Assignment 3

Drew Bonde

1. *[30 points]*

A. The grammar below specifies the *concrete syntax* of the LET language, as discussed in the class. The grammar includes an expression of the form `-(Expression, Expression)` that *subtracts* its two component `Expressions`.

```
Program     ::== Expression
Expression  ::== Number
Expression  ::== -(Expression , Expression)
Expression  ::== zero? (Expression)
Expression  ::== if Expression then Expression else Expression
Expression  ::== Identifier
Expression  ::== let Identifier = Expression in Expression
```

Add a new expression to the concrete syntax that ***adds*** its two component `Expressions`.

> `Expression ::== +(Expression , Expression)`

B. The figure below, *below the concrete syntax lines*, specifies the nodes in the abstract syntax tree for the LET language, as discussed in the class. For example, the node for the expression `-(Expression , Expression)` is `diff-exp(exp1 exp2)`

```
Program     ::== Expression
                 a-program (exp1)
Expression  ::== Number
                 const-exp (num)
Expression  ::== -(Expression , Expression)
                 diff-exp (exp1 exp2)
Expression  ::== zero? (Expression)
                 zero?-exp (exp1)
Expression  ::== if Expression then Expression else Expression
                 if-exp (exp1 exp2 exp3)
Expression  ::== Identifier
                 var-exp (var)
Expression  ::== let Identifier = Expression in Expression
                 let-exp (var exp1 body)
```

Add a node called `add-exp` for the add expression that you added to the concrete syntax in part (A).

> `add-exp (exp1 exp2)`

C. Write the *specification* (not implementation) of the *value-of function* for the node `add-exp` that you added to the abstract syntax in part (B). *Hint*: Use an inference rule, as discussed in the class, to write the specification.

```
      (value-of exp1 p) = val1 (value-of exp2 p) = val2
-------------------------------------------------------------
      (value-of (add-exp exp1 exp2) p)
        =  +(expval->num val1) (expval->num val2)
```

---

2. *[30 points]* Consider the following code in the LET language, discussed in the class. Line numbers are not part of the code.

```
Line 1: let x = 2 in
Line 2:   let y = 3 in
Line 3:     let x = x+y
Line 4:       x+y
```

A. True/False: Is this code valid in LET?

True.

B. If the code is valid, what are the values of `x` and `y` at each of lines 1-4?

1. x = 2
2. x = 2, y = 3
3. x = 2, y = 3
4. x = 5, y = 3

C. If the code is valid, what does the code evaluate to? Why? Please explain.

The code evaluates to 8. Up to line 3, `x` and `y` are equal to 2 and 3 respectively. However, line 3 indicates that `x` is equal to `x` + `y` in line 4, hence `x` is equal to 5 in line 4. Line 4 being just `x` + `y`, this will evaluate to 5.

---

3. *[40 points]*

A. Change `let Identifier = Expression in Expression` in the *concrete syntax* of LET language, as given in question 1.A, such that instead of one identifier and its value, now the expression can take *two identifiers with their values*.

```
Expression ::== let Identifier , Identifier = Expression , Expression in Expression
```

B. Change `let-exp (var exp1 body)` in the abstract syntax of LET, as given in question 1.B, to accomodate the new let expression in the concrete syntax in part (A).

```
let-exp (var1 var2 exp1 exp2 body)
```

C. Write the specification of value-of for the new `let-exp` in part (B).

```
    (value-of exp1 p) = val1 , val2
---------------------------------------
(value-of (let-exp var1 var2 exp1 exp2 body) p)
   = (value-of body [var1=val1][var2=val2]p)
```