P1 Interpreter 100% Complete

Julia Interpreter

Sharon Perry CPL 4308 Section 03

Ronak Patel
4-17-2022

The main loop of my program occurs in LexLoop where it keeps building lexemes and figuring out what they are. Build next lexeme calls find next lexeme and that function removes comments and whitespace until new lines are found. Build next lexeme builds the string one char at a time. LexLoop then uses that string to find out what type of lexeme it is, which can go into different categories. It can be a keyword, Identifier, or a literal. These are functionalized to return bool types for condition statements. Token types were created to account for the basic things I would run into via my test cases and the language grammar.

Some of the work I performed was fixing my previous scanner. I had a block remover put in that would remove tokens of \t. This allowed me to parse my file with the correct size. I used recursive descent parsing to build the structure of my program and scanned for tokens using the grammar chart. The grammar chart helped enable me to figure out whether there was an error in my test cases. The parses write to the console about what actions they're taking building out the BNF of the test cases. Another thing I had to keep track of was what values were given from ID's which I kept in a list. Using the Visual Studio Debugger tool was how I was able to mark out where the index should be for all test cases and let me map out exactly where I wanted it incremented which was immensely helpful to avoid out of bound errors or logic errors. Environment.Exit(0) would notify the user that a certain aspect of the program failed and exited after encountering an error.

The interpreter sprint had me reworking my parser so that it was formatted in a more indexable manner. I stuffed the parser results into a list that I accessed in the main file. This file allowed me to scan through the list until I hit the print statement. I then checked for whichever possible return values using the grammar rules of the language. So, I would scan ahead using pointer arithmetic to see if it was an ID or literal symbol. Based on that, I would know what

index my return value for the arithmetic expression would be. The only working Julia file that followed the grammar rule was test 1 considering the other two had errors so it worked flawlessly for that.

Files Used

Scanner.cs – scans for tokens, eliminates white space, and used TokenTypes to build a list of tokens.

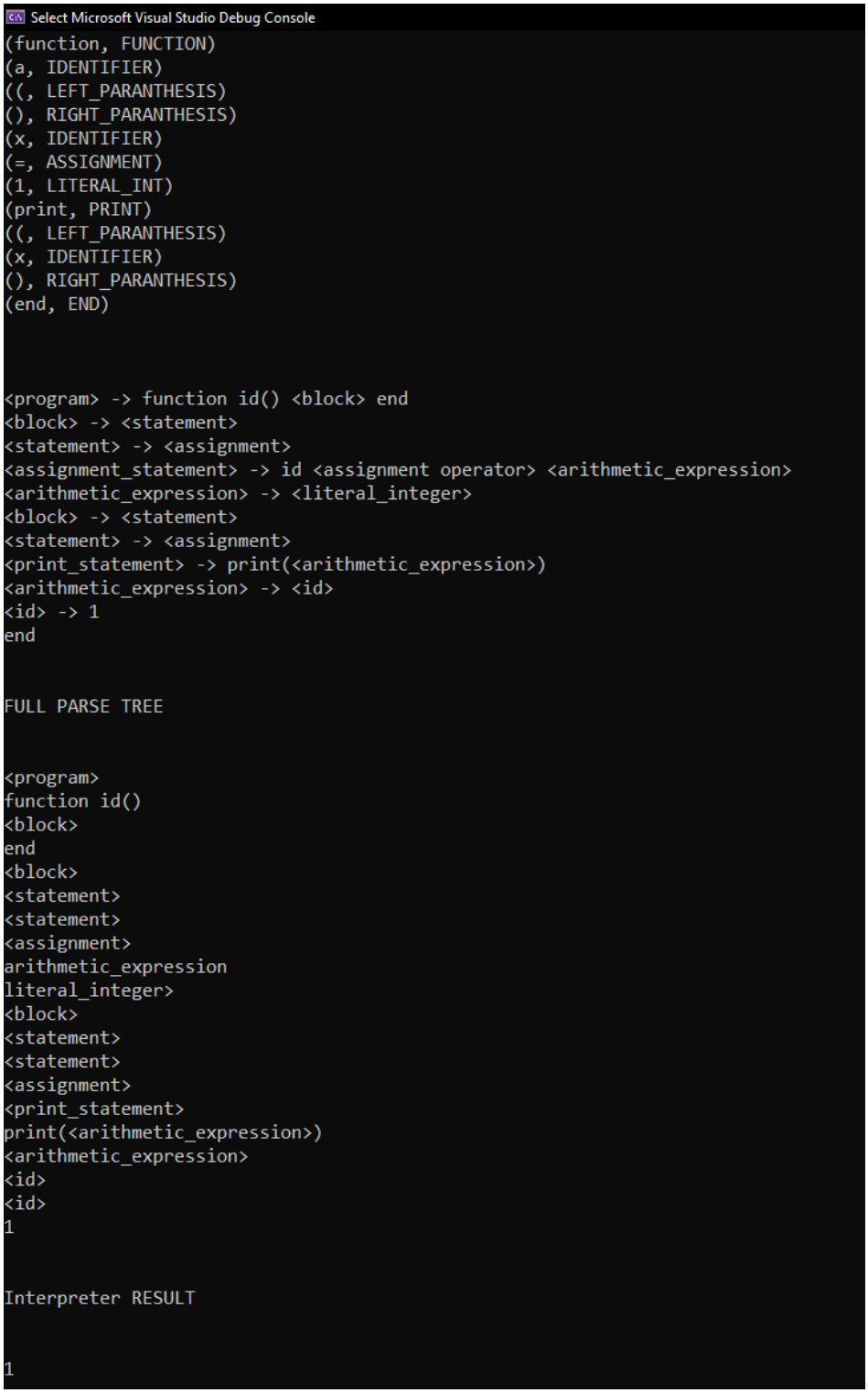
TokenTypes.CS – an enum type that had all my types I needed to classify my tokens.

Parser.CS – allowed me to read off my list of scanned tokens and verify if they accurate Julia files. If so, then it creates a BNF form of my code that allows me to interpret it in my main file.

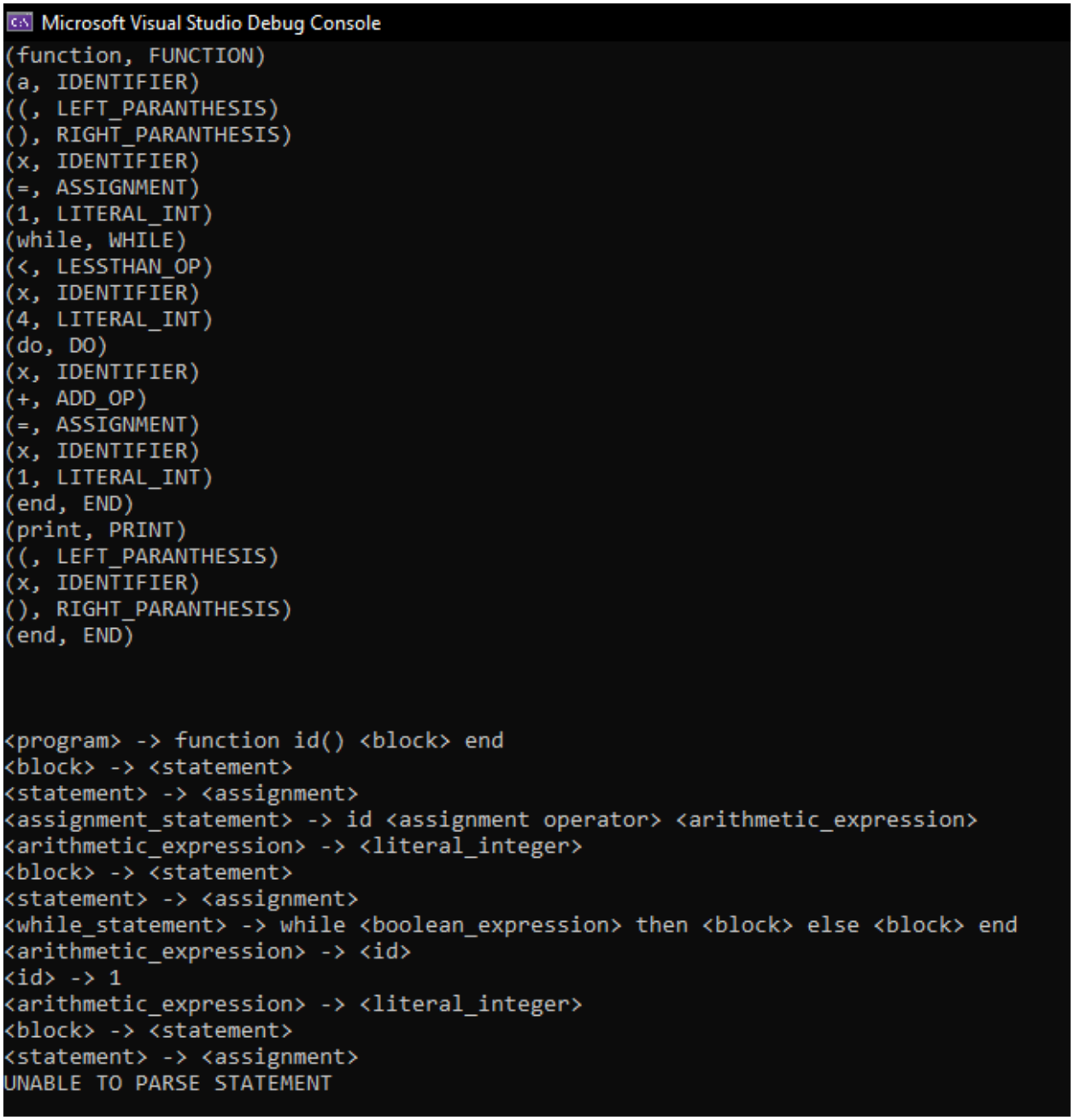
Program.cs – this main file held was returned to most of my lists. It also was where I wrote my interpreter result to.

Screenshots

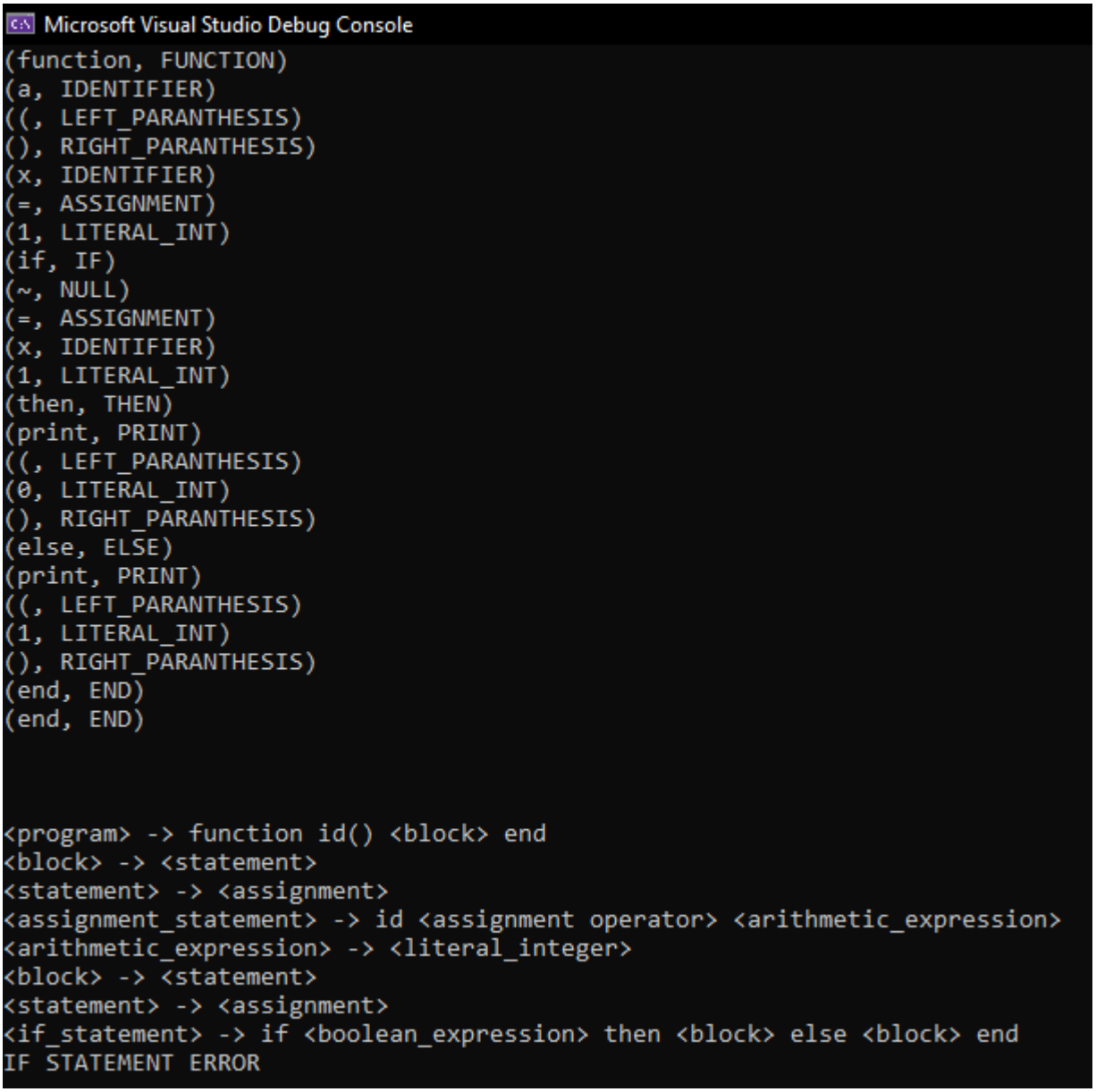
TEST 1



Test 2


Microsoft Visual Studio Debug Console
(function, FUNCTION)
(a, IDENTIFIER)
((, LEFT_PARANTHESIS)
(), RIGHT_PARANTHESIS)
(x, IDENTIFIER)
(=, ASSIGNMENT)
(1, LITERAL_INT)
(while, WHILE)
(<, LESSTHAN_OP)
(x, IDENTIFIER)
(4, LITERAL_INT)
(do, DO)
(x, IDENTIFIER)
(+, ADD_OP)
(=, ASSIGNMENT)
(x, IDENTIFIER)
(1, LITERAL_INT)
(end, END)
(print, PRINT)
((, LEFT_PARANTHESIS)
(x, IDENTIFIER)
(), RIGHT_PARANTHESIS)
(end, END)
<program> -> function id() <block> end
<block> -> <statement>
<statement> -> <assignment>
<assignment_statement> -> id <assignment operator> <arithmetic_expression>
<arithmetic_expression> -> <literal_integer>
<block> -> <statement>
<statement> -> <assignment>
<while_statement> -> while <boolean_expression> then <block> else <block> end
<arithmetic_expression> -> <id>
<id> -> 1
<arithmetic_expression> -> <literal_integer>
<block> -> <statement>
<statement> -> <assignment>
UNABLE TO PARSE STATEMENT

Test 3


Microsoft Visual Studio Debug Console
(function, FUNCTION)
(a, IDENTIFIER)
((, LEFT_PARANTHESIS)
(), RIGHT_PARANTHESIS)
(x, IDENTIFIER)
(=, ASSIGNMENT)
(1, LITERAL_INT)
(if, IF)
(~, NULL)
(=, ASSIGNMENT)
(x, IDENTIFIER)
(1, LITERAL_INT)
(then, THEN)
(print, PRINT)
((, LEFT_PARANTHESIS)
(0, LITERAL_INT)
(), RIGHT_PARANTHESIS)
(else, ELSE)
(print, PRINT)
((, LEFT_PARANTHESIS)
(1, LITERAL_INT)
(), RIGHT_PARANTHESIS)
(end, END)
(end, END)
<program> -> function id() <block> end
<block> -> <statement>
<statement> -> <assignment>
<assignment_statement> -> id <assignment operator> <arithmetic_expression>
<arithmetic_expression> -> <literal_integer>
<block> -> <statement>
<statement> -> <assignment>
<if_statement> -> if <boolean_expression> then <block> else <block> end
IF STATEMENT ERROR