Compilers and Interpreters Take Home Test SIMPLE Research Paper

1. Introduction

The main purpose of this document is to outline the steps I took to modify and create an interpreter that would create an Abstract Syntax Tree from a stream of tokens parsed from the input in order to directly execute the code according to the following SIMPLE grammar:

```
Program -> Statement P
P -> Program | ε
Statement -> display Expression St1 | assign id = Expression
           | while Expression do Program end
           | if Expression then Program St2
St1 -> read id | ε
St2 -> end | else Program end
Expression -> Expression relop AddExpr
           | AddExpr
AddExpr -> AddExpr + MultExpr
           | AddExpr – MultExpr
           | MultExpr
MultExpr -> MultExpr * NegExpr
           | MultExpr / NegExpr
           | NegExpr
NegExpr -> -Value
           | Value
Value -> id | number | (Expression)
```

Overall, the interpreter should correctly interpret the SIMPLE language and produce an Abstract Syntax Tree through recursive descent parsing, which utilizes a top-down parsing algorithm to build the tree or hierarchy of classes from the start symbol down. This tree allows the program to be easily evaluated in the later stages.

2. Overall Design

2a. Limitations and Semantic Issues

The main limitation of recursive descent parsing is that it only works on grammars with certain properties. The grammar cannot contain left recursion and should be completely left-factored, so that for each non-terminal, there are no two productions that have a common prefix of symbols on the right side of the production. The fundamental semantic issue that appears in our SIMPLE grammar is that although it has been left factored, it is left recursive for several non-terminals. In Expression, AddExpr, and MultExpr, the non-terminals are still on the right side of the production, so it's possible to go into an infinite recursive loop. This poses a problem for creating a recursive descent parser, but can be easily fixed and will be shown in the later sections of this document.

The following code below shows the left recursion present in our SIMPLE grammar (repeated non-terminals are highlighted):

Here, the non-terminal Expression is still on the right side of the production

```
Expression -> Expression relop AddExpr
            | AddExpr
```

This can also be seen in AddExpr

```
AddExpr -> AddExpr + MultExpr
         | AddExpr – MultExpr
         | MultExpr
```

as well as in MultExpr

```
MultExpr -> MultExpr * NegExpr
          | MultExpr / NegExpr
          | NegExpr
```

2b. Grammar Transformations

I will need to eliminate the left recursion seen in Expression, AddExpr, and MultExpr. The altered grammar will be highlighted in yellow.

1. Expression -> Expression relop AddExpr
        | AddExpr

For the left-recursive non-terminal Expression, one must first discard any rules in the form Expression -> Expression. In the productions, the sequence of non-terminals and terminals "Expression relop AddExpr" and the sequence of terminals "AddExpr" both end in the terminal "AddExpr," so Expression will ultimately always start with "AddExpr". Knowing this fact, expression can now be simplified by adding an additional productions called E' and making Expression only go to "AddExpr E' ". E' will have one production because Expression only had one production containing the non-terminal Expression. This new production should start with the terminal "relop" to break the left recursion. Epsilon must also be a production of E’ in order to end the grammar.

Would be replaced with

Expression -> AddExpr E’

E’ --> **relop** Expression E’

| ε

2. AddExpr -> AddExpr + MultExpr

| AddExpr – MultExpr

| MultExpr

For this left-recursive non-terminal AddExpr, similar to Expression, must have any rules in the form AddExpr -> AddExpr discarded. Though there are more productions present in this section, each production ends with “MultExpr”. We can conclude that AddExpr will ultimately start with “MultExpr” because that is the only production that does not contain non-terminals. Thus we should add another production A’ and make AddExpr only go to “MultExpr” and then “A’ ”, as AddExpr must start with “MultExpr”. Then, with A’, we can use multiple production rules of “- MultExpr A’,” “+ MultExpr A’,” and epsilon to end the grammar.

Would be replaced with

AddExpr -> MultExpr A’

A’ -> - MultExpr A’

| + MultExpr A’

| ε

3. MultExpr -> MultExpr * NegExpr

| MultExpr / NegExpr

| NegExpr

For this left-recursive non-terminal MultExpr, we will use the exact same method we used with AddExpr. In order to discard MultExpr -> MultExpr, we can add a new production M’. Like AddExpr in which each production ended with “MultExpr,” each production in MultExpr ends in “NegExpr”. Additionally, “NegExpr” is the only production without non-terminals, so we can conclude that MultExpr starts with “NegExpr”. Thus, we can alter MultExpr to go to “NegExpr M’ ”. M’ will go to “ * NegExpr M’ ” and “ / NegExpr M’ ” along with epsilon to end the grammar.

Would be replaced with

MultExpr -> NegExpr M’

M’ -> * NegExpr M’

| / NegExpr M’

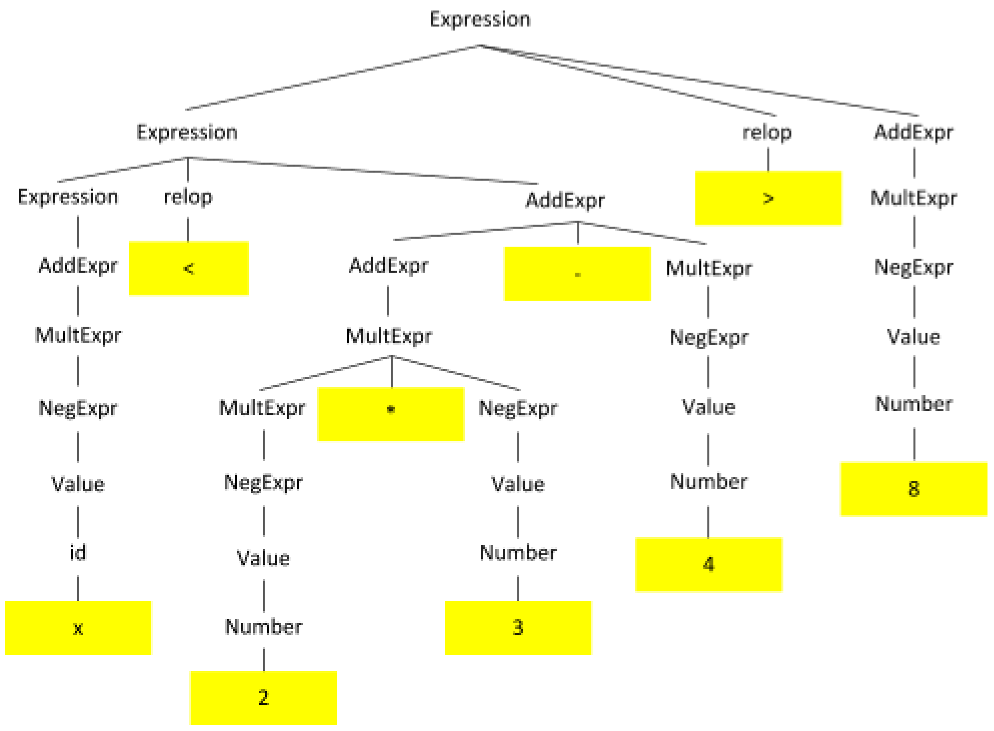
| ε

The final modified grammar, eliminated of left-recursion and suitable for recursive descent parsing:

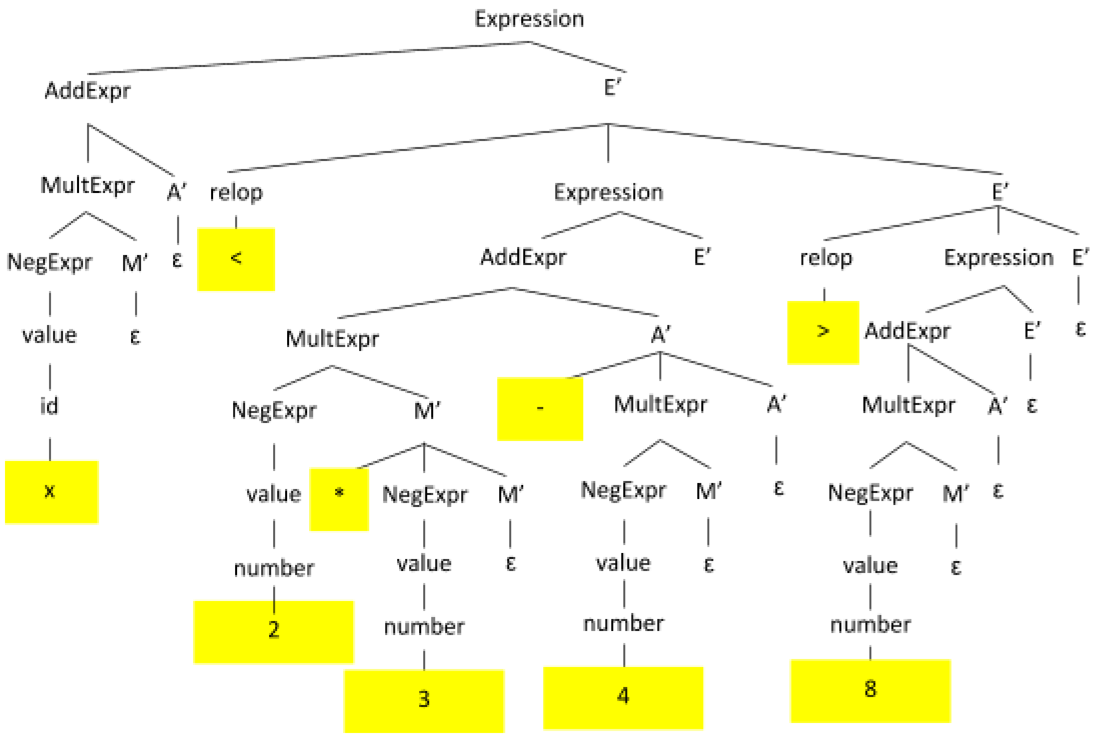
```
Program -> Statement P
P -> Program |  ε
Statement -> display Expression St1 | assign id = Expression
                | while Expression do Program end
                | if Expression then Program St2
St1 -> read id |  ε
St2 -> end | else Program end
Expression -> AddExpr E'
E' --> relop Expression E'
        |  ε
AddExpr -> MultExpr A'
A' -> - MultExpr A'
        | + MultExpr A'
        |  ε
MultExpr -> NegExpr M'
M' -> * NegExpr M'
        | / NegExpr M'
        |  ε
NegExpr -> -Value
        | Value
Value -> id | number | (Expression)
```

2c. Demonstration of Consistency Through Derivation Trees

These transformations do not change the original grammar. The parser can generate a derivation tree for the same expression from both the original and modified grammar. The next two figures show derivation trees for the Expression: x < 2 * 3 - 4 > 8 generated from the original grammar found in part 1 and the modified grammar found in part 2b. The expression (x < 2 * 3 - 4 > 8) tests the three changed productions of Expression & E' with "<" and ">", AddExpr & A' with "-" , and MultExpr & M' with "*".



The derivation tree for the Expression x < 2 * 3 - 4 > 8 using the original SIMPLE grammar.

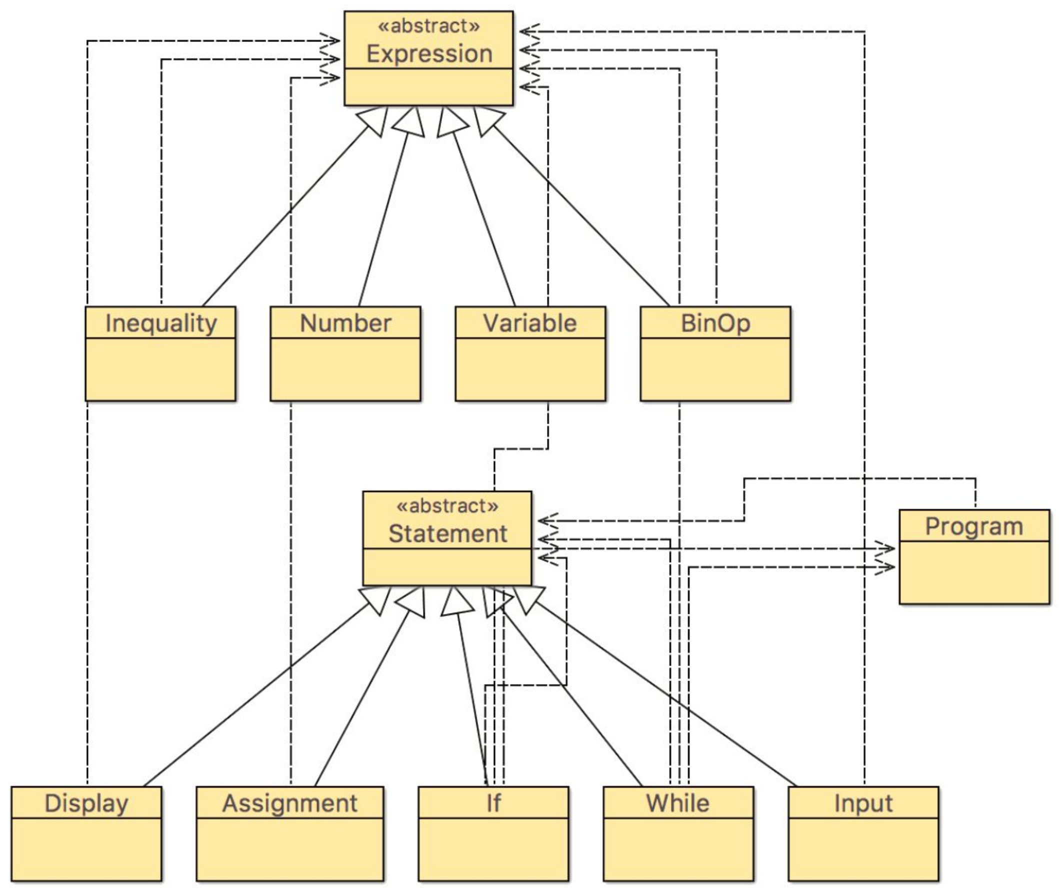


The derivation tree for the Expression x < 2 * 3 - 4 > 8 using the new SIMPLE grammar that eliminated left recursion.

Through these two demonstrations, it is clear that the parse tree of the modified grammar gives the same results as the original grammar. The modified grammar of part 2b eliminating left recursion is valid, as it produces the same output as the original grammar. Thus, this new grammar can be implemented by the SIMPLE Interpreter for recursive descent parsing.

2d. Object Decomposition Design

Here are the class hierarchies found in my abstract syntax tree package.



I based my new SIMPLE interpreter off the Statement and Expression hierarchy of my original Pascal interpreter, but I did make major modifications and added new objects to accommodate the SIMPLE grammar.

First, the Expression abstract class consists of a collection of objects that evaluate and returns an integer such as the value of number or the sum of two values. The Expressions hierarchy also has the Inequality class lumped into it.

Services of each object: The "Number" class creates a number object with a stored value that is returned when evaluated. The "Variable" class creates a variable object with a stored name and obtains the value of that variable with the name from the stored map in the Environment. The “BinOp” class takes in two expressions and an operator, effectively returning the sum, difference, product, or quotient the two expressions when evaluated. Finally, the “Inequality” class is used for conditions by taking multiple binary operators and expressions together. To make it most effective, it takes in a list of the numbers and a list of the operators and iterates completely through them, returning 0 if the inequalities are false and 1 if the inequalities are true.

On the other hand, the Statement abstract class consists of a collection of objects that executes actions in a specific environment such as printing lines, setting variables, or executing more statements.

Services of each object: The “Display” class takes in an Expression and displays, or prints out the evaluation of that Expression when executed. The “Assignment” class takes in a string and expression and sets the value of that string in the environment’s hashmap of variables to that value when executed. The “If” class takes in an Expression as the condition and two Programs, one for if the condition is true and one for if the condition is false (else statement), effectively performing an if statement when executed. Similarly, the “While” class takes in an Expression as the condition and one Program, continuously performing that statement while the condition is true. Finally, the “Input” class is unique, in that it prints and sets a new variable that the user inputs. The “Input” class will take in the name and expression and also the earlier expression to print later if it's false and 1 if it's true.

Program, which does not extend Statement or Expression, takes in a list of statements and executes them. This is used with the “While” and “If” classes in addition to when parsing the program in general. Overall, a hierarchical composition of objects with “Expression” and “Statement” as the parent classes is helpful for the interpreter to take in input and interpret it. When parsing a program, one can have a list of Statements and execute those, which will evaluate the Expressions which will evaluate to their separate objects accordingly.

2e. Test Strategy

In order to modify my code to match the SIMPLE grammar, I have to make sure the Interpreter correctly interprets all input commands and sees if it matches the SIMPLE grammar. Whenever the input command key word is “display”, the following expression would be printed on the console. Additionally, “read”, should be able to take in the user's input and assign it to a variable in the stored map. The key words "assign", "while", "if", and "else" all take in the following value, saving it in the stored map of variables, and execute according to the condition. Additionally, the interpreter won't properly execute tokens that aren't stated in the grammar and should not execute if the input does not match the grammar. For example, loops should be finished with "end", that would signify a stop to the program. If there is an extra "end", an error will occur. Thus, it is important to test nested loops to see that only the section of the program between the condition and "end" will occur. Similarly, "then" should follow an "if" condition and "do" should follow a "while" loop. Also, to test if the code works according to the grammar, I must create a Test input that contains "relop" tokens, addition or subtraction to test AddExpr, multiplication or division to test MultExpr, and negative values for NegExpr. The interpreter should output the value in the correct order of operations based off of the SIMPLE grammar's rules.

To modify my code to match the grammar, in my parseStatement method, which is supposed to parse a statement, it initially checks for if the current token is "display", "while", "if", or "assign". These are the four key words statements in the SIMPLE grammar should start with. In order to make sure the order of operations correctly matches the SIMPLE grammar's order, I created the methods parseExpression, parseTerm, and parseFactor. When evaluating an expression, parseExpression will first be called which is analogous to AddExpr, then calling parseTerm which is analogous to MultExpr, then calling parseFactor which is analogous to NegExpr. This organization of methods in which parseStatement calls ParseExpression which calls parseTerm which calls parseFactor is modeled after the grammar in which Expression -> AddExpr E', AddExpr -> MultExpr A', and MultExpr -> NegExpr M'.

Ultimately, to confirm my code works according to the grammar, I will run it through positive and negative test cases that check the keywords execute accordingly, the conditions with multiple operations are stored, and the order of operations is accurate.

3. Test Plan

For the Test Plan, we will be analyzing the expected output from the first and second SimpleTests using positive, negative, and edge test cases.

The first SimpleTest has the following test file input:

```
display 3
assign x = 1
display x read x
while x < 10 do
```

```
  display x
  assign x = x+1
  display x < 5 > 3
end
if x = 10 then
   display x
   assign x = 35
   display x
else
   display x
   assign x = 45
   display x
end
display x + 4
end
```

In the first simple test, first 3 is printed out. Then, 1 (x's new assignment) will print out. Then, whatever the user inputs as x will be read.

If x, the user's input, is less than 10, the program will go through a loop 10 - x times that prints out the value of x, prints out 0 because x cannot be < 5 > 3 and then adds 1 to x. This goes on until x hits 10. Once x hits 10, x, or 10, is printed. Then 35, x's new assignment, is printed. Finally, 39, or 35 + 4, is printed.

If x, the user's input, was not less than 10, then 45, x's new assignment, will be printed and then 49, or 45 + 4, will be printed.

To test the first SimpleTest, I should have three test cases. The first case is a **number less than 10** to check if the while loop works and will display the incrementing value of x along with "0", or false along with 10, 35, and 39 at the end. The second test is the **number 10** to test that the while loop will not occur and only 3, 1, 10, 35, and 39 are displayed. The third case is a **number greater than 10** to check that the while loop will not occur, 10 is not printed, and instead 3, 1, the number, 45, and 49 is printed.

The second SimpleTest has the following test file input:

```
display 4/3 read limit
assign x = 1
assign count = 0
while count < limit do
        display x
        assign count = count+1
        assign x = x*(x+1)
    end
    if count = limit
        then
        display count=limit
        if (x < 3)
            then
                display -600
            else
            display (x+1)*3
        end
    end
    if (limit+5)
        then
            display limit
    end
```

In the second simple test, first 1 is printed out. Then, whatever the user inputs as "limit" will be read and stored as a variable. Two more assignments are made of "x" with the value of 1 and "count" with the value of 0. Now there will be a loop, so that while count is less than limit, it does a specific program. This loop will iterate limit - count times, because count is incremented by one each iteration. In each iteration, the value of x is printed, and then x is multiplied by (itself + 1). Ultimately, x will turn out to be (x + (limit-count)! / x!) by the end of the while loop; this final value won't be printed until later though.

If it completed the loop or limit was set to 0, the resulting count is equal to limit, so it will print 1, or true for count being equal to limit. It now checks if x is less than 3 and will print -600. If x is not less than 3, it will print 3 times one more than x.

If it did not go through the loop, count was more than the user's input, so 1 would not be displayed and the program would not check for if x is less than 3.

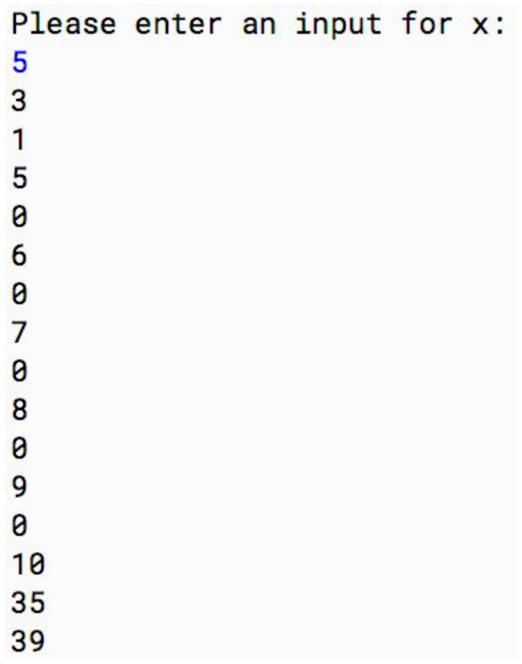
Finally, it prints the value of limit if limit + 5 is true. The way I interpreted this and the way I implemented my "If" class is that if limit + 5 = 0 then it's false, otherwise it will execute the statement. Thus, as long as limit is not equals to -5, limit will be printed at the end.

To test the second SimpleTest, I should have five test cases. The first test case is **a number greater than 0 and not 1** (0 is count) to check if the while loop will work and if 1 is printed out because count should be equal to limit by the end of the loop. At the end, (x+1)*3 should also be printed out as well as the value of limit, or the number/user's input. The second test case is the **number 0** check if the while loop will not occur and -600 will be printed. Although the while loop does not occur, count is equal to limit and x is less than 3. The third test case is the **number 1** to check if the while loop will occur and -600 is printed because x will be less than 3 after completing the while loop. The fourth test case is the **number less than 0 but not -5** . The while loop should not occur, and value of the number should be printed, as the number + 5 is not equal to 0. The final test case is the **number -5** to test if the final if the value of the number is not printed because (limit+5) is equal to 0.

4. Test Code Printout

First SimpleTest:

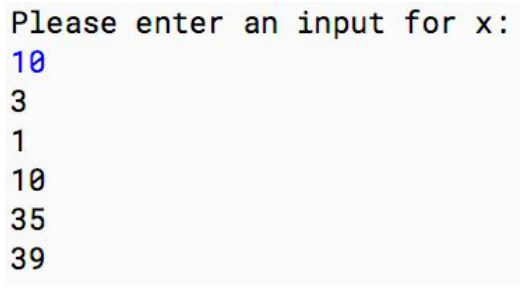
1. Input: 5 (number less than 10)



```
Please enter an input for x:
5
3
1
5
0
6
0
7
0
8
0
9
0
10
35
39
```

3 and 1 are first correctly printed out. Then a loop prints and iterates the values of x from x = 5 to 9, separated by printing out false, or 0. Finally, the value of x, 10, is accurately printed out along with 35 and 39.

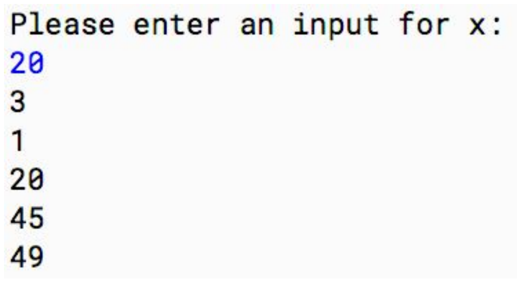
2. Input: 10



```
Please enter an input for x:
10
3
1
10
35
39
```

3 and 1 are first correctly printed out. Because 10 is not less than 10, it skips the loop. However, because x is equal 10, it prints the value of x, 10, and then 35 and 39.

3. Input: 20 (number greater than 10)

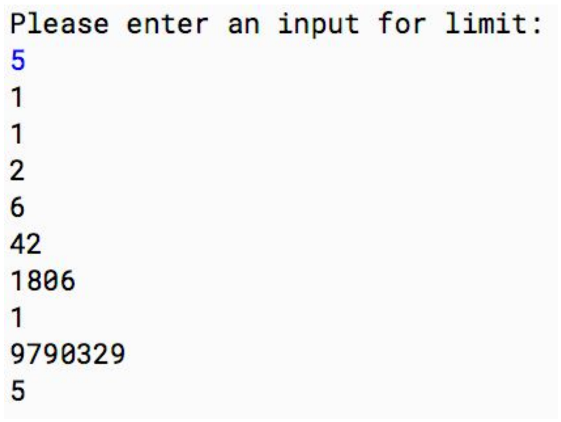


```
Please enter an input for x:
20
3
1
20
45
49
```

3 and 1 are first correctly printed out. Because 20 is not less than 10, it skips the loop. Additionally, because x is not equal 10, it prints the value of x, 20, and then 45 and 49.

Second SimpleTest:

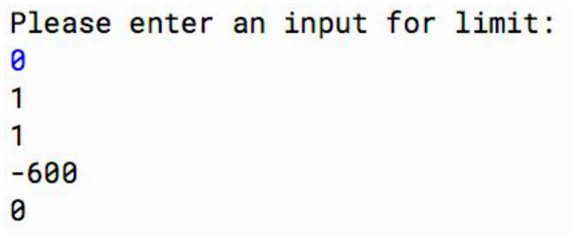
1. Input: 5 (number greater than 0 not 1)



```
Please enter an input for limit:
5
1
1
2
6
42
1806
1
9790329
5
```

1 is first correctly printed out. Then, limit, which is initially set to 1, is printed 5 (limit-count) times and each time x is multiplied by itself incremented. For example, it first prints 1, then 1*(1+1) or 2, then 2*3 or 6, then 6*7 or 42, then 42*43 or 1806. At the end, x should be equal to 1806*1807 or 3,263,442, but is not printed yet. Now, because count is equal to limit after the while loop, "true", or 1 is printed. Because x, which is now 3,263,442, is not less than 3, it prints out (x+1)*3 or 9,790,329. Finally, because limit+5, or 10, is not equal to 0, it prints the value of limit, 5.

2. Input: 0

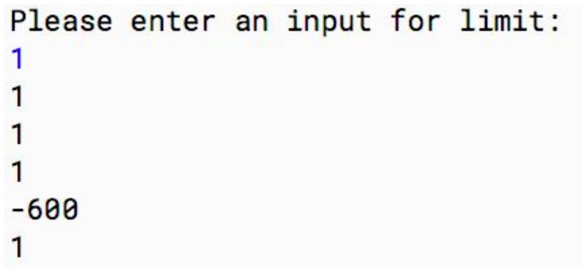


```
Please enter an input for limit:
0
1
1
-600
0
```

1 is first correctly printed out. Then, because limit, which is initially set to

0, is not greater than count, or 0, the while loop is skipped. However, because 0 is equal to 0, it goes into the if loop, printing true for count being equal to limit. Then, because x is still 1 from its initial assignment and less than 3, -600 is printed out. Finally, because 0 + 5 is not equal to 0, limit, or 0, is printed out.

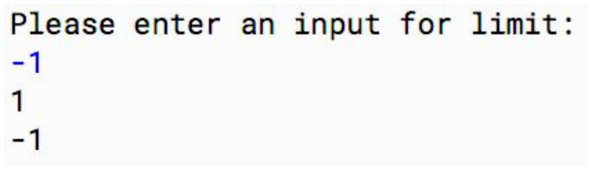
3. Input: 1



```
Please enter an input for limit:
1
1
1
1
-600
1
```

1 is first correctly printed out. Then, because limit, which is initially set to 1, is greater than count, or 0, the program goes into the while loop that iterates 1 (limit-count) time. In this single iteration, x, or 1, is printed out, and then count is incremented to 1 and x is set to 1*2 or 2. Now that count is equal to limit, it prints another 1 satisfying the count=limit condition. Because limit is less than 3, -600 is printed out.. Finally, because 0 + 5 is not equal to 0, limit, or 1, is printed out.

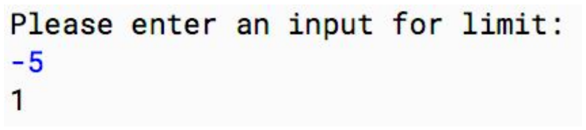
4. Input: -1 (number less than 0 not -5)



```
Please enter an input for limit:
-1
1
-1
```

1 is first correctly printed out. Then, because limit, which is initially set to -1, is not greater than count, or 0, the program goes skips the while loop. Count is not equal to limit, so it also skips the if statement. Finally, because 0 + 5 is not equal to 0, limit, or -1, is printed out.

5. Input: -5



```
Please enter an input for limit:
-5
1
```

1 is first correctly printed out. Then, because limit, which is initially set to -1, is not greater than count, or 0, the program goes skips the while loop. Count is not equal to limit, so it also skips the if statement. Finally, because -5 + 5 is equal to 0, the condition is false and nothing else is printed out.

From the test code printout, we can conclude that our Interpreter correctly parses both the first SimpleTest and the second SimpleTest. In the first test, the three inputs of 5, 10, and

20 cover all the conditions and give accurate results listed in the test plan. In the second test, the five inputs of 5, 0, 1, -1, and -5 cover all conditions and give accurate results listed in the test plan.

5. Results

My approach of tackling this SIMPLE interpreter with the grammar modifications to the original SIMPLE grammar remove left recursion, does effectively allow for recursive descent parsing. The class hierarchy I made with Expression and Statement as parent classes also work well.

6. Conclusion

In conclusion, this document shows that to create a SIMPLE Interpreter, one must modify it to be suitable for recursive descent parsing to implement an abstract syntax tree. To make it suitable, one must make sure it is left-factored and also make sure left recursion is eliminated. Furthermore, one should use a hierarchy of classes and multiple recursive methods according to the SIMPLE grammar for top-down parsing.