CSCI-GA-2110– Problem Set 3– Written Part (Solutions)

ARJUN PARASURAM PRASAD (ap9334)

1

```
(define (eval-env (env : Env) (sto : Store) (e : Expr)) : Result
    (type-case Expr e
        ...
        [ifC (guard e1 e2)
            (let* (
                [rguard (eval-env env sto guard)]
                [re1 (eval-env env (res-s rguard) e1)]
                [re2 (eval-env env (res-s rguard) e2)])
                (cond
                    [(equal? (res-v rguard) (boolV #true)) re1]
                    [(equal? (res-v rguard) (boolV #false)) re2]
                    [else (error 'eval-env "ifC guard was not a
boolean")]))] ...))
```

- The above implementation of ifC is INCORRECT as only e1 should update the store if the guard is true and e2 should only update the store if the guard is false. In this implementation of ifC both e1 and e2 update the store which is wrong.
- The CORRECT implementation of ifC is as follows:

  ```
  [ifC (guard e1 e2)
               (type-case Result (eval-env env sto guard)
                 [res (v1 sto1)
                      (cond
                        [(equal? #t (boolV-b v1))
                                 (eval-env env sto1 e1)]
                        [else
                         (eval-env env sto1 e2)])])]
  ```

  as you can see only 1 of e1 or e2 is evaluated with sto1 to update the store.

please turn over

2

```
(define (myfun b1 b2)
    (begin
        (set-box! b1 1)
        (set-box! b2 2)
        (+ (unbox b1) (unbox b2))))
```

- `myfun` will return 4 if both the parameters `b1` and `b2` point to the same location / are the same box object, as the update `(set-box! b2 2)` will update both the parameters to 2 (as they both reference the same location).
- E.g.

  ```
  (begin
      (define b (box 0))
      (myfun b b)
  )
  ```

  shall return 4.

3

```
(let(
        [fact (lambda (n) (if (equal? 0 n) 1 (* n (fact (+ n-1)))))]
    )
    (fact 5))
```

- In lexical scope, `fact` will call itself recursively and will try to find `fact` inside its own definition. But `fact` was not defined there - it was defined in `let` block outside. Therefore we get `unbound variable error`.
- In dynamic scoping, we look for `fact` in the current calling environment. Therefore when `fact` is called within its let binding definition, the recursive call can find the `fact` in the environment.
- Result: Dynamic scoping will allow the above segment of code to compute `5!` (i.e. `120`) successfully.