Introduction to Functional Languages

What does **functional** mean? Two main senses:

• Programs consist of functions with **no side effects**.

• Functions are supported as **"first-class" values**.

Grand Claim: functional programs are:

• clearer;

• easier to get right;

• easier to test;

• easier to transform;

• easier to parallelize;

• easier to prove things about;

• more "fun."

Important examples:

• Lisp, Scheme ("strict", dynamically typed, impure)

• Standard ML, CAML ("strict", statically typed, impure)

• Haskell, Gofer ("lazy", statically typed, pure)

What is a function?

Abstractly, a function is just a **mapping** from a set of possible input values (the **domain**) to a set of possible output values (the **co-domain**).

Function can be specified **intensionally**, by means of a rule that describes **how** to calculate the output for a given input, or **extensionally**, by means of a (perhaps infinite) table that describes **what** the output is for any given input.

Both views are extremely important. Intensional view allows us to program a machine; extensional view allows us to **abstract** away from the details of the program by focusing on the externally visible behavior.

```
Extensional spec:        Possible Intensional specs:

(x,f(x))                 f(x) = x * 2
--------
(1,2)                    f(x) = x + x
(2,4)
(3,6)                    f(x) = sqrt(x^4) / x + (1/(1/x))
(4,8)
...
```

Functional languages allow us to manipulate functions as just another kind of data. This often allows us to take a much **higher-level, declarative** approach to programming problems.

Functional Architecture

Programs are constructed by combining **functions** via **composition**; the output of one function is passed to the input of another.

Since there are no side-effects, all interaction between program components must be by passing explicit arguments and results. There are **no** shared variables or common data areas.

This means each function can stand on its own. It can be **tested** and debugged independently of any other functions.

(Also, since it's a pain to communicate information explicitly this way, programmers have an incentive to avoid unnecessary **coupling** between components – a sound software engineering goal.)

Standard ML

Standard ML is one of the best-developed examples of a functional language.

• Name originally derived from "**M**eta **L**anguage;" developed in Edinburgh in early 1980's.

• Uses so-called "eager" evaluation, like LISP/Scheme; more efficient (though less powerful) than so-called "lazy" languages such as Haskell.

• Strongly typed at compile time, unlike LISP/Scheme.

• Not doctrinaire; supports some side-effecting features for I/O, updatable store.

• Has good compilers and interpreters, including **SML of New Jersey** and **Moscow ML**.

• Has a full formal operational semantics.

Interactive System

SML provides a interactive "read-eval-print" loop for incremental program development.

You type value or function definitions; they are type-checked and compiled or interpreted, and are then usable in further definitions.

In Standard ML of New Jersey:

```
% sml
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- val a = 3;
val a = 3 : int
- val b = "hello";
val b = "hello" : string
- val c = a + 1;
val c = 4 : int
- val d = b + 1;
std_in:3.1-3.13 Error: operator and operand don't agree
                                          (tycon mismatch)
  operator domain: string * string
  operand:         string * int
  in expression:
    + : overloaded (b,1)
std_in:3.11 Error: overloaded variable not defined at type
  symbol: +
  type: string
- val b = c < 5;
val b = true : bool
- 2 + 2;
val it = 4 : int
```

Interactive System (continued)

Note that these are declarations binding **constants** to identifiers; they are **not** assignment statements! Redefining an identifier completely replaces the old binding.

In Moscow ML, similar except for form of type error message:

```
...
- val d = b + 1;
! Toplevel input:
! val d = b + 1;
!           ^
! Type clash: expression of type
!   int
! cannot be made to have type
!   string
```

Notice that, unless there is a type inconsistency, the system was able to **infer** the types of the declared identifiers **automatically**. In fact, SML almost never needs you to specify a type explicitly, although you always may:

```
- val c : int  = a + 1;
val c = 1 : int
```

Loading from Files

Of course, it's often handier to write code using an auxiliary editor. You can then compile your code using copy-and-paste, or via the `use` command. For example, if `fred.sml` contains the lines:

```
val a = 10;
val b = a + 20;
```

then we can load these definitions into the read-eval-print loop as follows:

```
% sml
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- use "fred.sml";
[opening fred.sml]
val a = 10 : int
val b = 30 : int
val it = () : unit
```

Note that the contents of the file are not echoed; just the "results."

Functions

Functions are (usually) defined using the `fun` keyword:

```
- fun addone x = x + 1;
val addone = fn : int -> int
- val a = addone ~1;
val a = 0 : int
- val b = addone a;
val b = 1 : int
```

Calling ("applying") a function is specified just by writing the function name followed by the argument; no parentheses are needed (though they are generally harmless).

All ML functions take exactly one argument; we'll see ways to get the effect of multiple arguments later.

Function Typing

Function definitions and applications must be well-typed too:

```
- fun g x = (x + 1) + true;
std_in:0.0-0.0 Error: operator and operand don't agree
                                          (tycon mismatch)
  operator domain: int * int
  operand:         int * bool
  in expression:
    + : overloaded ((+ : overloaded (<exp>,<exp>)),true)
- val c = addone "hello";
std_in:15.1-15.22 Error: operator and operand don't agree
                                          (tycon mismatch)
  operator domain: int
  operand:         string
  in expression:
    addone "hello"
```

Note that faulty function definitions are flagged as soon as the function is **defined**, rather than waiting until it is applied.

Pairs and Tuples

SML has a built-in type of **pairs**. You can pair together any two values. In fact, SML supports triples, quads, ..., arbitrary n-tuples.

```
- val z = (4,b);
val z = (4, "hello") : int * string
- val y = ("goodbye",not q,3.0);
val y = ("goodbye",false.3.0) : string * bool * real
```

Useful for passing **multiple arguments** to a function, or returning multiple results.

```
- fun calc (a,b,c) = (a + b + c + 10, a - b - c - 10);
val calc = fn : int * int * int -> int * int
- calc (1,2,3);
val it = (16,~14) : int * int
```

In fact, the built-in binary operators like `+` are really just pre-defined functions that take a pair argument: `a+b` is a shorthand for `(op +)(a,b)`.

Lists

SML also has a built-in data type of **lists**, i.e., sequences of zero or more values. You can make lists of anything, but everything in a given list must have the same type.

Lists have two interchangeable representations. The first derives from the **recursive** definition of what a list can be:

- A list can be **empty**, in which case it is represented by `nil`.
- A list can be formed by adding a value `x` onto the head of an existing list `y`, in which case it is represented by `x::y`, read "x cons y".

We can **construct** any list this way:

```
- val a = nil;
val a = [] : 'a list
- val b = 1::(2::a);
val b = [1,2] : int list
- val c = 3::4::b;
val c = [3,4,1,2] : int list
- val d = 5::6;
std_in:5.1-5.12 Error: operator and operand don't agree
                                               (tycon mismatch)
  operator domain: int * int list
  operand:         int * int
  in expression:
    :: (5,6)
```

Lists (continued)

The other way to write a list is to write the elements within square brackets, separated by commas, as illustrated in the values echoed by the interactive system. This is just a shorthand; in general,

$[a_1, a_2, \ldots, a_n] \equiv a_1 :: a_2 :: \ldots :: a_n :: nil$

We can also use `::` and `nil` as **patterns** in `case` expressions that "deconstruct" or **analyze** the contents of a list to obtain its constituent components.

```
- fun f c =
  case c of
    h::t => h + 1
  | nil => 0;
> val f = fn : int list -> int
- val a = f [3,2,1];
> val a = 4 : int
- val b = f [];
> val b = 0;
```

Recursive and Polymorphic Functions

Most useful functions on lists are **recursive**:

```
- fun length c =
  case c of
    h::t => 1 + (length t)
  | nil => 0;
> val length = fn : 'a list -> int
- length [1,2,3];
> val it = 3 : int;
- length [true,false,true,true];
> val it = 4 : int;
```

SML actually supports arbitrary user-defined recursive datatypes including queues, trees, etc.; the built-in `list` type is just a special case.

The `'a` in the type of `length` is a **type variable**: it indicates that the function can be applied to lists of **any** kind, as illustrated. This powerful feature is called **polymorphism**; it helps enable **code re-use**.

Higher-order Functions

Here's another function on lists:

```
- fun sum c =
  case c of
    h::t => h + (sum t)
  | nil => 0
> val sum = fn : int list -> int
- sum [1,2,3];
> val it = 6 : int;
```

Note the similarity in form to `length`. We can take advantage of this similarity to **abstract** the recursive pattern common to these functions and write them both as instances of a single **higher-order** function:

```
- fun foldr(f,a,c) =
    case c of
      h::t => f(h,foldr(f,a,t))
     | nil => a;
> val foldr = fn : ('a * 'b -> 'b) * 'b * 'a list -> 'b
- fun sum c = foldr (op+,0,c);
> val sum = fn : int list -> int
- fun addone (x,s) = s + 1;
> val addone = fn : 'a * int -> int
- fun length c = foldr(addone,0,c);
> val length = fn : 'a list -> int
```