Assignment 3

Dor Lebel 205872807 Nitzan Hochman 316264845

Part 1

1.1

"Let" is indeed a special form in L3 because it requires special evaluation rules, since the vars at the beginning of the expression are supposed to act as local variables for the body of the expression, and only then. (plus it is listed in the isSpecialForm?)

1.2

Semantic error 1 : ifExp with more then 3 CExp, e.g:

If ((> 3 4) 3 4 5)

Semantic error 2 : program without L3 at the beginning, e.g:

((* 3 3)) instead of (L3 (* 3 3))

Semantic error 3 : define var is not an identifier, e.g:

Define 3 3

Semantic error 4 : AppExp with undefined operator, e.g:

(L3 (square 3))

1.3.1

<program>::=(L3<exp>+)	//Program(exps:List(Exp))
<exp>::=<define>|<cexp>	/DefExp|CExp
<define>::=(define<var><cexp>)	/DefExp(var:VarDecl,val:CExp)
<var>::=<identifier>	/VarRef(var:string)
<cexp>::=<num-exp>	/NumExp(val:number)
| <bool-exp>	/BoolExp(val:boolean)
| <str-exp>	/StrExp(val:string)
| (lambda(<var>*)<cexp>+)	/ProcExp(args: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)
| (<cexp><cexp>*)	/AppExp(operator:CExp,operands:CExp[]))
| <value>	**/Value**
<binding>::=(<var><cexp>)	/Binding(var:VarDecl,val:Cexp)

<prim-op>::=+|
|*|/|<|>|=|not|and|or|eq?|string=?|
cons|car|cdr|pair?|number?|list|boolean?|symbol?|string?
#####L3
<num-exp>::=anumbertoken
<bool-exp>::=#t|#f
<str-exp> ::= a sequence of characters between double quotes
<var-ref>::=anidentifiertoken
<var-decl>::=anidentifiertoken
<sexp>::=symbol|number|bool|string|(<sexp>+.<sexp>)|(<sexp>*)

<value>:= sexp | closure **/SExp | Closure** #####L3

1.3.2

The interpreter checks if the CExp is of type value and if so returns the value v that the type calls (whether it's a string, bool…).

1.3.3

On the one side, it seems as though the changing type method is more efficient (because it uses less structures in-between interpretations) , but on the other hand, it incorporates a type into the ast which can produce incorrect structures.

1.4

Because every CExp is evaluated only once and only when it is needed for the evaluation of the return value.

1.5

Applicative faster:

((lambda (x) (* x x))(+ 3 5))

In this case, in applicative order the expression (+ 3 5) it will be evaluated only once, but in normal order it will be evaluated twice.

Normal faster:

((* 4 5)(- 4 8)(* 4 4) 1)

In applicative order all of the expressions will be evaluated, but in normal eval, none of them will (since only 1 is the returned).

Part 3

3.1

In the first example,

```
#lang lazy

(define x (-))

x
```

The output is <promise:x>. That's because the variable x is yet to be evaluated since the evaluation strategy is normal and so the define expression will only be evaluated when it is required.

In the second example,

```
#lang lazy

(define x (-))

1
```

The output is 1 because x never needs to be evaluated since scheme doesn't need any other calculations in order to return the value of 1.

The problem when handling the `define` expression stems from the fact that the program evaluates the value of x (the variable of the define expression) when it encounters it, although according to normal evaluation, the evaluation should only happen when x is actually needed in order to evaluate the return statement. Therefore, p3 should return the value 1 with no error.

The solution should be postponing the evaluation of the value in the define expression until it is needed only.

3.2

The output is incorrect because we are sending the define expression's value to be evaluated in L3-normal which uses the function evalDefineExp that returns the evaluated value of val instead of only binding it to the define expression's var (as normal evaluation should do).