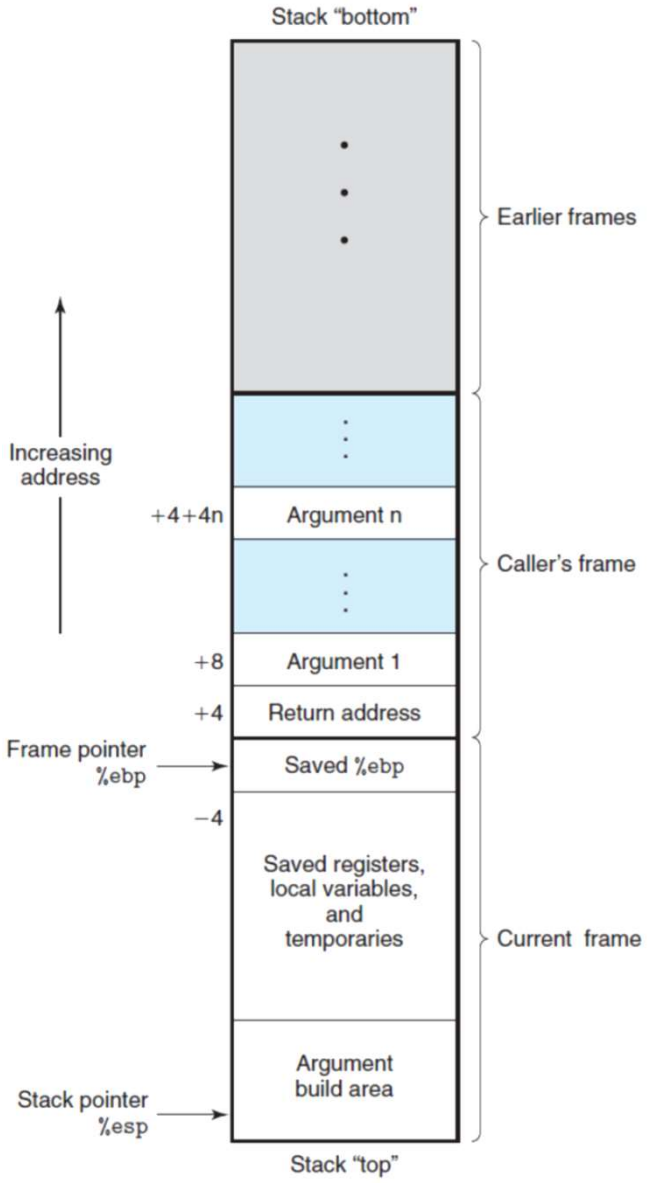
CSE216 Programming Abstractions

Tail Recursion,
Continuation Passing Style

YoungMin Kwon

Stack Frame

- Stack frame
  - When a function is invoked, a stack frame is created and it is pushed onto the stack
  - Contains
    - Function arguments
    - Local variables
    - Temporary variables
    - Return address …



Linear Recursion

- The first implementation of sum
  - Deep recursive processes can cause the stack overflow

```
let rec sum n =
    if n = 0
    then n
    else n + sum (n-1)
```

After returning from sum,
the result needs to be added to n

```
# sum 1000000;;
Stack overflow during evaluation (looping recursion?).
```

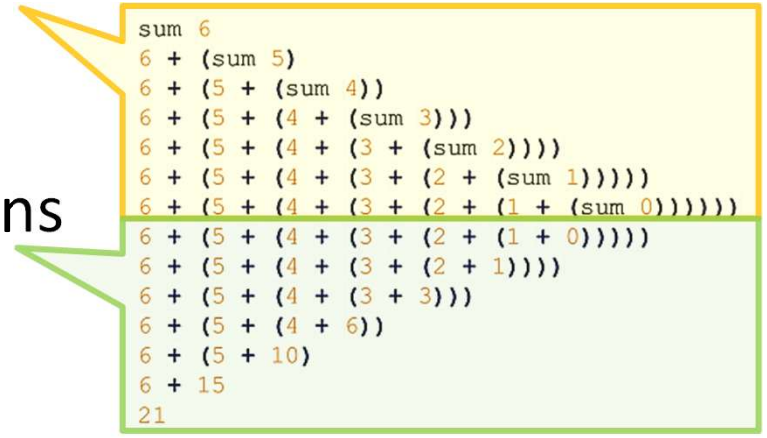
Linear Recursion

- Steps in sum 6

```
sum 6
6 + (sum 5)
6 + (5 + (sum 4))
6 + (5 + (4 + (sum 3)))
6 + (5 + (4 + (3 + (sum 2))))
6 + (5 + (4 + (3 + (2 + (sum 1)))))
6 + (5 + (4 + (3 + (2 + (1 + (sum 0))))))
6 + (5 + (4 + (3 + (2 + (1 + 0)))))
6 + (5 + (4 + (3 + (2 + 1))))
6 + (5 + (4 + (3 + 3)))
6 + (5 + (4 + 6))
6 + (5 + 10)
6 + 15
21
```

Linear Recursion

- First implementation of sum
  - After invoking sum, the result is added to n
    - Expansion occurs as the process builds up the chain of deferred operations
    - Contraction occurs as the operations actually performed



  - The interpreter needs to keep track of the deferred operations
    - Stack frames are pushed on to the stack

Tail Recursive Function

- The second implementation of sum
  - Revision to an iterative process

```
let sum n =
    let rec iter n acc =
        if n = 0
        then acc
        else iter (n-1) (acc+n) in
    iter n 0

# sum 1000000;;
- : int = 500000500000
```

No more computation
after returning from iter

```
sum 6
iter 6 0
iter 5 6
iter 4 11
iter 3 15
iter 2 18
iter 1 20
iter 0 21
21
```

Tail Recursive Function

- Second implementation of sum
  - No deferred operation after invoking sum
  - No expansion nor contraction

- Iterative process
  - A process whose state can be summarized by a fixed number of state variables

Tail Recursive Function

- Tail recursive function
  - A recursive function whose last action is the recursive call
  - Tail-call optimization: for a recursive function, instead of making a recursive call, jump to the beginning of the function
  - Converting a recursive function to a tail recursive function is not always straightforward

Tail Recursive Function

- Tail-call optimization

```
let sum n =
    let rec iter n acc =
        if n = 0
        then acc
        else iter (n-1) (acc+n) in
    iter n 0
```

⇒

```
int iter(int n, int acc) {
start:
    if(n = 0)
        return acc;
    else {
      /*update param variables*/
        n = n - 1;
        acc = acc + n;

      /*jump to start instead of
        making a recursive call*/
        goto start;
    }
}
```

Continuation Passing Style

- Continuation
  - A single parameter function that represents the rest of the program
  - A function written in CPS takes an extra parameter: continuation
  - The function computes its result and passes it to the continuation

Continuation Passing Style

- CPS makes operations explicit
  - Procedure return ⇒ call to a continuation
  - Intermediate values ⇒ given names
  - Order of argument eval ⇒ explicit in continuation
  - Tail call ⇒ call the procedure with the same continuation

Continuation Passing Style

- The third implementation of sum
  - Continuation Passing Style (CPS)

```
let rec sum n =
    if n = 0
    then n
    else n + sum (n-1)
```

continuation

```
let sum n =
    let rec iter n k =
        if n = 0
        then k n (*return => call to a continuation*)
        else iter (n-1) (fun x -> (*x: name of an intermediate
                                    result*)
                                  k (x + n)) in (*return => call to a
                                                  continuation*)

    iter n (fun x -> x)

# sum 1000000;;
-  : int = 500000500000
```

No deferred operation: iter is tail recursive

pass the result to continuation

Continuation Passing Style

- Steps of sum 6
  - Deferred operations are in the continuaions

```
sum 6
iter 6 (c0 = fun x -> x)
iter 5 (c1 = fun x -> c0 (x + 6))
iter 4 (c2 = fun x -> c1 (x + 5))
iter 3 (c3 = fun x -> c2 (x + 4))
iter 2 (c4 = fun x -> c3 (x + 3))
iter 1 (c5 = fun x -> c4 (x + 2))
iter 0 (c6 = fun x -> c5 (x + 1))
(c6 0)
(c5 1)
(c4 3)
(c3 6)
(c2 10)
(c1 15)
(c0 21)
21
```

Continuation Passing Style

- Continuation
  - On the return path

```
let rec iter n k =
    if n = 0
    then k n
    else iter (n-1) (fun x -> k (x + n))
```

    - If n = 0, pass 0 to continuation k: `fun x -> k (x + 1)`
    - The continuation will pass 1 to k: `fun x -> k (x + 2)`
    - The continuation will pass 3 to k: `fun x -> k (x + 3)`
    - The continuation will pass 6 to k: `fun x -> k (x + 4)`
    - …

Continuation Passing Style

- Tail-call case

```
let sum n =
    let rec iter n acc =
        if n = 0
        then acc
        else iter (n-1) (acc+n) in (*tail-call*)
    iter n 0
```

```
let sum n =
    let rec iter n acc k =
        if n = 0
        then k acc (*return => call to a continuation*)
        else iter (n-1) (acc+n) k in (*tail-call => call iter with
                                      the same continuation k*)
    iter n 0 (fun x -> x)
```

Continuation Passing Style

- Continuation
  - Parameter k can be moved into iter function

```
let sum n =
    let rec iter n =
        if n = 0
        then fun k -> k n                (*return a continuation*)
        else fun k ->
             (iter (n-1))                (*iter... returns a continuation*)
             (fun x -> k (n+x)) in       (*pass fun x... to the cont*)
    iter n (fun x -> x)                  (*id function*)

let _ = sum 1000000
```

Continuation Passing Style

- Exercise
  - Implement factorial function in CPS

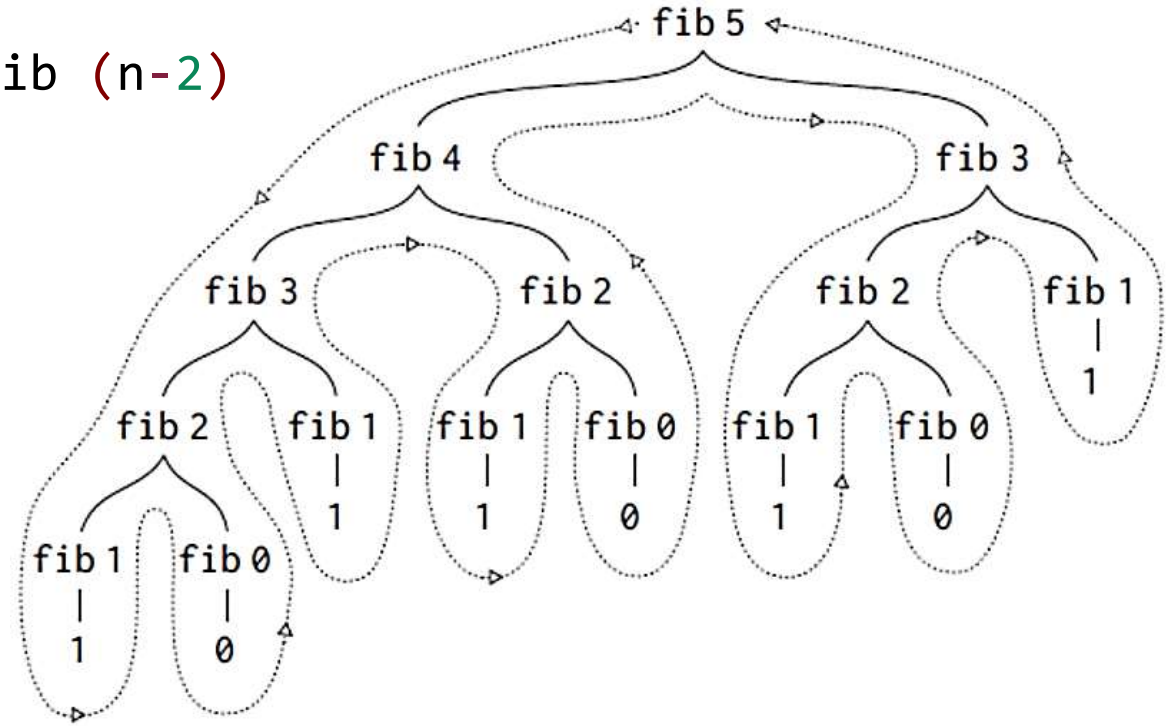
```
let rec fact n =
    if n = 1
    then n
    else n * fact (n-1)

let _ = fact 4
```

Tree Recursion

- Fibonacci number (recursive process)

```
let rec fib n =
    if n <= 1
    then n
    else fib (n-1) + fib (n-2)
```



Tree Recursion

- Fibonacci number (iterative process)

```
let fib n =
    (*state: a: n-1st fib no., b: n-2nd fib no.*)
    let rec f n a b =
        if n = 1
        then a
        else f (n-1) (a+b) a in
    f n 1 0
```

  - This implementation is efficient, but it is not always easy to find the states.

Tree Recursion

- Fibonacci number (CPS)

```
let fib n =
    let rec iter n k =
        if n <= 1
        then k n                          (*return => call to a continuation*)
        else iter (n-1) (fun x ->         (*eval order for x and y is fixed*)
             iter (n-2) (fun y ->         (*x, y: intermediate results*)
                  k (x + y))) in          (*return => call to a continuation*)
    iter n (fun x -> x)
```

  - Not as efficient as the state-based iterative form
  - However, easy to implement in a tail recursive form

Tree Recursion

- Fibonacci number (CPS)
  - Move k into iter fuction

```
let fib n =
    let rec iter n =
        if n <= 1
        then fun k -> k n
        else fun k ->
             iter (n-1) (fun x ->
             iter (n-2) (fun y ->
             k (x + y))) in
    iter n (fun x -> x)
```

Continuation Passing Style

- Exercise
  - Implement seq in CPS

```
(*f(n) = n                                  : if n < 3
         f(n-1) + 2*f(n-2) + 3*f(n-3) : otherwise
*)

let rec seq n =
    if n < 3
    then n
    else seq (n-1) + 2*seq (n-2) + 3*seq(n-3)

let _ = seq 5
```

Symbolic Derivative

- Symbolic Derivative
  - Given a symbolic expression, return its symbolic derivative

- Differentiation

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

Symbolic Derivative

- Symbolic expressions
  - Recursive type definitions

```
(*expression type*)
type expr = N of int              (*number*)
          | X | Y | Z             (*variable*)
          | Add of expr * expr    (*add expr*)
          | Mul of expr * expr    (*mul expr*)
```

  - Examples

```
let _ = N 1 (* 1 *)
let _ = Y   (* y *)

(* x^2 + 2x + 1 *)
let _ = Add( Add( Mul(X, X), Mul(N 2, X)), N 1)
```

Symbolic Derivative

- Helper functions

```
let add a b = (*add two exprs*)
    match (a, b) with
    | (N 0, y) -> y
    | (x, N 0) -> x
    | (N x, N y) -> N (x + y)
    | _ -> Add (a, b)

let mul a b = (*multiply two exprs*)
    match (a, b) with
    | (N 0, y) -> N 0
    | (N 1, y) -> y
    | (x, N 0) -> N 0
    | (x, N 1) -> x
    | (N x, N y) -> N (x * y)
    | _ -> Mul (a, b)
```

Symbolic Derivative

- Symbolic derivative

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

```
(*derivative of expr w.r.t. var*)
let rec deriv expr var =
    match expr with
    | N x -> N 0
    | X | Y | Z  -> if expr = var then N 1
                                  else N 0
    | Add (a, b) -> add (deriv a var) (deriv b var)
    | Mul (a, b) -> add (mul a (deriv b var))
                        (mul (deriv a var) b)
```

  - Example

```
let d = Add( Add( Mul(X, X), Mul(N 2, X)), N 1)

let dd = (deriv d X)
val dd : expr = Add (Add (X, X), N 2)
```

Symbolic Derivative

- to_str: convert a symbolic expr to a string

```
(*convert expr to string*)
let rec to_str expr =
    let open Printf in
    match expr with
    | N a -> sprintf "%d" a
    | X -> "x"
    | Y -> "y"
    | Z -> "z"
    | Add (a, b) -> sprintf "(%s + %s)" (to_str a) (to_str b)
    | Mul (a, b) -> sprintf "%s * %s"   (to_str a) (to_str b)
```

- Example

```
let _ = d  |> to_str
-  : string = "((x * x + 2 * x) + 1)"

let _ = dd |> to_str
- : string = "((x + x) + 2)"
```

Symbolic Derivative

- Test

```
(*test cases*)
let test () =
    let ( + ) = add in
    let ( * ) = mul in

    let a = X + N 3 in
    let b = N 2 * X + N 3 * Y in
    let c = X * Y * (X + N 3) in
    let d = X * X + N 2 * X + N 1 in

    [a; b; c; d] |> List.iter (fun exp ->
            let prn = Printf.printf "%s\n" in
            exp          |> to_str |> prn;
            deriv exp X |> to_str |> prn;
            deriv exp Y |> to_str |> prn)

let _ = test ()
```

```
Result:
(x + 3)
1
0

(2 * x + 3 * y)
2
3

x * y * (x + 3)
(x * y + y * (x + 3))
x * (x + 3)

((x * x + 2 * x) + 1)
((x + x) + 2)
0
```

Symbolic Derivative

- deriv in CPS

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

```
(*derivative of expr w.r.t. var*)
let rec deriv expr var k =
    match expr with
    | N x -> k (N 0)
    | X | Y | Z  -> if expr = var then k (N 1)
                                  else k (N 0)
    | Add (a, b) -> deriv a var (fun da ->
                    deriv b var (fun db ->
                    k (add da db)))
    | Mul (a, b) -> deriv a var (fun da ->
                    deriv b var (fun db ->
                    k (add (mul a db)
                           (mul da b))))
```

Symbolic Derivative

- to_str in CPS

```
(*convert expr to string*)
let rec to_str expr k =
    let open Printf in
    match expr with
    | N a -> k (sprintf "%d" a) (*return => call to a continuation*)
    | X -> k "x"                (*return => call to a continuation*)
    | Y -> k "y"
    | Z -> k "z"
    | Add (a, b) -> to_str a (fun sa -> (*eval order for a and b is fixed*)
                    to_str b (fun sb -> (*sa, sb: intermediate results*)
                    k (sprintf "(%s + %s)" sa sb)))
    | Mul (a, b) -> to_str a (fun sa ->
                    to_str b (fun sb ->
                    k (sprintf "%s * %s" sa sb))) (*return =>
                                                    call to a continuation*)
```

Symbolic Derivative

- Test

```
(*test cases*)
let test () =
    let ( + ) = add in
    let ( * ) = mul in

    let a = X + N 3 in
    let b = N 2 * X + N 3 * Y in
    let c = X * Y * (X + N 3) in
    let d = X * X + N 2 * X + N 1 in

    [a; b; c; d] |> List.iter (fun exp ->
            let prn = Printf.printf "%s\n" in
            to_str exp prn;
            deriv exp X (fun e -> to_str e prn);
            deriv exp Y (fun e -> to_str e prn))

let _ = test ()
```

```
Result:
(x + 3)
1
0

(2 * x + 3 * y)
2
3

x * y * (x + 3)
(x * y + y * (x + 3))
x * (x + 3)

((x * x + 2 * x) + 1)
((x + x) + 2)
0
```

Assignment 4

- Tiny PL in CPS
  - In this assignment, implement to_str and eval functions of Tiny PL in CPS
  - Download tiny_eval_cps.zip and implement to_str and eval functions in CPS
- Due date: 4/16/2024

Assignment 4

```
#use "globals.ml" (*global definitions*)
#use "scanner.ml" (*lexical analysis: string -> token list*)
#use "parser.ml"  (*parsing: token list -> parse tree*)
#use "eval.ml"    (*evaluating parse tree*)

(*print expr*)
let print expr =
    (*TODO: implement to_str in CPS*)
    let rec to_str e k =
…
    to_str expr (Printf.printf "%s\n")

(*evaluate expr in env*)
(*TODO: implement eval in CPS*)
let rec eval expr env k =
    let dropNUM  = …
    let dropBOOL = …
    let dropCLO  = …

    (*look up the value of a variable from an environment*)
    let rec lookup name env =
…
```