A Look at Functional Programming with Standard ML

TCS Developer's SIG
September 5, 2006

William Mitchell (whm)
Mitchell Software Engineering (.com)

Introduction

Imperative programming

Functional programming

Imperative Programming

Languages such as C, Pascal, and FORTRAN support programming in an *imperative* style.

Two fundamental characteristics of imperative languages:

"Variables"—data objects whose contents can be changed.

Support for iteration—a “while” control structure, for example.

Java supports object-oriented programming but methods are written in an imperative style.

Here is an imperative solution in Java to sum the integers from 1 to N:

```
int sum(int n)
{
    int sum = 0;

    for (int i = 1; i <= n; i++)
        sum += i;
    return sum;
}
```

Functional Programming

Functional programming is based on mathematical functions, which:

- Map values from a domain set into values in a range set
- Can be combined to produce more powerful functions
- Have no side effects

Functional programming, continued

Recall the imperative solution to sum 1...N:

```
int sum(int n)
{
    int sum = 0;

    for (int i = 1; i <= n; i++)
        sum += i;
    return sum;
}
```

A solution in a functional style using recursion:

```
int sum(int n)
{
    if (n == 1)
        return 1;
    else
        return n + sum(n - 1);
}
```

Note that there is no assignment or looping.

ML Basics

A little background

Interacting with ML

`val` declarations

Simple data types and operators

The conditional expression

ML—Background

Developed at Edinburgh University in the mid '70s by Mike Gordon, Robin Milner, and Chris Wadsworth.

Designed specifically for writing proof strategies for the Edinburgh LCF *theorem prover*. A particular goal was to have an excellent datatype system.

The name “ML” stands for "Meta Language".

ML is not a pure functional language. It does have some imperative features.

There is a family of languages based on ML. These slides use Standard ML of New Jersey (SML/NJ).

The SML/NJ home page is www.smlnj.org.

ML is not object-oriented

ML is not designed for object-oriented programming:

- There is no analog for the common notion of a class.
- Executable code is contained in functions, which may be associated with a structure but are often "free floating".
- Instead of "invoking a method" or "sending a message to an object", we "call functions".

  Example: A Java expression such as `xList.f(y)` might be expressed as `f(xList, y)` in ML.

- There is no notion of inheritance but ML does support polymorphism in various ways.

The OCaml (Objective Caml) language is derived from ML and has support for object-oriented programming.

Interacting with ML

SML/NJ is has a "read-eval-print" loop:

```
% sml
Standard ML of New Jersey, v110.57...
- 10 + ~20;
val it = ~10 : int

- 3.4 * 5.6 - 7.8;
val it = 11.24 : real

- size("testing");
val it = 7 : int

- "apple" <> "orange";
val it = true : bool

- Math.sqrt(2.0);           (* the sqrt function in the Math "structure" *)
val it = 1.41421356237 : real

- Math.pi;
val it = 3.14159265359 : real
```

Naming values—the val declaration

A *val declaration* can be used to specify a name for the value of an expression. The name can then be used to refer to the value.

```
- val radius = 2.0;
val radius = 2.0 : real

- radius;
val it = 2.0 : real

- val area = Math.pi * radius * radius;
val area = 12.5663706144 : real

- area;
val it = 12.5663706144 : real
```

Do not think of val as creating a variable.

It can be said that the above adds bindings for radius and area to our *environment*.

val declarations, continued

It is not an error to use an existing name in a subsequent val declaration:

```
- val x = 1;
val x = 1 : int

- val x = "abc";
val x = "abc" : string

- x;
val it = "abc" : string
```

Technically, the environment contains two bindings for x but only the latter binding is accessible.

The conditional expression

ML has an if-then-else construct. Example:

```
- if 1 < 2 then 3 else 4;
val it = 3 : int
```

This construct is called the "conditional expression".

It evaluates a boolean expression and then depending on the result, evaluates the expression on the then "arm" or the else "arm". The value of that expression becomes the result of the conditional expression.

A conditional expression can be used anywhere an expression can be used:

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

- x + (if x < 5 then x*x else x+x);
val it = 12 : int
```

How does ML's if-then-else compare to that in the C family (C / C++ / Java /C#)?

The char type

It is possible to make a one-character string but there is also a separate type, char, to represent single characters.

A char literal consists of a pound sign followed by a single character, or escape sequence, enclosed in double-quotes. Example, and some char-related functions:

```
- #"a";
val it = #"a" : char

- ord(#"b");
val it = 98 : int

- str(chr(97)) ^ str(chr(98));   (* ^ (caret) is string concatenation *)
val it = "ab" : string
```

Functions

Type consistency

Defining functions

Type deduction

Type variables

Loading source with `use`

A prelude to functions: type consistency

ML requires that expressions be *type consistent*. A simple violation is to try to add a real and an int:

```
- 3.4 + 5;
Error: operator and operand don't agree (tycon mismatch)
  operator domain: real * real
  operand:         real * int
  in expression:
    + : overloaded (3.4,5)
```

(tycon stands for "type constructor".)

Type consistency is a cornerstone of the design philosophy of ML.

There are no automatic type conversions in ML.

Type consistency, continued

Another context where type consistency appears is in the conditional operator: the expressions in both "arms" must have the same type.

Example:

```
- if "a" < "b" then 3 else 4.0;
Error: rules don't agree    (tycon mismatch)
  expected: bool -> int
  found:    bool -> real
  rule:
    false => 4.0
```

Function definition basics

A simple function definition:

```
- fun double(n) = n * 2;
val double = fn : int -> int
```

The body of a function is a single expression. The return value of the function is the value of that expression. There is no “return” statement.

The text "fn" is used to indicate that the value defined is a function, but the function itself is not displayed.

The text "int -> int" indicates that the function takes an integer argument and produces an integer result. ("->" is read as "to".)

Note that the type of the argument and the type produced by the function are not specified. Instead, *type deduction* was used.

Function definition basics, continued

Another example of type deduction:

```
- fun f(a, b, c, d) =
    if a = b then c + 1 else
      if a > b then c else b + d;
val f = fn : int * int * int * int -> int
```

The type of a function is described with a *type expression*.

The symbols * and -> are both *type operators*. * is left-associative and has higher precedence than ->, which is right-associative.

The type operator * is read as “cross”.

What is a possible sequence of steps used to determine the type of f?

Type variables and polymorphic functions

In some cases ML expresses the type of a function using one or more *type variables*.

A type variable expresses type equivalences among parameters and between parameters and the return value.

A function that simply returns its argument:

```
- fun f(a) = a;
val f = fn : 'a -> 'a
```

The identifier 'a is a type variable. The type of the function indicates that it takes a parameter of any type and returns a value of that same type, whatever it is.

```
- f(1);
val it = 1 : int
- f(1.0);
val it = 1.0 : real
- f("x");
val it = "x" : string
```

'a is read as "alpha", 'b as "beta", etc.

Type variables and polymorphic functions, continued

At hand:

```
- fun f(a) = a;
val f = fn : 'a -> 'a
```

The function f is said to be *polymorphic* because it can operate on a value of any type.

A polymorphic function may have many type variables:

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

- third(1, 2.0, "three");
val it = "three" : string
```

A function's type may be a combination of fixed types and type variables. A single type variable is sufficient for a function to be considered polymorphic.

A polymorphic function has an infinite number of possible *instances*.

Equality types

An *equality type variable* is a type variable that ranges over *equality types*. Instances of values of equality types, such as int, string, and char can be tested for equality. Example:

```
- fun equal(a,b) = a = b;
val equal = fn : ''a * ''a -> bool
```

The function equal can be called with any type that can be tested for equality. ''a is an equality type variable, distinguished by the presence of two apostrophes, instead of just one.

```
- equal(1,10);
val it = false : bool

- equal("xy", "x" ^ "y");
val it = true : bool
```

More-interesting types

Tuples

Pattern matching

Lists

List processing functions

Tuples

A *tuple* is an ordered aggregation of two or more values of possibly differing types.

```
- val a = (1, 2.0, "three");
val a = (1,2.0,"three") : int * real * string

- (1, 1);
val it = (1,1) : int * int

- (it, it);
val it = ((1,1),(1,1)) : (int * int) * (int * int)

- ((1,1), "x", (2.0,2.0));
val it = ((1,1),"x",(2.0,2.0)) : (int * int) * string * (real * real)
```

Tuples, continued

A function can return a tuple as its result:

```
- fun around(n) = (n-1, n+1);
val around = fn : int -> int * int

- around(5);
val it = (4,6) : int * int

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

- pair(1, "one");
val it = (1,"one") : int * string

- pair(it, it);
val it = ((1,"one"),(1,"one")): (int * string) * (int * string)
```

Tuples, continued

A function to put two integers in ascending order:

```
- fun order(x, y) = if x < y then (x, y) else (y, x);
val order = fn : int * int -> int * int

- order(3,4);
val it = (3,4) : int * int

- order(10,1);
val it = (1,10) : int * int
```

Pattern matching

Thus far, function parameter lists appear conventional but in fact the "parameter list" is a pattern specification.

Recall order:

```
fun order(x, y) =  if x < y then (x, y) else (y, x)
```

In fact, order has only one parameter: an (int * int) tuple.

The pattern specification (x, y) indicates that the name x is bound to the first value of the two-tuple that order is called with. The name y is bound to the second value.

Consider this:

```
- val x = (7, 3);
val x = (7,3) : int * int

- order x;
val it = (3,7) : int * int
```

Pattern matching, continued

In fact, all functions in our current ML world take one argument!

The syntax for a function call in ML is this:

function value

In other words, two values side by side are considered to be a function call.

Examples:

```
- val x = (7,3);
val x = (7,3) : int * int

- swap x;
val it = (3,7) : int * int

- size "testing";
val it = 7 : int
```

Pattern matching, continued

Patterns provide a way to bind names to components of values.

Imagine a 2x2 matrix represented by a pair of 2-tuples:

```
- val m = ((1, 2),
           (3, 4));
val m = ((1,2),(3,4)) : (int * int) * (int * int)
```

Elements of the matrix can be extracted with pattern matching:

```
- fun lowerRight((ul,ur),(ll,lr)) = lr;
val lowerRight = fn : ('a * 'b) * ('c * 'd) -> 'd

- lowerRight m;
val it = 4 : int
```

Underscores can be used in a pattern to match values of no interest. An underscore creates an *anonymous binding*.

```
- fun upperLeft ( (x, _), (_,  _) ) = x;
val upperLeft = fn : ('a * 'b) * ('c * 'd) -> 'a
```

Pattern matching, continued

The left hand side of a val expression is in fact a pattern specification:

```
- val (x, y, z) = (1, (2, 3),  (4, ("five", 6.0)));
val x = 1 : int
val y = (2,3) : int * int
val z = (4,("five",6.0)) : int * (string * real)
```

Pattern matching, continued

Functions may be defined using a series of patterns that are tested in turn against the argument value. If a match is found, the corresponding expression is evaluated to produce the result of the call. Also, literal values can be used in a pattern.

```
- fun f(1) = 10
    |  f(2) = 20
    |  f(n) = n;
val f = fn : int -> int
```

Usage:

```
- f(1);
val it = 10 : int

- f(2);
val it = 20 : int

- f(3);
val it = 3 : int
```

Pattern matching, continued

One way to sum the integers from 0 through N:

```
fun sum(n) = if n = 0 then 0 else n + sum(n-1);
```

A better way:

```
fun sum(0) = 0
  | sum(n) = n + sum(n - 1);
```

Lists

A list is an ordered collection of values of the same type.

One way to make a list is to enclose a sequence of values in square brackets:

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

- ["just", "a", "test"];
val it = ["just","a","test"] : string list

- [it, nil, [ ], it];   (* nil and [ ] are two ways to specify an empty list *)
val it = [["just","a","test"],[],[],["just","a","test"]] : string list list

- [(1, "one"), (2, "two")];
val it = [(1,"one"),(2,"two")] : (int * string) list
```

Note the type, int list, for example. list is another type operator. It has higher precedence than both * and ->.

Heads and tails

The hd and tl functions produce the *head* and *tail* of a list, respectively. The head is the first element. The tail is the list without the first element.

```
- val x = [1, 2, 3, 4];
val x = [1,2,3,4] : int list

- hd x;
val it = 1 : int

- tl x;
val it = [2,3,4] : int list

- tl it;
val it = [3,4] : int list

- hd( tl( tl x));
val it = 3 : int
```

Both hd and tl, as all functions in our ML world, are *applicative*. They produce a value but don't change their argument.

An important point

- Lists can't be modified. There is no way to add or remove list elements, or change the value of an element. (Ditto for tuples and strings.)

- Ponder this: Just as you cannot change an integer's value, you cannot change the value of a string, list, or tuple.

In ML, we never change anything. We only make new things.[1]

[1] Unless we use the imperative features of ML, which we won't be studying!

Simple functions with lists

The built-in length function produces the number of elements in a list:

```
- length [20, 10, 30];
val it = 3 : int

- length [ ];
val it = 0 : int
```

Problem: Write a function len that behaves like length. What type will it have?

Here's a start... (Note the use of a case instead of an "if".)

```
fun len [ ] = 0
  |   len L =
```

Problem: Write a function sum that calculates the sum of the integers in a list:

```
- sum([4,1,2,3]);
val it = 10 : int
```

Cons'ing up lists

A list may be constructed with the :: ("cons") operator, which forms a list from a compatible head and tail:

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

- 1::2::3::[ ];
val it = [1,2,3] : int list

- "x"::nil;
val it = ["x"] : string list
```

What's an example of an incompatible head and tail?

Note the type of the operator:

```
- op:: ;     (Just prefix the operator with "op")
val it = fn : 'a * 'a list -> 'a list
```

This operator is right associative.

Cons, continued

Problem: Write a function m_to_n(m, n) that produces a list of the integers from m through m inclusive. (Assume that m <= n.)

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

- m_to_n(~3, 3);
val it = [~3,~2,~1,0,1,2,3] : int list

- m_to_n(1, 0);
val it = [ ] : int list
```

A start:

```
fun m_to_n(m, n) = if m > n then [ ] else
```

Lists, strings, and characters

The explode and implode functions convert between strings and lists of characters:

```
- explode("boom");
val it = [#"b", #"o", #"o", #"m"] : char list

- implode([#"o", #"o", #"p", #"s", #"!"]);
val it = "oops!" : string

- explode("");
val it = [ ] : char list

- implode([ ]);
val it = "" : string
```

What are the types of implode and explode?

Problem: Write a function reverse(s), which reverses the string s. Hint: rev reverses a list.

Pattern matching with lists

In a pattern, :: can be used to describe a value. Example:

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

The first pattern is the basis case and matches an empty list.

The second pattern requires a list with at least one element. The head is bound to x and the tail is bound to xs.

Problem: Noting that x is never used, improve the above implementation.

Problem: Write a function drop2(L) that returns a copy of L with the first two values removed. If the length of L is less than 2, return L.

Try it!

Problem: Write a function member(v, L) that produces true iff v is contained in the list L.

```
- member(7,  [3, 7, 15]);
val it = true : bool
```

Problem: Write a function contains(s, c) that produces true iff the char c appears in the string s.

Problem: Write a function maxint(L) that produces the largest integer in the list L. Raise the exception Empty if the list has no elements.

Larger Examples

expand

travel

tally

expand

Consider a function that expands a string in a trivial packed representation:

```
- expand("x3y4z");
val it = "xyyyzzzz" : string

- expand("123456");
val it = "244466666" : string
```

Fact: The digits 0 through 9 have the ASCII codes 48 through 57. A character can be converted to an integer by subtracting from it the ASCII code for 0. Therefore,

```
fun ctoi(c) = ord(c) - ord(#"0")

fun is_digit(c) = #"0" <= c andalso c <= #"9"

- ctoi(#"5");
val it = 5 : int

- is_digit(#"x");
val it = false : bool
```

expand, continued

One more function:

```
fun repl(x, 0) = []
  | repl(x, n) = x::repl(x, n-1)
```

What does it do?

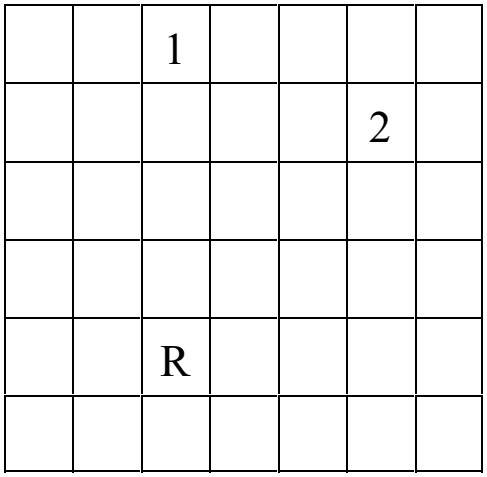
Finally, expand:

```
fun expand(s) =
    let
        fun expand'([ ]) = [ ]
          | expand'([c]) = [c]
          | expand'(c1::c2::cs) =
              if is_digit(c1) then
                  repl(c2, ctoi(c1)) @ expand'(cs)
              else
                  c1 :: expand'(c2::cs)
    in
        implode(expand'(explode(s)))
    end;
```

travel

Imagine a robot that travels on an infinite grid of cells. The robot's movement is directed by a series of one character commands: n, e, s, and w.

In this problem we will consider a function travel of type string -> string that moves the robot about the grid and determines if the robot ends up where it started (i.e., did it get home?) or elsewhere (did it get lost?).



If the robot starts in square R the command string nnnn leaves the robot in the square marked 1. The string nenene leaves the robot in the square marked 2. nnessw and news move the robot in a round-trip that returns it to square R.

travel, continued

Usage:

```
- travel("nnnn");
val it = "Got lost" : string

- travel("nnessw");
val it = "Got home" : string
```

How can we approach this problem?

travel, continued

One approach:

1. Map letters into integer 2-tuples representing X and Y displacements on a Cartesian plane.

2. Sum the X and Y displacements to yield a net displacement.

Example:

Argument value:	"nnee"
Mapped to tuples:	(0,1) (0,1) (1,0) (1,0)
Sum of tuples:	(2,2)

Another:

Argument value:	"nnessw"
Mapped to tuples:	(0,1) (0,1) (1,0) (0,-1) (0,-1) (-1,0)
Sum of tuples:	(0,0)

travel, continued

A couple of building blocks:

```
fun mapmove(#"n") = (0,1)
  |   mapmove(#"s") = (0,~1)
  |   mapmove(#"e") = (1,0)
  |   mapmove(#"w") = (~1,0)

fun sum_tuples([ ]) = (0,0)
  |   sum_tuples((x,y)::ts) =
    let
        val (sumx, sumy) = sum_tuples(ts)
    in
        (x+sumx, y+sumy)
    end
```

travel, continued

The grand finale:

```
fun travel(s) =
    let
        fun mk_tuples([ ]) = [ ]
          | mk_tuples(c::cs) = mapmove(c)::mk_tuples(cs)

        val tuples = mk_tuples(explode(s))

        val disp = sum_tuples(tuples)

    in
    if disp = (0,0) then
        "Got home"
    else
        "Got lost"
    end
```

Note that mapmove and sum_tuples are defined at the outermost level. mk_tuples is defined inside a let. Why?

Larger example: tally

Consider a function tally that prints the number of occurrences of each character in a string:

```
- tally("a bean bag");
a 3
b 2
  2
g 1
n 1
e 1
val it = () : unit
```

Note that the characters are shown in order of decreasing frequency.

How can this problem be approached?

tally, continued

Implementation:

```
(*
 * inc_entry(c, L)
 *
 *     L is a list of (char * int) tuples that indicate how many times a
 *  character has been seen.
 *
 *     inc_entry() produces a copy of L with the count in the tuple
 *  containing the character c incremented by one.  If no tuple with
 *  c exists, one is created with a count of 1.
 *)
fun inc_entry(c, [ ]) = [(c, 1)]
  |   inc_entry(c, (char, count)::entries) =
       if c = char then
          (char, count+1)::entries
       else
          (char, count)::inc_entry(c, entries)
```

tally, continued

```
(* mkentries(s) calls inc_entry() for each character in the string s *)

fun mkentries(s) =
    let
        fun mkentries'([ ], entries) = entries
          | mkentries'(c::cs, entries) =
            mkentries'(cs, inc_entry(c, entries))
    in
        mkentries'(explode s, [ ])
    end

(* fmt_entries(L) prints, one per line, the (char * int) tuples in L *)

fun fmt_entries(nil) = ""
  | fmt_entries((c, count)::es) =
    str(c) ^ " " ^ Int.toString(count) ^ "\n" ^ fmt_entries(es)
```

tally, continued

```
(*
 * sort, insert, and order_pair work together to provide an insertion sort
 *
 *    insert(v, L) produces a copy of the int list L with the int v in the
 * proper position.  Values in L are descending order.
 *
 *    sort(L) produces a sorted copy of L by using insert() to place
 *  values at the proper position.
 *
 *)
fun insert(v, [ ]) = [v]
  |  insert(v, x::xs) =
       if order_pair(v,x) then v::x::xs
                          else x::insert(v, xs)

fun sort([ ]) = [ ]
  |  sort(x::xs) = insert(x, sort(xs))

fun order_pair((_, v1), (_, v2)) = v1 > v2
```

tally, continued

With all the pieces in hand, tally itself is a straightforward sequence of calls.

```
(*
 * tally: make entries, sort the entries, and print the entries
 *)
fun tally(s) = print(fmt_entries(sort(mkentries(s))))
```

More with functions

Functions as values

Functions as arguments

A flexible sort

Curried functions

Functions as values

A fundamental characteristic of a functional language is that functions are values that can be used as flexibly as values of other types.

In essence, the fun declaration creates a function value and binds it to a name. Additional names can be bound to a function value with a val declaration.

```
- fun double(n) = 2*n;
val double = fn : int -> int

- val twice = double;
val twice = fn : int -> int

- twice;
val it = fn : int -> int

- twice 3;
val it = 6 : int
```

Note that unlike values of other types, no representation of a function is shown. Instead, "fn" is displayed. (Think flexibly: What could be shown instead of only fn?)

Functions as values, continued

Just as values of other types can appear in lists, so can functions:
It should be no surprise that functions can be elements of lists and tuples:

```
- (hd, 1, size, "x", length);
val it = (fn,1,fn,"x",fn)
  : ('a list -> 'a) * int * (string -> int) * string * ('b list -> int)

- val convs = [floor, ceil, trunc];
val convs = [fn,fn,fn] : (real -> int) list

- [it];
val it = [(fn,1,fn,"x",fn)]
  : (('a list -> 'a) * int * (string -> int) * string * ('b list -> int)) list
```

Using the "op" syntax we can work with operators as functions:

```
- op+;
val it = fn : int * int -> int

- op+(3,4);
val it = 7 : int
```

Functions as arguments

A function may be passed as an argument to a function.

This function simply *applies* a given function to a value:

```
- fun apply(F,v) = F(v);
val apply = fn : ('a -> 'b) * 'a -> 'b
```

Usage:

```
- apply(size, "abcd");
val it = 4 : int

- apply(swap, (3,4));
val it = (4,3) : int * int

- apply(length, apply(m_to_n, (5,7)));
val it = 3 : int
```

A function that uses other functions as values is said to be a *higher-order function*.

Could apply be written in Java? In C?

Functions as arguments, continued

Here is a function that applies a function to every element of a list and produces a list of the results:

```
fun applyToAll(_, [ ]) = [ ]
  |  applyToAll(f, x::xs) = f(x)::applyToAll(f, xs);
```

Usage:

```
- applyToAll(double, [10, 20, 30]);
val it = [20,40,60] : int list

- applyToAll(real, iota(5));  (* iota(n) produces [1,2,...,n] *)
val it = [1.0,2.0,3.0,4.0,5.0] : real list

- applyToAll(length, [it, it@it]);
val it = [5,10] : int list
```

We'll see later that applyToAll is really the map function from the library, albeit in a slightly different form.

A flexible sort

Recall order(ed)_pair, insert, and sort from the tally example. They work together to sort a (char * int) list.

```
fun ordered_pair((_, v1), (_, v2)) =  v1 > v2

fun insert(v, [ ]) = [v]
  |  insert(v, x::xs) = if ordered_pair(v,x) then v::x::xs else x::insert(v, xs)

fun sort([ ]) = [ ]
  |  sort(x::xs) = insert(x, sort(xs))
```

Consider eliminating ordered_pair and instead supplying a function to test whether the values in a 2-tuple are the desired order.

A flexible sort, continued

Here are versions of insert and sort that use a function to test the order of elements in a 2-tuple:

```
fun insert(v, [ ], isInOrder) = [v]
 |  insert(v, x::xs, isInOrder) =
       if isInOrder(v,x) then v::x::xs
                         else x::insert(v, xs, isInOrder)

fun sort([ ], isInOrder) = [ ]
 |  sort(x::xs, isInOrder) = insert(x, sort(xs, isInOrder), isInOrder)
```

Types:

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

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

What C library function does this version of sort resemble?

A flexible sort, continued

Sorting integers:

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

- sort([4,10,7,3], intLessThan);
val it = [3,4,7,10] : int list
```

We might sort (int * int) tuples based on the sum of the two values:

```
fun sumLessThan( (a1, a2), (b1, b2) ) = a1 + a2 < b1 + b2;

- sort([(1,1), (10,20), (2,~2), (3,5)], sumLessThan);
val it = [(2,~2),(1,1),(3,5),(10,20)] : (int * int) list
```

Problem: Sort an int list list based on the largest value in each of the int lists. Sorting

```
[[3,1,2],[50],[10,20],[4,3,2,1]]
```

would yield

```
[[3,1,2],[4,3,2,1],[10,20],[50]]
```

Curried functions

It is possible to define a function in *curried* form:

```
- fun add x y = x + y;        (Two arguments, x and y, not (x,y), a 2-tuple )
val add = fn : int -> int -> int
```

The function add can be called like this:

```
- add 3 5;
val it = 8 : int
```

Note the type of add: int -> (int -> int) (Remember that -> is right-associative.)

What add 3 5 means is this:

```
- (add 3) 5;
val it = 8 : int
```

add is a function that takes an int and produces a function that takes an int and produces an int. add 3 produces a function that is then called with the argument 5.

In general, a call like f x y z means (((f x) y) z).

Curried functions, continued

For reference: `fun add x y = x + y`. The type is `int -> (int -> int)`.

More interesting than `add 3 5` is this:

```
- add 3;
val it = fn : int -> int

- val plusThree = add 3;
val plusThree = fn : int -> int
```

The name `plusThree` is bound to a function that is a *partial instantiation* of `add`. (a.k.a. *partial application*)

```
- plusThree 5;
val it = 8 : int

- plusThree 20;
val it = 23 : int

- plusThree (plusThree 20);
val it = 26 : int
```

Curried functions, continued

For reference:

```
fun add x y = x + y
```

As a conceptual model, think of this expression:

```
val plusThree = add 3
```

as producing a result similar to this:

```
fun plusThree(y) = 3 + y
```

The idea of a partially applicable function was first described by Moses Schönfinkel. It was further developed by Haskell B. Curry. Both worked wtih David Hilbert in the 1920s.

What prior use have you made of partially applied functions?

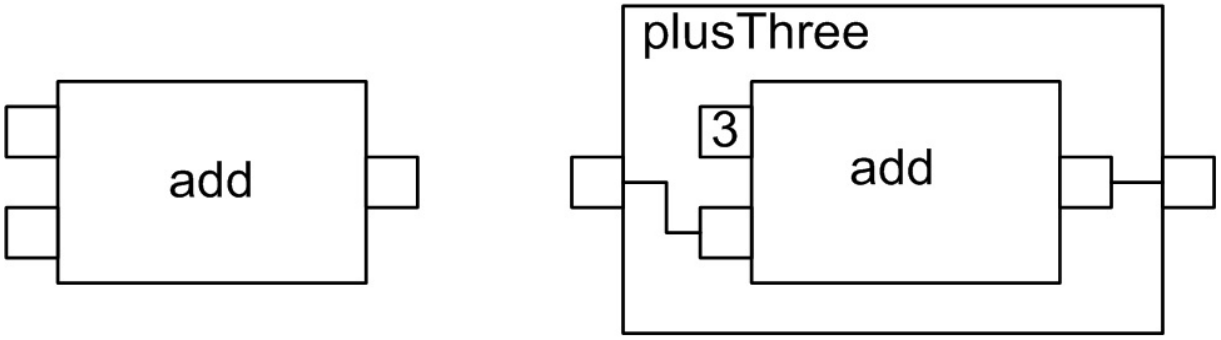
Curried functions, continued

For reference:

```
- fun add x y = x + y;
val add = fn : int -> int -> int

- val plusThree = add 3;
val plusThree = fn : int -> int
```

Analogy: A partially instantiated function is like a machine with a hardwired input value.



This model assumes that data flows from left to right.

Curried functions, continued

Here is a curried implementation of m_to_n (slide 76):

```
- fun m_to_n m n = if m > n then [ ] else m :: (m_to_n  (m+1)  n);
val m_to_n = fn : int -> int -> int list
```

Usage:

```
- m_to_n 1 7;
val it = [1,2,3,4,5,6,7] : int list
```

Problem: Create the function iota. (iota(3) produces [1,2,3].)

Curried functions, continued

Functions in the ML standard library (the "Basis") are often curried.

String.isSubstring returns true iff its first argument is a substring of the second argument:

```
- String.isSubstring;
val it = fn : string -> string -> bool

- String.isSubstring "tan" "standard";
val it = true : bool
```

We can create a partial application that returns true iff a string contains "tan":

```
- val hasTan = String.isSubstring "tan";
val hasTan = fn : string -> bool

- hasTan "standard";
val it = true : bool

- hasTan "library";
val it = false : bool
```

Curried functions, continued

In fact, the curried form is syntactic sugar. An alternative to `fun add x y = x + y` is this:

```
- fun add x =
    let
       fun add' y = x + y
    in
       add'
    end
val add = fn : int -> int -> int   (Remember associativity: int -> (int -> int) )
```

A call such as `add 3` produces an instance of `add'` where `x` is bound to `3`. That instance is returned as the value of the `let` expression.

```
- add 3;
val it = fn : int -> int

- it 4;
val it = 7 : int

- add' 3 4;
val it = 7 : int
```

List processing idioms with functions

Mapping

Anonymous functions

Predicate based functions

Reduction

travel, revisited

Mapping

The applyToAll function seen earlier applies a function to each element of a list and produces a list of the results. There is a built-in function called map that does the same thing.

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

- map size ["just", "testing"];
val it = [4,7] : int list

- map sumInts [[1,2,3],[5,10,20],[]];
val it = [6,35,0] : int list
```

Mapping is one of the idioms of functional programming.

There is no reason to write a function that performs an operation on each value in a list. Instead create a function to perform the operation on a single value and then map that function onto lists of interest.

Mapping, continued

Consider a partial application of map:

```
- val sizes = map size;
val sizes = fn : string list -> int list

- sizes ["ML", "Ruby", "Prolog"];
val it = [2,4,6] : int list

- sizes ["ML", "Icon", "C++", "Prolog"];
val it = [2,4,3,6] : int list
```

Mapping with curried functions

It is common to map with a partial application:

```
- val addTen = add 10;
val addTen = fn : int -> int

- map addTen (m_to_n 1 10);
val it = [11,12,13,14,15,16,17,18,19,20] : int list

- map (add 100) (m_to_n 1 10);
val it = [101,102,103,104,105,106,107,108,109,110] : int list
```

The partial application "plugs in" one of the addends. The resulting function is then called with each value in the list in turn serving as the other addend.

Mapping with anonymous functions

Here's another way to define a function:

```
- val double = fn(n) => n * 2;
val double = fn : int -> int
```

The expression being evaluated, `fn(n) => n * 2`, is a simple example of a *match expression*. It provides a way to create a function "on the spot".

If we want to triple the numbers in a list, instead of writing a `triple` function we might do this:

```
- map (fn(n) => n * 3) [3, 1, 5, 9];
val it = [9,3,15,27] : int list
```

The function created by `fn(n) => n * 3` never has a name. It is an anonymous function. It is created, used, and discarded.

The term *match expression* is ML-specific. A more general term for an expression that defines a nameless function is a *lambda expression*.

Predicate-based functions

The built-in function List.filter applies function F to each element of a list and produces a list of those elements for which F produces true. Here's one way to write filter:

```
- fun filter F [ ] = [ ]
    |   filter F (x::xs) = if (F x) then x::(filter F xs)
                                    else (filter F xs);
val filter = fn : ('a -> bool) -> 'a list -> 'a list
```

It is said that F is a *predicate*—inclusion of a list element in the result is predicated on whether F returns true for that value.

Consider the following.

```
- val f = List.filter (fn(n) => n mod 2 = 0);
val f = fn : int list -> int list

- f [5,10,12,21,32];
val it = [10,12,32] : int list

- length (f (m_to_n 1 100));
val it = 50 : int
```

Predicate-based functions, continued

The function String.tokens uses a predicate to break a string into "tokens":

```
- Char.isPunct;
val it = fn : char -> bool

- String.tokens Char.isPunct "a,bc:def.xyz";
val it = ["a","bc","def","xyz"] : string list
```

Problem: What characters does Char.isPunct consider to be punctuation?

Real-world application: A very simple grep

The UNIX grep program searches files for lines that contain specified text. Imagine a very simple grep in ML:

```
- grep;
val it = fn : string -> string list -> unit list

- grep "sort" ["all.sml","flexsort.sml"];
all.sml:fun sort1([ ]) = [ ]
all.sml: |  sort1(x::xs) =
all.sml:    insert(x, sort1(xs))
flexsort.sml:fun sort([ ], isInOrder) = [ ]
flexsort.sml: |  sort(x::xs, isInOrder) = insert(x, sort(xs, isInOrder), isInOrder)
val it = [(),()] : unit list
```

We could use SMLofNJ.exportFn to create a file that is executable from the UNIX command line, just like the real grep.

A simple grep, continued

Implementation

```
fun grepAFile text file =
    let
        val inputFile = TextIO.openIn(file);
        val fileText = TextIO.input(inputFile);
        val lines = String.tokens (fn(c) => c = #"\n") fileText
        val linesWithText = List.filter (String.isSubstring text) lines
        val _ = TextIO.closeIn(inputFile);
    in
        print(concat(map (fn(s) =>  file ^ ":" ^ s ^ "\n") linesWithText))
    end;

fun grep text files = map (grepAFile text) files;
```

Look: No loops, no variables, no recursion (at this level)!

Reduction of lists

Another idiom is *reduction* of a list by repeatedly applying a binary operator to produce a single value. Here is a simple reduction function:

```
- fun reduce F [ ] = raise Empty
    |  reduce F [x] = x
    |  reduce F (x::xs) = F(x, reduce F xs)
val reduce = fn : ('a * 'a -> 'a) -> 'a list -> 'a
```

Usage:

```
- reduce op+ [3,4,5,6];
val it = 18 : int
```

What happens:

```
op+(3, reduce op+ [4,5,6])
   op+(4, reduce op+ [5,6])
      op+(5, reduce op+ [6])
```

Or,

```
op+(3, op+(4, op+(5,6)))
```

Reduction, continued

More examples:

```
- reduce  op^  ["just", "a", "test"];
val it = "justatest" : string

- reduce op* (iota 5);
val it = 120 : int
```

Problem: How could a list like [[1,2],[3,4,5],[6]] be turned into [1,2,3,4,5,6]?

Reduction, continued

Because reduce is curried, we can create a partial application:

```
- val concat = reduce op^;   (* mimics built-in concat *)
val concat = fn : string list -> string

- concat ["xyz", "abc"];
val it = "xyzabc" : string

- val sum = reduce  op+ ;
val sum = fn : int list -> int

- sum(iota 10);
val it = 55 : int

- val max = reduce (fn(x,y) => if x > y then x else y);
val max = fn : int list -> int

- max [5,3,9,1,2];
val it = 9 : int
```

Another name for reduction is "folding".

travel, revisited

Here's a version of travel that uses mapping and reduction instead of explicit recursion:

```
fun dirToTuple(#"n") = (0,1)
  | dirToTuple(#"s") = (0,~1)
  | dirToTuple(#"e") = (1,0)
  | dirToTuple(#"w") = (~1,0)

fun addTuples((x1 , y1), (x2, y2)) = (x1 + x2, y1 + y2);

fun travel(s) =
    let
        val tuples = map dirToTuple (explode s)
        val displacement = reduce addTuples tuples
    in
        if displacement = (0,0) then "Got home"
                                else "Got lost"
    end
```

How confident are we that it is correct?

Even more with functions

Composition

Manipulation of operands

The composition operator (○)

There is a composition operator in ML:

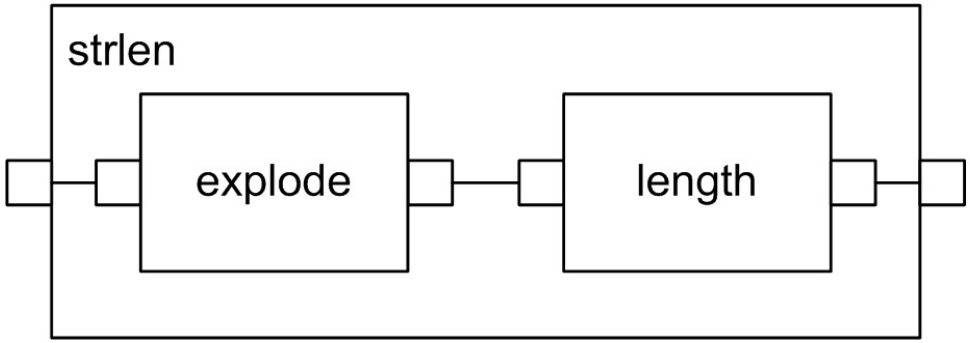
```
- op o;     (* lower-case "Oh" *)
val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Two functions can be composed into a new function:

```
- val strlen = length o explode;
val strlen = fn : string -> int

- strlen "abc";
val it = 3 : int
```

Analogy: Composition is like bolting machines together.



Composition, continued

Problem: Using composition, create a function to reverse a string.

Problem: Create a function to reverse each string in a list of strings and reverse the order of strings in the list. (Example: `f ["one","two","three"]` would produce `["eerht","owt","eno"]`.)

Problem: Compute the sum of the odd numbers between 1 and 100, inclusive. Use only composition and applications of `op+`, `iota`, `isEven`, `reduce`, `filter`, and `not` (`bool -> bool`).

Another way to understand composition

Composition can be explored by using functions that simply echo their call.

Example:

```
- fun f(s) = "f(" ^ s ^ ")";
val f = fn : string -> string

- f("x");
val it = "f(x)" : string
```

Two more:

```
fun g(s) = "g(" ^ s ^ ")";

fun h(s) = "h(" ^ s ^ ")";
```

Compositions:

```
- val fg = f o g;
val fg = fn : string -> string

- fg("x");
val it = "f(g(x))" : string

- val ghf = g o h o f;
val ghf = fn : string -> string

- ghf("x");
val it = "g(h(f(x)))" : string

- val q = fg o ghf;
val q = fn : string -> string

- q("x");
val it = "f(g(g(h(f(x)))))" : string
```

"Computed" composition

Because composition is just an operator and functions are just values, we can write a function that computes a composition. compN f n composes f with itself n times:

```
- fun compN f 1 = f
    |   compN f n = f o compN f (n-1);
val compN = fn : ('a -> 'a) -> int -> 'a -> 'a
```

Usage:

```
- val f = compN double 3;
val f = fn : int -> int

- f 10;
val it = 80 : int

- compN double 10 1;
val it = 1024 : int

- map (compN double) (iota 5);
val it = [fn,fn,fn,fn,fn] : (int -> int) list
```

Could we create compN using folding?

Manipulation of operands

Consider this function:

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

Usage:

```
- c op+ 3 4;
val it = 7 : int

- c op^ "a" "bcd";
val it = "abcd" : string
```

What is it doing?

What would be produced by the following partial applications?

```
c op+

c op^
```

Manipulation of operands, continued

Here's the function again, with a revealing name:

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

Consider:

```
- op+;
val it = fn : int * int -> int

- val add = curry op+;
val add = fn : int -> int -> int

- val addFive = add 5;
val addFive = fn : int -> int

- map addFive (iota 10);
val it = [6,7,8,9,10,11,12,13,14,15] : int list

- map (curry op+ 5) (iota 10);
val it = [6,7,8,9,10,11,12,13,14,15] : int list
```

Manipulation of operands, continued

For reference:

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

For a moment, think of a partial application as textual substitution:

`val add = curry op+`	is like	`fun add x y = op+(x, y)`
`val addFive = curry op+ 5`	is like	`fun addFive y = op+(5, y)`

Bottom line:

If we have a function that takes a 2-tuple, we can easily produce a curried version of the function.

Manipulation of operands, continued

Recall repl from slide 165:

```
- repl("abc", 4);
val it = "abcabcabcabc" : string
```

Let's create some partial applications of a curried version of it:

```
- val stars = curry repl "*";
val stars = fn : int -> string

- val arrows = curry repl " ---> ";
val arrows = fn : int -> string

- stars 10;
val it = "**********" : string

- arrows 5;
val it = " --->  --->  --->  --->  ---> " : string

- map arrows (iota 3);
val it = [" ---> "," --->  ---> "," --->  --->  ---> "] : string list
```

Manipulation of operands, continued

Sometimes we have a function that is curried but we wish it were not curried. For example, a function of type 'a -> 'b -> 'c that would be more useful if it were 'a * 'b -> 'c.

Consider a curried function:

```
- fun f x y = g(x,y*2);
val f = fn : int -> int -> int
```

Imagine that we'd like to map f onto an (int * int) list. We can't! (Why?)

Problem: Write an uncurry function so that this works:

```
- map (uncurry f) [(1,2), (3,4), (5,6)];
```

Important: The key to understanding functions like curry and uncurry is that without partial application they wouldn't be of any use.

Manipulation of operands, continued

The partial instantiation curry repl "x" creates a function that produces some number of "x"s, but suppose we wanted to first supply the replication count and then supply the string to replicate.

Example:

```
- five;      (Imagine that 'five s' will call 'repl(s, 5)'.)
val it = fn : string -> string

- five "*";
val it = "*****" : string

- five "<x>";
val it = "<x><x><x><x><x>" : string
```

Manipulation of operands, continued

Consider this function:

```
- fun swapArgs f x y = f y x;
val swapArgs = fn : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
```

Usage:

```
- fun cat s1 s2 = s1 ^ s2;
val cat = fn : string -> string -> string

- val f = swapArgs cat;
val f = fn : string -> string -> string

- f  "a"  "b";
val it = "ba" : string

- map (swapArgs (curry op^) "x") ["just", "a", "test"];
val it = ["justx","ax","testx"] : string list
```

Manipulation of operands, continued

```
- val curried_repl = curry repl;
val curried_repl = fn : string -> int -> string

- val swapped_curried_repl = swapArgs curried_repl;
val swapped_curried_repl = fn : int -> string -> string

- val five = swapped_curried_repl 5;
val five = fn : string -> string

- five "*";
val it = "*****" : string

- five "<->";
val it = "<-><-><-><-><->" : string
```

Or,

```
- val five = swapArgs (curry repl) 5;
val five = fn : string -> string

- five "xyz";
val it = "xyzxyzxyzxyzxyz" : string
```

Example: optab

Function optab(F, N, M) prints a table showing the result of F(n,m) for each value of n and m from 1 to N and M, respectively. F is always an int * int -> int function.

Example:

```
- optab;
val it = fn : (int * int -> int) * int * int -> unit

- optab(op*, 5, 7);
        1   2   3   4   5   6   7
   1    1   2   3   4   5   6   7
   2    2   4   6   8  10  12  14
   3    3   6   9  12  15  18  21
   4    4   8  12  16  20  24  28
   5    5  10  15  20  25  30  35
val it = () : unit
```

optab, continued

```
- optab(add, 3, 4);
        1   2   3   4
   1    2   3   4   5
   2    3   4   5   6
   3    4   5   6   7
val it = () : unit
```

```
val repl = concat o xrepl;

fun rightJustify width value =
   repl(" ", width-size(value)) ^ value

fun optab(F, nrows, ncols) =
   let
      val rj = rightJustify 4  (* assumes three-digit results at most *)

      fun intsToRow (L) = concat(map (rj o Int.toString) L) ^ "\n"

      val cols = iota ncols

      fun mkrow nth = intsToRow(nth::(map (curry F nth) cols))

      val rows = map mkrow (iota nrows)
   in
      print((rj "") ^ intsToRow(cols) ^ concat(rows))
   end
```

The datatype declaration

New types with datatype

New types can be defined with the datatype declaration. Example:

```
- datatype Shape =
    Circle of real
  | Square of real
  | Rectangle of real * real
  | Point;
datatype Shape
  = Circle of real | Point | Rectangle of real * real | Square of real
```

This defines a new type named Shape. An instance of a Shape is a value in one of four forms:

A Circle, consisting of a real (the radius)

A Square, consisting of a real (the length of a side)

A Rectangle, consisting of two reals (width and height)

A Point, which has no data associated with it. (Debatable, but good for an example.)

Shape: a new type

At hand:

```
datatype Shape =
    Circle of real
  | Square of real
  | Rectangle of real * real
  | Point
```

This declaration defines four constructors. Each constructor specifies one way that a Shape can be created.

Examples of constructor invocation:

```
- val r = Rectangle (3.0, 4.0);
val r = Rectangle (3.0,4.0) : Shape

- val c = Circle(5.0);
val c = Circle 5.0 : Shape

- val p = Point;
val p = Point : Shape
```

Shape, continued

A function to calculate the area of a Shape:

```
- fun area(Circle radius) = Math.pi * radius * radius
    | area(Square side) = side * side
    | area(Rectangle(width, height)) = width * height
    | area(Point) = 0.0;
val area = fn : Shape -> real
```

Usage:

```
- val r = Rectangle(3.4,4.5);
val r = Rectangle (3.4,4.5) : Shape

- area(r);
val it = 15.3 : real

- area(Circle 1.0);
val it = 3.14159265359 : real
```

Speculate: What will happen if the case for Point is omitted from area?

Shape, continued

A Shape list can be made from any combination of Circle, Point, Rectangle, and Square values:

```
- val c = Circle(2.0);
val c = Circle 2.0 : Shape

- val shapes = [c, Rectangle (1.5, 2.5), c, Point, Square 1.0];
val shapes = [Circle 2.0,Rectangle (1.5,2.5),Circle 2.0,Point,Square 1.0]
  : Shape list
```

We can use map to calculate the area of each Shape in a list:

```
- map area shapes;
val it = [12.0,78.5398163397,0.0] : real list
```

What does the following function do?

```
- val f = (foldr op+ 0.0) o (map area);
val f = fn : Shape list -> real
```