CSE216 Programming Abstractions

Algebraic Data Type

YoungMin Kwon

Algebraic Data Type

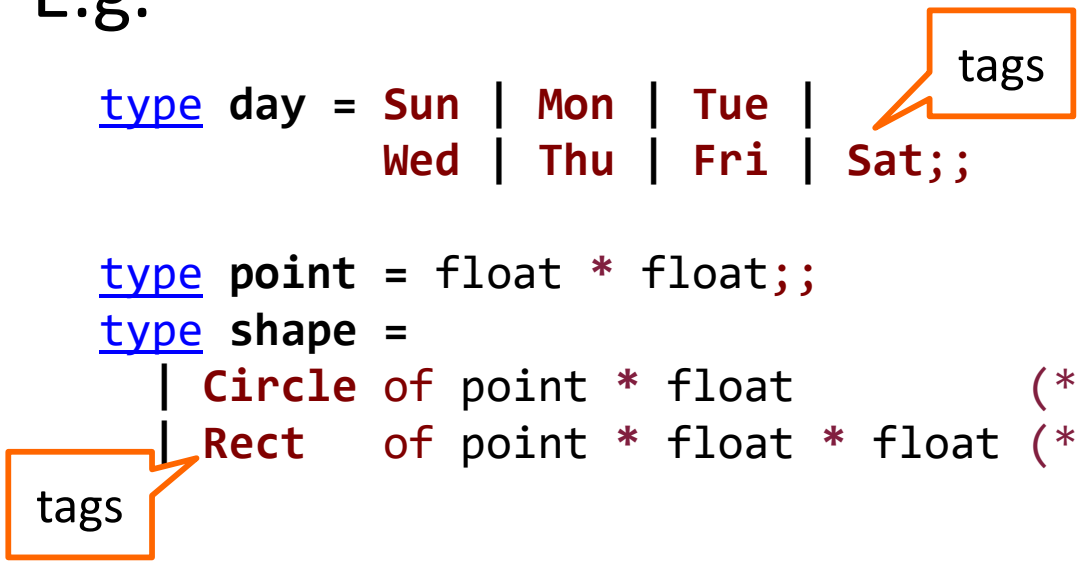
- Recursive data type
  - Data type for values that contain other values of the same type

```
class List<E> {
    E value;
    List<E> next;
}
```

- Algebraic data type
  - A data type representing one of many possibilities
  - Possibly recursive sum type of product types
    - Product types: tuple (Cartesian product), record, function

Variants

- Variants
  - Data that may represent multiple different forms
  - Each form is marked by an explicit tag
  - E.g.




```
type day = Sun | Mon | Tue |
           Wed | Thu | Fri | Sat;;

type point = float * float;;
type shape =
  | Circle of point * float          (* center, radius *)
  | Rect   of point * float * float (* left-top,
                                       width, height *);;

Circle ((1.,2.), 3.)
```

Variants

- Accessing data
  - Pattern matching
    - Use match or function

```
let area shape =
    let pi = 3.141592 in
    match shape with
    | Circle (c, r)    -> pi *. r *. r
    | Rect   (_, w, h) -> w *. h

area (Circle ((1.,2.), 3.))
```

Ignore the top-left point

Variants

- Example: ANSI color code

```
type base_color = Black | Red     | Green | Yellow |
                  Blue  | Magenta | Cyan  | White;;

type weight = Regular | Bold;;

type color =
    | Basic of base_color * weight
    | RGB   of int * int * int
    | Gray  of int;;
```

```
# #use "color.ml";;
…
Hello World !!!
- : unit = ()
```

```
let clr_to_str clr =
    let base_clr_to_num clr =
        match clr with
        | Black -> 0    | Red -> 1      | Green -> 2    | Yellow -> 3
        | Blue -> 4     | Magenta -> 5  | Cyan -> 6     | White -> 7 in

    let weight_to_num = function
        | Regular -> 0
        | Bold    -> 8 in

    let clr_to_num clr =
        match clr with
        | Basic (clr, wgt) -> (base_clr_to_num clr) + (weight_to_num wgt)
        | RGB (r, g, b)    -> 16 + b + g * 6 + r *36
        | Gray i           -> 232 + i in

    Printf.sprintf "\027[38;5;%dm" (clr_to_num clr)
```

```
let reset = "\027[0m"
let yellow = clr_to_str (Basic (Yellow, Bold))
let green  = clr_to_str (Basic (Green,  Bold))

let _ = Printf.printf "%sHello %sWorld %s!!!\n" yellow green reset
```

```
# #use "color.ml";;
…
Hello World !!!
- : unit = ()
```

- In Windows 10
  - To test the example in windows 10 terminal (cmd.exe), run the following first

```
> REG ADD HKCU\CONSOLE /f /v VirtualTerminalLevel /t REG_DWORD /d 1
The operation completed successfully.
>
```

Parametric Polymorphism

- Type variable
  - Example

```
# (+);;
- : int -> int -> int = <fun>
# (>);;
- : 'a -> 'a -> bool = <fun>
```

Variable names beginning with ' (single quote) are for type variables

  - 'a is a type variable
    - Any type
  - > is a polymorphic function (generic function)
    - > is defined for any types as long as the two parameters are of the same type

Lists as Variants

- Defining a List with variants

```
type 'a list = Nil
             | Cons of 'a * 'a list;;
```

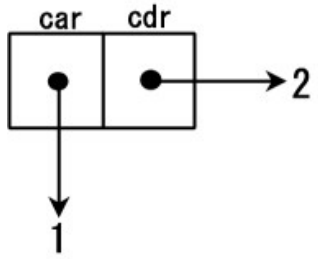
  - List type is a recursive data type
    - 'a list contains 'a list

  - Example

```
# Cons (1, Cons (2, Cons (3, Nil)));;
- : int list = Cons (1, Cons (2, Cons (3, Nil)))
```

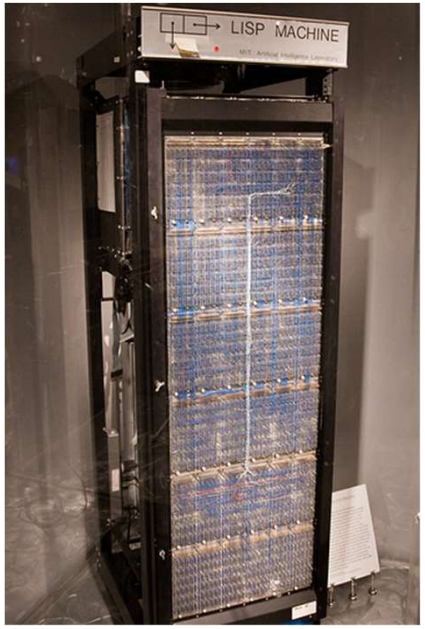
Lists as Variants

- Constructor and accessors




```
(*constructors*)
let nil () = Nil                    (*[] operator*)
let cons a b = Cons (a, b)          (*:: operator*)

(*accessors*)
let car = function
    | Cons (a, _) -> a
    | _ -> assert false

let cdr = function
    | Cons (_, b) -> b
    | _ -> assert false
```

Any other case



Lists as Variants

- Max element of a list

```
(*max of a list*)
let max lst =
    let rec iter m lst =
        if lst = nil () then
            m
        else
            let a = car lst in
            if a > m then iter a (cdr lst)
                     else iter m (cdr lst) in
    iter (car lst) (cdr lst)

(*define a list*)
let lst = cons 1 (cons 2 (cons 3 (nil ())))

(*compute its max*)
let _= max lst
```

Lists as Variants

- Max element of the list

```
(*define a list*)
let lst= nil () |> cons 1 |> cons 2 |> cons 3 |> cons 4 |> cons 5
                |> cons 6 |> cons 7 |> cons 8 |> cons 9 |> cons 10

let _ = Printf.printf "%d\n" (car lst)
let _ = Printf.printf "%d\n" (car (cdr lst))
(*compute its max*)
let _= max lst
```


```
10
9
- : int = 10
```

Lists as Variants 2

- Alternative representation of a list

```
(*List using a function: first attempt*)

(*type list*)
type 'a list = Nil
             | Fun of (bool -> 'a list);;

(*constructors*)
let nil () = Nil
let cons a b =
    Fun (fun z -> if z then a else b)

# let a = cons 1 (nil ());;
Error: This expression has type int but an expression
       was expected of type 'a list
```

a and b should be of the same type

Lists as Variants 2

- Alternative representation of a list

```
(*List using a function*)

(*type list*)
type 'a list = Nil
             | Val of 'a
             | Fun of (bool -> 'a list);;

(*constructors*)
let nil () = Nil
let cons a b =
    Fun (fun z -> if z then Val a else b)
```

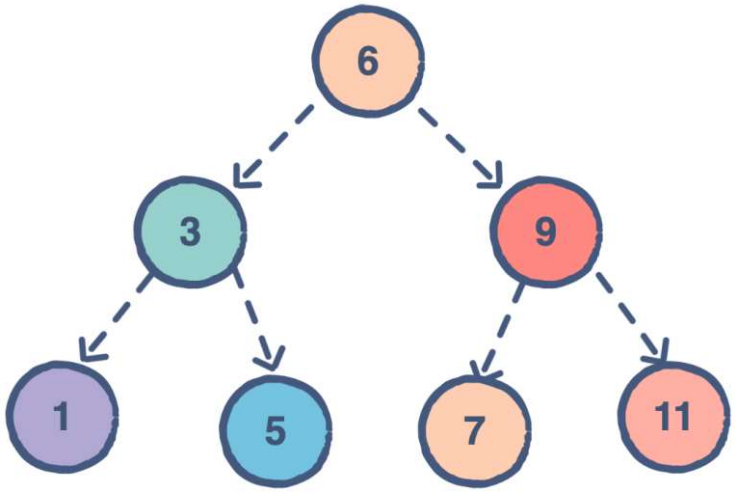
A type error without Val tag:
Val a and b should be the same type

Lists as Variants 2

```
(*accessors*)
let car c =
    let remove_tag = function Val a -> a | _ -> assert false in
    match c with
    | Fun f -> f true |> remove_tag
    | _ -> assert false

let cdr = function
    | Fun f -> f false
    | _ -> assert false

(*the same max function can be used using
  the alternative representation of a list*)
```

Example: Binary Search Tree

```
type 'a tree = Empty
             | Node of 'a * 'a tree * 'a tree;;

(*constructors *)
let empty () =
    Empty

let node e l r =
    Node (e, l, r)

(*accessors*)
let element = function
    | Node (e, l, r) -> e
    | _ -> assert false

let left = function
    | Node (e, l, r) -> l
    | _ -> assert false

let right = function
    | Node (e, l, r) -> r
    | _ -> assert false
```



Example: Binary Search Tree

```
(*add to BST*)
let rec add e n =
    if n = empty () then
        node e (empty ()) (empty ()) (*add e to a leaf node*)
    else
        let e' = element n in
        let l  = left n in
        let r  = right n in
        if e <= e' then
            node e' (add e l) r      (*add e to the left subtree*)
        else
            node e' l (add e r)      (*add e to the right subtree*)
```

Example: Binary Search Tree

```
(*inorder tree traversal*)
let rec inorder f n =
    if n = empty () then
        ()
    else begin
        inorder f (left n);          (*visit left subtree*)
        f (element n);               (*apply f to the element*)
        inorder f (right n)          (*visit right subtree*)
    end
```

Without begin and end, this semicolon will divide (if..then..else..) and f (..)..

Example: Binary Search Tree

```
let t = empty () |> add 5 |> add 3 |> add 7 |> add 6
                 |> add 9 |> add 1 |> add 2 |> add 4
                 |> add 8 |> add 0

let _= inorder (fun x -> Printf.printf "%d, " x) t
```

Tiny Programming Language

- Tiny: a toy programming language similar to Lisp or Scheme
  - Literal expressions
    - `1, 2, 3, …, true, false, …`
  - Conditional expression
    - `(if (>= x y) x y), …`
  - Function application (prefix expr)
    - `(+ 1 2), (and true false), (foo 1 2), …`
  - Function definition
    - `(lambda (x) (+ x 1)),`
      `(lambda (x y) (+ x y)), …`

Tiny Programming Language

- S-expression
  - *atom* is an S-expression
  - (*x* . *y*) is an S-expression, where *x* and *y* are S-expressions
  - E.g.
    - 1, (1 . 2), (1 . (2 . (3 . NIL))), …

- List: (*x y*) stands for (*x* . (*y* . *NIL*))
  - (1 2 3) means (1 . (2 . (3 . NIL)))

  - Tiny (Lisp, Scheme) uses S-expression for its syntax
    - A Tiny (Lisp, Scheme) program is a list

Tiny: Examples

```
//max
(  (lambda (x y)
        (if (>= x y) x y))
    1 2)

//sum
(   (lambda (f) (f f)) //no let rec: how to make a recursion?
    (lambda (self x)
        (if (= x 0)
            0
            (+  x
                (self self (- x 1)))))
    10)

//gcd
(   (lambda (f) (f f))
    (lambda (self x y)
        (if (= x y)
            x
            (if (>= x y)
                (self self (- x y) y)
                (self self (- y x) x))))
    14 21)
```

Tiny: expr type for a parse tree

```
type expr
    = NUM  of int    (*number*)
    | BOOL of bool   (*Boolean*)
    | VAR  of string (*variable*)

    (*arithmetic exprs*)
    | ADD of expr * expr | SUB of expr * expr

    (*comparators*)
    | EQ of expr * expr  | GE of expr * expr

    (*logical exprs*)
    | AND of expr * expr | OR of expr * expr | NOT of expr

    (*conditional expr*)
    | IF  of expr * expr * expr

    (*function definition: parameter, body*)
    | FUN of string * expr

    (*closure: parameter, body, environment*)
    | CLO of string * expr * (string * expr) list

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

Tiny: parsing and evaluation

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

let max = parse
            "( (lambda (x y)\
                    (if (>= x y) x y))\
                1 2)"


print max (*parse tree*)
APP(APP(FUN(x, FUN(y,
                  IF(GE(VAR(x), VAR(y)),
                     VAR(x),
                     VAR(y)))),
        NUM(1)),
    NUM(2))


eval max [] |> print (*eval result*)
NUM(2)
```

Tiny: parsing and evaluation

```
let sum = parse
            "(  (lambda (f) (f f))\ (*recursion for lambda*)
                (lambda (self x)\
                    (if (= x 0)\
                        0\
                        (+  x\
                            (self self (- x 1)))))\
                10)"

print sum (*parse tree*)
APP(APP(FUN(f,
            APP(VAR(f), VAR(f))),
        FUN(self, FUN(x,
                      IF(EQ(VAR(x), NUM(0))
                         NUM(0),
                         ADD(VAR(x),
                             APP(APP(VAR(self), VAR(self)),
                                 SUB(VAR(x), NUM(1))))))),
    NUM(10))

eval sum [] |> print (*eval result*)
NUM(55)
```

Tiny: eval

- eval expr env
  - eval function evaluates expr in env
- Evaluation of literals
  - The value of the literals itself

```
let rec eval expr env =
…

    match expr with
    | BOOL b -> BOOL b
    | NUM n  -> NUM n
…
```

Tiny: eval

- Evaluation of variables
  - Lookup the variable from the environment
  - env is a list of name-value bindings

```
let rec eval expr env =
…
    (*look up the value of a variable from an environment*)
    let rec lookup name env =
        match env with
        | [] -> print (VAR name); assert false
        | (n, e)::rest -> if name = n
                          then e
                          else lookup name rest in

    match expr with
…
    | VAR v  -> lookup v env
…
```

Tiny: eval

- Evaluation of primitive function applications
  - Evaluate the operand expressions
  - Apply the primitive operator to the operands

```
let dropNUM  = function NUM  n -> n | assert false in
let dropBOOL = function BOOL b -> b | assert false in
let dropCLO  = function CLO (v, e, ev) -> (v, e, ev) | assert false in
…
| ADD (a, b)    -> eval a env |> fun x ->
                   eval b env |> fun y ->
                   NUM (dropNUM x + dropNUM y)
| EQ (a, b)     -> eval a env |> fun x ->
                   eval b env |> fun y ->
                   BOOL (dropNUM x = dropNUM y)
| AND (a, b)    -> eval a env |> fun x ->
                   eval b env |> fun y ->
                   BOOL (dropBOOL x && dropBOOL y)
```

Tiny: eval

- Evaluation of conditional expression
  - Evaluate the condition expression
  - Depending on the value of the condition, either evaluate t-expression or f-expression

```
…
    | IF (c, t, f) -> eval c env |> fun x ->
                       if dropBOOL x then eval t env
                                     else eval f env
…
```

Tiny: eval

- Evaluation of function definitions
  - The result is a closure, with
    - The formal parameter name
    - The body of the function
    - The environment at the moment of the evaluation

```
…
    (*closure remembers the env. when the fun. def. is evaluated*)
    | FUN (v, e)    -> CLO (v, e, env)

    (*evaluate the body in the extended env. of closure*)
    | APP (f, a)    -> eval f env  |> fun clo ->
                       eval a env  |> fun x ->
                       dropCLO clo |> fun (v, e, ev) ->
                       eval e ((v, x)::ev)
…
```

Tiny: Closure

- Why closures have an environment
  - Static scoping vs dynamic scoping

```
let add = fun x -> fun y -> x + y
=> eval fun x -> fun y -> x + y, []
=> add:  CLO (x, fun y -> x + y, [])
```

```
let inc = add 1
=> eval fun y -> x + y, [(x, 1)]
=> inc:  CLO (y, x + y, [(x, 1)])
```

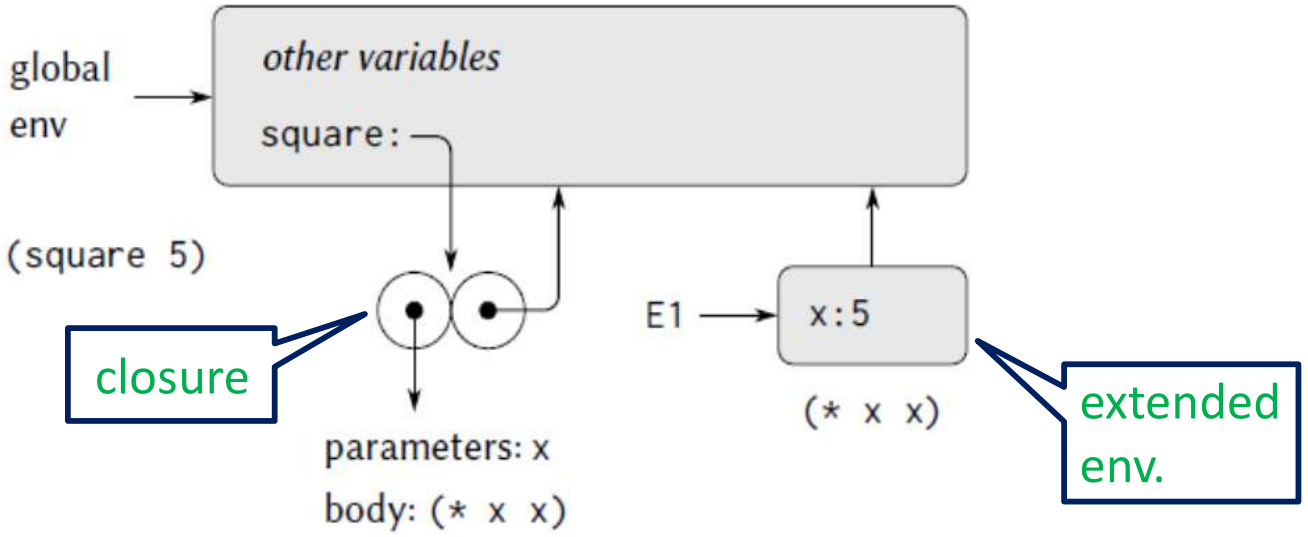
```
let foo = fun f -> fun x -> f x
=> eval fun f -> fun x -> f x  []
=> foo:  CLO (f, fun x -> f x, [])
```

```
let bar = foo inc
=> eval fun x -> f x  [(f, inc)]
=> bar:  CLO (x, f x, [(f, inc)])
```

```
let _ = bar 2
=> eval f x [(x, 2); (f, inc)]
=> f: CLO (y, x + y, [(x, 1)])
   x: 2 (*this x and x in CLO of f are
          different variables*)
   eval x + y [(y, 2); (x, 1)]
=> 3
```

Tiny: Environment

- Environment
  - A name to value map
- When evaluating a function definition
  - A closure is created with the environment of the moment

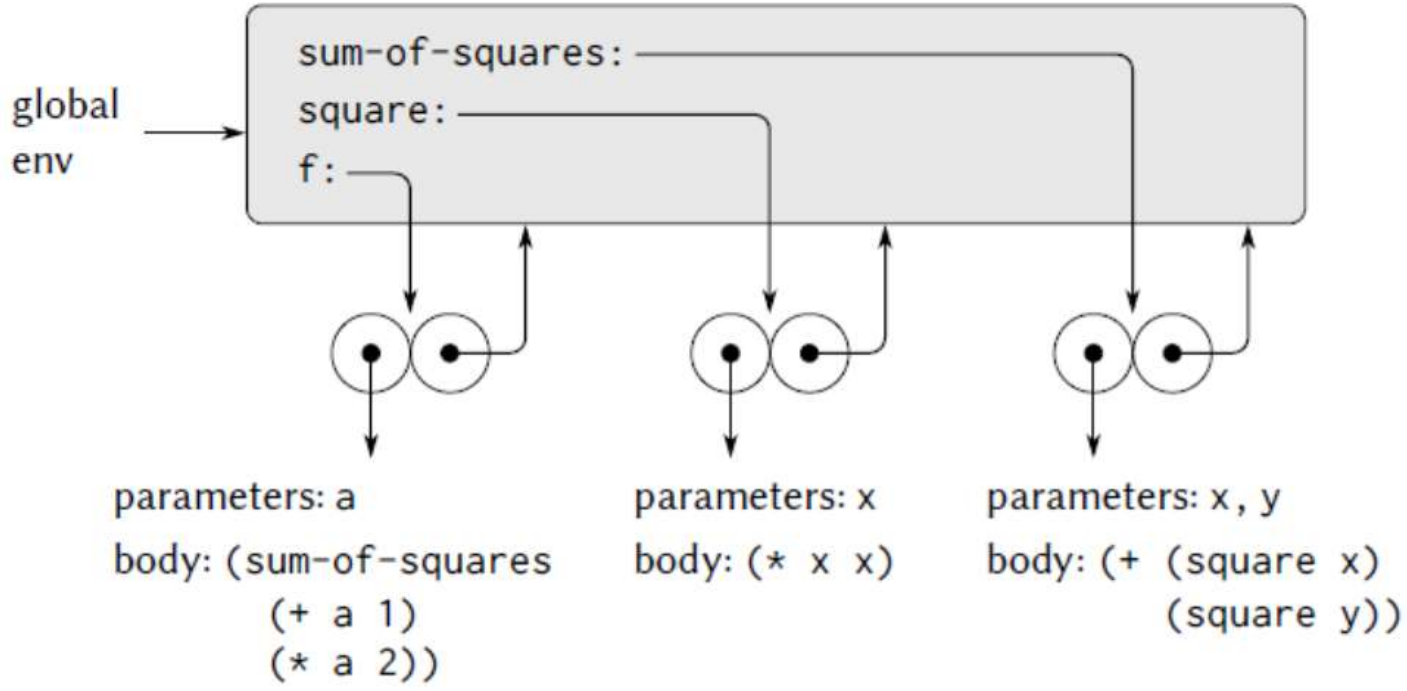



```
let square = fun x -> x * x;;
```

```
square 5;;
```

Tiny: Environment

```
let square
    = fun x -> x * x;;

let sum_of_squares
    = fun x y -> square x + square y;;

let f
    = fun a -> sum_of_squares (a+1) (a*2);;
```



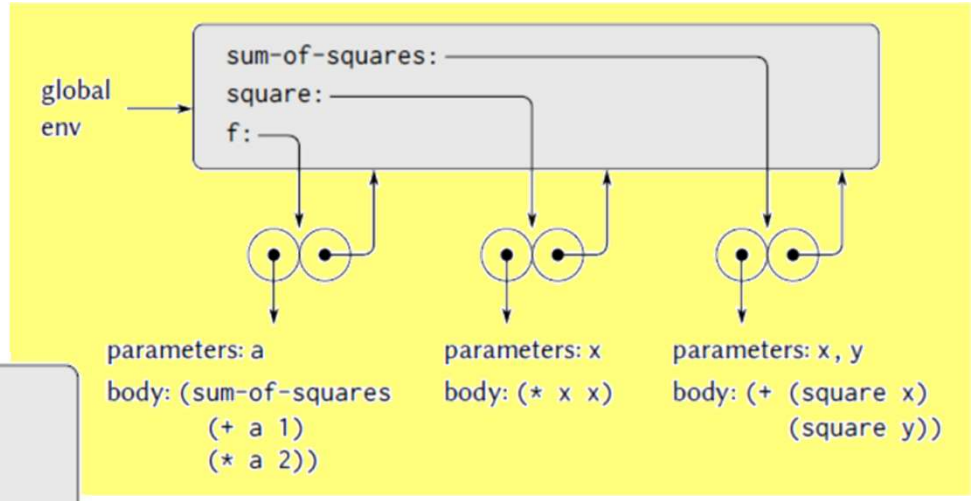
Tiny: eval

- Evaluation of function applications
  - Evaluate the function and the actual parameter expr
  - Extend the env of closure with the binding of
    - The formal parameter name and the actual parameter value
  - Evaluate the body expr in the extended environment

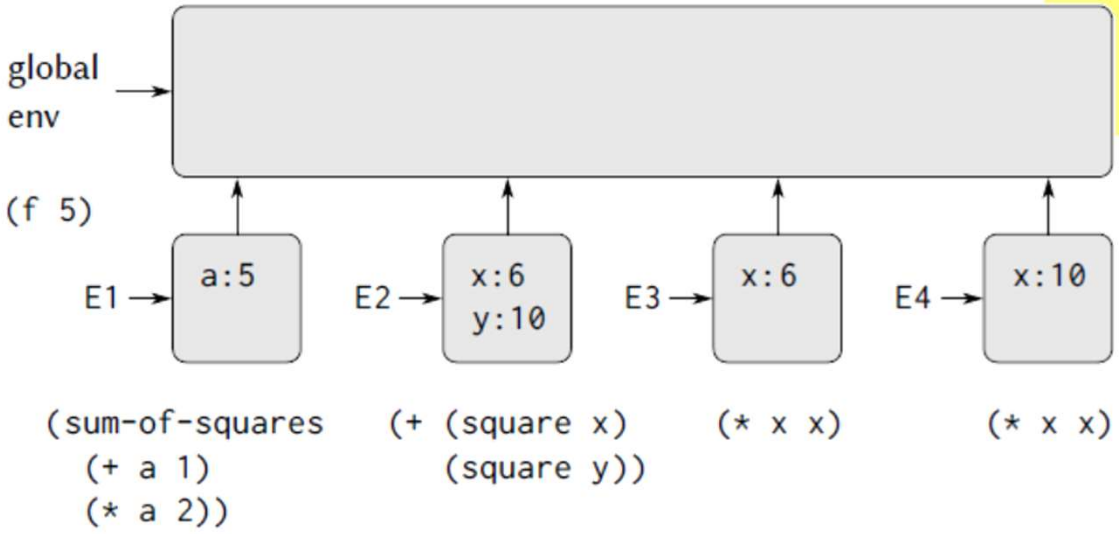
```
(*closure remembers the env. when the fun. def. is evaluated*)
| FUN (v, e)    -> CLO (v, e, env)

(*evaluate the body in the extended env. of closure*)
| APP (f, a)    -> eval f env  |> fun clo ->
                   eval a env  |> fun x ->
                   dropCLO clo |> fun (v, e, ev) ->
                   eval e ((v, x)::ev)
```

Tiny: Environment

- When applying a function to a parameter
  - Environment is extended with the binding of
    - formal parameter name and actual parameter value




```
f 5;;
```



Optional: Polymorphic Variants

- Polymorphic variants
  - Do not need explicit type declaration
  - Tags begin with backtick (`)
  - The types are inferred automatically

```
# let int_or_str b = if b then `Int 3
                           else `Str "hello";;
val int_or_str : bool -> [> `Int of int | `Str of string ] = <fun>

# int_or_str true;;
- : [> `Int of int | `Str of string ] = `Int 3

# int_or_str false;;
- : [> `Int of int | `Str of string ] = `Str "hello"
```

BST with Polymorphic Variants

```
(*Polymorphic variants*)

(*constructors*)
let empty () =
    `Empty

let node e l r =
    `Fun (function
              | "e" -> `Val e
              | "l" -> l
              | "r" -> r
              | _ -> assert false)
```

Wrap e with `Val to fix type error

BST with Polymorphic Variants

```
(*accessors*)
let element = fun n ->
    let remove_tag = function `Val e -> e | _ -> assert false in
    match n with
    | `Fun f -> f "e" |> remove_tag
    | _ -> assert false


let left = function
    | `Fun f -> f "l"
    | _ -> assert false


let right = function
    | `Fun f -> f "r"
    | _ -> assert false
```