## Programming Languages and Compilers (CS 421)



Elsa L Gunter 2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

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

9/23/15



#### Mapping Functions Over Lists

```
# 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]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

## 4

#### Mapping Recursion

 One common form of structural recursion applies a function to each element in the structure

```
# 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]
```

9/23/15



#### Mapping Recursion

 Can use the higher-order recursive map function instead of direct recursion

```
# let doubleList list =
   List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

Same function, but no explicit rec



#### Your turn now

Write a function

make\_app: (('a -> 'b) \* 'a) list -> 'b list that takes a list of function – input pairs and gives the result of applying each function to its argument. Use map, no explicit recursion.

let make\_app I =

9/23/15 5



9/23/15

#### Folding Recursion

 Another common form "folds" an operation over the elements of the structure

```
# let rec multList list = match list
with [] -> 1
| x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48

Computes (2 * (4 * (6 * 1)))
```

```
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 multList list = match list with

[] -> 1 | x::xs -> x * multList xs;;

val multList : int list -> int = <fun>

# multList [2;3;4];;

-: int = 24
```

```
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 multList list = match list with

[] -> [1] x::xs -> x * multList xs;;

val multList : int list -> int = <fun>

# multList [2;3;4];;

-: int = 24
```

```
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];;

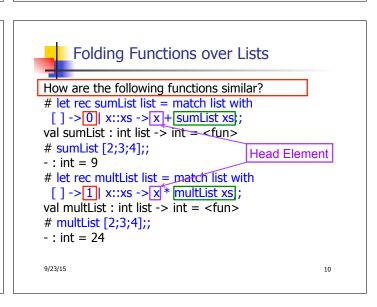
-: int = 9

# let rec multList list = match list with

[] -> [1] x::xs -> x * multList xs;

val multList : int list -> int = <fun>
# multList [2;3;4];;

-: int = 24
```



```
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 multList list = match list with

[] -> 1] x::xs -> x * multList xs;

val multList: int list -> int = <fun>

# multList [2;3;4];;

-: int = 24
```

```
Folding Functions over Lists

How are the following functions similar?

# let rec sumList list = match list with

[]->[] x::xs -> | x + | Rec value | ;

val sumList : int list -> int = <fun>

# sumList [2;3;4];;

-: int = 9

# let rec multList list = match list with

[]->[] | x::xs -> | x * | Rec value | ;

val multList : int list -> int = <fun>

# multList [2;3;4];;

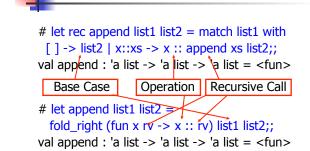
-: int = 24R
```

```
Folding Recursion

multList folds to the right
Same as:

# let multList list =
List.fold_right
(fun x -> fun rv -> x * rv)
list 1;;

val multList : int list -> int = <fun>
# multList [2;4;6];;
-: int = 48
```



**Encoding Recursion with Fold** 

# append [1;2;3] [4;5;6];; -: int list = [1; 2; 3; 4; 5; 6]

9/23/15 15



## Your turn now

## Try Problem 1 on ML2

9/23/15 16



let rec length I =
 match I with [] -> 0
 | (a :: bs) -> 1 + length bs

How do you write length with fold\_right, but no explicit recursion?

9/23/15 17



let rec length I =
 match I with [] -> 0
 | (a :: bs) -> 1 + length bs

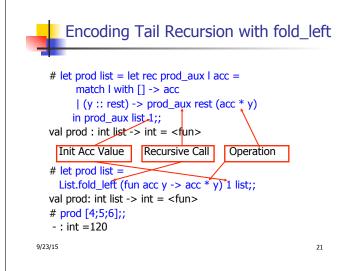
How do you write length with fold\_right, but no explicit recursion?

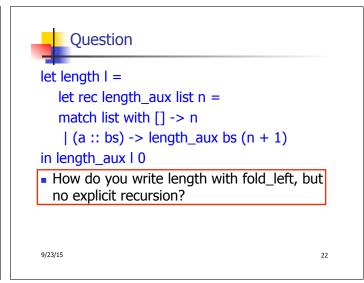
let length list =
List.fold\_right (fun x -> fun n -> n + 1) list 0

9/23/15 18

```
# 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
    (fun () -> print_string)
    ()
    ["hi"; "there"];;
hithere-: unit = ()
```





```
# 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 [x<sub>1</sub>; x<sub>2</sub>;...;x<sub>n</sub>] = f(...(f (f a x<sub>1</sub>) x<sub>2</sub>)...)x<sub>n</sub>

# 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 [x<sub>1</sub>; x<sub>2</sub>;...;x<sub>n</sub>] b = f x<sub>1</sub>(f x<sub>2</sub> (...(f x<sub>n</sub> b)...))
```

```
-
```

#### Recall

What is its running time?

9/23/15

25

27



- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

9/23/15 26



#### Tail Recursion - Example

```
# let rec rev_aux list revlist =
match list with [] -> revlist
| x :: xs -> rev_aux xs (x::revlist);;
val rev_aux : 'a list -> 'a list -> 'a list = <fun>
# let rev list = rev_aux list [];;
val rev : 'a list -> 'a list = <fun>
■ What is its running time?
```

9/23/15



#### Comparison

- poor\_rev [1,2,3] =
- (poor\_rev [2,3]) @ [1] =
- ((poor\_rev [3]) @ [2]) @ [1] =
- (((poor\_rev [ ]) @ [3]) @ [2]) @ [1] =
- (([] @ [3]) @ [2]) @ [1]) =
- ([3] @ [2]) @ [1] =
- (3:: ([ ] @ [2])) @ [1] =
- [3,2] @ [1] =
- **3** :: ([2] @ [1]) =
- **3** :: (2:: ([ ] @ [1])) = [3, 2, 1]

9/23/15 28



#### Comparison

- rev [1,2,3] =
- rev\_aux [1,2,3] [ ] =
- rev\_aux [2,3] [1] =
- rev\_aux [3] [2,1] =
- rev\_aux [ ] [3,2,1] = [3,2,1]

4

#### Folding - Tail Recursion

```
# let rev list =
fold_left
(fun I -> fun x -> x :: I) //comb op
[] //accumulator cell
list
```

9/23/15 29

9/23/15



#### Folding

- Can replace recursion by fold\_right in any forward 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 primitive recursive definition

9/23/15 31



#### **Continuations**

- A programming technique for all forms of "non-local" control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially it's a higher-order function version of GOTO

9/23/15 32



#### Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an extra argument to which to pass its result; outer procedure "returns" no result
- Function receiving the result called a continuation
- Continuation acts as "accumulator" for work still to be done

9/23/15 33



#### Continuation Passing Style

 Writing procedures such that all procedure calls take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)

9/23/15 34



#### Continuation Passing Style

- A compilation technique to implement nonlocal control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics
- Possible intermediate state in compiling functional code

9/23/15 35

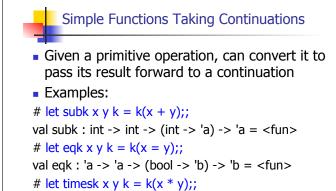


#### Why CPS?

- Makes order of evaluation explicitly clear
- Allocates variables (to become registers) for each step of computation
- Essentially converts functional programs into imperative ones
  - Major step for compiling to assembly or byte code
- Tail recursion easily identified
- Strict forward recursion converted to tail recursion
  - At the expense of building large closures in heap

9/23/15 36

```
Example
Simple reporting continuation:
# let report x = (print_int x; print_newline());; val report : int -> unit = <fun>
Simple function using a continuation:
# let addk a b k = k (a + b);; val addk : int -> int -> (int -> 'a) -> 'a = <fun>
# addk 22 20 report;; 2
- : unit = ()
```



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

4

## Your turn now

# Try Problem 5 on MP4 Try modk

9/23/15



9/23/15

#### Nesting Continuations

```
# let add_three x y z = (x + y) + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three x y z= let p = x + y in p + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three_k x y z k =
    addk x y (fun p -> addk p z k);;
val add_three_k : int -> int -> (int -> 'a)
    -> 'a = <fun>
```

9/23/15 40



#### add\_three: a different order

- # let add\_three x y z = x + (y + z);;
- How do we write add\_three\_k to use a different order?
- let add three k x y z k =



### Your turn now

## Try Problem 6 on MP4

9/23/15 41

9/23/15

42

```
Recursive Functions

Recall:

# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);;
    val factorial : int -> int = <fun>
# factorial 5;;
    - : int = 120
```

```
# let rec factorial n =
let b = (n = 0) in (* First computation *)
if b then 1 (* Returned value *)
else let s = n - 1 in (* Second computation *)
let r = factorial s in (* Third computation *)
n * r in (* Returned value *);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Recursive Functions

# let rec factorialk n k =
eqk n 0
(fun b -> (\* First computation \*)
if b then k 1 (\* Passed value \*)
else subk n 1 (\* Second computation \*)
(fun s -> factorialk s (\* Third computation \*)
(fun r -> timesk n r k))) (\* Passed value \*)
val factorialk : int -> int = <fun>
# factorialk 5 report;;
120
-: unit = ()



- To make recursive call, must build intermediate continuation to
  - take recursive value: r
  - build it to final result: n \* r
  - And pass it to final continuation:
  - times n r k = k (n \* r)

9/23/15 46

Example: CPS for length

let rec length list = match list with [] -> 0 | (a :: bs) -> 1 + length bs

What is the let-expanded version of this?

Example: CPS for length

let rec length list = match list with [] -> 0
| (a :: bs) -> 1 + length bs

What is the let-expanded version of this?

let rec length list = match list with [] -> 0
| (a :: bs) -> let r1 = length bs in 1 + r1

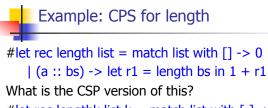
9/23/15 47 9/23/15 48



#### Example: CPS for length

9/23/15

723/13



#let rec lengthk list k = match list with [ ] -> k 0
| x :: xs -> lengthk xs (fun r -> addk r 1 k);;

val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
# lengthk [2;4;6;8] report;;

4

-: unit = ()

9/23/15



## Your turn now

## Try Problem 8 on MP4

9/23/15



#### **CPS for Higher Order Functions**

- In CPS, every procedure / function takes a continuation to receive its result
- Procedures passed as arguments take continuations
- Procedures returned as results take continuations
- CPS version of higher-order functions must expect input procedures to take continuations

9/23/15 52



#### Example: all

```
#let rec all p I = match I with [] -> true
| (x :: xs) -> let b = p x in
if b then all p xs else false
```

val all : ('a -> bool) -> 'a list -> bool = <fun>

• What is the CPS version of this?

4

51

53

#### Example: all

What is the CPS version of this?

#let rec allk pk l k =

9/23/15

54

50

#### Example: all

55



9/23/15 56

9/23/15

#### Example: all

9/23/15

57

59

## Example: all

9/23/15 58

## 4

#### Example: all

9/23/15

Example: all

60



#### **Terms**

- A function is in <u>Direct Style</u> when it returns its result back to the caller.
- A Tail Call occurs when a function returns the result of another function call without any more computations (eq tail recursion)
- A function is in Continuation Passing Style when it, and every function call in it, passes its result to another function.
- Instead of returning the result to the caller, we pass it forward to another function.

9/23/15 61



#### Terminology

- Tail Position: A subexpression s of expressions e, such that if evaluated, will be taken as the value of e
  - if (x>3) then x + 2 else x 4
  - let x = 5 in x + 4
- Tail Call: A function call that occurs in tail position
  - if (h x) then f x else  $(x \pm g x)$

9/23/15 62



#### Terminology

- Available: A function call that can be executed by the current expression
- The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).
  - if (h x) then f x else (x + g x)
  - if (h x) then (fun x -> f x) else (g (x + x))



9/23/15



#### **CPS Transformation**

- Step 1: Add continuation argument to any function definition:
  - let f arg =  $e \Rightarrow$  let f arg k = e
  - Idea: Every function takes an extra parameter saying where the result goes
- Step 2: A simple expression in tail position should be passed to a continuation instead of returned:
  - return  $a \Rightarrow k a$
  - Assuming a is a constant or variable.
  - "Simple" = "No available function calls."

9/23/15 64



#### **CPS Transformation**

- Step 3: Pass the current continuation to every function call in tail position
  - return f arg  $\Rightarrow$  f arg k
  - The function "isn't going to return," so we need to tell it where to put the result.



63

65

#### **CPS Transformation**

- Step 4: Each function call not in tail position needs to be converted to take a new continuation (containing the old continuation as appropriate)
  - return op (f arg)  $\Rightarrow$  f arg (fun r -> k(op r))
  - op represents a primitive operation
  - return  $f(g arg) \Rightarrow g arg (fun r-> f r k)$

9/23/15

9/23/15

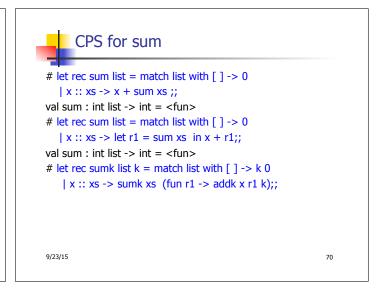
```
Example
```

```
After:
    Before:
                                                                                                                                                                                                                                                                                                            let rec add_listk lst k =
let rec add list lst =
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           (* rule 1 *)
match lst with
                                                                                                                                                                                                                                                                                                              match Ist with
                  []->0
                                                                                                                                                                                                                                                                                                            |[]-> k 0 (* rule 2 *)
\mid 0 :: xs \rightarrow add_{ist} xs \mid 0 :: xs \rightarrow add_{is
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               (* rule 3 *)
    | x :: xs -> (+) x
                                                                                                                                                                                                                                                                                                              | x :: xs -> add_listk xs
                             (add_list xs);;
                                                                                                                                                                                                                                                                                                                                                                                              (fun r -> k ((+) x r));;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                  (* rule 4 *)
9/23/15
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          67
```

```
# let rec sum list = match list with [] -> 0
| x :: xs -> x + sum xs ;;
val sum : int list -> int = <fun>

9/23/15
```

# let rec sum list = match list with [] -> 0
| x :: xs -> x + sum xs ;;
val sum : int list -> int = <fun>
# let rec sum list = match list with [] -> 0
| x :: xs -> let r1 = sum xs in x + r1;;



# let rec sum list = match list with [] -> 0
| x :: xs -> x + sum xs ;;

val sum : int list -> int = <fun>
# let rec sum list = match list with [] -> 0
| x :: xs -> let r1 = sum xs in x + r1;;

val sum : int list -> int = <fun>
# let rec sumk list k = match list with [] -> k 0
| x :: xs -> sumk xs (fun r1 -> addk x r1 k);;

val sumk : int list -> (int -> 'a) -> 'a = <fun>
# sumk [2;4;6;8] report;;

20
- : unit = ()

9/23/15



69

#### Other Uses for Continuations

- CPS designed to preserve order of evaluation
- Continuations used to express order of evaluation
- Can be used to change order of evaluation
- Implements:
  - Exceptions and exception handling
  - Co-routines
  - (pseudo, aka green) threads

9/23/15 72

```
# exception Zero;;
exception Zero;;
exception Zero
# let rec list_mult_aux list =
    match list with [] -> 1
    | x :: xs ->
    if x = 0 then raise Zero
        else x * list_mult_aux xs;;
val list_mult_aux : int list -> int = <fun>
```

```
# let list_mult list =
try list_mult_aux list with Zero -> 0;;
val list_mult : int list -> int = <fun>
# list_mult [3;4;2];;
-: int = 24
# list_mult [7;4;0];;
-: int = 0
# list_mult_aux [7;4;0];;
Exception: Zero.
```



#### **Exceptions**

- When an exception is raised
  - The current computation is aborted
  - Control is "thrown" back up the call stack until a matching handler is found
  - All the intermediate calls waiting for a return values are thrown away

9/23/15 75

## Implementing Exceptions

```
# let multkp m n k =
let r = m * n in
  (print_string "product result: ";
  print_int r; print_string "\n";
  k r);;
val multkp : int -> int -> (int -> 'a) -> 'a
  = <fun>
```

9/23/15 76



9/23/15

#### Implementing Exceptions

77

#### Implementing Exceptions

```
# list_multk [3;4;2] report;;
product result: 2
product result: 8
product result: 24
24
- : unit = ()
# list_multk [7;4;0] report;;
0
- : unit = ()
```