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

Quiz (20 Minutes)

- Please **check in** before starting
- Then navigate to https://us.prairietest.com/
- Close all other tabs
- Start the quiz when ready
- Load one question in advance (two open instances)
- Please run the **test scripts** *before* submitting
- Note that there may be **more tests** for grading
- Let us know if you run into any issues
- Let us know if you'd like to check out

Three Minute Break

Objectives for Today

- Today, we will cover **tail recursion** a bit more
- We will focus on examples, capturing the importance of the **accumulator**
- We will use this to lead in to something called **continuation-passing style**, which has similar accumulation behavior, but accumulates the remaining **work to be done** rather than values
- This style, which we'll cover more next class, is **super useful** for compilers and interpreters

Thanks, all, for patience and help!

See Piazza

Questions from last time?

More Tail Recursion

Tail Recursion

- Tail Recursion form of **Structural Recursion** (recurse on substructures)
- In **tail** recursion, **first build the intermediate result**, then **call the function recursively**
- Build answer **as you go**, typically using an **accumulator** or **auxiliary function**
- Corresponds to **folding left** (with caveats)

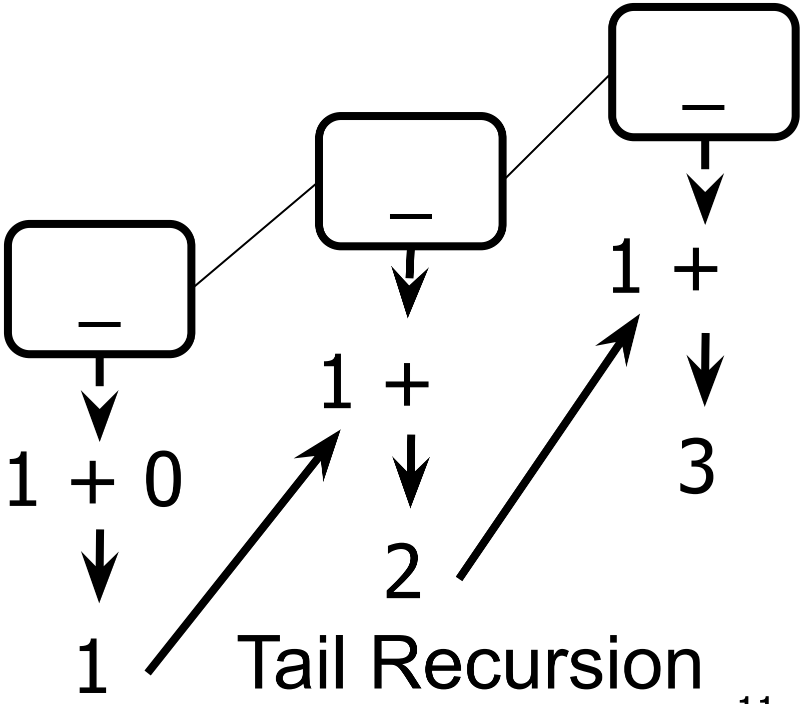
Tail Recursion

- Tail Recursion form of **Structural Recursion** (recurse on substructures)
- In **tail** recursion, **first build the intermediate result**, then **call the function recursively**
- Build answer **as you go**, typically using an **accumulator** or **auxiliary function**
- Corresponds to **folding left** (with caveats)

Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```

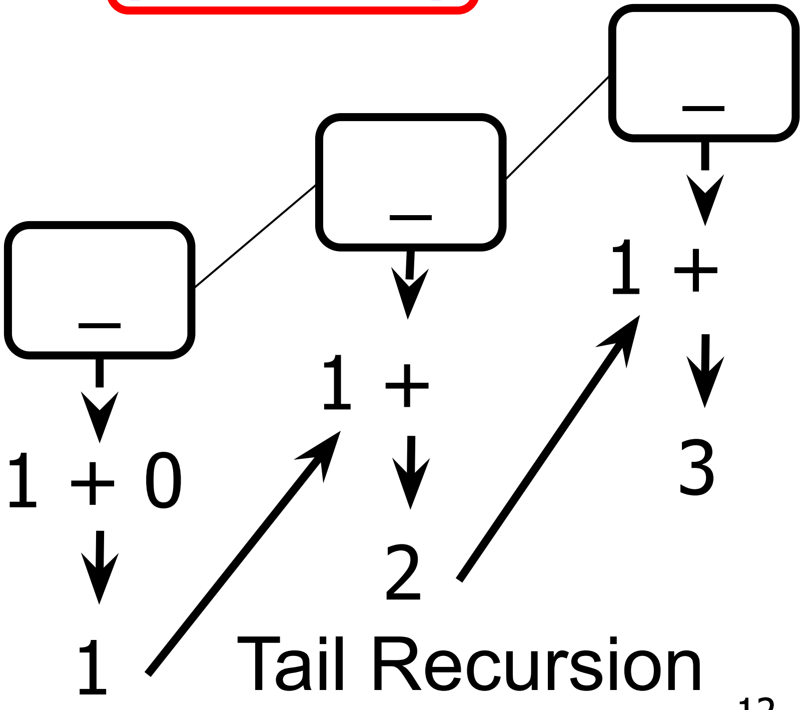


Tail Recursion

Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```

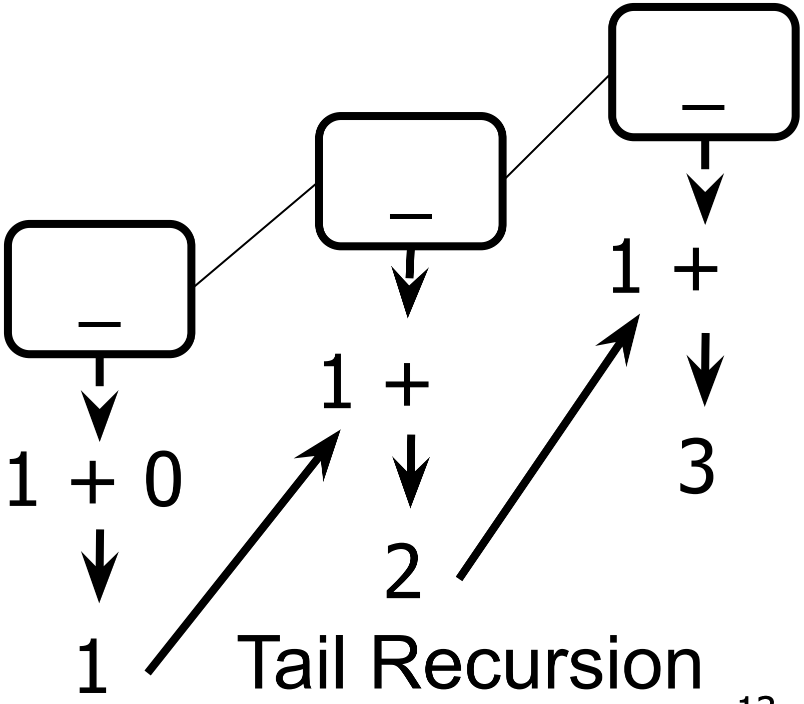


Tail Recursion

Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```



Tail Recursion

Your turn: num_neg – tail recursive

```
let num_neg list =
  let rec num_neg_aux list curr_neg =
    ???
```

What to do first?

```
  in num_neg_aux list ???
```

Your turn: num_neg – tail recursive

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

  in num_neg_aux list ???
```

Your turn: num_neg – tail recursive

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

  in num_neg_aux list ???
```

Base case?

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) ->
        ???

  in num_neg_aux list ???
```

The accumulated value!

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) ->
        ???

  in num_neg_aux list ???
```

Recursive case?

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 list ???
```

Recursive call is *last*.

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 list ???
```

How to accumulate (i.e., update curr_neg)?

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 list ???
```

Add 1 if the head is negative; otherwise change nothing.

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 list ???
```

The real work here happens in the accumulator. But we need an initial value for that accumulator. What should that be?

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 list 0
```

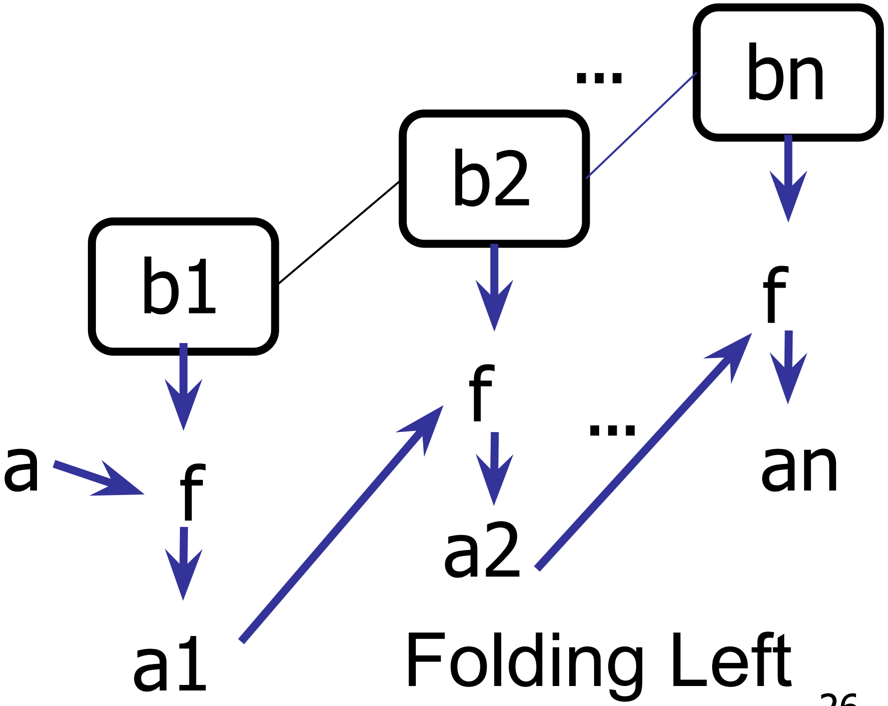
Questions so far?

Tail recursion with fold_left

Tail Recursion by **fold_left**

```
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>
```



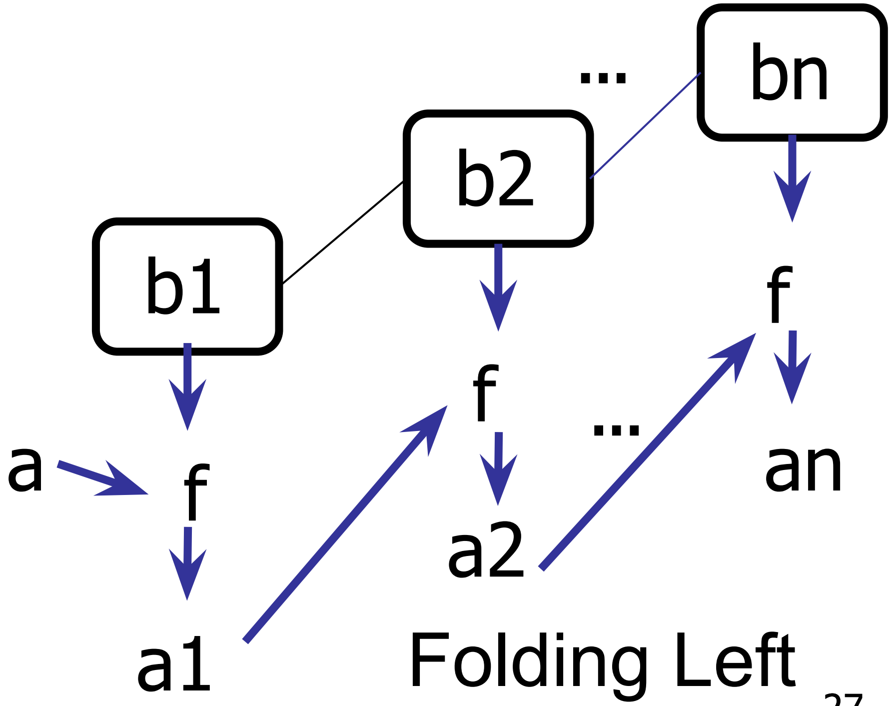
Folding Left

Tail Recursion by **fold_left**

```
let rec fold_left f a list =
  match list with
  | [ ] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
```

Argument a is the accumulated value.

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



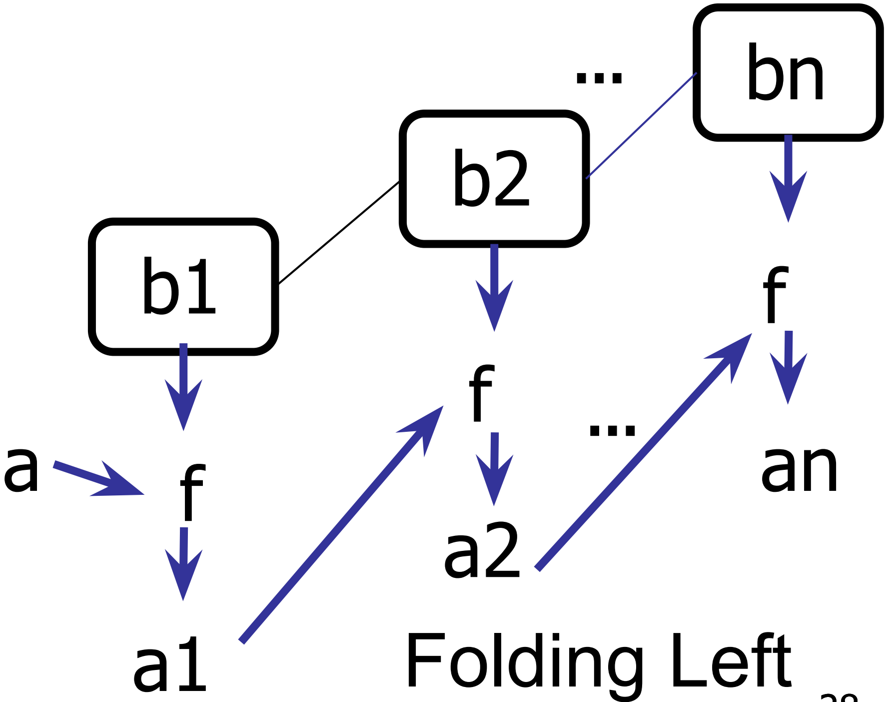
Folding Left

Tail Recursion by **fold_left**

```
let rec fold_left f a list =
  match list with
  | [ ] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
```

Operator f does the actual accumulation!

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

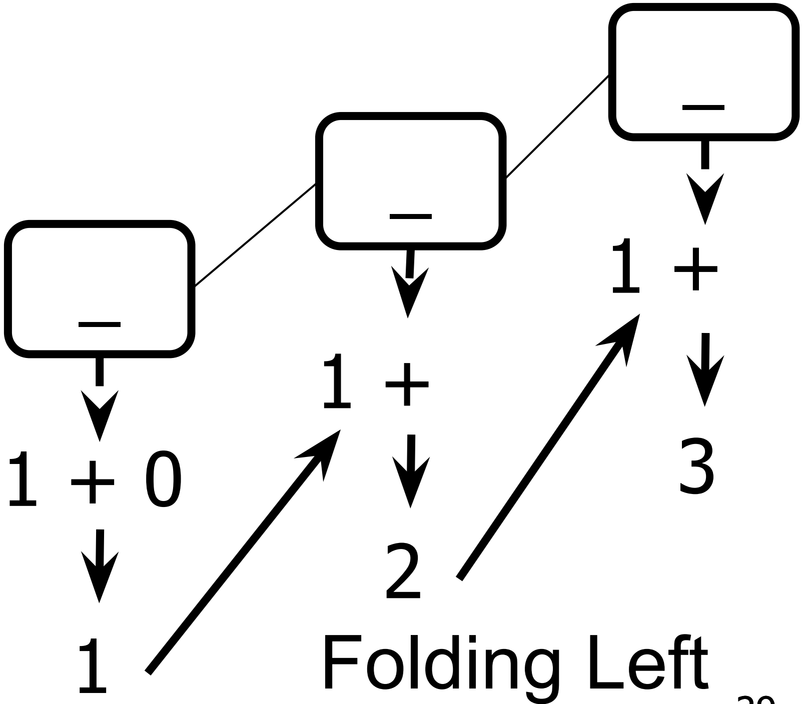


Folding Left

Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc)

let length =
  length_aux list 0
```



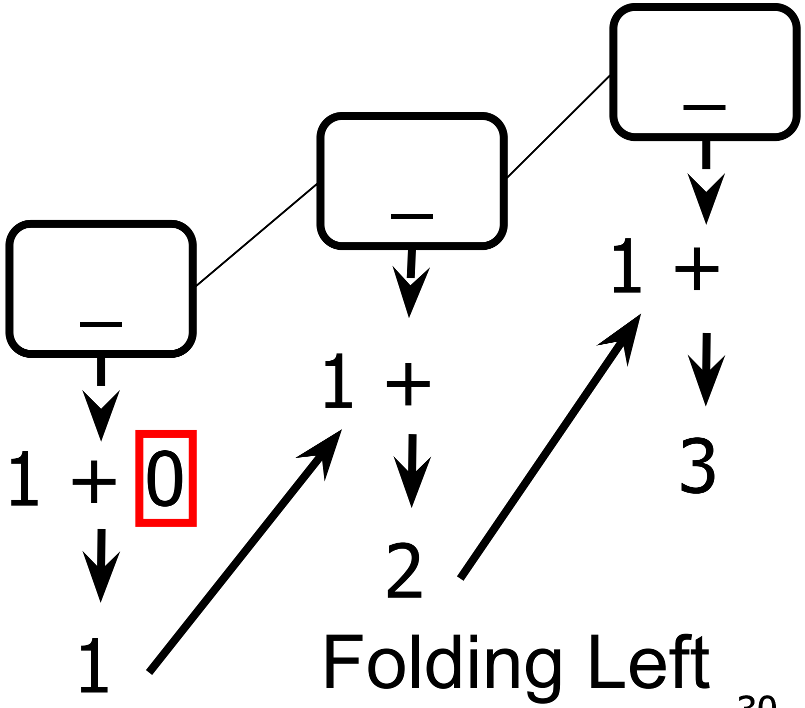
Folding Left

Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc)

let length =
  length_aux list 0
```

base case / id



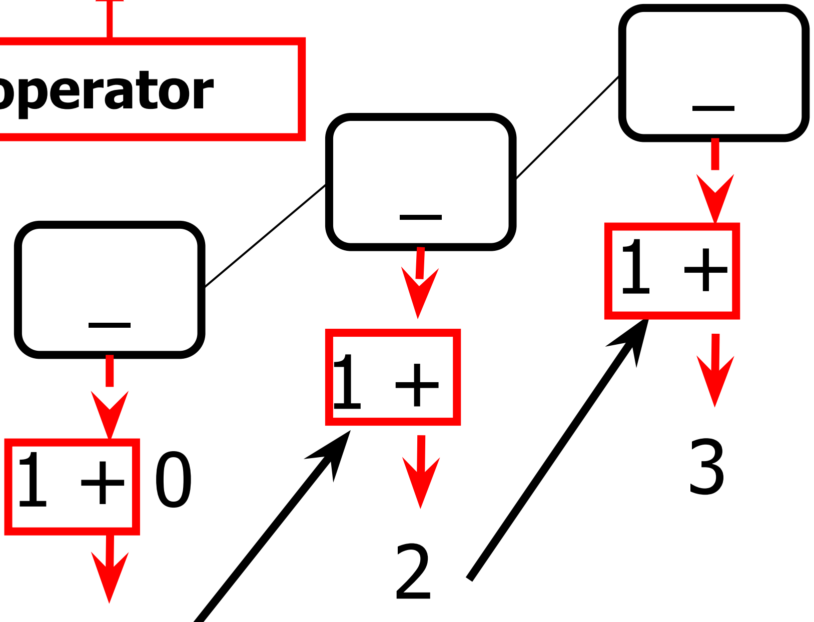
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc)

let length =
  length_aux list 0
```

operator

base case / id



Folding Left

Tail Recursion - Length

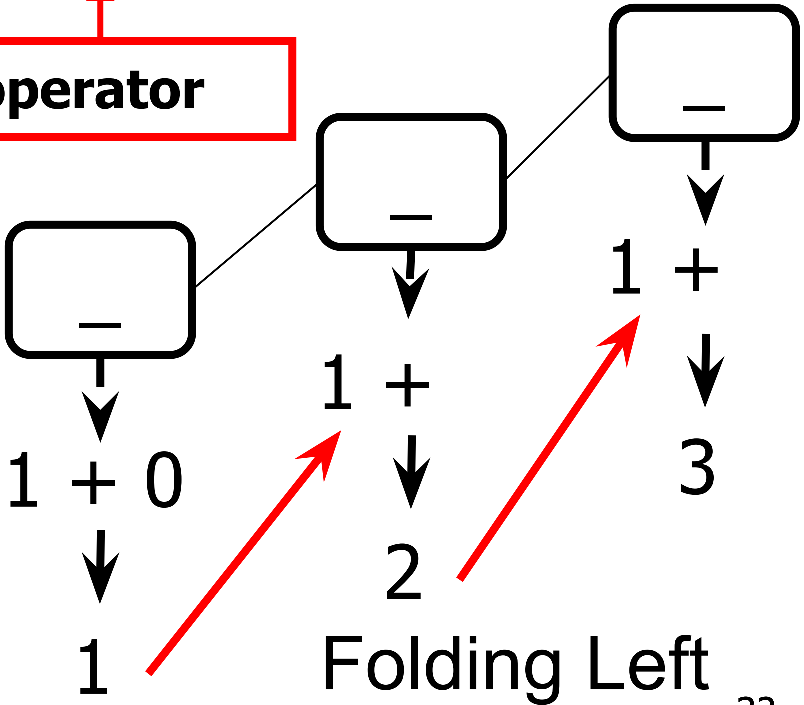
```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc)

let length =
  length_aux list 0
```

recursion (last)

operator

base case / id



Folding Left

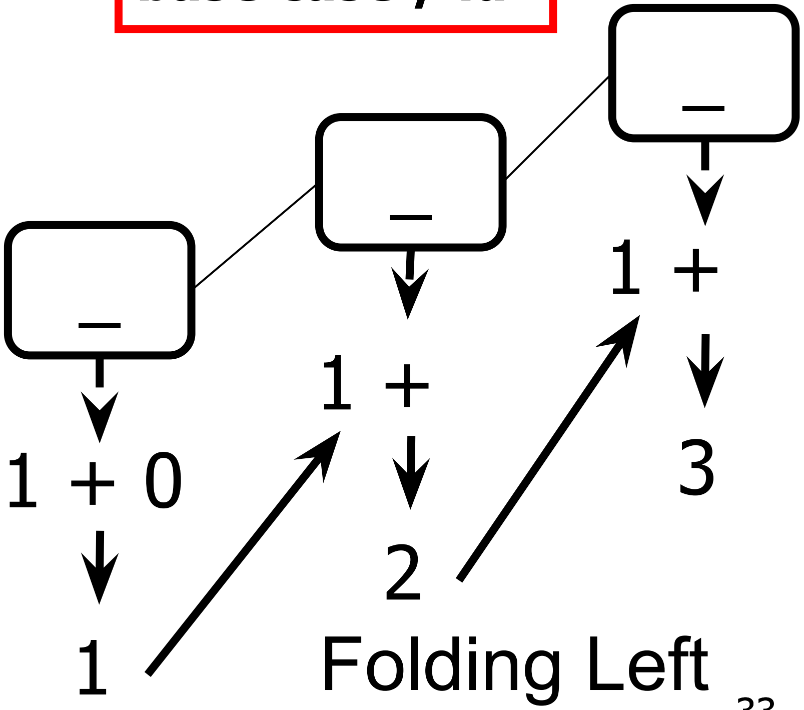
Tail Recursion - Length

```
let length list =
 fold_left (fun acc -> fun _ -> 1 + acc) list 0
```

recursion (last)

operator

base case / id



Folding Left

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 list 0
let num_neg list =
  fold_left ?? ?? list
```

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 list 0
```

```
let num_neg list =
  fold_left ?? ?? list
```

What is the base case—the initial accumulated value?

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 list 0
let num_neg list =
  fold_left ?? 0 list
```

Zero, so that's what we pass for the last argument.

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 list 0
```

```
let num_neg list =
  fold_left ?? 0 list
```

What operator do we use to update the accumulator?

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 list 0
```

This whole thing here:

```
let num_neg list =
  fold_left ?? 0 list
```

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 list 0
```

This whole thing here:

```
let num_neg list =
  fold_left (fun r x -> if x < 0 then 1 + r else r) 0 list
```

So we abstract it into a function:

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 list 0
let num_neg list =
  fold_left (fun x r -> if x < 0 then 1 + r else r) list 0
```

Folding Left

Your turn: num_neg – tail recursive

```
(* Concise, captures essence of accumulation *)
let num_neg list =
  fold_left (fun x r -> if x < 0 then 1 + r else r) list 0
```

Folding Left

Questions so far?

Continuations, Briefly

Continuations

- What if, rather than accumulating values, we **accumulate** the **work that remains** to be done?
- Then we get these things called ***continuations***.
- It turns out this is *very useful* for **"non-local" control flow**, like:
    - non-local jumps
    - exceptions
    - general conversion of non-tail calls to tail calls
- Essentially a higher-order function version of **GOTO**

Continuations

- What if, rather than accumulating values, we **accumulate** the **work that remains** to be done?
- Then we get these things called ***continuations***.
- It turns out this is *very useful* for **"non-local" control flow**, like:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially 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 is called a **continuation**
  - Continuation acts as **"accumulator"** for **work still to be done**

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 is called a **continuation**
  - Continuation acts as **"accumulator"** for **work still to be done**

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 is called a **continuation**
  - Continuation acts as **“accumulator”** for **work still to be done**

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 is called a **continuation**
  - Continuation acts as **“accumulator”** for **work still to be done**

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

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

Why CPS?

Reasoning:

- Explicit **order of evaluation**

Compilation:

- **Variables/registers** for **each step** of computation
- **Functional** to **imperative**
- Nice **IR** on the way to assembly or byte code

Optimization:

- **Tail recursion** easy to identify
- **Strict forward recursion** becomes **tail recursion** (at the expense of building large closures in heap)

Why CPS?

Reasoning:

- Explicit **order of evaluation**

Compilation:

- **Variables/registers** for **each step** of computation
- **Functional** to **imperative**
- Nice **IR** on the way to assembly or byte code

Optimization:

- **Tail recursion** easy to identify
- **Strict forward recursion** becomes **tail recursion** (at the expense of building large closures in heap)

Why CPS?

Reasoning:

- Explicit **order of evaluation**

Compilation:

- **Variables/registers** for **each step** of computation
- **Functional** to **imperative**
- Nice **IR** on the way to assembly or byte code

Optimization:

- **Tail recursion** easy to identify
- **Strict forward recursion** becomes **tail recursion** (at the expense of building large closures in heap)

Why CPS?

Reasoning:

- Explicit **order of evaluation**

Compilation:

- **Variables/registers** for **each step** of computation
- **Functional** to **imperative**
- Nice **IR** on the way to assembly or byte code

Optimization:

- **Tail recursion** easy to identify
- **Strict forward recursion** becomes **tail recursion** (at the expense of building large closures in heap)

Other Uses for Continuations

- **Changing order of evaluation**
- **Implementing**:
  - **Exceptions** and exception handling
  - **Coroutines**
  - (pseudo, aka green) **threads**

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 = ()
```

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 = ()
```

Questions?

Reminders

- **Midterm 1** in **CBTF** 9/14-9/16—**please sign up!**
- I'll post about the **first extra credit** on Piazza very soon this week.
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help

Next Class