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.

```ml
- 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:

```ml
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", ... "xxxxxxxxxxx"]. (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:

\[ \text{fun larger}(f1, f2, x) = \text{if } f1(x) > f2(x) \text{ then } f1 \text{ else } f2; \]

\[
\text{val larger = fn : ('a -> int) * ('a -> int) * 'a -> 'a -> int}
\]

\[\text{- val g = larger(double, square, 5);} \]
\[\text{val g = fn : int -> int}
\]

\[\text{- g(5);} \]
\[\text{val it = 25 : int}
\]

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

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

\[\text{- (larger(double, square, ~4)) (10);} \]
\[\text{val it = 100 : int}
\]
A flexible sort

Recall order(ed)_pair, insert, and sort from tally (slide 112). They work together to sort a (char * int) list.

\[
\text{fun ordered\_pair((\_, v1), (_, v2)) = v1 > v2}
\]

\[
\begin{align*}
\text{fun insert(v, \[])} & = [v] \\
& | \quad \text{insert(v, x::xs) = if ordered\_pair(v,x) then v::x::xs else x::insert(v, xs)}
\end{align*}
\]

\[
\begin{align*}
\text{fun sort([\[])} & = [\[] \\
& | \quad \text{sort(x::xs) = insert(x, sort(xs))}
\end{align*}
\]

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:

```ml
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:

```ml
- 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:

```ml
- 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:

```ml
- (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 with 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:

```ml
- 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:

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

Note that the expression

```plaintext
f 10 20 30;
```

is evaluated like this:

```plaintext
((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:

```plaintext
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 \texttt{mul} to multiply two integers. Using a partial application, use a \texttt{val} binding to create a function equivalent to \texttt{fun double(n) = 2 * n}.

Here is a curried implementation of \texttt{m_to_n} (slide 76):

\begin{verbatim}
- 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
\end{verbatim}

Problem: Create the function \texttt{iota}, described on slide 78. (\texttt{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 * int
  operand: int -> int
  in expression:
  add double
Curried functions, continued

Problem—fill in the blanks:

```ml
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:

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

A curried version of `sort`:

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

Usage:

```ml
- 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 \texttt{fun add x y = x + y} is this:

\small
\begin{verbatim}
- fun add x =
  let
    fun add' y = x + y
  in
    add'
  end
val add = fn : int -> int -> int   (Remember associativity: int -> (int -> int) )
\end{verbatim}

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

\small
\begin{verbatim}
- add 3;
  val it = fn : int -> int

- it 4;
  val it = 7 : int

- add' 3 4;
  val it = 7 : int
\end{verbatim}

\normalsize
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.

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

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

Consider a partial application of `map`:

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

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

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

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

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

(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);

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:

\[
\text{val double = fn(n) => n * 2;}
\]
\[
\text{val double = fn : int -> int}
\]

The expression being evaluated, \( fn(n) => n * 2 \), is a simple example of a \textit{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 \texttt{triple} function we might do this:

\[
\text{map (fn(n) => n * 3) [3, 1, 5, 9];}
\]
\[
\text{val it = [9,3,15,27] : int list}
\]

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 \textit{match expression} is ML-specific. A more general term for an expression that defines a nameless function is a \textit{lambda expression}. 

Mapping with anonymous functions, continued

Explain the following:

- \( \text{map (fn(s) => (size(s), s)) } ["\text{just}", "\text{try}", "\text{it}"] \);
  \[\text{val it} = [(4,"\text{just}"),(3,"\text{try}"),(2,"\text{it}")] : (\text{int} \times \text{string}) \text{ list} \]

Problem: Recall this mapping of a partial application:

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

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 \textit{true}. Here's one way to write \textit{filter}:

\[
- \begin{align*}
\text{fun filter } F \ [ ] &= [ ] \\
\text{filter } F \ (x::xs) &= \text{if } (F \ x) \ \text{then } x::(\text{filter } F \ xs) \\
&\quad \text{else } (\text{filter } F \ xs);
\end{align*}
\]

val filter = fn : ('a -> bool) -> 'a list -> 'a list

It is said that \( F \) is a \textit{predicate}—inclusion of a list element in the result is predicated on whether \( F \) returns true for that value.

Problem: Explain the following.

\[
- \begin{align*}
\text{val } f &= \text{List.filter } (\text{fn}(n) => n \ mod \ 2 = 0); \\
\text{val } f &= \text{fn} : \text{int list} -> \text{int list}
\end{align*}
\]

- \( f \ [5,10,12,21,32] \);
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 \texttt{List.partition}:

\begin{verbatim}
- 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
\end{verbatim}

\texttt{String.tokens} uses a predicate to break a string into tokens:

\begin{verbatim}
- Char.isPunct;
  val it = fn : char -> bool

- String.tokens Char.isPunct "a,bc:def.xyz";
  val it = ["a","bc","def","xyz"] : string list
\end{verbatim}

Problem: What characters does \texttt{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:

```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 \texttt{grep}, continued

Implementation

\begin{verbatim}
  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;
\end{verbatim}

Notes:
- \texttt{TextIO.openIn} opens a file for reading.
- \texttt{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:

```ml
- 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:

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

What happens:

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

Or,

```ml
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 \texttt{foldl} and \texttt{foldr} produce different results. In some they don't:

\begin{itemize}
  \item \texttt{- foldr op^ "!' "["a","list","of","strings"];}
        \texttt{val it = "alistofstrings!" : string}
  \item \texttt{- foldl op^ "!' "["a","list","of","strings"];}
        \texttt{val it = "stringsoflista!" : string}
  \item \texttt{- foldr op+ 0 [5,3,9,2];}
        \texttt{val it = 19 : int}
  \item \texttt{- foldl op+ 0 [5,3,9,2];}
        \texttt{val it = 19 : int}
  \item \texttt{- foldl op@ [ ] [[1,2],[3],[4,5]];}
        \texttt{val it = [4,5,3,1,2] : int list}
  \item \texttt{- foldr op@ [ ] [[1,2],[3],[4,5]];}
        \texttt{val it = [1,2,3,4,5] : int list}
\end{itemize}

What characteristic of an operation leads to different results with \texttt{foldl} and \texttt{foldr}?
travel, revisited

Here's a version of travel (slide 107) that uses mapping and reduction (folding) instead of explicit recursion:

```ml
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 \texttt{compose} function that applies two functions in turn:

\[- \textbf{fun} \ \texttt{compose}(F,G,x) = F(G(x)); \]
\[- \text{val} \ \texttt{compose} = \text{fn} : ('a -> 'b) * ('c -> 'a) * 'c -> 'b \]

Usage:

\[- \texttt{length}; \]
\[- \text{val} \ \text{it} = \text{fn} : 'a \text{ list} -> \text{int} \]

\[- \texttt{explode}; \]
\[- \text{val} \ \text{it} = \text{fn} : \text{string} -> \text{char list} \]

\[- \texttt{compose(length, explode, "testing")}; \]
\[- \text{val} \ \text{it} = 7 : \text{int} \]

Could we create a function \( \texttt{composeAll}([f1, f2, \ldots \ fn], x) \) that would call \( f1(f2(\ldots fn(x))) \)?
The composition operator (\circ)

There is a composition operator in ML:

- \texttt{op o; (* lower-case "Oh" *)}
  \begin{verbatim}
  val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
  \end{verbatim}

Two functions can be composed into a new function:

- \texttt{val strlen = length \circ explode;}
  \begin{verbatim}
  val strlen = fn : string -> int
  \end{verbatim}

- \texttt{strlen "abc");}
  \begin{verbatim}
  val it = 3 : int
  \end{verbatim}

Consider the types with respect to the type of \texttt{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  (WRONG!!!)`

Assuming a composition is valid, the type is based only on the input of the rightmost function and the output of the leftmost function.
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","ows","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. \texttt{compN} \( f \ n \) composes \( f \) with itself \( n \) times:

\begin{verbatim}
  - fun compN f 1 = f
    |  compN f n = f o compN f (n-1);

  val compN = fn : ('a -> 'a) -> int -> 'a -> 'a
\end{verbatim}

Usage:

\begin{verbatim}
- 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
\end{verbatim}

Could we create \texttt{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:

```ml
- 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:

```ml
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:

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

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

```ml
- 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 = [" ---> "," ---> ",&quot; ---> "] : 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:

\[
\text{fun} \ f \ x \ y = g(x,y*2);
\]

\[
\text{val} \ f = \text{fn} : \text{int} -> \text{int} -> \text{int}
\]

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:

\[
\text{map (uncurry } f) [(1,2), (3,4), (5,6)];
\]

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:

- \texttt{fun swapArgs f x y = f y x;}
  \texttt{val swapArgs = fn : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c}

Usage:

- \texttt{fun cat s1 s2 = s1 ^ s2;}
  \texttt{val cat = fn : string -> string -> string}

- \texttt{val f = swapArgs cat;}
  \texttt{val f = fn : string -> string -> string}

- \texttt{f "a" "b";}
  \texttt{val it = "ba" : string}

- \texttt{map (swapArgs (curry op^) "x") ["just", "a", "test"];}
  \texttt{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

```ml
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
```

```ml
- 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
```