

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

[http://www.cs.uiuc.edu/class
/sp07/cs421/](http://www.cs.uiuc.edu/class/sp07/cs421/)

Based in part on slides by Mattox Beckman, as updated
by Vikram Adve and Gul Agha

First Class Types

- A type is *first class* if it can be
 - **Passed** as an argument
 - **Assigned** as a value
 - **Returned** as a result
- Examples:
 - C: scalars, pointers, structures
 - C++: same as C, plus classes
 - Scheme, LISP: scalars, lists (s-expressions), functions
 - ML: same as Scheme, plus user defined data types

First Class Types

- The kind of data that can be manipulated well in a language largely determines for which applications the language is well suited
- The ability to treat functions as data is one of the strengths of applicative programming languages

Higher Order Functions

- A function is *higher-order* if it takes a function as an argument or returns one as a result
- Example:

```
# let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

- The type $('a \rightarrow 'b) \rightarrow ('c \rightarrow 'a) \rightarrow 'c \rightarrow 'b$ is a higher order type because of $('a \rightarrow 'b)$ and $('c \rightarrow 'a)$ and $\rightarrow 'c \rightarrow 'b$

Higher Order Functions

- What are the types of the following functions:

```
# plus_two;;
```

```
- : int -> int = <fun>
```

```
# compose plus_two plus_two;;
```

?

```
# compose plus_two;;
```

?

Higher Order Functions

- What are the types of the following functions:

```
# plus_two;;
```

```
- : int -> int = <fun>
```

```
# compose plus_two plus_two;;
```

```
- : int -> int = <fun>
```

```
# compose plus_two;;
```

```
- : ('_a -> int) -> '_a -> int = <fun>
```

Higher Order Functions

- What do the following functions do?

```
# plus_two;;
```

```
- : int -> int = <fun>
```

```
# compose plus_two plus_two;;
```

```
- : int -> int = <fun>
```

```
# compose plus_two;;
```

```
- : ('_a -> int) -> '_a -> int = <fun>
```

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;
```

```
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;
```

```
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;
```

```
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

```
# let thrice f = compose f (compose f f);;
```

```
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- Is this the only way?

Reversing Arguments

```
# let flip f a b = f b a;;  
val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>  
# map ((-) 1) [5;6;7];;  
- : int list = [-4; -5; -6]  
# map (flip (-) 1) [5;6;7];;  
- : int list = [4; 5; 6]  
# let (-) = flip (-);;  
val ( - ) : int -> int -> int = <fun>  
# 2 - 5;;  
- : int = 3
```

Partial Application

```
# (+);;
```

```
- : int -> int -> int = <fun>
```

```
# (+) 2 3;;
```

```
- : int = 5
```

```
# let plus_two = (+) 2;;
```

```
val plus_two : int -> int = <fun>
```

```
# plus_two 7;;
```

```
- : int = 9
```

- Patial application also called *sectioning*

Lambda Lifting

- You must remember the rules for evaluation when you use partial application

```
# let add_two = (+) (print_string "test\n"; 2);;  
test
```

```
val add_two : int -> int = <fun>
```

```
# let add2 = (* lambda lifted *)
```

```
    fun x -> (+) (print_string "test\n"; 2) x;;
```

```
val add2 : int -> int = <fun>
```

Lambda Lifting

```
# thrice add_two 5;;
```

```
- : int = 11
```

```
# thrice add2 5;;
```

```
test
```

```
test
```

```
test
```

```
- : int = 11
```

- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied

Lambda Lifting and “Unknown Types”

- Recall `compose plus_two`:

```
# let f1 = compose plus_two;;
```

```
val f1 : ('_a -> int) -> '_a -> int = <fun>
```

- Compare to lambda lifted version:

```
# let f2 = fun g -> compose plus_two g;;
```

```
val f2 : ('a -> int) -> 'a -> int = <fun>
```

- What is the difference?

Lambda Lifting and “Unknown Types”

- ‘_a can only be instantiated once for an expression

```
# f1 plus_two;;
```

```
- : int -> int = <fun>
```

```
# f1 List.length;;
```

Characters 3-14:

```
f1 List.length;;
```

```
^^^^^^^^^^^^
```

This expression has type 'a list -> int but is here used with type int -> int

Lambda Lifting and “Unknown Types”

- ‘a can be repeatedly instantiated

```
# f2 plus_two;;  
- : int -> int = <fun>  
  
# f2 List.length;;  
- : 'a list -> int = <fun>
```

Curried vs Uncurried

```
val add_three : int -> int -> int -> int = <fun>
```

How does it differ from

```
# let add_triple (u,v,w) = u + v + w;;
```

```
val add_triple : int * int * int -> int = <fun>
```

add_three is *curried*;

add_triple is uncurried

curry and uncurry

```
# let curry f x y = f (x,y);;
```

```
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c =  
<fun>
```

```
# let uncurry f (x,y) = f x y;;
```

```
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c =  
<fun>
```

- Remember these!

curry and uncurry

```
# (+);;
```

```
- : int -> int -> int = <fun>
```

```
# let plus = uncurry (+);;
```

```
val plus : int * int -> int = <fun>
```

```
# plus (3,4);;
```

```
- : int = 7
```

```
# curry plus 3 4;;
```

```
- : int = 7
```

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
```

```
  [] -> 0 | x::xs -> x + sumlist xs;;
```

```
val sumlist : int list -> int = <fun>
```

```
# sumlist [2;3;4];;
```

```
- : int = 9
```

```
# let rec prodlist list = match list with
```

```
  [] -> 1 | x::xs -> x * prodlist xs;;
```

```
val prodlist : int list -> int = <fun>
```

```
# prodlist [2;3;4];;
```

```
- : int = 24
```

Folding

```
# let rec fold_left f a list = match list
  with [] -> a | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
fold_left f a [x1; x2; ...; xn] = f(...(f (f a x1) x2)... )xn
```

```
# let rec fold_right f list b = match list
  with [] -> b | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
fold_right f [x1; x2; ...; xn] b = f x1(f x2 (...(f xn b)...))
```

Folding

```
# let sumlist list = fold_right (+) list 0;;
```

```
val sumlist : int list -> int = <fun>
```

```
# sumlist [2;3;4];;
```

```
- : int = 9
```

```
# let prodlist list = fold_right ( * ) list 1;;
```

```
val prodlist : int list -> int = <fun>
```

```
# prodlist [2;3;4];;
```

```
- : int = 24
```

Folding

- Can replace recursion by `fold_right` in any primitive recursive definition
 - Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by `fold_left` in any tail recursive definition

Encoding Recursion with Fold

```
# let rec append list1 list2 = match list1 with  
  [] -> list2 | x::xs -> x :: append xs list2;;  
val append : 'a list -> 'a list -> 'a list = <fun>
```

Base Case

Operation

Recursive Call

```
# let append list1 list2 =  
  fold_right (fun x y -> x :: y) list1 list2;;  
val append : 'a list -> 'a list -> 'a list = <fun>  
# append [1;2;3] [4;5;6];;  
- : int list = [1; 2; 3; 4; 5; 6]
```

Combining Lists of Functions

```
# let rec complist flist = match flist with
  [ ] -> (fun x -> x)
  | f::fs -> compose f (complist fs);;
val complist : ('a -> 'a) list -> 'a -> 'a = <fun>
```

Why isn't type more general, like compose?

```
# complist [( - ) 1; ( * ) 3; plus_two] ;;
```

```
- : int -> int = <fun>
```

```
# complist [( - ) 1; ( * ) 3; plus_two] 5;;
```

```
- : int = -20
```

Can you write this with fold_right?

Repeating n Times

```
# let rec repeat n f x =  
  match n with 0 -> x | _ -> f (repeat (n - 1) f x);;  
val repeat : int -> ('a -> 'a) -> 'a -> 'a = <fun>  
# repeat 8 (fun x -> x * 2) 1;;  
- : int = 256  
# let rec iter n f x =  
  match n with 0 -> x | _ -> iter (n - 1) f (f x);;  
val iter : int -> ('a -> 'a) -> 'a -> 'a = <fun>  
# iter 8 (fun x -> x * 2) 1;;  
- : int = 256
```

- Which is more efficient?

Mapping

- What do these functions have in common?

```
# let rec inclist list = match list with [ ] -> [ ]
```

```
| x :: xs -> (1 + x) :: inclist xs;;
```

```
val inclist : int list -> int list = <fun>
```

```
# inclist [2;3;4];;
```

```
- : int list = [3; 4; 5]
```

```
# let rec doublelist list = match list with [ ] -> [ ]
```

```
| x :: xs -> (2 * x) :: doublelist xs;;
```

```
val doublelist : int list -> int list = <fun>
```

```
# doublelist [2;3;4];;
```

```
- : int list = [4; 6; 8]
```

Recall Map

```
# let rec map f list =  
  match list  
  with [] -> []  
  | (h::t) -> (f h) :: (map f t);;  
val map : ('a -> 'b) -> 'a list -> 'b list =  
  <fun>  
  
# map plus_two fib5;;  
- : int list = [10; 7; 5; 4; 3; 3]
```

Mapping

```
# let inclist = map ((+) 1);;
val inclist : int list -> int list = <fun>
# inclist [2;3;4];;
- : int list = [3; 4; 5]
# let doublelist = map ((* ) 2);;
val doublelist : int list -> int list = <fun>
# doublelist [2;3;4];;
- : int list = [4; 6; 8]
```

Map from Fold

```
# let map f list =
```

```
  fold_right (fun x y -> f x :: y) list [ ];;
```

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# map ((+)1) [1;2;3];;
```

```
- : int list = [2; 3; 4]
```

- Can you write `fold_righth` (or `fold_left`) with just `map`? How, or why not?

Related Function: Zip

```
# let rec zip list1 list2 =  
  match (list1,list2) with ([ ], _) -> []  
  | (_, [ ]) -> []  
  | (x::xs, y:: ys) -> (x,y)::zip xs ys;;  
val zip : 'a list -> 'b list -> ('a * 'b) list =  
  <fun>  
  
# zip [1;2;3] [4;5;6];;  
- : (int * int) list = [(1, 4); (2, 5); (3, 6)]
```

Zipwith

```
# let rec zipwith f list1 list2 =
match (list1,list2) with ([ ], _) -> []
| (_, [ ]) -> []
| (x::xs, y::ys) -> f x y ::zipwith f xs ys;;
val zipwith : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c
  list = <fun>
# zipwith (+) [1;2;3] [4;5;6];;
- : int list = [5; 7; 9]
# zipwith (fun x y -> (x,y)) [1;2;3] [4;5;6];;
- : (int * int) list = [(1, 4); (2, 5); (3, 6)]
```

Zip from Zipwith

- How do you write zip using zipwith, and no explicit recursion?

Zip from Zipwith

- How do you write zip using zipwith, and no explicit recursion?

```
# let zip = zipwith (fun x y -> (x,y));;  
val zip : '_a list -> '_b list -> ('_a * '_b) list  
= <fun>  
  
# zip [1;2;3] [4;5;6];;  
- : (int * int) list = [(1, 4); (2, 5); (3, 6)]
```

Problems

- Write a function `flipuc` that flips the arguments to an uncurried function, using just `curry`, `flip` and `uncurry`
- Write a function that has type
$$('a \rightarrow 'b) \rightarrow 'a * 'c \rightarrow 'b$$
- Use `fold_right` to write a function that takes a list and returns it.
- Use `fold_right` to write a function to remove all negative elements from a list

Problem 1

- Write a function `flipuc` that flips the arguments to an uncurried function, using just `curry`, `flip` and `uncurry`

```
# let flipuc f = uncurry (flip (curry f));;
val flipuc : ('a * 'b -> 'c) -> 'b * 'a -> 'c = <fun>
# let cons (x,xs) = x::xs;;
val cons : 'a * 'a list -> 'a list = <fun>
# let snoc = flipuc cons;;
val snoc : 'a list * 'a -> 'a list = <fun>
# snoc(snoc ([1],2),3);;
- : int list = [3; 2; 1]
```

Problem 2

- Write a function that has type

$$('a \rightarrow 'b) \rightarrow 'a * 'c \rightarrow 'b$$

```
# let app_fst f (a,b) = f a;;
```

```
val app_fst : ('a -> 'b) -> 'a * 'c -> 'b = <fun>
```

```
# app_fst ((+) 1) (3, 7);;
```

```
- : int = 4
```

```
# app_fst ((+) 1) (4, "hi");;
```

```
- : int = 5
```

Problem 3

- Use `fold_right` to write a function that takes a list and returns it.

```
# let listId list =  
  fold_right (fun x xs -> x::xs) list [];;  
val listId : 'a list -> 'a list = <fun>  
# listId [1;2;3];;  
- : int list = [1; 2; 3]
```

Problem 4

- Use `fold_right` to write a function to remove all negative elements from a list

```
# let gezero list =  
  fold_right  
  (fun x xs -> if x >= 0 then x::xs else xs)  
  list [ ];;  
  
val gezero : int list -> int list = <fun>  
  
# gezero [1;0;3;-5;7;-2];;  
- : int list = [1; 0; 3; 7]
```

- Related to filter function (in next homework)