17/05/18

Liav Rabi, id: 205688328
Benny Peresetsky, id: 310727771

PPL – Assignment 3 – Part 1

1. There are few differences between primitive operator to special form and those are:
   a. Special form can be nested while primitive operator cannot.

```
;;          | <varRef>                          / VarRef(var:string)
;;          | ( lambda ( <var>* ) <cexp>+ )     / ProcExp(params:VarDecl[], body:CExp[]))
;;          | ( if <cexp> <cexp> <cexp> )       / IfExp(test: CExp, then: CExp, alt: CExp)
;;          | ( let ( binding* ) <cexp>+ )      / LetExp(bindings:Binding[], body:CExp[]))
;;          | ( quote <sexp> )                  / LitExp(val:SExp)

; <prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
;                | cons | car | cdr | list? | number?
;                | boolean? | symbol? | string?      ##### L3
```

   We can see that primitive operators are atomic and the special form expressions there are sub expression of the type CExp – that can be nested.
   b. For each special form there is a special evaluation rule while all primitive operators have the same evaluation rule.

```
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) :
```

   c. primitive operator evaluation is built in in the interpreter.
   d. Special form is a compound expression while primitive operators are atomic expressions (same as code fragment in a).

2. <u>shortcut semantics</u>:

```
evalOR(<OR-exp exp>,env) =>
  let test:Value = eval(first(exp))
  if test is considered a true value
    return test
  else
    isEmpty(rest(exp)) ? return test:
    evalOR(<OR-exp rest(exp)>, env)
```

   <u>non-shortcut semantics:</u>

```
evalOR(<OR-exp exp>,env) =>
  let test:Value = reduce((a, b) => a || b, false, exp.map(eval))
  return test
```

3. We prefer the representation of "VarRef" because it is much easier to add a primitive to the interpreter(we only need to add a proper binding in the initialization of the GE), while in the representation of "PrimOp" we need to add expression type of "PrimOp" which makes the language more complex – if there are changes we need to apply to the language, In "VarRef" representation will be apply single change while in "PrimOp" we will apply several changes.

4. Applicative order implements an eager approach in evaluation: arguments are evaluated immediately, before the closure is reduced – therefore it can evaluate expression which is not needed or being used in the program.

```
(define test
 (lambda (x y)
  (if (= x 0)
     0
     y)))

(define zero-div
 (lambda (n)
  (/ n 0))) ; division by zero!

(test 0 (zero-div 5))
```

We can notice in the above example that while in normal order (zero-div 5) will not be evaluated and therefore we will not get an exception while in applicative order we will.

5. Normal order implements a lazy approach in evaluation: it avoids evaluating arguments until the last moment it is necessary – therefore it can evaluate the same expression more than once. That is the reason why applicative is more efficient.

```
(define square (lambda (x) (* x x)))
(define sum-of-squares (lambda (x y) (+ (square x) (square y))))
(define f (lambda (a) (sum-of-squares (+ a 1) (* a 2)))
(f 5)
```

In normal order the expression of (* 5 2) evaluated twice while in applicative order it would be evaluated once.

6. There are several reasons for switching from substitution model to the environment model and those are:
   a. Substitution requires repeated analysis of procedure bodies
   b. The operations on the ASTs copy the structure of the whole AST – extensive memory allocation \ garbage collection when dealing with large programs.
   c. The substitution interpreter is slow that is barely usable.

   This is an applyClosure function in L4 (environment model):

```
const applyClosure4 = (proc: Closure4, args: Value4[]): Value4 | Error => {
    let vars = map((v: VarDecl) => v.var, proc.params);
    return evalExps(proc.body, makeExtEnv(vars, args, proc.env));
}
```

   This is an applyClosure function in L3 (substitution model):

```
const applyClosure = (proc: Closure, args: Value[], env: Env): Value | Error => {
    let vars = map((v: VarDecl) => v.var, proc.params);
    let body = renameExps(proc.body);
    let litArgs = map(valueToLitExp, args);
    return evalExps(substitute(body, vars, litArgs), env);
```

7. <u>Equivalent example:</u>

```
(define square (lambda (x) (* x x)))
(define sum-of-squares (lambda (x y) (+ (square x) (square y))))
(define f (lambda (a) (sum-of-squares (+ a 1) (* a 2)))
(f 5)
```

In both normal and applicative order, the program will return 136.

Non-equivalent example:

```
(L3
    (define try
          (lambda (a b)
               (if (= a 0)
                    1
                    b)))
     (try 0 (/ 1 0)))
```

In normal order, this program returns 1. In applicative order, it throws a divide by 0 exception.

8. In normal order we do not need to turn the values back into expression before we apply the substitution in the body, since the rands are passed as non-evaluated expressions. Therefore, there is no need of the valueToLitExp procedure.

9. In environment model expressions are evaluated with respect to an environment.
   The evaluation rule for procedure application is modified, so to replace substitution (and renaming) by environment creation.
   The environment model is also lazy approach, therefore it will substitute the vars only when their values is needed.

10. The evaluation of let expression does involve creation of a closure at all models we encountered in L3 (applicative order and normal order).
    As we have been shown in class, In L3, let expression is a syntactic abbreviation of proc expression (lambda) and when a proc expression is evaluated, the makeClosure function is applied.
    Sample from L3 applicative – interpreter:

```
isProcExp(exp) ? evalProc(exp, env) :

const evalProc = (exp: ProcExp, env: Env): Value =>
    makeClosure(exp.args, exp.body);
```

Sample from L3 normal – interpreter:

```
    isProcExp(exp) ? makeClosure(exp.args, exp.body) :
    // This is the difference between applicative-eval and normal-eval
    // Substitute the arguments into the body without evaluating them first.
```

In L4, there are 2 options to evaluate a let expression. The first is the same as in L3, a syntactic abbreviation of a proc expression. The second is evaluating the expression with a new special evaluation rule to let expression.

Sample from L4 – interpreter (syntactic abbreviation):

```
const evalProc4 = (exp: ProcExp4, env: Env): Closure4 =>
    makeClosure4(exp.args, exp.body, env);
```

Sample from L4 – interpreter (special evaluation rule):

```
const evalLet4 = (exp: LetExp4, env: Env): Value4 | Error => {
    const vals = map((v) => L4applicativeEval(v, env), map((b) => b.val, exp.bindings));
    const vars = map((b) => b.var.var, exp.bindings);
    if (hasNoError(vals)) {
        return evalExps(exp.body, makeExtEnv(vars, vals, env));
    } else {
        return Error(getErrorMessages(vals));
    }
}
```