

Assignment 2 : Ruby

Work Statement

Initially, the work was to be divided between the lexer designed by Jonathan and the parser designed by Anthony. However, as the project progressed, both team members began to add onto both portions of the program. After a certain amount of progress was made with the parser, we worked on the code exclusively together in order to ensure code consistency, though we sent each other various versions of the code as we updated it.

Design

First, “tokenArray” is created as a new array. It is then populated with nextToken, which takes the designated filename and opens the corresponding file. The function then splits all possible tokens into positions within an array that is returned and stored as the new value of tokenArray (in main). Comments beginning with “/” were disregarded entirely and thrown out upon reaching the lexer, never being added to the array which would be sent to the parser. Several auxiliary functions assist in this process, such as whatOperator, whatKeyword, splitOperator, removeOnlySpace, and removeNewLine.

When the tokenArray is populated, it is then sent to program. This is the beginning of the recursive descent analyzer (the parser). From program, the full list of sequential characters within the entire file (local arr) is checked with stmts to see if the return value is true. If so, the exit message of “This is a valid program!” is displayed. Else, “This program is invalid!” will be displayed. Stmt then checks to see if there is a valid stmt followed by a “;”, or one followed by more stmts, otherwise returning false. Within Stmt, the different types of statement which can be observed in a valid simpl program are represented by scanning for keywords with regex and getTokenKind followed by reading in elements and doing boolean checks to make sure that the tokens between the two are of the proper format (as in the grammar). Stmt and StmtS both return true or false, whereas the other functions call down to lower levels of the diagram and eventually return the type of whatever terminal is being read (such as identifier or integer or perhaps “true” or “false”). Functionally, these methods are still acting as a boolean type of value, meaning that the only thing being checked at each level is whether or not the level beneath it (according the diagram) is nil or not. At many points in the program, nil is returned or the program will immediately shut down, yielding an invalid program message so as to make things easier with coding for all these layers of recursion. The logic follows along with the BNF grammar diagram recursively as intended on every level.

Other functions such as parenthesisBalance and isPrimaryOp were used at points in the program to help with better parsing. For example, the former function helped with nested parentheses by returning the index of the closing parenthesis associated with the opening parenthesis at the index given as an argument (usually used when elements begin with “(“). The latter function was used to determine if a given string was equal to any of the valid mulops or addops in order to disallow any double inputs of operators to be considered valid.

Difficulties

We had disagreements on how some of the methods should be structured and both made separate attempts to complete them, but could not do so until we began working together on the project full time. Also, as we have surmised from the sheer enormity of our code, we probably over-complicated the program and made it somewhat of a disorganized mess. However, luckily, it does work.

Pros of this Language

Ruby was very easy to pick up on. The overall structure felt similar to Python, yet even easier because one did not have to worry about the strict nature of indentation when writing code. Features such as regex were also helpful and easily accessible. The print statements were quick and easy to type and I loved being able to print an array using only one statement. Having print and puts both available also helped for differing types of debugging printouts needed. Arrays were also much more accessible in general as one need not declare the size and could shift or pop elements easily without having to resize or build all new arrays. The language itself was very fun and easy to use. It was much more enjoyable than scheme and felt very lightweight in its use, as opposed to languages like Java, where even a simple print statement is quite a bit more code than just “puts”.

Cons of this Language

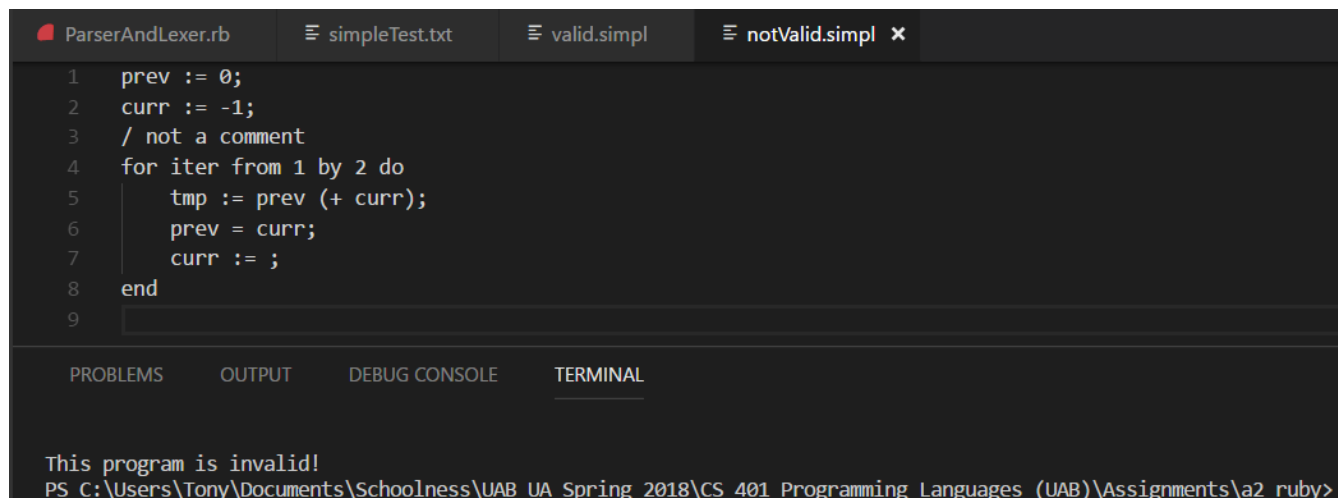
Honestly, it did not feel like there was much of a downside to this language at all, but perhaps due to it feeling so lightweight and easy to use, we ended up writing more code than needed without trying to condense much until the very end. Additionally, on Anthony’s computer, there ended up being exactly one point of the program that yielded a stack inconsistency error. This was never resolved, but for some reason, a blank print statement on a particular line of the program allowed us to circumvent this problem, which never occurred on Jonathan’s computer.

Test Cases

Screenshots taken in MS Visual Studio Code IDE

The full file simpleText.simpl will also be included to show accuracy of these tests. Within said file, any commented out line can have the comments removed and the program will return invalid.

Example invalid statements :



The screenshot shows the MS Visual Studio Code IDE with four tabs open: ParserAndLexer.rb, simpleTest.txt, valid.simpl, and notValid.simpl. The notValid.simpl tab is active, displaying a Ruby script with several syntax errors. The script is as follows:

```
1 prev := 0;
2 curr := -1;
3 / not a comment
4 for iter from 1 by 2 do
5     tmp := prev (+ curr);
6     prev = curr;
7     curr := ;
8 end
9
```

Below the code editor, the TERMINAL panel is visible, showing the output of the program:

```
This program is invalid!
PS C:\Users\Tony\Documents\Schoolness\UAB UA Spring 2018\CS 401 Programming Languages (UAB)\Assignments\a2 ruby>
```

Example valid statements:

```
ParserAndLexer.rb  simpleTest.simpl x
4  if a < 1 and b < 2 then
5      jj:=1+2+3-4+5+6-(3+3);
6      for itr from 0 to M-1 do
7          bb:= 2;
8          pt:=(((4/3)+1));
9      end;
10     pp:=(((3*3)+1));
11     hh:=1+(2+3)+3*4;
12 end;
13
14 if a < 3 and b < 4 then
15     if a < 2 and b < 5 then
16         aa:= 1+1;
17         bb:= 2;
18         cc:=3;
19     else
20         dd:=1;
21         ee:=56;
22     end;
23 end;
24
25 for iter from 0 to N-1 by (1+2) do          // iterative fibonacci
26     tmp := prev + curr;
27     bb := (1+2);
28     if a < 2 and b < 5 then
29         prev := curr;
30     else
31         dd:=1;
32         ee:=56;
33     end;
34 curr := tmp;
35 a := b; end
36 ;

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

This is a valid program!
PS C:\Users\Tony\Documents\Schoolness\UAB UA Spring 2018\CS 401 Programming Languages (UAB)\Assignments\a2 ruby> 
```