and `orelse` be?

A Crucial Fact

Only the values is scope at the point of the function definition matter. Subsequent redefinitions of the the identifier are irrelevant!:

```
- let fun f x = x + a + 1
  in let val a = 20
     in (f 1, f 2)
     end
  end;
std_in:2.19 Error: unbound variable or constructor: a
```

```
- let val a = 10
  in let fun f x = x + a + 1
     in let val a = 20
        in (f 1, f 2)
        end
     end
 end;
> val it = (12,13) : int * int
```

Function Application and Scope

As in other languages, function applications are evaluated by:

- evaluating the actual argument;
- binding the resulting values to the formal parameter of the function;
- evaluating the body of the function with those bindings in effect; and
- returning the value of that body expression as the function result.

```
- let fun f x = x + 1
  in (f 1, f 2)
  end;
> val it = (2,3) : int * int
```

What if a function body mentions a non-local identifier (a **free variable**)? Such an identifier must be in scope, with a value, at the point where the function is **defined**:

```
- let val a = 10
  in let fun f x = x + a + 1
     in (f 1, f 2)
     end
  end;
> val it = (12,13) : int * int
```

Operators

The standard arithmetic, string, and logical operators are just "built-in" functions; they generally obey all the same rules as user-defined functions.

Most of the standard binary operators are **infix**, so that you write them between their operands, rather than in usual **prefix** style for function application, e.g.

```
2 + 2
```

instead of

```
plus(2,2)
```

Actually, **any** function can be made **infix** by an appropriate declaration, and its associativity and precedence can also be declared. E.g., the "built-in" declarations for the integers are:

```
infix 7 * div
infix 6 + -
```

Once an symbol is declared infix, it can be still be used in a prefix fashion by adding the keyword `op`, e.g., can write

```
(op+)(2,2)
```

Anonymous functions

`fun` declarations bind functions to names. ML also has **function expressions** which allow you to define **anonymous** functions.

For example:

```
fn x => x + 1
```

is the (anonymous) function that adds `1` to its argument.

We can use a function expression wherever a function name would make sense, e.g.,

```
- (fn x => x + 1) 2;
> val it = 3 : int
```

or, a little more curiously,

```
- (if 8 > 3 then fn y => y + 7 else fn z => z + 2) 2;
> val it = 9 : int
```

Naturally, function expressions can also be bound to names, e.g.,

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

Anonymous functions (continued)

In fact, the usual function declaration syntax (using `fun`)

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

is just a derived form for the `fn` binding above.

Note: `fn` ≢ `fun` !!

We'll see good uses for anonymous functions later.

Meanwhile, what should it mean to **evaluate** a function expression?

Clearly, the body of the function should not be evaluated (this happens only later, when the function is **applied**).

One thing that does happen (in the underlying implementation) is that the values of any free variables in the body are **recorded** for use when the function is later applied. You don't need to worry about this explicitly, but it can be handy to remember when you're trying to figure out how free variables work!