Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Objectives for Today

- Today, we will continue where we left off Tuesday with **continuation-passing style (CPS),** which is **super useful** for compilers and interpreters
- We will learn how to write more interesting functions in CPS, like how to **nest continuations**
- We will then see how we can **transform** functions written in OCaml into CPS
- CPS transformation is **useful** and **important**!

Questions from last time?

Continuation-Passing Style (CPS)

Continuation Passing Style

- **Continuation Passing Style (CPS):** Writing functions such that all functions calls take a **continuation** to which to **pass** the result, and return no result
- CPS is useful as:
  - A **compilation technique** to implement non-local control flow (especially useful in interpreters)
  - A **formalization** of non-local control flow in denotational semantics
  - A **possible intermediate state** in compiling functional code

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;;
42
- : unit = ()
```

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;;
42
- : unit = ()
```

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;;
42
- : unit = ()
```

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;;
42
- : unit = ()
```

Simple Functions Taking Continuations

- Given a **primitive operation**, can convert it to pass its result forward to a continuation
- **More 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>
```

Simple Functions Taking Continuations

- Given a **primitive operation**, can convert it to pass its result forward to a continuation
- **More 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>
```

Simple Functions Taking Continuations

- Given a **primitive operation**, can convert it to pass its result forward to a continuation
- **More 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>
```

Simple Functions Taking Continuations

- Given a **primitive operation**, can convert it to pass its result forward to a continuation
- **More 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>
# subk (22, 20) report;;
```

What happens?

Nesting Continuations

Nesting Continuations

```
(* Asked last class: can we compose these? Yes *)
# let add_triple (x, y, z) =
    (x + y) + z;;
val add_triple
  : int * int * int -> int = <fun>
```

Nesting Continuations

```
(* Asked last class: can we compose these? Yes *)
# let add_triple (x, y, z) =
    (x + y) + z;;
val add_triple
  : int * int * int -> int = <fun>
```

Nesting Continuations

```
(* Asked last class: can we compose these? Yes *)
# let add_triple (x, y, z) =
    let p = (x + y) in p + z;;
val add_triple
  : int * int * int -> int = <fun>
```

Nesting Continuations

```
(* Asked last class: can we compose these? Yes *)
# let add_triple (x, y, z) =
    let p = x + y in p + z;;
val add_triple
  : int * int * int -> int = <fun>
```

Nesting Continuations

```
(* Asked last class: can we compose these? Yes *)
# let add_triple (x, y, z) =
    let p = x + y in p + z;;
val add_triple
  : int * int * int -> int = <fun>
```

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> 'a) -> 'a = <fun>
(* Asked last class: can we compose these? Yes *)
# let add_triple (x, y, z) =
    let p = x + y in p + z;;
val add_triple
  : int * int * int -> int = <fun>
```

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> 'a) -> 'a = <fun>
(* Asked last class: can we compose these? Yes *)
# let add_triple_k (x, y, z) k =
    let p = x + y in p + z;; (* WIP *)
```

Nesting

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> 'a) -> 'a = <fun>
(* Asked last class: can we compose these? Yes *)
# let add_triple_k (x, y, z) k =
     let p = x + y in p + z;; (* WIP *)
```

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> 'a) -> 'a = <fun>
(* Asked last class: can we compose these? Yes *)
# let add_triple_k (x, y, z) k =
    let p = x + y in p + z;; (* WIP *)
```

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> 'a) -> 'a = <fun>
(* Asked last class: can we compose these? Yes *)
# let add_triple_k (x, y, z) k =
    addk (x, y) (fun p -> addk (p, z) k);;
```

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> ’a) -> ’a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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>
```

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> ’a) -> ’a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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>
```

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> ’a) -> ’a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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_triple_k (1, 2, 3) report;;
```

What happens?

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> ’a) -> ’a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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_triple_k (1, 2, 3) report;;
```

What happens?

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> ’a) -> ’a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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_triple_k (1, 2, 3) report;;
```

What happens?

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> 'a) -> 'a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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_triple_k (1, 2, 3) report;;
```

What happens?

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> ’a) -> ’a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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_triple_k (1, 2, 3) report;;
```

What happens?

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> ’a) -> ’a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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_triple_k (1, 2, 3) report;;
```

What happens?

Nesting Continuations

```
# let addk (a, b) k = k (a + b);;
val addk
  : int * int -> (int -> ’a) -> ’a = <fun>
(* Asked last class: can we compose these? Yes *)
# 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_triple_k (1, 2, 3) report;;
6
- : unit = ()
```

Questions so far?

add_triple: A Different Order

How do we write add_triple_k to use a **different** order?

```
# let add_triple (x, y, z) =
    x + (y + z);;
```

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a **different** order?

```
# let add_triple (x, y, z) =
    x + (y + z);;
```

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;
```

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;
```

```
# let add_triple_k (x, y, z) k =
    ???;;
```

Your turn!

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;
```

```
# let add_triple_k (x, y, z) k =
    ???;;
```

Lift first computation to CPS

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;

# let add_triple_k (x, y, z) k =
    addk (y, z) ???;;
```

Lift first computation to CPS

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;
```

```
# let add_triple_k (x, y, z) k =
    addk (y, z) ???;;
```

What is the continuation?

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;

# let add_triple_k (x, y, z) k =
    addk (y, z) (fun r -> ???);;
```

What is the continuation?

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;
```

```
# let add_triple_k (x, y, z) k =
    addk (y, z) (fun r -> ???);;
```

Lift second computation to CPS

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;

# let add_triple_k (x, y, z) k =
    addk (y, z) (fun r -> addk (x, r) ??);;
```

Lift second computation to CPS

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;
```

```
# let add_triple_k (x, y, z) k =
    addk (y, z) (fun r -> addk (x, r) ??);;
```

What happens after the final addk?

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;

# let add_triple_k (x, y, z) k =
    addk (y, z) (fun r -> addk (x, r) k);;
```

Done!

Nesting

add_triple: A Different Order

How do we write add_triple_k to use a different order?

```
# let add_triple (x, y, z) =
    let r = y + z in x + r;;

# let add_triple_k (x, y, z) k =
    addk (y, z) (fun r -> addk (x, r) k);;

# add_triple_k (1, 2, 3) report;;
```

Nesting

add_triple: Both Orders

How do we write add_triple_k to use a different order?

```
(* (x + y) + z *)
let add_triple_k (x, y, z) k =
  addk (x, y) (fun p -> addk (p, z) k)
(* x + (y + z) *)
let add_triple_k (x, y, z) k =
  addk (y, z) (fun r -> addk (x, r) k)
```

Nesting

add_triple: Both Orders

How do we write add_triple_k to use a different order?

```
(* (x + y) + z *)
let add_triple_k (x, y, z) k =
  addk (x, y) (fun p -> addk (p, z) k)
(* x + (y + z) *)
let add_triple_k (x, y, z) k =
  addk (y, z) (fun r -> addk (x, r) k)
```

Nesting

add_triple: Both Orders

How do we write add_triple_k to use a different order?

```
(* (x + y) + z *)
let add_triple_k (x, y, z) k =
  addk (x, y) (fun p -> addk (p, z) k)
(* x + (y + z) *)
let add_triple_k (x, y, z) k =
  addk (y, z) (fun r -> addk (x, r) k)
```

Nesting

add_triple: Both Orders

How do we write add_triple_k to use a different order?

```
(* (x + y) + z *)
let add_triple_k (x, y, z) k =
  addk (x, y) (fun p -> addk (p, z) k)
(* x + (y + z) *)
let add_triple_k (x, y, z) k =
  addk (y, z) (fun r -> addk (x, r) k)
```

Nesting

add_triple: Both Orders

How do we write add_triple_k to use a different order?

```
(* (x + y) + z *)
let add_triple_k (x, y, z) k =
  addk (x, y) (fun p -> addk (p, z) k)
(* x + (y + z) *)
let add_triple_k (x, y, z) k =
  addk (y, z) (fun r -> addk (x, r) k)
```

Nesting

Questions so far?

CPS and Recursion

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
```

Terminology

- 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

- **Direct Style**:

```
# let rec factorial n =
    if n = 0 then
      1
    else
      n * factorial (n - 1);;
```

To simplify transformation to CPS, make order of execution explicit first.

Recursive Functions

- **(Refactoring) Direct Style**:

```
# let rec factorial n =
    if n = 0 then
      1
    else
      n * factorial (n - 1);;
```

Recursive Functions

- **(Refactoring) Direct Style**:

```
# let rec factorial n =
    let b = (n = 0) in (* first computation *)
    if b then
      1
    else
      n * factorial (n - 1);;
```

Recursive Functions

- **(Refactoring) Direct Style**:

```
# let rec factorial n =
    let b = (n = 0) in (* first computation *)
    if b then
      1
    else
      n * factorial (n - 1);;
```

Recursive Functions

- **(Refactoring) Direct Style**:

```
# let rec factorial n =
    let b = (n = 0) in (* first computation *)
    if b then
      1
    else
      let r = factorial (n - 1) in
      n * r;;
```

Recursive Functions

- **(Refactoring) Direct Style**:

```
# let rec factorial n =
    let b = (n = 0) in (* first computation *)
    if b then
      1
    else
      let r = factorial (n - 1) in
      n * r;;
```

Recursive Functions

- **(Refactoring) Direct Style**:

```
# let rec factorial n =
    let b = (n = 0) in (* first computation *)
    if b then
      1
    else
      let s = n - 1 in (* second computation *)
      let r = factorial s in  (* third computation *)
      n * r;;
```

Recursive Functions

- **(Refactored) Direct Style**:

```
# 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 *)
```

Recursive Functions

- **(Refactored) Direct Style**:

```
# 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 *)
```

Rather than **return** these values, we will **pass** them forward.

Recursive Functions

- **Continuation Passing Style**:

```
# let rec factorialk n k =
    eqk (n, 0) (fun b -> (* first computation *)
      if b then
        k 1 (* passed value *)
      else
        subk (n, 1) (fun s -> (* second computation *)
          factorialk s (fun r -> (* third computation *)
            timesk (n, r) k));; (* passed value *)
```

Rather than **return** these values, we will **pass** them forward.

Recursive Functions

- **Continuation Passing Style**:

```
# let rec factorialk n k =
    eqk (n, 0) (fun b -> (* first computation *)
     if b then
       k 1 (* passed value *)
     else
       subk (n, 1) (fun s -> (* second computation *)
        factorialk s (fun r -> (* third computation *)
          timesk (n, r) k));; (* passed value *)
```

Rather than **return** these values, we will **pass** them forward.

Recursive Functions

- **(Refactored) Direct Style**:

```
# 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 *)
```

These stay in the **same order**, but are **transformed** to CPS.

Recursive Functions

- **Continuation Passing Style**:

```
# let rec factorialk n k =
    eqk (n, 0) (fun b -> (* first computation *)
     if b then
       k 1 (* passed value *)
     else
       subk (n, 1) (fun s -> (* second computation *)
         factorialk s (fun r -> (* third computation *)
           timesk (n, r) k)));; (* passed value *)
```

These stay in the **same order**, but are **transformed** to CPS.

Recursive Functions

- **Continuation Passing Style**:

```
# let rec factorialk n k =
    eqk (n, 0) (fun b -> (* first computation *)
     if b then
       k 1 (* passed value *)
     else
       subk (n, 1) (fun s -> (* second computation *)
         factorialk s (fun r -> (* third computation *)
           timesk (n, r) k)));; (* passed value *)
```

Recursive Functions

- **Continuation Passing Style**:

```
# let rec factorialk n k =
    eqk (n, 0) (fun b -> (* first computation *)
     if b then
       k 1 (* passed value *)
     else
       subk (n, 1) (fun s -> (* second computation *)
       factorialk s (fun r -> (* third computation *)
          timesk (n, r) k)));; (* passed value *)
```

How to transform **recursive call**?

Recursive Functions

- **(Refactored) Direct Style**:

```
# 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 *)
```

How to transform **recursive call**?

Recursive Functions

- To transform a **recursive call** to CPS, must build intermediate continuation to:
  - take recursive value,
  - build it to final result, and
  - pass it to final continuation.

```
let r = factorial s in
n * r
```

Recursive Functions

- To transform a **recursive call** to CPS, must build intermediate continuation to:
  - **take recursive value**,
  - build it to final result, and
  - pass it to final continuation.

```
let r = factorial s in
n * r
```

Recursive Functions

- To transform a **recursive call** to CPS, must build intermediate continuation to:
  - take recursive value,
  - **build it to final result**, and
  - pass it to final continuation.

let **r** = factorial s in
n * r

Recursive Functions

- To transform a **recursive call** to CPS, must build intermediate continuation to:
  - take recursive value,
  - build it to final result, and
  - **pass it to final continuation.**

```
factorialk s (fun r ->
  timesk (n, r) k)
```

Questions so far?

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) -> let r = length bs in 1 + r
```

What is the let-expanded version of this?

Example: CPS for length

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

What is the CPS version of this?

Example: CPS for length

```
let rec lengthk list k = (* WIP *)
  match list with
  | [] -> 0
  | (a :: bs) -> let r = length bs in 1 + r
```

What is the CPS version of this?

Example: CPS for length

```
let rec lengthk list k = (* WIP *)
  match list with
  | [] -> k 0
  | (a :: bs) -> let r = length bs in 1 + r
```

What is the CPS version of this?

Example: CPS for length

```
let rec lengthk list k = (* WIP *)
  match list with
  | [] -> k 0
  | (a :: bs) -> let r = lengthk bs in 1 + r
```

What is the CPS version of this?

Example: CPS for length

```
let rec lengthk list k = (* WIP *)
  match list with
  | [] -> k 0
  | (a :: bs) -> let r = lengthk bs in 1 + r
```

What is the CPS version of this?

Example: CPS for length

```
let rec lengthk list k = (* WIP *)
  match list with
  | [] -> k 0
  | (a :: bs) -> lengthk bs (fun r -> 1 + r)
```

What is the CPS version of this?

Example: CPS for length

```
let rec lengthk list k = (* WIP *)
  match list with
  | [] -> k 0
  | (a :: bs) -> lengthk bs (fun r -> 1 + r)
```

What is the CPS version of this?

Example: CPS for length

```
let rec lengthk list k = (* WIP *)
  match list with
  | [] -> k 0
  | (a :: bs) -> lengthk bs (fun r -> addk (r, 1) k)
```

What is the CPS version of this?

Example: CPS for length

```
let rec lengthk list k =
  match list with
  | [] -> k 0
  | (a :: bs) -> lengthk bs (fun r -> addk (r, 1) k)

# lengthk [2; 4; 6; 8] report;;
4
- : unit = ()
```

Example: CPS for length

```
let rec lengthk list k =
  match list with
  | [] -> k 0
  | (a :: bs) -> lengthk bs (fun r -> addk (r, 1) k)

# lengthk [2; 4; 6; 8] report;;
4
- : unit = ()
```

Example: CPS for length

```
let rec lengthk list k =
  match list with
  | [] -> k 0
  | (a :: bs) -> lengthk bs (fun r -> addk (r, 1) k)

# lengthk [2; 4; 6; 8] report;;
4
- : unit = ()
```

CPS for sum

```
# let rec sum list =
    match list with
    | [ ] -> 0
    | x :: xs -> x + sum xs;;
val sum : int list -> int = <fun>
# let rec sumk list k =
    match list with
    | [ ] -> k 0
    | x :: xs -> sumk xs  (fun r -> addk (x, r) k);;
val sumk : int list -> (int -> 'a) -> 'a = <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 sumk list k =
    match list with
    | [ ] -> k 0
    | x :: xs -> sumk xs  (fun r -> addk (x, r) k);;
val sumk : int list -> (int -> 'a) -> 'a = <fun>
```

CPS for sum

```
# let rec sum list =
    match list with
    | [ ] -> 0
    | x :: xs -> let r = sum xs in x + r;;
val sum : int list -> int = <fun>
# let rec sumk list k =
    match list with
    | [ ] -> k 0
    | x :: xs -> sumk xs (fun r -> addk (x, r) k);;
val sumk : int list -> (int -> 'a) -> 'a = <fun>
```

CPS for sum

```
# let rec sum list =
    match list with
    | [ ] -> 0
    | x :: xs -> let r = sum xs in x + r;;
val sum : int list -> int = <fun>
# let rec sumk list k =
    match list with
    | [ ] -> k 0
    | x :: xs -> sumk xs (fun r -> addk (x, r) k);;
val sumk : int list -> (int -> 'a) -> 'a = <fun>
```

CPS for sum

```
# let rec sum list =
    match list with
    | [ ] -> 0
    | x :: xs -> let r = sum xs in x + r;;
val sum : int list -> int = <fun>
# let rec sumk list k =
    match list with
    | [ ] -> k 0
    | x :: xs -> sumk xs (fun r -> addk (x, r) k);;
val sumk : int list -> (int -> 'a) -> 'a = <fun>
```

CPS for sum

```
# let rec sum list =
    match list with
    | [ ] -> 0
    | x :: xs -> let r = sum xs in x + r;;
val sum : int list -> int = <fun>
# let rec sumk list k =
    match list with
    | [ ] -> k 0
    | x :: xs -> sumk xs (fun r -> addk (x, r) k);;
val sumk : int list -> (int -> 'a) -> 'a = <fun>
```

CPS and Higher-Order Functions

CPS for Higher Order Functions

- In CPS, every function takes a **continuation** to receive its result
- Accordingly:
  - **Functions** passed as **arguments** take **continuations**
  - **Functions** returned as **results** take **continuations**
- CPS version of higher-order functions must **expect input functions** 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>
```

Example: all

```
# let rec allk (p, l) k = (* WIP *)
    match l with
    | [] -> true
    | x :: xs ->
        let b = p x in
        if b then
          all (p, xs)
        else
          false;;
```

Example: all

```
# let rec allk (p, l) k = (* WIP *)
    match l with
    | [] -> ?? true
    | x :: xs ->
        let b = p x in
        if b then
          all (p, xs)
        else
          false;;
```

What do we do with the **returned** value?

Example: all

```
# let rec allk (p, l) k = (* WIP *)
    match l with
    | [] -> k true
    | x :: xs ->
        let b = p x in
        if b then
          all (p, xs)
        else
          false;;
```

We **pass it forward**.

Example: all

```
# let rec allk (p, l) k = (* WIP *)
    match l with
    | [] -> k true
    | x :: xs ->
        let b = p x in
        if b then
          all (p, xs)
        else
          false;;
```

What do we do here?

Example: all

```
# let rec allk (p, l) k = (* WIP *)
    match l with
    | [] -> k true
    | x :: xs ->
        let b = p x in
        if b then
          all (p, xs)
        else
          false;;
```

We need to assume that **input function p** has been **transformed to CPS** already.

Example: all

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

We need to assume that **input function pk** has been **transformed to CPS** already.

Example: all

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

Now we can transform these to CPS in the standard way.

Example: all

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

Now we can transform these to CPS in the standard way.

Example: all

```
# 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>
```

Example: all

```
# 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>
```

Questions so far?

CPS Transformation

CPS Transformation

- **Step 1:** Add **continuation argument** to any function definition
  - let f arg = e ⇒ 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
  - a ⇒ **k** a
  - Assuming a is a constant or variable.
  - "Simple" = "No available function calls."

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
  - a $\Rightarrow$ **k** a
  - Assuming a is a constant or variable.
  - “Simple” = “No available function calls.”

CPS Transformation

- **Step 3: Pass** the **current continuation** to every function call in **tail** position
  - f arg ⇒ f arg **k**
  - The function "isn't going to return," so we need to tell it where to put the result.
- **Step 4:** Each function call **not in tail** position needs to be converted to **take a new continuation** (containing the old continuation as appropriate)
  - **op** (f arg) ⇒ f arg **(fun r -> k (op r))**
  - **op** represents a primitive operation
  - **g** (f arg) ⇒ f arg **(fun r -> g r k)**

CPS Transformation

- **Step 3: Pass** the **current continuation** to every function call in **tail** position
  - f arg ⇒ f arg **k**
  - The function "isn't going to return," so we need to tell it where to put the result.
- **Step 4:** Each function call **not in tail** position needs to be converted to **take a new continuation** (containing the old continuation as appropriate)
  - **op** (f arg) ⇒ f arg **(fun r -> k (op r))**
  - **op** represents a primitive operation
  - **g** (f arg) ⇒ f arg **(fun r -> g r k)**

Example

Before:

```
let rec sum lst =
match lst with
| [ ] -> 0
| 0 :: xs -> sum xs
| x :: xs ->
    (+) x (sum xs);;
```

After:

```
let rec sumk lst k = (* 1 *)
match lst with
| [ ] -> k 0 (* 2 *)
| 0 :: xs -> sumk xs k (* 3 *)
| x :: xs -> (* 4 *)
    sumk xs (fun r -> k ((+) x r));;
```

Example

Before:

```
let rec sum lst =
match lst with
| [ ] -> 0
| 0 :: xs -> sum xs
| x :: xs ->
    (+) x (sum xs);;
```

After:

```
let rec sumk lst k = (* 1 *)
match lst with
| [ ] -> k 0 (* 2 *)
| 0 :: xs -> sumk xs k (* 3 *)
| x :: xs -> (* 4 *)
    sumk xs (fun r -> k ((+) x r));;
```

Example

Before:

```
let rec sum lst =
match lst with
| [ ] -> 0
| 0 :: xs -> sum xs
| x :: xs ->
    (+) x (sum xs);;
```

After:

```
let rec sumk lst k = (* 1 *)
match lst with
| [ ] -> k 0 (* 2 *)
| 0 :: xs -> sumk xs k (* 3 *)
| x :: xs -> (* 4 *)
    sumk xs (fun r -> k ((+) x r));;
```

Example

Before:

```
let rec sum lst =
match lst with
| [ ] -> 0
| 0 :: xs -> sum xs
| x :: xs ->
    (+) x (sum xs);;
```

After:

```
let rec sumk lst k = (* 1 *)
match lst with
| [ ] -> k 0 (* 2 *)
| 0 :: xs -> sumk xs k (* 3 *)
| x :: xs -> (* 4 *)
    sumk xs (fun r -> k ((+) x r));;
```

Example

Before:

```
let rec sum lst =
match lst with
| [ ] -> 0
| 0 :: xs -> sum xs
| x :: xs ->
    (+) x (sum xs);;
```

After:

```
let rec sumk lst k = (* 1 *)
match lst with
| [ ] -> k 0 (* 2 *)
| 0 :: xs -> sumk xs k (* 3 *)
| x :: xs -> (* 4 *)
    sumk xs (fun r -> k ((+) x r));;
```

Questions so far?

Other Applications

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

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

Exceptions - Example

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

Other Applications

Exceptions - Example

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

Exceptions - Example

```
# let list_mult list =
   try mul_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
# mul_aux [7;4;0];;
Exception: Zero.
```

Exceptions - Example

```
# let list_mult list =
   try mul_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
# mul_aux [7;4;0];;
Exception: Zero.
```

Exceptions - Example

```
# let list_mult list =
    try mul_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
# mul_aux [7;4;0];;
Exception: Zero.
```

Exceptions - Example

```
# let list_mult list =
   try mul_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
# mul_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**

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>
```

Implementing Exceptions

```
# let rec mul_aux list k kexcp =
    match list with
    | [ ] -> k 1
    | x :: xs ->
        if x = 0 then
          kexcp 0
        else
          mul_aux xs (fun r -> multkp (x, r) k) kexcp;;
val mul_aux : int list -> (int -> 'a) -> (int -> 'a) -> 'a = <fun>
# let list_multk list k = mul_aux list k k;;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```

Exception Handler

Other Applications

Implementing Exceptions

```
# let rec mul_aux list k kexcp =
    match list with
    | [ ] -> k 1
    | x :: xs ->
        if x = 0 then
          kexcp 0
        else
          mul_aux xs (fun r -> multkp (x, r) k) kexcp;;
val mul_aux : int list -> (int -> 'a) -> (int -> 'a) -> 'a = <fun>
# let list_multk list k = mul_aux list k k;;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```

Exception Handler

Raise Exception

Other Applications

Implementing Exceptions

```
# let rec mul_aux list k kexcp =
    match list with
    | [ ] -> k 1
    | x :: xs ->
        if x = 0 then
          kexcp 0
        else
          mul_aux xs (fun r -> multkp (x, r) k) kexcp;;
val mul_aux : int list -> (int -> 'a) -> (int -> 'a) -> 'a = <fun>
# let list_multk list k = mul_aux list k k;;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```

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 = ()
```

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 = ()
```

Questions?

Takeaways

- We saw how to transform functions written in **direct style** to functions written in **continuation-passing style (CPS),** which is **super useful** for compilers and interpreters
- We also saw how to use continuations to implement **exceptions**—one of many features we can implement with continuations

Next Class

- **I will be away!** This absence is actually a planned absence.
- I will **record the lecture ahead of time** and post it for you all to watch.
- I will **announce** when it is ready.
- **There will not be an in-person lecture.**
- I will also miss **office hours**, but I will pay very close attention to Piazza.
- I will post (extra) extra credit today. (Please don't let it distract you from the midterm.)