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 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 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 largest = 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, <u>the pattern is the literal for unit</u>. 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 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
```

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?