Functional programming

- Functional style makes heavy use of **functions as values**
- Hence, functional languages provide powerful constructs for manipulating functions

Higher-order functions

- Functions that operate on functions
- Most languages have at least a weak form, e.g. C's quicksort (in stdlib.h):

```
void qsort(void *base,
           size_t nmem,
           size_t size,
           int (*compar)(const void*,
                         const void *));
```

Using C's qsort

```
int double_greater(const void* r1,
                   const void* r2){
  return (*(double *)r1 > *(double *)r2);
}

double array[5] =
     { 1.2, 24.2, 4.2, 9.0, 17.3 };
qsort(array, 5, sizeof(double),
      double_greater);
```

ML: real higher order functions

```
fun qsort(greaterThan:('a * 'a) -> bool,
          nil: 'a list):'a list = ...

val foo = [1.2, 24.2, 4.2, 9.0, 17.3];
val sorted = qsort(op >, foo);
```

qsort implementation

```
fun qsort(greaterThan, nil)  = nil
  | qsort(greaterThan, pivot::rest) =
    let
        fun partition(pivot, nil) = (nil, nil)
          | partition(pivot, x::xs) =
            let val (lessOrEqual, greater) =
                                partition(pivot, xs);
            in  if greaterThan(x, pivot)
                    then (lessOrEqual, x::greater)
                    else (x::lessOrEqual, greater)
            end;
        val (lessOrEqual, greater) =
                                partition(pivot, rest);
    in
        qsort(greaterThan, lessOrEqual)
             @ pivot::qsort(greaterThan, greater)
    end;
```

Anonymous functions

```
(* identity function *)
- fn x => x;
val it = fn : 'a -> 'a

(* adds 5 to its argument *)
- fn x => x + 5;
val it = fn : int -> int

(* constructs and immediately applies function *)
- (fn x => x + 5) 4;
val it = 9 : int
```

Functions as parameters

```
datatype ComplexNum = Complex of real * real;

val myNums = [Complex(0.0, 1.0),
              Complex(1.2, 0.6),
              Complex(2.5, 3.2)];

val sortedByReals =
    qsort(fn (Complex(r1,_), Complex(r2,_))
              => r1 > r2, myNums);

val sortedByImag =
    qsort(fn (Complex(_,i1), Complex(_,i2))
              => i1 > i2, myNums);
```

Functions as return values

```
(* Makes a single-argument function that
  adds x to its argument *)
fun makeAddX(x) = fn y => x + y;

(* A function that adds 5 to any int *)
val add5 = makeAddX(5);

(* foo = 9. *)
val foo = add5 4;
```

Less silly return values

```
datatype complexPart = RealPart
                     | ImagPart;

fun sortBy(RealPart) =
     (fn (Complex(r1,_), Complex(r2,_))
          => r1 > r2)
  | sortBy(ImagPart) =
     (fn (Complex(_,i1), Complex(_,i2))
          => i1 > i2);

qsort(sortBy(RealPart), myNums);
```

Currying

- Named after Haskell Curry
  - (There's also a language called Haskell. Yes, it has currying.)
- Key insight: any multi-argument function can be written using single arguments and higher-order functions:

a * b -> c ° **a -> b -> c**

Hand curried addition?

```
val add = fn (x, y) = x + y;
val foo = add(3, 4);

val handCurriedAdd =
     fn (x) => fn(y) => x + y;

val addFive   = handCurriedAdd 5;
val eleven    = addFive 6;
val seventeen = (handCurriedAdd 8) 9;
val twenty    = handCurriedAdd 10 10;
```

ML built-in currying syntax

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

- curriedAdd 5;
val it = fn : int -> int

- val thirty = curriedAdd 15 15;
val thirty = 30 : int
```

More complex currying

```
(* curried qsort no comma between params *)
fun qsort (greaterThan:('a * 'a) -> bool)
          (elems:'a list) = ...

(* Not taking advantage of currying. *)
fun sortComplexByReal elems =
      qsort (sortBy(RealPart)) elems);

(* Takes advantage of curried syntax *)
val sortComplexByImag = qsort (sortBy(RealPart));
```

Exercise

```
fun composeUnaryIntOps
        (f:int -> int)
        (g:int -> int) =
            fn x => f (g x);
```