CSE216 Programming Abstractions

Simple Programming Language (SPL)

YoungMin Kwon

SPL: Syntax

```
type expr = B of bool   (*Boolean*)
          | N of int    (*number*)
          | V of string (*variable*)
          (*arithmetic exprs*)
          | Add of expr * expr | Sub of expr * expr
          (*predicates*)
          | Equ of expr * expr | Leq 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, environment, body*)
          | Clo of string * (string * expr) list * expr
          (*function application: operator, operand*)
          | App of expr * expr
          (*TODO add a definition for let binding:
            let variable = expr in expr*)
```

SPL: Evaluation

```
let rec eval expr env =
    let getNum  = function N n -> n | e -> print e; assert false in
    let getBool = function B b -> b | e -> print e; assert false in
    let getClo  = function Clo (v, ev, e) -> (v, ev, e)
                         | e -> print e; assert false in

    match expr with
    | B b -> B b
    | N n -> N n
    | V v -> (*TODO: look up v from env*)

    | Add (a, b)   -> eval a env |> fun x ->
                      eval b env |> fun y ->
                      N (getNum x + getNum y)
    | Sub (a, b)   -> (*TODO*)

    | Equ (a, b)   -> eval a env |> fun x ->
                      eval b env |> fun y ->
                      B (getNum x = getNum y)
    | Leq (a, b)   -> (*TODO*)
```

SPL: Evaluation

```
| And (a, b)    -> eval a env |> fun x ->
                   eval b env |> fun y ->
                   B (getBool x && getBool y)
| Or (a, b)     -> (*TODO*)
| Not a         -> eval a env |> fun x ->
                   B (not (getBool x))

| If (c, t, f) -> (*TODO: eval c, if true eval t; else eval f*)

| Fun (v, e)    -> (*TODO: make a closure with v, env, and e*)
| App (f, a)    -> (*TODO: eval the body of f in the extended env
                           with a binding of param of f and a*)

(*TODO: add a case for Let*)
| _ -> assert false
```

SPL: Evaluation

```
let test () =
    let sum = Fun ("self", Fun ("x",
                    Let ("sum", App (V "self", V "self"),
                    Let ("dec", Fun ("x", Sub (V "x", N 1)),
                    If (Equ (V "x", N 0),
                        N 0,
                        Add (V "x",
                             App (V "sum",
                                  App (V "dec", V "x")))))))) in

    print (eval (App (App (sum, sum), N 10)) [])

let _ = test ()
```