# CSE216 Programming Abstractions

## Lazy Evaluation

YoungMin Kwon

# Lazy Evaluation

- Function application
  - Arguments (without being evaluated) are stored in an environment as a thunk
  - Delay evaluating the actual parameters until they are necessary
- When the variable is actually used
  - The thunk is forced

# Tiny: Lazy Evaluation

- Convert Tiny to a Lazy evaluator
  - Download tiny_eval.zip
  - Update expr type in globals.ml
  - Update eval function in eval.ml

  - Implement the test programs in reci_lazy.ml

# Tiny: Lazy Evaluation

```
(* TODO: in globals.ml,
   add TNK (thunk) type to expr type.
   thunk is product of expr and env in which the expr will be evaluated
   list is a list of string and expr pairs
*)
type expr
    = NUM  of int    (*number*)
    | BOOL of bool   (*Boolean*)
    | VAR  of string (*variable*)
    (*arithmetic exprs*)
    | ADD of expr * expr | SUB of expr * expr
…
    (*function definition: parameter, body*)
    | FUN of string * expr
    (*closure: parameter, body, environment*)
    | CLO of string * expr * (string * expr) list

    (*lazy eval, thunk: expr, env*)
    | TNK of expr * (string * expr) list

    (*function application: operator, operand*)
    | APP of expr * expr
```

```
(*evaluate expr in env*)
let rec eval expr env =
…

    (* TODO: add force function to eval.
       for thunk parameters, force evaluates its expr in its env;
       for others, force simply returns the parameter
    *)

    let force = function
        | TNK (e, ev) -> eval e ev (*lazy*)
        | x -> x in
```

```
(*evaluate expr in env*)
let rec eval expr env =
…
    match expr with
    | BOOL b -> BOOL b
    | NUM n  -> NUM n

    (* TODO: for variable expressions,
       lookup the thunk bound to the variable name from env
       and force it
    *)
    | VAR v  -> lookup v env |> force
```

```
(*evaluate expr in env*)
let rec eval expr env =
…
    match expr with
…
    | FUN (v, e)    -> CLO (v, e, env)

    (* TODO: for function applications,
       instead of evaluating the parameter, add it to the env
       as a thunk bound to the formal parameter name
    *)
    (*lazy: thunk a and env without evaluating a*)
    | APP (f, a)    -> eval f env  |> fun clo ->
                       dropCLO clo |> fun (v, e, ev) ->
                       eval e ((v, TNK (a, env))::ev)
…
```

```
(*   reci_lazy.ml
*)
…

(*--Helpers-----------------------*)
(*eval_str: evaluates string expression in env*)
let eval_str str_expr env =
    parse str_expr
    |> fun expr -> eval expr env


(*define: extends env with the value of string expression
*)
let define name str_expr env =
    eval_str str_expr env
    |> fun v -> (name, v)::env
```

```
(* extend env with basic functions
*)
let str_rec =
    "(lambda (f)
        (f f))"
let str_cons =
    "(lambda (x y z)
        (if z x y))"
let str_car =
    "(lambda (x)
        (x true))"
let str_cdr =
    "(lambda (x)
        (x false))"

let env = []
        |> define "rec"  str_rec
        |> define "cons" str_cons
        |> define "car"  str_car
        |> define "cdr"  str_cdr
```

```
(*TODO: implement max function*)
let str_max =


let _ = env
        |> define "max" str_max
        |> eval_str "(max 1 2)"
        |> print
```

```
(*TODO: implement max function*)
let str_max =
    "(lambda (x y)
        (if (>= x y) x y))"

let _ = env
        |> define "max" str_max
        |> eval_str "(max 1 2)"
        |> print

(*-- move this line to the next TODO when done ----------
```

```
(*TODO: implement gcd function
    hint: use rec
*)
let str_gcd =


let _ = env
        |> define "gcd" str_max
        |> eval_str "(gcd 30 42)"
        |> print
```

```
(*TODO: implement gcd function
    hint: use rec
*)
let str_gcd =
    "(rec (lambda (gcd x y)
            (if (= x y)
                x
                (if (>= x y)
                    (rec gcd (- x y) y)
                    (rec gcd (- y x) x)))))"

let _ = env
        |> define "gcd" str_max
        |> eval_str "(gcd 30 42)"
        |> print
```

```
(*TODO: implement index function
    it takes a stream and a number and returns
    the number-th element of the stream
*)
let str_index =


(*TODO: implement the stream of natural numbers
*)
let str_nat =


let _ = env
        |> define "index" str_index
        |> define "nat"   str_nat
        |> eval_str "(index nat 10)"
        |> print
```

```
(*TODO: implement index function
    it takes a stream and a number and returns
    the number-th element of the stream
*)
let str_index =
    "(rec (lambda (index strm n)
            (if (= n 0)
                (car strm)
                (rec index (cdr strm) (- n 1)))))"

(*TODO: implement the stream of natural numbers
*)
let str_nat =
    "(rec (lambda (nat n)
            (cons n (rec nat (+ n 1))))
            0)"

let _ = env
        |> define "index" str_index
        |> define "nat"   str_nat
        |> eval_str "(index nat 10)"
        |> print
```