Kennesaw State University

Department of Computer Science

College of Computing and Software Engineering

CS 4308 Concepts of Programming Languages

Section W01

Project Part 3

Myron Woods

myronwoods46@students.kennesaw.edu

May 1, 2018

Initial Problem Statement:

The first part of this project stipulates the creation of a complete scanner. This scanner would then process a Lua program with a given set of lexical grammar. After the program is processed, tokens are to be derived from the Lua code. Then in the second part, a parser should check the logic and syntax of the code. The last part should then interprete these strings of tokens and produce a logical output.

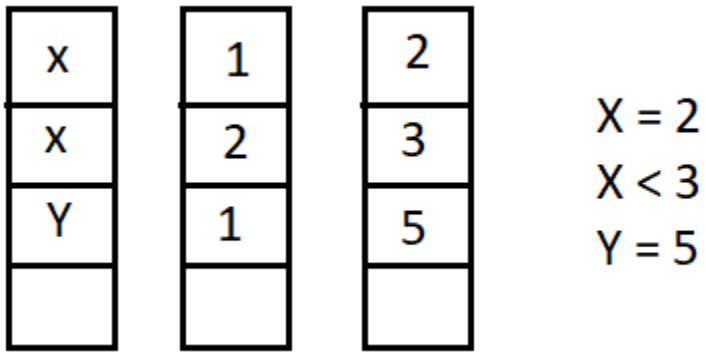
Purpose of the Report/Assignment:

The purpose of this assignment is to enforce an understanding of language rules and statutes. The project will have students create a scanner. This scanner must then extract tokens from a given input source. A parser will then check the logic of the tokens. The parser should catch logic and syntax errors and display error messages accordantly. The interpreter should then produce an output. The subsequent report should give a detail account of the methods used to achieve this.

Project Description:

After the scanner extracts tokens from an input source and the parser checks for logic and syntax, we can then move on to the last part; the interpreter. I decided to go with a "grid" system or a series of arrays that interact with each other. I will go through each of the major parts of the interpreter and explain the intuition behind them.

<u>Declaring variables and evaluating mathematical expressions:</u>

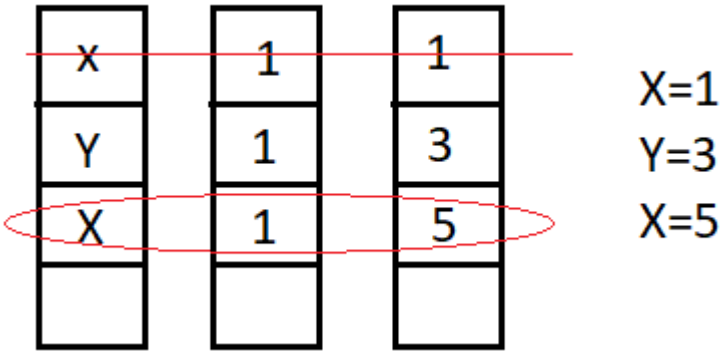


COLUMN 1 = Variable COLUMN 2 = Relation COLUMN 3 = Value

The most simple expression the interpreter has to deal with is assign statements and comparisons. Three arrays are used. One char array that stores the variables. Another integer array that stores the "relationship" value. Relation just means if the variable is being assigned or having another operation performed on it. If the variable is being assigned then we put a 1 in the column, if the variable is being compared, then a 2 is put in the column. The interpreter will need these relation values later in order to do other statements in the grammar. Last is a value array that stores the value for the corresponding variable.

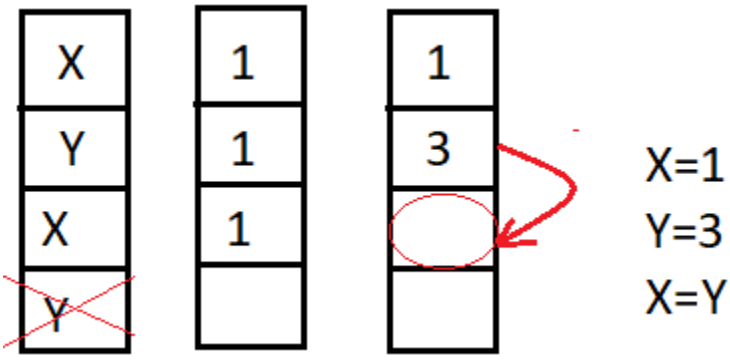
In order to fully flush out this this system above, we must also account for situations where a variable is being assigned or compared to an integer.

- Double assigning



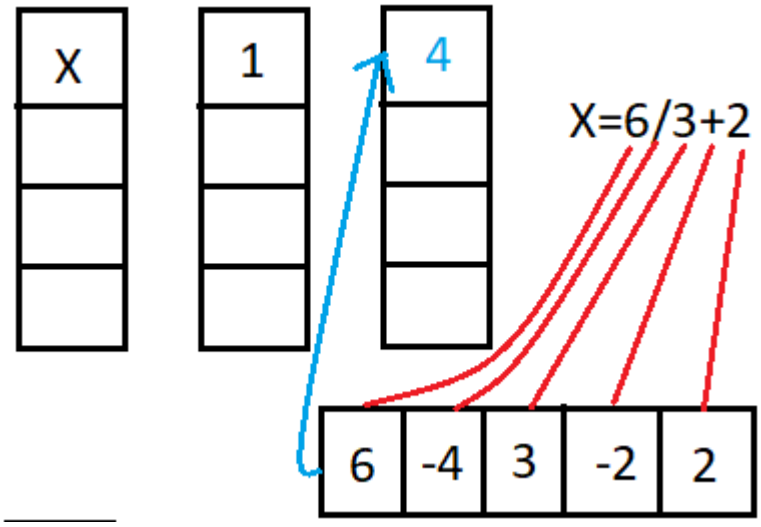
Here we can see X originally has a value of 1, but later is changed to 5. The interpreter just uses the last time the variable was assigned as the current value.

- Setting a variable to a variable



Here we have X being set to Y. The way the interpreter handles this is by looking at the value array. Here we see that X is being assigned, but it has no value. The interpreter will recognize this and understands that Y is not being declared. The interpreter determines that X is being assigned the value of Y. The interpreter then searches the variable array to see the last time Y was declared and assigned; it then returns that value of 3 to X.

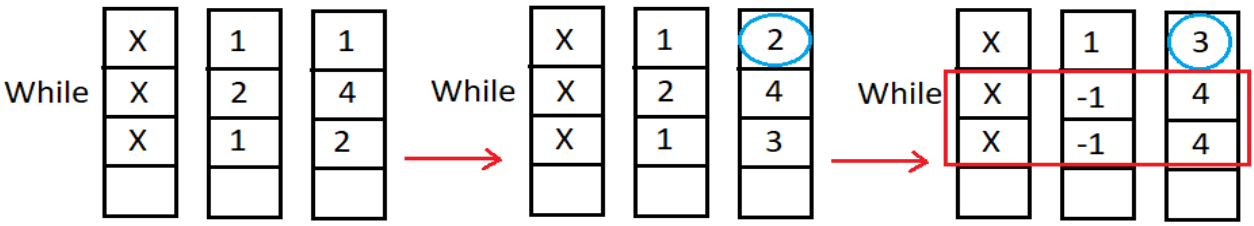
- Evaluating mathematical expressions



When the interpreter sees a math operator, it understands that a calculation needs to occur. In the example above, the interpreter sees the division sign and knows that it needs to calculate something. The interpreter will then send everything after the assign statement into another array. Since we can't put math operators into an array, we can use an integer representation for them. In the example, -4 is being used to represent division. In the actual code, a number like –99999999 is used to avoid a situation where the calculation results in a math operation. Ex. 4-8=-4. Of course, orders of operations must be followed. In the example above, 6 –4 3 would be evaluated before 3 –2 2. After the expression is evaluated, it is then sent back to the value array. (Math operations involving comparisons operate similarly.)

Loops, repeats, and if statements:

Now that we have of way of declaring variables and storing a value with them, we can now move on to other functions like loops. I lumped if statements with loops; as it can be seen as a loop that runs zero or one times.



X = 1
While X < 3
X = X+1

Here we have a while loop, with operation X = X+1 nested inside. In the first set of arrays we see X being set to 1. The loop is then evaluated. If the condition is true, the contents inside the loop are executed. Any variable value changes inside the loop are assigned to the index of the last time the variable was assigned BEFORE the loop. In the example above, X=1 was the last time X was assigned before the loop. We then change that to the value calculated inside the loop; as seen in the second group of arrays. X now equals 2. This process continues until the condition is no longer true. In the last set of arrays we see that X now equals 3. Three is not less than 3 so the loop is not executed. All lines associated with the loop are now set to –1. This way the interpreter knows the loop is done and not to reference any variables or values associated with the loop. If and repeat statements also behave similarly, except when their Boolean expression may be evaluated at a different time.

Print statements:

Print statements are relatively simple. They simply print the expression associated with. Even when the print statement is nested within an if statement or a loop, it will either print the line associated with it or not.

Here's how the output looks inside the compiler

Input:

```
x = 6
y = 2
while x < 8 {
x = ( x + 1 )  }
if x < 9 {
y = ( x + 2 ) }
print x
print y
```

JGRASP IDE:

```
ID_TOK, ASSIGN_TOK, LITERAL_INTEGER_TOK,
ID_TOK, ASSIGN_TOK, LITERAL_INTEGER_TOK,
WHILE_TOK, ID_TOK, COMP_TOK, LITERAL_INTEGER_TOK,
ID_TOK, ASSIGN_TOK, ID_TOK, MATH_TOK, LITERAL_INTEGER_TOK,
IF_TOK, ID_TOK, COMP_TOK, LITERAL_INTEGER_TOK,
ID_TOK, ASSIGN_TOK, ID_TOK, MATH_TOK, LITERAL_INTEGER_TOK,
PRINT_TOK, ID_TOK,
PRINT_TOK, ID_TOK,

PROGRAM EXECUTED
8
10

 ----jGRASP: operation complete.
```

*Prints the string of tokens for each line, if no errors were found the parser returns

"PROGRAM EXECUTED" and then the interpreter reads and executes the lines.

Program WITH ERRORS:

Input:

```
x = 6
y = 2
while x < / 8 {
x = < 8 }
print x
```

JGRASP IDE:

```
ID_TOK, ASSIGN_TOK, LITERAL_INTEGER_TOK,
ID_TOK, ASSIGN_TOK, LITERAL_INTEGER_TOK,
WHILE_TOK, ID_TOK, COMP_TOK, MATH_TOK, LITERAL_INTEGER_TOK,
ID_TOK, ASSIGN_TOK, COMP_TOK, LITERAL_INTEGER_TOK,
PRINT_TOK, ID_TOK,

ERROR ON LINE 3: COMP_TOK AND MATH_TOK CANNOT BE NEXT TO EACH OTHER
ERROR ON LINE 4: CANNOT ASSIGN COMPARISON TOKEN
ERROR ON LINE 4: CANNOT COMPARE PREVIOUS STATEMENT

 ----jGRASP: operation complete.
```

*Program recognizes errors and does not execute.

Citations:

Nguyen, Loc. "Lexical analyzer: an example." *Lexical analyzer: an example - CodeProject*, www.codeproject.com/Articles/833206/Lexical-analyzer-an-example.

"Compiler/Lexical analyzer." *Compiler/Lexical analyzer - Rosetta Code*, rosettacode.org/wiki/Compiler/lexical_analyzer.

"Writing a Parser in Java: The Tokenizer." *Cogito Learning*, Apr. 2013, cogitolearning.co.uk/2013/04/writing-a-parser-in-java-the-tokenizer/.

"How to Build a Parser by Hand." *Jayconrod.com*, Feb. 2014, jayconrod.com/posts/65/how-to-build-a-parser-by-hand.

Mcmanis, Chuck. "Build an Interpreter in Java -- Implement the Execution Engine." *JavaWorld*, JavaWorld, 1 July 1997, www.javaworld.com/article/2076974/learn-java/build-an-interpreter-in-java----implement-the-execution-engine.html.

Concepts of Programming Languages: Concepts of Programming Languages, 11/E