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
Your turn: num_neg – tail recursive

# let num_neg list =
Your turn: `num_neg` – tail recursive

```ocaml
# let num_neg list =
  let rec num_neg_aux list curr_neg =
    match list with [ ] -> curr_neg
    | (x::xs) ->
      if x < 0 then
        (num_neg_aux xs _)
      in num_neg_aux
```

9/19/23
Your turn: num_neg – tail recursive

```ocaml
# let num_neg list =
  let rec num_neg_aux list curr_neg =
    match list with [] ->
    | (x :: xs) ->
      in num_neg_aux ? ?
```

9/19/23
Your turn: num_neg – tail recursive

```ocaml
# let num_neg list = 
let rec num_neg_aux list curr_neg =
  match list with [] -> curr_neg
  | (x :: xs) ->

  in num_neg_aux ? ?
```
Your turn: num_neg – tail recursive

# let num_neg list =
let rec num_neg_aux list curr_neg =
  match list with [] -> curr_neg
  | (x :: xs) ->
    num_neg_aux xs ?
  in num_neg_aux ? ?
Your turn: num_neg – tail recursive

# let num_neg list =
let rec num_neg_aux list curr_neg =
  match list with
  | [] -> curr_neg
  | (x :: xs) ->
    num_neg_aux xs
    (if x < 0 then 1 + curr_neg
     else curr_neg)
in num_neg_aux ? ?
Your turn: `num_neg` – tail recursive

```ocaml
# let num_neg list =
let rec num_neg_aux list curr_neg =
  match list with [] -> curr_neg
  | (x :: xs) ->
    num_neg_aux xs
    (if x < 0 then 1 + curr_neg
     else curr_neg)
in num_neg_aux list
```

9/19/23
Your turn: num_neg – tail recursive

```ocaml
# let num_neg list =
let rec num_neg_aux list curr_neg =
  match list with [] -> curr_neg
  | (x :: xs) ->
    num_neg_aux xs
    (if x < 0 then 1 + curr_neg else curr_neg)
in num_neg_aux list 0
```
How can we write length with tail recursion?

```plaintext
let length list =
  let rec length_aux list acc_length =
    match list with
      | [] -> acc_length
      | (x::xs) -> length_aux xs (1 + acc_length)
    in length_aux list 0
```

- **Initial acc value**: 0
- **Accumulated value**: `acc_length`
- **Combining operation**: `1 + acc_length`
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 [x_1; x_2;...;x_n] = f(...(f (f a x_1) x_2)...x_n)

# 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_1; x_2;...;x_n] b = f x_1(f x_2 (...(f x_n b)...))
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
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
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
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)
Continuation Passing Style

- A compilation technique to implement non-local control flow, especially useful in interpreters.

- A formalization of non-local control flow in denotational semantics

- Possible intermediate state in compiling functional code
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
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
Example

- Simple reporting continuation:
  ```ocaml
  # let report x = (print_int x; print_newline());
  val report : int -> unit = <fun>
  ```

- Simple function using a continuation:
  ```ocaml
  # let addk (a, b) k = k (a + b);
  val addk : int * int -> (int -> 'a) -> 'a = <fun>
  # addk (22, 20) report;
  2
  - : unit = ()
  ```
Simple Functions Taking Continuations

- Given a primitive operation, can convert it to pass its result forward to a continuation

- Examples:
  
  # let subk (x, y) k = k(x - y);;
  val subk : int * int -> (int -> 'a) -> 'a = <fun>
  # let eqk (x, y) k = k(x = y);;
  val eqk : 'a * 'a -> (bool -> 'b) -> 'b = <fun>
  # let timesk (x, y) k = k(x * y);;
  val timesk : int * int -> (int -> 'a) -> 'a = <fun>
Nesting Continuations

# let add_triple (x, y, z) = (x + y) + z;;
val add_triple : int * int * int -> int = <fun>

# let add_triple (x,y,z)=let p = x + y in p + z;;
val add_triple : int * int * int -> int = <fun>

# let add_triple_k (x, y, z) k =
   addk (x, y) (fun p -> addk (p, z) [k]);;
val add_triple_k: int * int * int -> (int -> 'a) -> 'a = <fun>
add_three: a different order

- # let add_triple (x, y, z) = x + (y + z);
- How do we write add_triple_k to use a different order?

- let add_triple_k (x, y, z) k =
add_three: a different order

- # let add_triple (x, y, z) = x + (y + z);;
- How do we write add_triple_k to use a different order?

- let add_triple_k (x, y, z) k =
  addk (y,z) (fun r -> addk(x,r) [k])
Recall:

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

- A function is in Direct Style when it returns its result back to the caller.
- 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 giving the computation after the call.
Recursive Functions

# 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 (* 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 -> 'a) -> 'a = <fun>

# factorialk 5 report;;
120
- : unit = ()
Recursive Functions

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

```ocaml
def length list = match list with
    | [] -> 0
    | (a :: bs) -> 1 + length bs
```

What is the let-expanded version of this?

```ocaml
def length list = match list with
    | [] -> 0
    | (a :: bs) -> let r1 = length bs in 1 + r1
```
Example: CPS for length

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

What is the CSP version of this?
Example: CPS for length

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

What is the CSP version of this?

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

val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>

# lengthk [2;4;6;8] report;;

4
- : unit = ()
```
CPS for sum

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

val sum : int list -> int = <fun>
```
CPS for sum

# 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;;
CPS for sum

# 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) ;;
CPS for sum

# 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 = ()
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
Example: all

#let rec all (p, l) = match l 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?
Example: all

```ocaml
#let rec all (p, l) = match l 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?

```ocaml
#let rec allk (pk, l) k =
```
Example: all

```ocaml
default

let rec all (p, l) = match l 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?

```ocaml
default

let rec allk (pk, l) k = match l with
    [] -> true
```

Example: all

```ocaml
#let rec all (p, l) = match l 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?

```ocaml
#let rec allk (pk, l) k = match l with [] -> k true
```
Example: all

```ocaml
#let rec all (p, l) = match l 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?

```ocaml
#let rec allk (pk, l) k = match l with [] -> k true
  | (x :: xs) ->
```

Example: all

```ocaml
#let rec all (p, l) = match l 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?

```ocaml
#let rec allk (pk, l) k = match l with [] -> k true
  | (x :: xs) -> pk x
```
Example: all

```ocaml
#let rec all (p, l) = match l 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?
#let rec allk (pk, l) k = match l with [] -> k true
  | (x :: xs) -> pk x
      (fun b -> if b then
          else
        )
```
Example: all

```ocaml
#let rec all (p, l) = match l 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?

```ocaml
#let rec allk (pk, l) k = match l with [] -> k true
  | (x :: xs) -> pk x
    (fun b -> if b then allk (pk, xs) k else k false)

val allk : ('a -> (bool -> 'b) -> 'b) * 'a list -> (bool -> 'b) -> 'b = <fun>
```