Functional Programming Makes a Comeback

Chuck Allison

Better Software 2008

Observation

- Popular object-oriented languages have some degree of support for functional programming
  - Python, Ruby, C++, Java, C#, D, F#, Scala, Groovy
- C++, Java, and C# are adding even more support
- What's so cool about FP?

Objective

- To appreciate the contribution the functional style of programming makes to problem solving
- To become familiar with the functional style of programming in modern languages

Agenda

- What is Functional Programming?
- History of FP
- **ML** – The first Modern FP Language
- FP in Other Modern Languages
  - Python, D, C++, Scala, C#

What is Functional Programming?

The First Programming Language!

Functional Programming

- A style (paradigm) of programming where **functions** are the basic building blocks
  - Functions are used for what they *return*
  - *Not* for achieving side effects (e.g., *assignment*)
- Functions are "first-class" entities
  - Can be passed as arguments, returned as results
  - Can be created "on-the-fly"
- FP programs focus more on *what* you want
  - Not so much on *how* to compute it

Key FP Features

- Higher-order functions
  - Functions can be passed and returned
- Nested functions
  - With *closures* (a type of delegate)
- Partial function application
  - Aka "currying"
- No assignment statement
  - High-level (and thread-safe) programming
- No loops
  - Recursion preferred

History of Functional Programming

Before Computers…

- …there was *computation*
- Mathematical operations and functions
- Symbolic manipulation
    - a key focus of 20th Century mathematics

Great Moments in Computation

- Turing Machines
  - Where imperative programming originated
  - Led to FORTRAN, Algol, C, etc.
- Church's Lambda Calculus
  - Led to Lisp, Scheme, ML, Haskell, etc.
- Both happened in the 1930s!

Milestones in Functional Programming

- 1936 – Church's Lambda Calculus
- 1958 – First release of Lisp
- Early 1970s – ML
  - Static typing
  - Type inference
  - Function templates ("parametric polymorphism")
- Mid 1970's – Scheme (Lisp for the masses)
  - Block scoping; tail recursion optimization
- 1990 – Haskell (**S**oftware **T**ransactional **M**emory)
- 1998 – Erlang (Fault tolerant, Message Passing)

Introduction to ML

The First Modern Functional Programming Language

ML Topics

- Getting Started
  - Types, Expressions, Bindings
  - Functions, Type Inference, Tuples
- Lists and Recursion
  - List operations, Pattern-Matching, Type Variables
- Higher-order Functions
  - Lambda expressions (anonymous functions)
  - Currying, Folding
  - Nested Functions and Closures

The ML Interpreter

```
$ sml
Standard ML of New Jersey v110.67 [built: Thu Nov 15 10:18:08
2007]
- 1 + 2;
val it = 3 : int
- 1 - 2;
val it = ~1 : int
- "the" ^ "end";
val it = "theend" : string
- 2.0 + 3.0;
val it = 5.0 : real
- 2.0 + 5;
stdIn:5.1-5.8 Error: operator and operand don't agree [literal]
  operator domain: real * real
  operand:         real * int
  in expression:
    2.0 + 5
```

```
- #"a";
val it = #"a" : char
- 1 = 2;
val it = false : bool
- 1 > 2 andalso 3 > 2;
val it = false : bool
- 1 < 2 orelse 3 > 2;
val it = true : bool
- 1.0 = 2.0;
stdIn:9.1-9.10 Error: operator and operand don't agree
[equality type required]
  operator domain: ''Z * ''Z
  operand:         real * real
  in expression:
    1.0 = 2.0
```

```
- true orelse 1 div 0 = 0;
val it = true : bool
- if 1 > 0 then "greater" else "not"; (* an expression *)
val it = "greater" : string
- 2 / 5;
stdIn:11.1-11.6 Error: operator and operand don't agree
[literal]
  operator domain: real * real
  operand:         int * int
  in expression:
    2 / 5
- 2.0 / 5.0;
val it = 0.4 : real
- 2 div 5;
val it = 0 : int
```

Basic ML Types

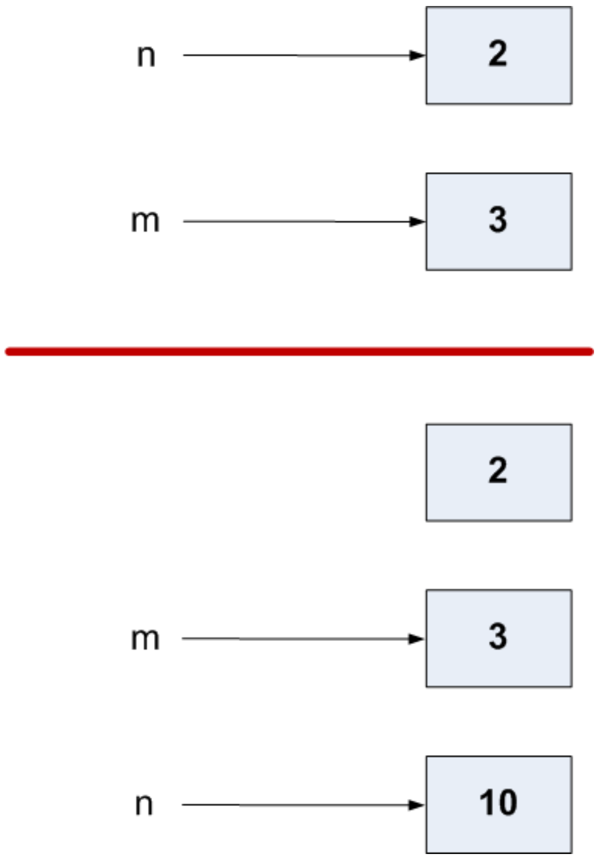
- int
- real
  - Not an "equality type"
  - Can't mix with **int**
- string
- char
- bool
  - **andalso** and **orelse** are *short-circuiting*

Binding Variables

- **val** keyword
- The type of the initializer expression becomes the type of the initialized variable
  - "type inference"
- It's not really "variable"
  - i.e., it's not mutable
  - can only be **initialized**
- But variables can be "rebound"

Variable Bindings

```
- val n = 2;
val n = 2 : int
- val m = n + 1;
val m = 3 : int
- val n = 10;
val n = 10 : int
- m;
val it = 3 : int
```



Basic Operators

- For Integers:
  - +, -, *, div, mod, ~
- For Reals:
  - +, -, *, /, ~
- For Strings:
  - ^ (concatenation)

Functions in ML

- ML functions take exactly 1 argument

- That argument can be an aggregate
  - tuple, list, etc.

- Parentheses not needed:
  - f x;

Calling Functions in ML

```
- floor 2.5;
val it = 2 : int
- real 2;
val it = 2.0 : real
- explode "hello";
val it = [#"h",#"e",#"l",#"l",#"o"] : char list
- implode [#"h",#"e",#"l",#"l",#"o”];
val it = "hello" : string
- floor 2.6 + 1;
val it = 3 : int
- floor (2.6 + 2.0);
val it = 4 : int
```

Commonly Used Functions

- Numeric conversions:
  - real, floor, ceil, round, trunc
- Character-to-integer conversion:
  - chr, ord
- Character-to-string conversion:
  - str, explode, implode

Defining Functions in ML

```
- fun square x = x * x;
val square = fn : int -> int
- square 2;
val it = 4 : int
- square 2 + 2;
val it = 6 : int
- square (2 + 2);
val it = 16 : int
```

Annotating Functions with Types

```
- fun square x:real = x*x;     (* annotate argument type *)
val square = fn : real -> real
- square 2;
stdIn:2.1-2.9 Error: operator and operand don't agree [literal]
  operator domain: real
  operand:         int
  in expression:
    square 2
- square 2.0;
val it = 4.0 : real
- square (real 2);             (* Note paren placement! *)
val it = 4.0 : real
- fun square (x:real):real = x*x; (* annotate return type *)
val square = fn : real -> real
```

Defining Functions in ML

- **fun** keyword
- function name
- single parameter = expression;

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

(x,y) is a *tuple*
  ```

Tuples

- Can hold an *arbitrary number* of elements
  - Of *any* type
- Accessed positionally with #1, #2, etc.
- Can perform tuple assignment

Tuple Access and Assignment

```
- val twonums = (2,3);
val twonums = (2,3) : int * int
- #1 twonums;
val it = 2 : int
- #2 twonums;
val it = 3 : int
- val (x,y) = twonums;
val x = 2 : int
val y = 3 : int
- x = #1 twonums;
val it = true : bool
```

Summary

Getting Started with ML

- ML is *strongly typed*
  - no mixing of types in expressions
  - types are statically determined
- Variables are *bound to values*
  - the value can't be changed
  - but the variable can be rebound to another value
- Functions take a *single argument* and return a *single value*
  - Function bodies are a *single expression*

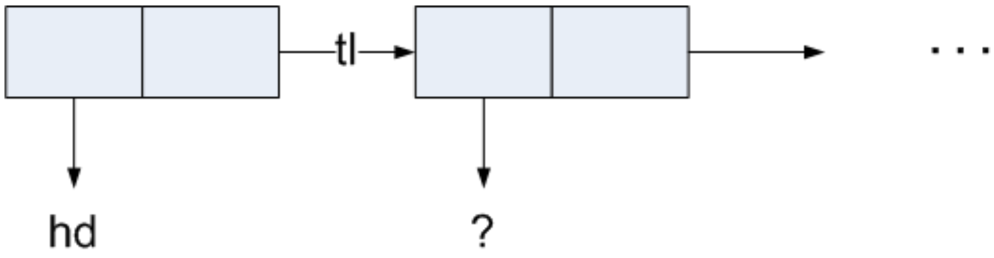
Exercises

Getting Started

- Write an ML function that takes a real number and returns its cube (3^{rd} power)
- Write an ML function that returns the smallest of 3 integers
- Write an ML function that returns the sum of its 3 integer arguments

Lists in ML

- Must be homogeneous
  - i.e., each element must be of the same type
- Stored as linked lists of pairs of pointers
  - first element is the *head* (refers to a value)
  - second element is the *tail* (refers to rest of list)
    - is itself a list



Using Lists

```
- val a = [1,2,3];
val a = [1,2,3] : int list
- hd a;
val it = 1 : int
- tl a;
val it = [2,3] : int list
- hd (tl a);
val it = 2 : int
- null a;
val it = false : bool
- null [];
val it = true : bool
- nil;
val it = [] : 'a list
```

Basic List Operations

- Concatenation: **@**
  - [1,2] @ [3,4] => [1,2,3,4]
- Construct a node: **::**
  - 1::[2,3,4] => [1,2,3,4]
- Determine head: **hd**
  - hd [1,2,3,4] => 1
- Determine tail: **tl**
  - tl [1,2,3,4] => [2,3,4]
- Length: **length**

Writing List-processing Functions

- Done with *recursion* to visit each list element
- Typical pattern:
  - if list is empty
        process base case of recursion
    else
        process head, recurse on tail

Writing a Length Function for Lists

```
fun mylen x =
    if null x then 0
    else 1 + length (tl x);
```

Another Recursive Function

```
(* Sum of squares of 0 through n *)
- fun sumsq n =
=     if n = 0 then 0
=     else n*n + sumsq (n-1);
val sumsq = fn : int -> int
- sumsq 2;
val it = 5 : int
- sumsq 3;
val it = 14 : int
```

Pattern Matching

```
- fun sumsq 0 = 0
= |   sumsq n = n*n + sumsq (n-1);
val sumsq = fn : int -> int
- sumsq 2;
val it = 5 : int
- sumsq 3;
val it = 14 : int
- sumsq 0;
val it = 0 : int
```

mylen with Pattern Matching

```
- fun mylen nil = 0
= |   mylen (h::t) = 1 + mylen t; (* h is not used *)
val mylen = fn : 'a list -> int   (* a type variable *)
- mylen [];
val it = 0 : int
- mylen [1,2,3];
val it = 3 : int
```

Unused Variables

- If a variable won't be used, you can use the *underscore* for its name:
```
|    mylen (_::t) = 1 + mylen t;
```

- This applies in other contexts:
```
- var (_,y) = (1,2);
val y = 2 : int
```

Type Variables

- Notice the `'a` in the definition of `mylen`
- The operations of `mylen` are *type independent*
- Therefore, the type of the list can *vary*

```
- mylen ["hello", "goodbye"];
val it = 2 : int
```

- This is a form of *polymorphism*
  - "Parametric Polymorphism"
  - The inspiration for C++ function templates
- The type is fixed when the statement is *compiled*

Testing For List Membership

```
- fun member (_, nil) = false
= |   member (x, h::t) = x = h orelse member (x,t);
stdIn:27.26 Warning: calling polyEqual (* Ignore this *)
val member = fn : ''a * ''a list -> bool
- member (2,[1,2]);
val it = true : bool
- member (3,[1,2]);
val it = false : bool
- member (1, nil);
val it = false : bool
```

The ''a Type Variable

- Remember that reals can't be compared for equality
- The `member` function requires equality types
- The type variable `''a` stands for any equality type
  - can't call this function on a list of reals!

Multiple Type Variables

```
- fun outer (x,_,z) = (x,z);
val outer = fn : 'a * 'b * 'c -> 'a * 'c
- outer (1,2,3);
val it = (1,3) : int * int
- outer ("a","b","c");
val it = ("a","c") : string * string
```

Reversing a List

```
- fun reverse nil = nil
= |   reverse (h::t) = reverse t @ [h];
val reverse = fn : 'a list -> 'a list
- reverse [1,2,3];
val it = [3,2,1] : int list
- rev ["a","b","c"];    (* built-in function *)
val it = ["c","b","a"] : string list
```

Defining Local Variables

- The only locals we've seen are parameters
- The **let** expression defines local bindings
- They are used in the function body
  - Which is a single expression, remember

The `let` Expression

```
- fun days2ms days =
=     let
=         val hours = days * 24.0
=         val minutes = hours * 60.0
=         val seconds = minutes * 60.0
=     in
=         seconds * 1000.0
= end;
val days2ms = fn : real -> real
- days2ms 1.5;
val it = 129600000.0 : real
```

A Local Function Definition

```
fun union (x, nil) = x
|   union (x, head::rest) =
    let
        fun member (_, nil) = false
        |   member (x, h::t) = x = h orelse member (x,t)
    in
        if member(head, x) then union(x,rest)
        else head::union(x,rest)
    end;

- union (["a","b","c"],["b","c","d"]);
val it = ["d","a","b","c"] : string list
```

Summary
Lists and Recursion

- Lists have a head and a tail
  - the empty list is denoted by **nil**
- Patterns are matched in the order they appear
- ML allows parametric polymorphism
  - implicit type variables
- Place local bindings in a **let** block

Exercises

Lists and Recursion

- Write a function named **`repeats`** that determines if a list has two adjacent equal elements
- Write a function named **`unique`** that returns elements of a sorted list but ignoring duplicates.
- After reviewing the code or union, write a binary function named **`intersection`**, that returns only those elements common to both its input lists.

Functions are First-Class Entities

- Functions are like other values in that:
  - they can be *passed* as arguments to other functions
  - they can be *returned* from functions
  - they can be *bound* to variables

- A function that accepts or returns another function is a called a *higher-order* function
  - very useful!

Using Functions as Objects

```
- length;
val it = fn : 'a list -> int
- val f = length;
val f = fn : 'a list -> int
- fun apply (f,x) = f x;
val apply = fn : ('a -> 'b) * 'a -> 'b
- apply (f, [1,2,3]);
val it = 3 : int
```

Using Operator Functions

```
- op <;
val it = fn : int * int -> bool
- (op <) (3,4);
val it = true : bool
- val g = op <;
val g = fn : int * int -> bool
- g(4,3);
val it = false : bool
```

Quicksort in ML

```
fun quicksort (cmp, nil) = nil
|   quicksort (cmp, pivot::rest) =
    let
       fun partition nil = (nil,nil)
       |   partition(x::xs) =
          let
             val (below, above) = partition xs
          in
             if cmp(x,pivot) then (x::below, above)
             else (below, x::above)
          end;
       val (below, above) = partition(rest)
    in
       quicksort(cmp, below) @ [pivot] @ quicksort(cmp, above)
    end;
```

Using Quicksort

```
- val words = ["go","ahead","make","my","day"];
val words = ["go","ahead","make","my","day"] : string list
- quicksort(String.<,words);
val it = ["ahead","day","go","make","my"] : string list
- quicksort(String.>,words);
val it = ["my","make","go","day","ahead"] : string list
```

Anonymous Functions

- Called *lambda expressions* in other FP languages
- Sometimes it is more convenient to create a function on the fly
- Uses **fn arg => expr** syntax

```
- quicksort(fn (x,y) => x < y, [3,2,1]);
val it = [1,2,3] : int list
- quicksort(fn (x,y) => x > y, [1,2,3]);
val it = [3,2,1] : int list
```

Currying

- Named after Haskell Curry
- A flexible way of providing *multiple arguments* to a functions
- Allows *partial function evaluation*
  - So you can provide the other arguments later
- Technique:
  - For all but the last parameter, *a function is returned* that takes the next parameter
  - The last returned function returns the actual value

Currying Syntax

```
- fun f a = fn b => a + b;
val f = fn : int -> int -> int
- f 1;
val it = fn : int -> int
- f 1 2;
val it = 3 : int
- val g = f 1;
val g = fn : int -> int
- g 2;
val it = 3 : int
```

Currying Shorthand

```
- fun f a b = a + b;
val f = fn : int -> int -> int
- f 1;
val it = fn : int -> int
- f 1 2;
val it = 3 : int
- val g = f 1;
val g = fn : int -> int
- g 2;
val it = 3 : int
```

A Curried Quicksort

```
fun quicksort cmp L = if null L then nil else
    let
        val (pivot, rest) = (hd L, tl L)
        fun partition nil = (nil,nil)
        |   partition(x::xs) =
            let
                val (below, above) = partition xs
            in
                if cmp(x,pivot) then (x::below, above)
                else (below, x::above)
            end;
        val (below, above) = partition(rest)
    in
        quicksort cmp below @ [pivot] @ quicksort cmp above
    end;
```

Using the Curried Quicksort

```
- use "/Users/chuck/sort2.sml";
[opening /Users/chuck/sort2.sml]
val quicksort = fn : ('a * 'a -> bool) -> 'a list -> 'a list
val it = () : unit
- val sortasc = quicksort (op <);
val sortasc = fn : int list -> int list
- sortasc [3,2,1];
val it = [1,2,3] : int list
- sortasc [5,4,3];
val it = [3,4,5] : int list
- val sortdesc = quicksort (op >);
val sortdesc = fn : int list -> int list
- sortdesc [1,2,3];
val it = [3,2,1] : int list
- sortdesc [3,4,5];
val it = [5,4,3] : int list
```

Standard Higher-Order Functions

- **`map`**
  - Applies a unary function to each list element
  - Returns the resulting list
- **`foldl`**
  - Reduces a list to a value
  - Applies a binary function to each element with the accumulated value
  - Works left-to-right
- **`foldr`**
  - Like **foldl** but works right-to-left
- All are *curried*

Using map

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- map (fn x => x + 1) [1,2,3];
val it = [2,3,4] : int list
- val add1 = map (fn x => x + 1);
val add1 = fn : int list -> int list
- add1 [1,2,3];
val it = [2,3,4] : int list
- add1 [2,3,4];
val it = [3,4,5] : int list

- map (op +) [(1,2),(3,4),(5,6)];
val it = [3,7,11] : int list
```

Using foldl

```
(* Add list elements *)
- foldl (op +) 0 [1,2,3];  (* (((0+1)+2)+3), or… *)
val it = 6 : int           (* op+(3,op+(2,op+(1,0))) *)

(* Multiply them *)
- foldl (op * ) 1 [2,3,4]; (* op*(4,op*(3,op*(2,0))) *)
val it = 24 : int

(* Sum of squares: f(3,f(2,f(1,1))) *)
- foldl (fn (x, sofar) => sofar + x*x) 0 [1,2,3];
val it = 14 : int
```

Leveraging Currying

```
- val addup = foldl (op +) 0;
val addup = fn : int list -> int
- addup [1,2,3];
val it = 6 : int
- addup [2,3,4];
val it = 9 : int
- val concat = foldl (op ^) "";
val concat = fn : string list -> string
- concat ["how","now","brown","cow"];
val it = "cowbrownnowhow" : string
```

Using foldr

```
- val concat = foldr (op ^) "";
val concat = fn : string list -> string
- concat ["how","now","brown","cow"];
val it = "hownowbrowncow" : string
- val append5 = foldr (op ::) [5];
val append5 = fn : int list -> int list
- append5 [1,2,3];
val it = [1,2,3,5] : int list
```

Question

- **append5** is a little too specific

- How can we write a *generic* **append**?
  - i.e., build **append(n)** on-the-fly

A Generic **append**

```
- fun append n = foldr (op ::) [n];
val append = fn : 'a -> 'a list -> 'a list
- val append3 = append 3;
val append3 = fn : int list -> int list
- append3 [0,1,2];
val it = [0,1,2,3] : int list
```

Nested Functions and Closures

- **append3** made a *partial call* to **append**
  - A *function*, not a value, was returned
- The returned function used a binding from *outside* of its scope (**n**)
- The binding for **n** needs to be available after **append** returns
- What **append** actually returned is a *closure*
  - a function coupled with its *lexical environment*

More Examples

```
- fun bor bools = foldr (fn (a, b) => a orelse b) false bools;
val bor = fn : bool list -> bool
- bor [false,true,false];
val it = true : bool
- fun member x L = bor (map (fn y => x = y) L);
stdIn:82.5 Warning: calling polyEqual
val member = fn : ''a -> ''a list -> bool
- member 5 [3,4,5];
val it = true : bool
```

Design Exercise: Function Composition

- Data processing is often *a sequence of transformations* on data
  - e.g., remove punctuation, then change to lower case, then change all e's to 3's
- Packaging a sequence of functions into a single, comoposite function is called *function composition*
- **`f(s) <==> threes(lower(nopunct(s)))`**
- Just as currying allows reuse of a partially-evaluated function, composition allows a *sequence of operations* to be *reused as a unit*

Solution Approach

- We will be given a list of unary functions
  - This example requires the input and output types to be the same
- We need to return a unary function that applies each original function in reverse list order to obtain the final result
- Sounds like a job for lists and **foldr**

Using compose

```
use "/Users/chuck/compose.sml";
val compose = fn : ('a -> 'a) list -> 'a -> 'a
val it = () : unit
- fun add1 x = x + 1;
val add1 = fn : int -> int
- fun mult3 x = x*3;
val mult3 = fn : int -> int
- fun sub5 x = x - 5;
val sub5 = fn : int -> int
- val f = compose [add1,mult3,sub5];
val f = fn : int -> int
- f(1) ;
val it = ~11 : int
- f(20);
val it = 46 : int
```

Implementing **compose**

```
fun compose flist =
    fn x => foldr (fn (f, sofar) => f sofar) x flist;
```

We'll see this again in other languages…

FP Summary So Far

- Variables do not change
  - no *shared memory* problems (globals, threads, etc.)
- No loops
  - => no loop errors
  - use recursion instead
- Very high-level programming
  - facilitated by higher-order functions, anonymous functions, nested functions, currying
  - concise code!

FYI

- **OCaml** is an object-oriented ML
- Compiles to native code
  - runs very fast!
- Supports procedural, functional, and OO programming
- **F#** on .NET

Exercises

Higher-Order Functions

- Write a curried version of **union**; use **foldl** or **foldr**
- Repeat for **intersection**
- Write a curried version of **append**
  - Hint: use **foldr**; then “cons” (::) elements of the first list with the second

Functional Programming in Other Languages

Topics
Other Languages

- FP in Python
- FP in D
- FP in C++
- FP in Scala

About Python

- Python is a *dynamically typed* language
  - there is no "compile time"
  - dynamic OO programming
- Interpreted (but no JIT compiler)
- Easy to learn, read
  - indentation is *required*
- Lists and tuples are *indexable*
  - Lists are *mutable*; tuples are not

Lists in Python

```
>>> L=[1,2,2,3,3,3]
>>> for n in L: print L.count(n),
1 2 2 3 3 3
>>> L.index(2)
1
>>> L.append(5)
>>> L
[1, 2, 2, 3, 3, 3, 5]
>>> L.extend([5,5,5,5])
>>> L
[1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5]
>>> for i in range(4): L.insert(6+i, 4)
>>> L
[1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]
```

Slices

```
words = "now is the time".split()
print words
print words[1]
print words[0:2]
print words[1:]
print words[:2]
print words[-1]

''' Output:
['now', 'is', 'the', 'time']
is
['now', 'is']
['is', 'the', 'time']
['now', 'is']
time
'''
```

Defining Functions in Python

- **def** keyword
- Arguments can be *collected* into a *tuple parameter*
- Tuples can be *flattened* into arguments
- Python supports *nested functions* and *closures*

Functions in Python

```
def h(x):
    return x + 2

def r(s):
    return s*2

# g calls f on x:
def g(f, x):
    return f(x)

print g(h,3)        # prints 5
print g(r,'two')    # prints twotwo
#print g(2,3)       # error: 2 is not callable
```

Arguments and Tuples

```
def varargs(*args):
    for arg in args:
        print arg

varargs("one","two")
varargs(3,4,5)

''' Output:
one
two
3
4
5
'''
```

```
def fixargs(a,b):
    print 'a =', a
    print 'b =', b

pair = (1,"two")
fixargs(*pair)

''' Output:
a = 1
b = two
'''
```

Quicksort in Python

Uses List Comprehensions

```
def qsort(L):
   if len(L) <= 1: return L
   return qsort([lt for lt in L[1:] if lt < L[0]]) \
        + [L[0]] \
        + qsort([gt for gt in L[1:] if gt >= L[0]])
```

FP in Python

- **map** = same as ML
- **foldl** = **reduce**
- There is no **foldr**
  - but you can easily traverse lists backwards with the **reversed** iterator
- Currying is not directly supported
  - easily provided with a 7-line “wrapper” function

Python FP Examples

```
>>> map(lambda x: -x, [1,2,3])
[-1, -2, -3]
>>> [-x for x in [1,2,3]]
[-1, -2, -3]
>>> map(lambda x,y: x+y, [1,2,3],[4,5,6])
[5, 7, 9]
>>> map(operator.add, [1,2,3],[4,5,6])
[5, 7, 9]
>>> reduce(operator.add, map(lambda x: -x, [1,2,3]))
-6
>>> [reduce(operator.add, x) for x in [(1,2), (3,4)]]
[3, 7]
>>> [x for x in [1,2,3] if x > 2]
[3]
```

compose in Python

```
def compose(*funs):
    return lambda x: reduce(lambda z,f: f(z), \
        reversed(funs), x)

def add1(x):
    return x + 1

def mult3(x):
    return x * 3

def sub5(x):
    return x - 5

f = compose(add1,mult3,sub5)
print f(1)      # -11
print f(20)     # 46
```

Exercise
The Last One!

- Implement **union** in Python
  - takes the two sets as input
- Implement **addn** in Python
  - use it to add 5 to an existing integer list, returning a new list

The D Programming Language

- A “Modern C++”
  - higher-level, cleaner syntax
- Supports systems programming
  - and generates native executables
- Garbage collected
- Other features
  - automated unit testing
  - contract programming
  - Python-like module system
  - FP!

Introducing D

A Word Count Program (output on next slide)

```
void wc(string filename) {
    auto words = split(cast(string) read(filename));
    int[string] counts;
    foreach (word; words)
        ++counts[word];
    foreach (w; counts.keys.sort)
        writefln("%s: %d", w, counts[w]);
}

// A simple driver: process all files arguments
void main(string[] args) {
    foreach(f; args[1..$]) {    // Start at second arg ([1])
        writefln("\n%s:", f);
        wc(f);
    }
}
```

wc.txt:
%d",: 1
([1]): 1
(f;: 1
(w;: 1
(word;: 1
++counts[word];: 1
//: 2
=: 1
all: 1
arg: 1
w,: 1
wc(f);: 1
wc(string: 1
words: 1
words): 1
writefln("%s:: 1
writefln("\n%s:",: 1
{: 3
}: 3

FP in D

- Does not have **map**, **foldr**, or **foldl**
  - but it has **foreach** and **foreach_reverse**
- Supports *nested functions* and *closures*
  - Closures in D are called *delegates*
  - Delegates couple a function with either an *enclosing function*, an *object*, or a *class*

compose in D

non-generic

```
alias int function(int) F;
alias int delegate(int) D;

D compose(F[] funs) {
   int doit(int n) {
      int result = n;
      foreach_reverse (f; funs)
         result = f(result);
      return result;
   }
   return &doit;
}
```

Using compose

```
void main() {
   F[] funs;
   funs ~= function int(int x){return x+1;};
   funs ~= function int(int x){return x*3;};
   funs ~= function int(int x){return x-5;};
   auto c = compose(funs);  // type inference
   writeln(c(1));         // -11
   writeln(c(20));        // 46
}
```

A Generic **compose**

```
T delegate(T) compose(T)(T function(T)[] funs)
{
   T doit(T n) {
      T result = n;
      foreach_reverse (f; funs)
         result = f(result);
      return result;
   }
   return &doit;
}
```

Using the Generic **compose**

```
void main() {
   string function(string)[] sfuns;
   sfuns ~= function string(string s) {return s ~ 's';};
   sfuns ~= function string(string s) {return s[1..$];};
   auto c2 = compose(sfuns);
   writeln(c2("stale"));   // "tales"
}
```

FP in C++

- Uses *function objects*
  - objects with a function-call *operator* (**operator()**)
  - the object's data constitutes the "closure"
- **map = transform**
- **foldl = accumulate**
- "Lists" can be arrays, vectors, linked-lists, etc.
  - any STL-conforming "sequence"
- 50+ sequence algorithms in the standard library

Defining a C++ Function Object

```
#include <algorithm>
#include <iostream>
using namespace std;

class addn {
    int n;
public:
    addn(int n) : n(n) {}
    int operator()(int x) {
        return x + n;
    }
};
```

Using addn

```
int main() {
    addn add5(5);
    cout << add5(10) << endl;       // 15

    int a[] = {1,2,3,4,5};
    transform(a, a+5, a, addn(10)); // 11 12 13 14 15
    for (int i = 0; i < 5; ++i)
        cout << a[i] << ' ';
    cout << endl;
}
```

Selected C++ Function Objects

Predicates

```
equal_to
not_equal_to
greater
less
greater_equal
less_equal
logical_and
logical_or
logical_not
```

Arithmetic

```
plus
minus
multiplies
divides
modulus
negate
```

A Simple Filter

```
// Add an input integer to each number in a file

int main(int argc, char* argv[]) {
    int n = 0;

    // Get n from command line
    if (argc > 1)
        n = atoi(argv[1]);

    ifstream inf("nums.dat");
    ofstream outf("nums.out");
    transform(istream_iterator<int>(inf),
              istream_iterator<int>(),
              ostream_iterator<int>(outf," "),
              bind2nd(plus<int>(),n));
}
```

Using accumulate

```
int main() {
    int a[] = {1,2,3,4};
    cout << accumulate(a, a+4, 0) << endl;

    string s[] = {"eat","my","dust"};
    string result = accumulate(s, s+3, string());
    cout << result << endl;

    cout << accumulate(a,a+4,1,multiplies<int>()) << endl;
}

/* Output:
10
eatmydust
24
*/
```

compose in C++

```
typedef int (*Fun)(int);

class Composer {
private:
    const vector<Fun>& funs;
    static int apply(int sofar, Fun f) {
        return f(sofar);
    }
public:
    Composer(const vector<Fun>& fs) : funs(fs) {}
    int operator()(int x) const {
        return accumulate(funs.rbegin(), funs.rend(),
                          x, apply);
    }
};
```

Using compose

```
int add1(int x) {
    return x + 1;
}
int mult3(int x) {
    return x * 3;
}
int sub5(int x) {
    return x - 5;
}
```

```
int main() {
    vector<Fun> funs;
    funs.push_back(add1);
    funs.push_back(mult3);
    funs.push_back(sub5);
    Composer comp(funs);
    cout << comp(1) << endl;    // -11
    cout << comp(20) << endl; // 46
}
```

Scala

- A FP front-end to the JVM
  - statically typed
  - type inference
- Pretty much a copy of ML
  - pattern matching
  - **foldright**, **foldleft**, etc.

compose in Scala

```
object Compose {
 def compose3[T](flist: List[(T) => T]): (T) => T =
  (x: T) => flist.foldRight(x)
            ((f: (T) => T, sofar: T) => f(sofar))

 def main(args: Array[String]) {
   val addOne = (x: Int) => x + 1
   val addTwo = (x: Int) => x + 2
   val addThree = (x: Int) => x + 3
   val addFour = compose(List(addOne,addOne,addOne,addOne))
   println(addFour(1))  // 5
   val addSix = compose(List(addOne, addTwo, addThree))
   println(addSix(1))   // 7
 }
}
```

union in Scala

```
object Union {
 def union[T](a: List[T], b: List[T]): List[T] =
   (a, b) match {
     case (x, Nil) => x
     case (x, head :: rest) => {
       if (x contains head)
         union(x, rest)
       else
         head :: union(x, rest)
     }
   }

 def main(args: Array[String]) {
   println(union(List("a", "b", "c"), List("b", "c", "d")))
 }
}
```

FP in C# 3.0

- As in D, *delegates* act as closures
- Lambdas via *anonymous delegates*
- Type inference with **var**

addn in C# 3.0

```
public static Func<int, int> addn(int n)
{
    return new Func<int, int>(x => x + n);
}

…

var f2 = addn(5);
Console.WriteLine(f2(2));     // 7
```

compose in C# 3.0

```
public static Func<T, T> Compose<T>(IEnumerable<Func<T, T>> funcs)
    {
        return new Func<T, T>(i =>
          {
            T result = i;
            foreach (var func in funcs.Reverse())
            {
              result = func(result);
            }
            return result;
          });
    }
```

Using compose

```
IEnumerable<Func<int, int>> t = new List<Func<int, int>>
{
    new Func<int, int>(x => x + 1),
    new Func<int, int>(x => x * 3),
    new Func<int, int>(x => x - 5)
};

…

var c = Compose(t);
Console.WriteLine(c(1));            // -11
Console.WriteLine(c(20));           // 46
```

The Future of FP

D 3.0

- Will add “pure functions”
  - functions that don’t change state
- Will add a bunch of algorithms
- Will support full FP and STM (a la Haskell)

Java

- Closures proposal
  - somewhat controversial
- Inner Classes are a poor-man's closure
- Algorithms have been around via JGL for over 10 years

C++0x

- More flexible lambda expressions
- More flexible function-argument binding