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

A simple function to double a number:

```
int Double(int n)
{
    return n * 2;
}
```

Usage:

```
int result = Double(Double(Double(2)));
```

A function can be thought of as a rule of correspondence.

Functional programming, continued

A function to compute the maximum of two integers:

```
int max(int a, int b)
{
    if a > b then return a else return b;
}
```

Two uses of max can be combined to produce a function to compute the largest of three integers:

```
int largest(int x, int y, int z)
{
    return max(x, max(y, z));
}
```

The largest of six values can be computed by combining max and largest:

```
max(largest(a, b, c), largest(d, e, f))

largest(max(a,b), max(c,d), max(e,f))
```

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

Quick history of ML

Interacting with ML

Identifiers and `val` declarations

Simple data types and operators

ML—Background

ML will be our vehicle for studying functional programming.

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. We'll be using Standard ML of New Jersey (SML/NJ).

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

See the class website for information on running SML/NJ on lectura and on Windows.

ML is not object-oriented

ML is not designed for object-oriented programming:

- There is no analog for Java's 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 interactive—the user types an expression, it is evaluated, and the result is printed.

```
% sml
Standard ML of New Jersey, v110.57...
- 3+4;
val it = 7 : int

- 3.4 * 5.6 / 7.8;
val it = 2.44102564102564 : real

- 3 - 4;
val it = ~1 : int

- 10 + ~20;
val it = ~10 : int
```

Note the prompt is a minus sign. Use a semicolon to terminate the expression.

To use some Lisp terminology, we can say that SML/NJ provides a “read-eval-print loop”.

Interacting with ML, continued

The result of the last expression is given the name it and can be used in subsequent expressions:

```
- 10 + ~20;
val it = ~10 : int

- it * it + it;
val it = 90 : int
```

Input can be broken across several lines:

```
- 5
= *
= 6 + 7
= ;
val it = 37 : int
```

Note that the equal signs at the start of the lines are being printed as a secondary prompt—the expression is recognized as being incomplete.

Interacting with ML, continued

ML has a number of predefined functions. Some of them:

```
- real(4) ;
val it = 4.0 : real

- floor(3.45);
val it = 3 : int

- ceil(3.45);
val it = 4 : int

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

Interacting with ML, continued

For organizational purposes, many functions and constants are grouped in *structures* such as Math, Int, and Real:

```
- Math.sqrt(2.0);
val it = 1.41421356237 : real

- Int.sign(~300);
val it = ~1 : int

- Real.min(1.0, 2.0);
val it = 1.0 : real

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

In some ways, an ML structure is similar to a Java class that has only static methods and fields.

Functions with simple names, like ceil and size, are typically in a structure, with an alias for the simple name. For example, real(x) is another name for Real.fromInt(x).

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 = 3.4;
val x = 3.4 : real

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

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

Technically, the environment contains three bindings for x, but only one—the last one—is accessible.

Identifiers

There are two types of identifiers: alphanumeric and symbolic.

An alphanumeric identifier begins with a letter or apostrophe and is followed by any number of letters, apostrophes, underscores, and digits. Examples:

```
x, minValue, a', C''
```

Identifiers starting with an apostrophe are *type variables*. Examples:

```
'a, ''a, 'b
```

Identifiers, continued

A symbolic identifier is a sequence of one or more of these characters:

```
+ - / * < > = ! @ # $ % ^ & ` ~ \ | ? :
```

Examples:

```
- val \/ = 3;       (* Not a “V” — it’s two slashes *)
val \/ = 3 : int

- val <---> = 10;
val <---> = 10 : int

- val |\| = \/ + <--->;
val |\| = 13 : int
```

The two character sets can’t be mixed. For example, `<x>` is not a valid identifier.

Comparisons and boolean values

ML has a set of comparison operators that compare values and produce a result of type bool:

```
- 1 < 3;
val it = true : bool

- 1 > 3;
val it = false : bool

- 1 = 3;
val it = false : bool

- 1 <> 1+1;
val it = true : bool
```

ML does not allow real numbers to be tested for equality or inequality:

```
- 1.0 = 1.0;
stdIn:5.1-5.10 Error: operator and operand don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         real * real
```

Comparisons and boolean values, continued

The logical operators andalso and orelse provide logical conjunction and disjunction:

```
- 1 <= 3 andalso 7 >= 5 ;
val it = true : bool

- 1 = 0 orelse 3 = 4;
val it = false : bool
```

The values true and false may be used literally:

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

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

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 <u>expression</u>".

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

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 Java?

The conditional expression, continued

For reference:

```
if *boolean-expr* then *expr1* else *expr2*
```

Problem: Using nested conditional expressions, write an expression having a result of -1, 0, or 1 depending on whether n is negative, zero, or positive, respectively.

```
val n = <some integer>

val sign =
```

Note that the boolean expression is not required to be in parentheses. Instead, the keyword then is required.

There is no else-less form of the conditional expression. Why?

What construct in Java is most similar to this ML construct?

Strings

ML has a string data type to represent a sequence of zero or more characters.

A string literal is specified by enclosing a sequence of characters in double quotes:

```
- "testing";
val it = "testing" : string
```

Escape sequences may be used to specify characters in a string literal:

\n	newline
\t	tab
\\	backslash
\"	double quote
\010	any character; value is in decimal (000-255)
\^A	control character (must be capitalized)

Example:

```
- "\n is \^J is \010";
val it = "\n is \n is \n" : string
```

Strings, continued

Strings may be concatenated with the **^** (caret) operator:

```
- "Standard " ^ "M" ^ "L";
val it = "Standard ML" : string

- it ^ it ^ it;
val it = "Standard MLStandard MLStandard ML" : string
```

Strings may be compared:

```
- "aaa" < "bbb";
val it = true : bool

- "aa" < "a";
val it = false : bool

- "some" = "so" ^ "me";
val it = true : bool
```

Based on the above, how are strings in ML different from strings in Java?

String functions

The size function produces the length of a string:

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

The structures Int, Real, and Bool each have a toString function to convert values into strings.

```
- Real.toString( Math.sqrt(2.0) );
val it = "1.41421356237" : string
```

String.substring does what you'd expect:

```
- String.substring("0123456789", 3, 4);
val it = "3456" : string
```

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:

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

- #"\010";
val it = #"\n" : char
```

String.sub extracts a character from a string:

```
- String.sub("abcde", 2);
val it = #"c" : char
```

The char type, continued

The chr(n) function produces a char having the value of the n'th ASCII character. The ord(c) function calculates the inverse of chr.

```
- chr(97);
val it = #"a" : char

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

The str(c) function returns a one-character string containing the character c.

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

What are some pros and cons of having a character type in addition to a string type?

Operator summary (partial)

Integer arithmetic operators:

    + - * div mod ~ (unary)

Real arithmetic operators:

    + - * / ~ (unary)

Comparison operators (int, string; bool and real for some):

    = <> < > <= >=

Boolean operators:

    andalso orelse not (unary)

Operator summary (partial), continued

Precedence:

```
             not

  * / div quot rem mod

            + - ^

   =  <>  <  >  <=  >=

     andalso orelse
```

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.

What automatic type conversions does Java provide?

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

At hand:

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

Examples of usage for double:

```
- double(double(3));
val it = 12 : int

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

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

- f(5);
val it = 10 : int
```

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?

Function definition basics, continued

More simple functions:

```
- fun sign(n) = if n < 0 then ~1
                else if n > 0 then 1 else 0;
val sign = fn : int -> int

- fun max(a,b) = if a > b then a else b;
val max = fn : int * int -> int

- fun max3(x,y,z) = max(x,max(y,z));
val max3 = fn : int * int * int -> int

- max3(~1, 7, 2);
val it = 7 : int
```

How was the type of max3 deduced?

Function definition basics, continued

Problem: Define functions with the following types. The functions don't need to do anything practical; only the type is important.

```
string -> int

real * int -> real

bool -> string

int * bool * string -> real
```

Problem: Make up some more and solve them, too.

Have you previously used a system that employs type deduction?

Function definition basics, continued

Problem: Write a function `even(n)` that returns true iff (if and only if) `n` is an even integer.

Problem: Write a function `sum(N)` that returns the sum of the integers from 1 through `N`. Assume `N` is greater than zero.

Problem: Write a function `countTo(N)` that returns a string like this: "1...2...3", if `N` is 3, for example. Use `Int.toString` to convert `int` values to strings.

Function definition basics, continued

Sometimes, especially when overloaded arithmetic operators are involved, we want to specify a type other than that produced by default.

Imagine we want a function to square real numbers. The obvious definition for a square function produces the type int -> int:

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

Solution: We can explicitly specify a type:

```
- fun real(x:real) = x * x;
val real = fn : real -> real
```

Two other solutions:

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

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, 3);
val it = 3 : int

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

Type variables and polymorphic functions

A function's type may be a combination of fixed types and type variables:

```
- fun classify(n, a, b) = if n < 0 then a else b;
val classify = fn : int * 'a * 'a -> 'a

- classify(~3, "left", "right");
val it = "left" : string

- classify(10, 21.2, 33.1);
val it = 33.1 : real
```

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
```

Another example:

```
- fun equal3(a,b,c) = a = b andalso b = c;
val equal3 = fn : ''a * ''a * ''a -> bool
```

Practice

Problem: Define functions having the following types:

```
''a * int * ''a -> real

''a * ''b * ''a * ''b -> bool

''a * 'b * ''a -> 'b
```

Problem: Make up some more and solve them, too.

Loading source code with use

SML source code can be loaded from a file with the use function:

```
% cat funcs.sml
fun double(n) = n * 2

fun bracket(s) = "{" ^ s ^ "}"

% sml
- use("funcs.sml");
[opening funcs.sml]
val double = fn : int -> int
val bracket = fn : string -> string

- double(3);
val it = 6 : int

- bracket("abc");
val it = "{abc}" : string
- ^D
```

Loading source with use, continued

Calls to use can be nested:

```
% cat test.sml
use("funcs.sml");

(* Test cases *)
val r1 = double(3);
val r2 = bracket("abc");
val r3 = bracket(bracket(""));

% sml
- use("test.sml");
[opening test.sml]
[opening funcs.sml]
val double = fn : int -> int
val bracket = fn : string -> string
val r1 = 6 : int
val r2 = "{abc}" : string
val r3 = "{{}}" : string
```

Note the use of r1, r2, ... to associate results with test cases.

Loading source with use, continued

When developing code, you might have an editor open and an SML session open, too. In that session you might do this:

```
- fun run() = use("test.sml");
val run = fn : unit -> unit

- run();
[opening test.sml]
[opening funcs.sml]
val double = fn : int -> int
val bracket = fn : string -> string
val r1 = 6 : int
val r2 = "{abc}" : string
val r3 = "{{}}" : string

...edit your source files...
- run();

...repeat...
```

Running tests with redirection

On both lectura and at a Windows command prompt, you can use *input redirection* to feed source code into sml:

```
c:\372> sml < test.sml
Standard ML of New Jersey v110.57 [built: Mon Nov 21 21:46:28 2005]
- [opening funcs.sml]
...
val r1 = 6 : int
val r2 = "{abc}" : string
val r3 = "{{}}" : string
```

On lectura:

```
% sml < test.sml
Standard ML of New Jersey v110.57 [built: Mon Nov 21 21:46:28 2005]
- [opening funcs.sml]
...
```

On both Windows and lectura, sml exits after processing redirected input.

The unit type

All ML functions return a value but in some cases, there's little practical information to return from a function.

use is such a function. Note its type, and the result of a call:

```
- use;
val it = fn : string -> unit

- use("x.sml");
 [opening x.sml]
...Output from evaluating expressions in x.sml...
val it = () : unit
```

There is only one value having the type unit. That value is represented as a pair of parentheses.

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

Is there an analog to unit in Java?

Simple exceptions

Like Java, ML provides exceptions.

Here is a simple example:

```
exception BadArg;
 fun sum(N) = if N < 1 then raise BadArg
                         else if N = 1 then 1
                                       else N + sum(N-1);
```

Usage:

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

- sum(~10);
uncaught exception BadArg
  raised at: big.sml:26.35-26.41
```

We'll learn how to catch exceptions if the need arises.

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)
```

Problem: Specify tuples with the following types:

```
string * int

string * (int * int)

(real * int) * int
```

Tuples, continued

Problem: What is the type of the following values?

```
(1 < 2, 3 + 4, "a" ^ "b")

(1, (2, (3,4)))

(#”a”, (2.0, #”b”), (“c”), 3)

(((1)))
```

A tuple may be drawn as a tree.

```
("x", 3)          (1,2,(3,4))          ((1,2),(3,4))
```

Tuples, continued

A function can return a tuple as its result:

```
- 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)

- val c = "a" and i = 1;
val c = "a" : string
val i = 1 : int

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

Does Java have a language element that is equivalent to a tuple?

Problem: Write a function `rev3` that reverses the sequence of values in a 3-tuple. What is its type?

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

Consider a swap function:

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

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

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

We can swap the result of order:

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

Problem: Write a function descOrder that orders an int * int in descending order.

Pattern matching, continued

In fact, all ML functions in our current 372 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

Consider a couple of errors:

```
- 1 2;
stdIn:15.1-15.4 Error: operator is not a function [literal]
  operator: int
  in expression:
    1 2

- swap order x;
stdIn:65.1-65.13 Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z * 'Y
  operand:         int * int -> int * int
  in expression:
    swap order
```

Explain them! Fix the second one.

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 (i,r,s) = (1, 2.0, "three");
val i = 1 : int
val r = 2.0 : real
val s = "three" : string
```

Which of the following are valid? If valid, what bindings result?

```
val (x, y, z) = (1, (2, 3),  (4, (5, 6)));

val (x, (y, z)) = (1, 2, 3);

val ((x, y), z) = ((1, 2), (3, 4));

val x = (1, (2,3), (4,(5,6)));
```

Pattern matching, continued

Consider a function that simply returns the value 5:

```
- fun five() = 5;
val five = fn : unit -> int

- five ();
val it = 5 : int
```

In this case, the pattern is the literal for unit. Alternatively, we can bind a name to the value of unit and use that name:

```
- val x = ();
val x = () : unit

- five x;
val it = 5 : int
```

Is five(x) valid?

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);
```

Pattern matching, continued

The set of patterns for a function may be cited as being *non-exhaustive*:

```
- fun f(1) = 10
    | f(2) = 20;
stdIn:10.5-11.14 Warning: match nonexhaustive
          1 => ...
          2 => ...
```

This warning indicates that there is at least one value of the appropriate type (int, here) that isn't matched by any of the cases.

Calling f with such a value produces an exception:

```
- f(3);
uncaught exception nonexhaustive match failure
```

A non-exhaustive match warning can indicate incomplete reasoning.

It's just a warning—f works fine when called with only 1 or 2.

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, it];
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 ->.

What is the analog for a (int * string) list in Java?

Obviously, list is a postfix operator. How might a prefix alternative be represented?

Lists, continued

An empty list can be represented with [ ] or the literal nil:

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

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

- [ [ ], nil, [1], [7, 3] ];
val it = [[],[],[1],[7,3]] : int list list
```

Why is ‘a list used as the type of an empty list?

Recall that all elements of a list must be of the same type. Which of the following are valid?

```
[[1], [2], [[3]]];

[ [ ],[1, 2]];

[ [ ], [ [ ] ] ];
```

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.

Speculate: What are the types of hd and tl?

Heads and tails, continued

Problems:

Given val x = [1, 2, 3, 4], what is the value of hd(tl x))? How about hd(hd (tl x))?

What is the value of tl [3]?

Given val x = [[1], [2], [3]], extract the 2.

What is the value of hd(tl(hd([[1,2],[3,4]])))?

Specify a list x for which hd(hd(hd x)) is 3.

Given val x = [length], what is the value of hd x x?

List equality

Lists can be tested for equality if their component types are equality types. Equality is determined based on the complete contents of the list. (I.e., it is a *deep comparison*.)

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

- [1] <> [1,2];
val it = true : bool

- [("a", "b")] = [("b", "a")];
val it = false : bool
```

Some comparisons involving empty lists produce a warning about “polyEqual”:

```
- [ ] = [ ];
stdIn:27.4 Warning: calling polyEqual
val it = true : bool
```

For our purposes, this warning can be safely ignored.

A VERY 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?

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

```
- sum([1,2,3,4]);
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

- 1::[ ];
val it = [1] : 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
```

Cons, continued

Note that :: is right-associative. The expression

```
1::2::3::nil
```

is equivalent to

```
1::(2::(3::nil))
```

What is the value and type of the following expressions?

```
(1,2)::nil

[1]::[[2]]

nil::nil

nil::nil::nil

length::nil
```

Cons, continued

Recall that all elements must be the same type, i.e., they are homogenous. Note the error produced when this rule is violated with the **[...]** syntax:

```
- [1, "x"];
Error: operator and operand don't agree [literal]
  operator domain: int * int list
  operand:         int * string list
  in expression:
    1 :: "x" :: nil
```

The [...] form is *syntactic sugar*—we can live without it, and use only cons, but it sweetens the language a bit.

Cons, continued

Problem: Write a function m_to_n(m, n) that produces a list of the integers from m through n 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
```

What would m_to_n look like in Java? Which is faster, ML or Java?

Problem: Calculate the sum of the integers from 1 to 100.

Sidebar: A Java model of ML lists

```
public class List {
  private Object head;
  private List tail;

  public List(Object head, List tail) {
    this.head = head;   this.tail = tail;
  }

  public static List cons(Object head, List tail) {
    return new List(head, tail);
  }

  public static void main(String args[ ]) {
    List L1 = List.cons("z", null);
    List L2 = List.cons("x", List.cons("y", L1));

    System.out.println(L1);  // Output: [z]
    System.out.println(L2);  // Output: [x, y, z]

    List L3 = List.cons(L1, List.cons(L2, null));
    System.out.println(L3);  // Output: [[z], [x, y, z]]
  }
```

A good exercise is to model elements of new language in a language we know.

Here is a simple model of ML lists in Java.

Problems:
(1) Draw the structures for L1, L2, and L3.

(2) Implement toString() in a functional style. Don't worry about empty lists.

List concatenation

The @ operator concatenates two lists:

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

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

- op@ ;
val it = fn : 'a list * 'a list -> 'a list
```

Problem: Write a function iota(n) that produces a list of the integers between 1 and n inclusive. Don't use m_to_n. Example:

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

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.

Lists, strings, and characters, continued

The concat function forms a single string out of a list of strings:

```
- concat(["520", "-", "621", "-", "4632"]);
val it = "520-621-4632" : string

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

What is the type of concat?

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 sum_evens(L) that returns the sum of the even values in L, an int list.

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.

Pattern matching, continued

What's an advantage of using a pattern to work with a list rather than the `hd` and `tl` functions?

Hint: Consider the following two implementations of `sum`:

```
fun sum(L) = hd(L) + sum(tl(L));

fun sum(x::xs) = x + sum(xs);
```

Practice

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.

Pattern construction

A pattern can be:

- A literal value such as 1, "x", true (but not a real)
- An identifier
- An underscore
- A tuple composed of patterns
- A list of patterns in [ ] form
- A list of patterns constructed with :: operators

Note the recursion.

Pattern construction, continued

Unfortunately, a pattern cannot contain an arbitrary expression:

```
- fun f(n > 0) = n     (* not valid! *)
   |   f(n) = n;
stdIn:1.5-2.13 Error: non-constructor applied to argument in pattern: >
```

Note "non-constructor" in the message. In a pattern, operators like :: are known as *constructors*.

An identifier cannot appear more than once in a pattern:

```
- fun equals(x, x) = true        (* not valid! *)
    |  equals(_) = false;
stdIn:1.5-3.24 Error: duplicate variable in pattern(s): x
```

Practice

What bindings result from the following val declarations?

```
val [ [ (x, y) ] ] = [ [ ( 1, 2) ] ];


val [ [ x, y ] ] = [ [ 1, 2, 3 ] ];


val [(x,y)::z] = [ [ ( 1, (2 ,3) ) ] ];


val (x, ( y::ys, x ) ) = (1, ([2,3,4], (1, 2) ) );
```

A batch of odds and ends

let expressions

Producing output

Common problems

let expressions

A let expression can be used to create name/value bindings for use in a following expression to improve clarity and/or efficiency.

One way to write a function:

```
fun calc(x, y, z) = f1(g(x + y) - h(z)) + f2(g(x + y) - h(z))
```

An alternative with let:

```
fun calc(x,y,z) =
    let
        val diff = g(x+y) - h(z)
    in
        f1(diff) + f2(diff)
    end
```

Would it be practical for a compiler to make the above transformation automatically, using CSE (common subexpression elimination)?

let expressions, continued

General form of a let expression:

```
let
    declaration1
    declaration2
    ...
    declarationN
in
    expression
end
```

The value of *expression* is the value produced by the overall let expression. The name/value binding(s) established in the declaration(s) are only accessible in *expression*.

```
- val result = let val x = 1 val y = 2 in x + y end;
val result = 3 : int

- x;
stdIn:2.1 Error: unbound variable or constructor: x
```

let expressions, continued

A cute example of let from Ullman, p.78:

```
fun hundredthPower(x:real) =
  let
    val four = x*x*x*x
    val twenty = four*four*four*four*four
  in
    twenty*twenty*twenty*twenty*twenty
  end
```

Usage:

```
- hundredthPower(10.0);
val it = 1.0E100 : real
```

let expressions, continued

A function to count the number of even and odd values in a list of integers and return the result as int * int:

```
fun count_eo([ ]) = (0,0)
 |  count_eo(x::xs) =
    let
       val (even,odd) = count_eo(xs)
    in
       if x mod 2 = 0 then (even+1,odd)
                               else (even,odd+1)
    end
```

Usage:

```
- count_eo([7,3,5,2]);
val it = (1,3) : int * int

- count_eo([2,4,6,8]);
val it = (4,0) : int * int
```

Would it be as easy to write without the let?

let expressions, continued

Imagine a function remove_min(L) that produces a tuple consisting of the smallest integer in L and a list consisting of the other values, possibly in a different order.

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

- remove_min([3,2,4]);
val it = (2,[3,4]) : int * int list

- remove_min([3,4]);
val it = (3,[4]) : int * int list

- remove_min([4]);
val it = (4,[ ]) : int * int list
```

let expressions, continued

remove_min can be used to write a function that sorts a list:

```
fun remsort([ ]) = []
 |  remsort(L) =
        let
            val (min, remain) = remove_min(L)
        in
            min::remsort(remain)
        end
```

Usage:

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

let expressions, continued

A common technique is to define "helper" functions inside a function using a let expression.

Consider a function that returns every Nth element in a list:

```
- every_nth([10,20,30,40,50,60,70], 3);
val it = [30,60] : int list
```

Implementation:

```
fun every_nth(L, n) =
  let
    fun select_nth([ ],_,_) = [ ]
      |  select_nth(x::xs, elem_num, n) =
          if elem_num mod n = 0 then
             x::select_nth(xs, elem_num+1, n)
          else
             select_nth(xs, elem_num+1, n)
  in
    select_nth(L, 1, n)
  end;
```

Simple output

The print function writes its argument, a string, to standard output.

```
- print("abc");
abcval it = () : unit

- print("i = " ^ Int.toString(i) ^ "\n");   (* assume i = 7 *)
i = 7
val it = () : unit
```

A function to print the integers from 1 through N:

```
fun printN(n) =
    let
        fun printN'(0) = ""
          |  printN'(n) = printN'(n - 1) ^ Int.toString(n) ^ "\n"
  in
      print(printN'(n))
  end
```

Note the similarity between this function and countTo, on slide 37 (1...2...3). Could a generalization provide both behaviors?

Simple output, continued

Imagine a function to print name/value pairs:

```
- print_pairs([("x",1), ("y",10), ("z",20)]);
x 1
y 10
z 20
val it = () : unit
```

Problem: Write it!

Common problems

When loading source code sml typically cites the line and position in the line of any errors that are encountered:

```
% cat -n errors.sml     ( -n produces numbered output )
      1  fun count_eo([ ]) = (0,0)
      2   |  count_eo(x::xs) =
      3    let
      4      (even,odd) = count_eo(xs)
      5    in
      6      if x mud 2 = 0 then (even+1,odd)
      7                     else (even,Odd+1)
      8    end
```

Loading:

```
- use "errors.sml";
[opening errors.sml]
errors.sml:4.5 Error: syntax error: inserting  VAL
errors.sml:6.10-6.13 Error: unbound variable or constructor: mud
errors.sml:7.31-7.34 Error: unbound variable or constructor: Odd
```

Common problems, continued

Infinite recursion:

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

Usage:

```
- sum(5);
...no response...
^C
Interrupt
```

Common problems, continued

Type mismatch when calling a function:

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

- fun f(x) = double(3.0 * x);
stdIn:3.27 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         real
  in expression:
    double (3.0 * x)
```

Type mismatch when recursively calling a function:

```
- fun f(x,y) = f(x);
Error: operator and operand don't agree [circularity]
  operator domain: 'Z * 'Y
  operand:         'Z
  in expression:
    f x
```

Common problems, continued

A non-exhaustive match warning can indicate incomplete reasoning, typically a missing basis case to terminate recursion:

```
- fun len(x::xs) = 1 + len(xs);
Warning: match nonexhaustive
          x :: xs => ...

- len([1,2,3]);
uncaught exception nonexhaustive match failure
  raised at: stdIn:368.3
```

Use of fun instead of | (or-bar) for a function case:

```
- fun f(1) = "one"
  fun f(n) = "other";
Warning: match nonexhaustive
          1 => ...

val f = <hidden-value> : int -> string
val f = fn : 'a -> string
```

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?

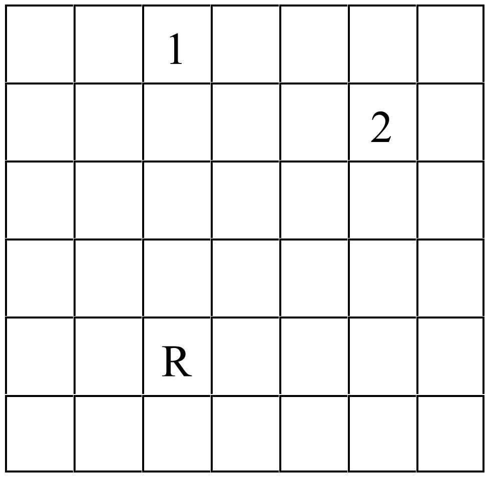
Finally, expand:

```
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:

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

- hd convs;
val it = fn : real -> int

- it 4.3;
val it = 4 : int

- (hd (tl convs)) 4.3;
val it = 5 : int
```

What is the type of the list [hd]?

What is the type of [size, length]?

Functions as values, continued

It should be no surprise that functions can be elements of a tuple:

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

- [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:

```
- swap(pair(op~, op^));
val it = (fn,fn) : (string * string -> string) * (int -> int)

- #2(it) 10;             (* #n(tuple) produces the nth value of the tuple *)
val it = ~10 : int
```

What are some other languages that allow functions to be treated as values, at least to some extent?

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

Consider the following function:

```
fun f(f', x, 1) = f'(x)
 |  f(f', x, n) = f(f', f'(x), n - 1)
```

What does it do?

What is its type?

Give an example of a valid use of the function.

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));
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

- applyToAll(implode,
        applyToAll(rev,
              applyToAll(explode, ["one", "two", "three"])));
val it = ["eno","owt","eerht"] : string list
```

Functions as arguments, continued

Here's a roundabout way to calculate the length of a list:

```
- val L = explode "testing";
val L = [#"t",#"e",#"s",#"t",#"i",#"n",#"g"] : char list

- fun one _ = 1;
val one = fn : 'a -> int

- sumInts(applyToAll(one, L));
val it = 7 : int
```

Problem: Create a list like ["x", "xx", "xxx", ... "xxxxxxxxxx"]. (One to ten "x"s.)

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

Functions that produce functions

Consider a function that applies two specified functions to the same value and returns the function producing the larger integer result:

```
- fun larger(f1, f2, x) = if f1(x) > f2(x) then f1 else f2;
val larger = fn : ('a -> int) * ('a -> int) * 'a -> 'a -> int

- val g = larger(double, square, 5);
val g = fn : int -> int

- g(5);
val it = 25 : int

- val h = larger(sum, len, [0, 0, 0]);
val h = fn : int list -> int

- h([10,20,30]);
val it = 3 : int

- (larger(double, square, ~4)) (10);
val it = 100 : int
```

A flexible sort

Recall order(ed)_pair, insert, and sort from tally (slide 112). 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.

Is add(3,5) valid?

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?

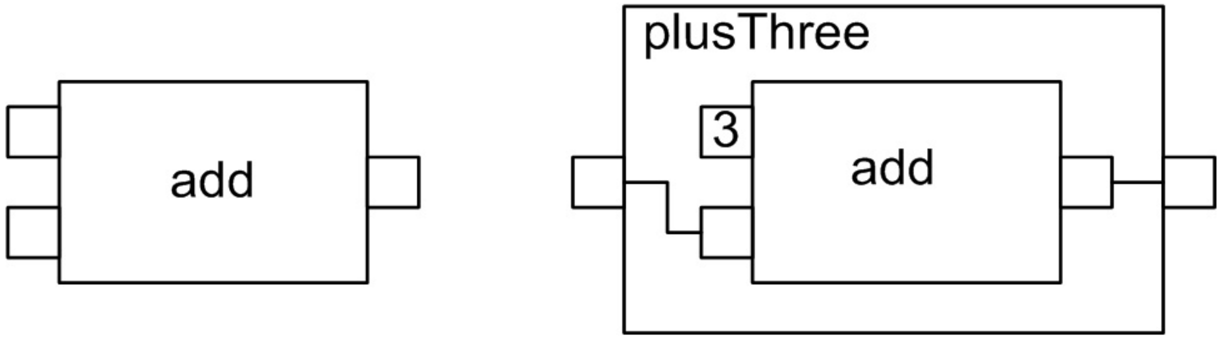
Curried functions, continued

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

Consider another function:

```
- fun f a b c = a*2 + b*3 + c*4;
val f = fn : int -> int -> int -> int

- val f_a = f 1;
val f_a = fn : int -> int -> int

        fun f_a(b,c) = 1*2 + b*3 + c*4

- val f_a_b = f_a 2;
val f_a_b = fn : int -> int

        fun f_a_b(c) = 1*2 + 2*3 + c*4
                              8       + c*4;

- f_a_b 3;
val it = 20 : int

- f_a 5 10;
val it = 57 : int
```

Curried functions, continued

At hand:

```
- fun f a b c = a*2 + b*3 + c*4;
val f = fn : int -> int -> int -> int
```

Note that the expression

```
f 10 20 30;
```

is evaluated like this:

```
((f 10) 20) 30
```

In C, it's said that "declaration mimics use"—a declaration like int f() means that if you see the expression f(), it is an int. We see something similar with ML function declarations:

```
fun add(x,y) = x + y        Call: add (3, 4)

fun add x y = x + y         Call: add 3 4
```

Curried functions, continued

Problem: Define a curried function named mul to multiply two integers. Using a partial application, use a val binding to create a function equivalent to fun double(n) = 2 * n.

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

- val L = m_to_n ~5 5;
val L = [~5,~4,~3,~2,~1,0,1,2,3,4,5] : int list
```

Problem: Create the function iota, described on slide 78. (iota(3) produces [1,2,3].)

Curried functions, continued

What's happening here?

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

- add double(3) double(4);
Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         int -> int
  in expression:
    add double
```

Curried functions, continued

Problem—fill in the blanks:

```
fun add x y z = x + y + z;


val x = add 1;


val xy = x 2;


xy 3;


xy 10;


x 0 0;
```

Curried functions, continued

Here is sort from slide 125:

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

A curried version of sort:

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

Usage:

```
- val intSort = sort  intLessThan;
val int_sort = fn : int list -> int list

- int_sort [4,2,1,8];
val it = [1,2,4,8] : int list
```

Why does the curried form have the function as the first argument?

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
```

See the Resources page on the website for a link to documentation for the Basis.

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/folding

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

Contrast the types of applyToAll and map. Which is more useful?

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

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

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, continued

Here's one way to generate a string with all the ASCII characters:

```
- implode (map chr (m_to_n 0 127));
val it =
  "\^@\^A\^B\^C\^D\^E\^F\a\b\t\n\v\f\r\^N\^O\^P\^Q\^R\^S\^T\^U\^V\^W\^X\^Y\^Z\^[
\^\\^]\^^\^_ !\"#$%&'()*+,-./0123456789:;<=>?@ABCDE#"
  : string
```

(Note that the full value is not shown—the trailing # indicates the value was truncated for display.)

Problem: Write a function `equalsIgnoreCase` of type `string * string -> bool`. The function `Char.toLower` (`char -> char`) converts upper-case letters to lower case and leaves other characters unchanged.

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.

Remember that map is curried, too:

```
- val addTenToAll = map (add 10);
val addTenToAll = fn : int list -> int list

- addTenToAll [3,1,4,5];
val it = [13,11,14,15] : int list
```

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*.

Mapping with anonymous functions, continued

Explain the following:

```
- map (fn(s) => (size(s), s)) ["just", "try", "it"];
val it = [(4,"just"),(3,"try"),(2,"it")] : (int * string) list
```

Problem: Recall this mapping of a partial application:

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

Do the same thing but use an anonymous function instead.

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.

Problem: Explain 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

Another predicate-based function is List.partition:

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

- List.partition (fn(s) => size(s) <= 3) ["a", "test", "now"];
val it = (["a","now"],["test"]) : string list * string list
```

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;
```

Notes:

- TextIO.openIn opens a file for reading.
- TextIO.input reads an entire file and returns it as a string.
- Study the use of anonymous functions, mapping, and partial application.
- No loops, no variables, no recursion at this level.

How much code would this be in Java? Do you feel confident the code above is correct?

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
```

Reduction, continued

Another name for reduction is "folding"; There are two built-in reduction/folding functions: foldl and foldr. Contrast their types with the implementation of reduce shown above:

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

- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

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

Here's an example of foldr:

```
- foldr op+ 0 [5,3,9,2];
val it = 19 : int
```

What are the differences between reduce and foldr?

Speculate: What's the difference between foldl and foldr?

Reduction, continued

At hand:

```
- foldr;                                          (* foldl has same type *)
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- reduce;
val it = fn : ('a * 'a -> 'a) -> 'a list -> 'a
```

Our reduce has two weaknesses: (1) It can't operate on an empty list. (2) The operation must produce the same type as the list elements.

Consider this identity operation:

```
- foldr op:: [ ] [1,2,3,4];
val it = [1,2,3,4] : int list
```

Here are the op:: (cons) operations that are performed:

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

Note that the empty list in op::(4, [ ]) comes from the call. (Try it with [10] instead of [ ].)

Reduction, continued

At hand:

```
- foldr op:: [ ] [1,2,3,4];
val it = [1,2,3,4] : int list
```

foldl (note the "L") folds from the left, not the right:

```
- foldl op:: [ ] [1,2,3,4];
val it = [4,3,2,1] : int list
```

Here are the op:: calls that are made:

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

Reduction, continued

In some cases foldl and foldr produce different results. In some they don't:

```
- foldr op^ "!" ["a","list","of","strings"];
val it = "alistofstrings!" : string

- foldl op^ "!" ["a","list","of","strings"];
val it = "stringsoflista!" : string

- foldr op+ 0 [5,3,9,2];
val it = 19 : int

- foldl op+ 0 [5,3,9,2];
val it = 19 : int

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

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

What characteristic of an operation leads to different results with foldl and foldr?

travel, revisited

Here's a version of travel (slide 107) that uses mapping and reduction (folding) 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 = foldr addTuples (0,0) tuples
   in
      if displacement = (0,0) then "Got home"
                                  else "Got lost"
   end
```

How confident are we that it is correct? Would it be longer or shorter in Java?

Even more with functions

Composition

Manipulation of operands

Composition of functions

Given two functions F and G, the *composition* of F and G is a function C that for all values of x, C(x) = F(G(x)).

Here is a primitive compose function that applies two functions in turn:

```
- fun compose(F,G,x) = F(G(x));
val compose = fn :   ('a -> 'b) * ('c -> 'a) * 'c -> 'b
```

Usage:

```
- length;
val it = fn : 'a list -> int

- explode;
val it = fn : string -> char list

- compose(length, explode, "testing");
val it = 7 : int
```

Could we create a function composeAll([f1, f2, ... fn], x) that would call f1(f2(...fn(x)))?

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
```

Consider the types with respect to the type of `op o`:

```
'a is 'a list
'b is int
'c is string

(('a list -> int) * (string -> 'a list)) -> (string -> int)
```

Composition, continued

When considering the type of a composed function only the types of the leftmost and rightmost functions come into play.

Note that the following three compositions all have the same type. (Yes, the latter two are doing some "busywork"!)

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

- length o explode o implode o explode;
val it = fn : string -> int

- length o rev o explode o implode o rev o explode;
val it = fn : string -> int
```

A COMMON ERROR is to say the type of `length o explode` is something like this:

`string -> 'a list -> int` **<u>(WRONG!!!)</u>**

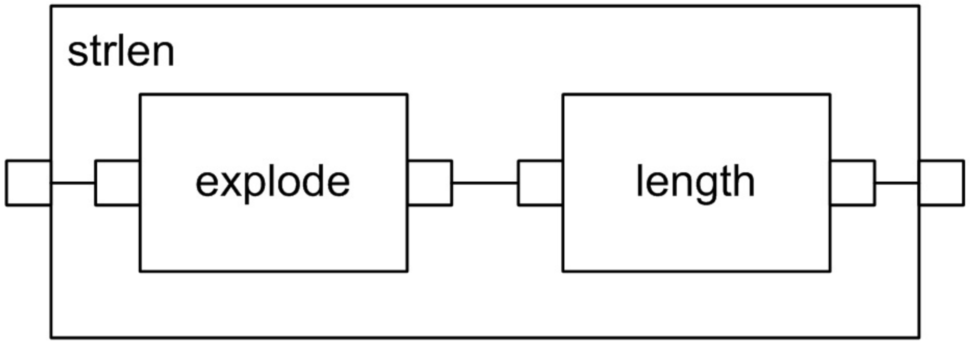
<u>Assuming a composition is valid, the type is based only on the input of the rightmost function and the output of the leftmost function.</u>

Composition, continued

For reference:

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

Analogy: Composition is like bolting machines together.



Because these machine models assume left to right data flow, explode comes first.

Composition, continued

Recall order and swap:

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

fun swap(x, y) = (y, x)
```

A descOrder function can be created with composition:

```
- val descOrder = swap o order;
val descOrder = fn : int * int -> int * int

- descOrder(1,4);
val it = (4,1) : int * int
```

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"].)

Composition, continued

Problem: Create two functions second and third, which produce the second and third elements of a list, respectively:

```
- second([4,2,7,5]);
val it = 2 : int

- third([4,2,7,5]);
val it = 7 : int
```

Problem: The function xrepl(x, n) produces a list with n copies of x:

```
- xrepl(1, 5);
val it = [1,1,1,1,1] : int list
```

Create a function repl(s, n), of type string * int -> string, that produces a string consisting of n copies of s. For example, repl("abc", 2) = "abcabc".

Problem: Compute the sum of the odd numbers between 1 and 100, inclusive. Use only composition and applications of op+, iota, isEven, foldr, 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

```
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
```

```
- 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
```

Data structures with datatype

A shape datatype

An expression model

An infinite lazy list

A simple 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 = [12.5663706144,3.75,12.5663706144,0.0,1.0] : real list
```

What does the following function do?

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

A model of expressions using datatype

Here is a set of types that can be used to model a family of ML-like expressions:

```
datatype ArithOp = Plus | Times | Minus | Divide;

type Name = string          (* Makes Name a synonym for string *)

datatype Expression =
    Let of (Name * int) list * Expression
  | E   of Expression * ArithOp * Expression
  | Seq of Expression list
  | Con of int
  | Var of Name;
```

Note that it is recursive—an Expression can contain other Expressions.

Problem: Write some valid expressions.

Expression, continued

The expression 2 * 4 is described in this way:

```
E(Con 2, Times, Con 4))
```

Consider a function that evaluates expressions:

```
- eval(E(Con 2, Times, Con 4));
val it = 8 : int
```

The Let expression allows integer values to be bound to names. The pseudo-code

```
let a=10, b=20, c=30
in a + (b * c)
```

can be expressed like this:

```
- eval(Let([("a",10),("b",20),("c",30)],
      E(Var "a", Plus, E(Var "b", Times, Var "c"))));
val it = 610 : int
```

Expression, continued

Let expressions may be nested. The pseudo-code:

```
let a = 1, b = 2
in a + ((let b = 3 in b*3) + b)
```

can be expressed like this:

```
- eval(Let([("a",1),("b",2)],
      E(Var "a", Plus,
         E(Let([("b",3)],   (* this binding overrides the first binding of "b" *)
           E(Var "b", Times, Con 3)), Plus, Var "b"))));
val it = 12 : int
```

The Seq expression allows sequencing of expressions and produces the result of the last expression in the sequence:

```
- eval(Seq [Con 1, Con 2, Con 3]);
val it = 3 : int
```

Problem: Write eval.

Expression, continued

Solution:

```
fun lookup(nm, nil) = 0
  | lookup(nm, (var,value)::bs) = if nm = var then value else lookup(nm, bs);

fun eval(e) =
  let
    fun eval'(Con i, _) = i
      | eval'(E(e1, Plus, e2), bs) = eval'(e1, bs) + eval'(e2, bs)
      | eval'(E(e1, Minus, e2), bs) = eval'(e1, bs) - eval'(e2, bs)
      | eval'(E(e1, Times, e2), bs) = eval'(e1, bs) * eval'(e2, bs)
      | eval'(E(e1,Divide,e2), bs) = eval'(e1, bs) div eval'(e2,bs)
      | eval'(Var v, bs) = lookup(v, bs)
      | eval'(Let(nbs, e), bs) = eval'(e, nbs @ bs)
      | eval'(Seq([ ]), bs) = 0
      | eval'(Seq([e]), bs) = eval'(e, bs)
      | eval'(Seq(e::es), bs) = (eval'(e,bs); eval'(Seq(es),bs))
    in
        eval'(e, [ ])
  end;
```

How can eval be improved?

An infinite lazy list

A *lazy list* is a list where values are created as needed.

Some functional languages, like Haskell, use *lazy evaluation*—values are not computed until needed. In Haskell the infinite list 1, 3, 5, ... can be created like this: [1,3 ..].

```
% hugs
Hugs> head [1,3 ..]
1

Hugs> head (drop 10 [1,3 ..])
21
```

Of course, you must be careful with an infinite list:

```
Hugs> length [1,3 ..]
(...get some coffee...check mail...^C)
{Interrupted!}

Hugs> reverse [1,3 ..]
ERROR - Garbage collection fails to reclaim sufficient space
```

An infinite lazy list, continued

ML does not use lazy evaluation but we can approach it with a data structure that includes a function to compute results only when needed.

Here is a way to create an infinite head/tail list with a datatype:

```
datatype 'a InfList = Nil
                |       Cons of 'a * (unit -> 'a InfList)

fun head(Cons(x, _)) = x;
fun tail(Cons(_, f)) = f();[1]
```

Note that 'a is used to specify that values of any (one) type can be held in the list.

A Cons constructor serves as a stand-in for op::, which can't be overloaded.

Similarly, we provide head and tail functions that mimic hd and tl but operate on a Cons.

[1]Adapted from *ML for the Working Programmer* L.C. Paulson

An infinite lazy list, continued

```
datatype 'a InfList = Nil
            |   Cons of 'a * (unit -> 'a InfList)

fun head(Cons(x,_)) = x;
fun tail(Cons(_,f)) = f();
```

Here's what we can do with it:

```
- fun byTen n = Cons(n, fn() => byTen(n+10));
val byTen = fn : int -> int InfList

- byTen 100;
val it = Cons (100,fn) : int InfList

- tail it;
val it = Cons (110,fn) : int InfList

- tail it;
val it = Cons (120,fn) : int InfList
```

Try it!

An infinite lazy list, continued

More fun:

```
fun toggle "on" = Cons("on", fn() => toggle("off"))
  |  toggle "off" = Cons("off", fn() => toggle("on"))

- toggle "on";
val it = Cons ("on",fn) : string InfList

- tail it;
val it = Cons ("off",fn) : string InfList

- tail it;
val it = Cons ("on",fn) : string InfList

- tail it;
val it = Cons ("off",fn) : string InfList
```

Problem: Write `drop(L,n)`:

```
- drop(byTen 100, 5);
val it = Cons (150,fn) : int InfList
```

An infinite lazy list, continued

Problem: Create a function repeatValues(L) that infinitely repeats the values in L.

```
- repeatValues;
val it = fn : 'a list -> 'a InfList

- repeatValues (explode "pdq");
val it = Cons (#"p",fn) : char InfList

- tail it;
val it = Cons (#"d",fn) : char InfList

- tail it;
val it = Cons (#"q",fn) : char InfList

- tail it;
val it = Cons (#"p",fn) : char InfList

- tail it;
val it = Cons (#"d",fn) : char InfList
```