Theoretical questions Assignment 3:

Eden Abadi

Noam Weiler

Part 1:

1. For each “Special form” a special evaluation rule exists, while the evaluation of “Primitive operators” is built in the interpreter and not explained by the semantic of the language.
   Examples from the interpreter code:

   While the expression of a primitive operator stays the same, for the special form if we have a specific evaluation function.

```
const L3applicativeEval = (exp: CExp | Error, env: Env): Value | Error =>
    isError(exp)  ? exp :
    isNumExp(exp) ? exp.val :
    isBoolExp(exp) ? exp.val :
    isStrExp(exp) ? exp.val :
    isPrimOp(exp) ? exp :
    isVarRef(exp) ? applyEnv(env, exp.var) :
    isLitExp(exp) ? exp.val :
    isIfExp(exp) ? evalIf(exp, env) :
    isProcExp(exp) ? evalProc(exp, env) :
    isAppExp(exp) ? L3applyProcedure(L3applicativeEval(exp.rator, env),
                                     map((rand) => L3applicativeEval(rand, env),
                                         exp.rands),
                                     env) :
    Error(`Bad L3 AST ${exp}`);

export const isTrueValue = (x: Value | Error): boolean | Error =>
    isError(x) ? x :
    ! (x === false);

const evalIf = (exp: IfExp, env: Env): Value | Error => {
    const test = L3applicativeEval(exp.test, env);
    return isError(test) ? test :
        isTrueValue(test) ? L3applicativeEval(exp.then, env) :
        L3applicativeEval(exp.alt, env);
};
```

2. Assumes or-exp is <curr_exp, or_tail>
   A. Eval(<or-exp>,env) =>
   if len exp is empty
   Return False

   else

   Let curr_exp:Value = eval(exp.curr_exp)

   If curr_exp is considered a true value

   Return True

   Else

Return eval(exp.or_tail,env)

B. Eval(<or-exp>,env) =>
If exp is empty
Return false
else

Return eval(curr_exp,env) || eval(or_tail,env)

3. We would prefer VarRef and that's because PrimOp is defined inside the interpreter while we can easily change the implementation of VarRef as we please defined in the GE – we can change it and create it by defining new binding to the primitive operation.
4. We would like to switch from applicative order to normal order when the calculation is very long and not needed at some of the evaluation process, normal evaluation is done in a "lazy" while – computed only when needed and therefore some times more efficient, also in some cases applicative eval can fail/go into an infinite loop, while normal eval wont - For example:

```
1. (define test
2.   (lambda (x y)
3.     (if (= x 0)
4.         0
5.         y)))
6.
7. (define zero-div
8.   (lambda (n)
9.     (/ n 0))) ; division by zero!
10.
11. (test 0 (zero-div 5))
```

In this example normal eval will return 0 because it hasn't computed y at this point, while applicative eval will throw an exception of division by zero while evaluating y.

5. There some cases when the first evaluation can save us some time later - this can happen when this specific evaluation happens several times in the code, for example:
(define long_calc
(lambda (x)
(if (> 5 x)
(display x)
(display (+ 5 x)))))

(define square (lambda(x) (*x x)))

(long_calc (squre 3))

In this example 9 will be calculated 3 times instead of 1 time at first.

6. The environment model is an optimization of the substitution applicative model of the operational semantics. It changes the way we map variables to their values. Instead of eagerly substituting variables by their values when we apply a closure, we leave the body of the closure untouched, and maintain an environment data structure on the

side. The main problem of the substitution approach is that substitution requires repeated analysis of procedure bodies. In every application, the entire procedure body is repeatedly renamed, substituted and reduced. These operations on ASTs actually **copy** the structure of the whole AST - leading to extensive memory allocation / garbage collection when dealing with large programs. Therefore, environment model might save us computation time and memory.
For example:
(define no_op (lambda (x) (let (x lambda(x) (let (lambda(x) …)) and so on.

7. Equivalent example:
   (define impl_display (lambda(x) (display x)))

   Non- Equivalent example:

```
1. (define loop (lambda (x) (loop x)))
2.
3. (define g (lambda (x y) y))
4.
5. (g (loop 0) 7)
```

   normal-eval will return 7 while applicative-eval will get into infinite loop.

8. Since we are delaying the evaluation to the point where we need the value, the arguments are received as cexp. One a cexp is evaluated, it is used, so there is no need to turn it into a cexp.
9. Since we don't perform substitution, the value will not be placed in the AST. when we come across a var-ref we will have the variable with its matching value in the environment ready for use.
10. We learned in class two ways of evaluating let expression – the first one is syntactic abbreviation and the other one is special form evaluation such as if expression for example. In the first one we create a closure and replace the let with the matching lambda definition it creates, in the second one we evaluate by its special form.

```
11.  // LET: Direct evaluation rule without syntax expansion
12.  // compute the values, extend the env, eval the body.
13.  const evalLet4 = (exp: LetExp4, env: Env): Value4 | Error => {
14.      const vals = map((v) => L4applicativeEval(v, env), map((b) =>
   b.val, exp.bindings));
15.      const vars = map((b) => b.var.var, exp.bindings);
16.      if (hasNoError(vals)) {
17.          return evalExps(exp.body, makeExtEnv(vars, vals, env));
18.      } else {
19.          return Error(getErrorMessages(vals));
20.      }
21.  }
```