Dominick DiLeo

Programming Language Concepts

Due December 1st, 2021

Assessment 2

**PRACTICAL- 50 points**

**In this problem you will be designing code that should be able to take in a file that is analyzed for lexical and syntactical correctness. This programming language should allow for the following type of statements:**

**• switch**

**• foreach**

**• for**

**• while**

**• do-while**

**• block**

**• if**

**• assignment**

**• return**

**Imagine this programming language also includes the following rules:**

**<program> VOID MAIN ’(‘ ’)’ <block>**

**<block> ‘{‘ { <statement> ; } ‘}’**

**\*\*\* Code should be able to detect syntax and lexical errors. You may choose the syntactical**

**structure of each statement, but they must be that of another programming language that you specify in the comments and in a word doc where you list the grammar rules of your language.**

First I will “specify in the comments and in a word doc where you list the grammar rules of your language” as the instructions said. My code will basically do what the grammar does but just with a bunch of nested if statements and each nonterminal is a function call

**<program> and <block> are as defined in the instructions above**

**<statement> 🡪**

switch<swith\_statement> |

for<for\_or\_foreach\_statement> |

while<while \_statement> |

do<dowhile \_statement> |

if<if \_statement> |

identifier<assignment\_statement> |

return<return \_statement>

(c based) also remember that <statement> before already included the keyword switch, while, if, etc so I do not include those keywords in the non terminals below

**<switch\_statement> 🡪**

‘(’identifier‘)’ ‘{’case:{<statement>}break;{case:{<statement>}break;}[default:{<statement>}]}

**<for\_or\_foreach\_statement> 🡪**

‘(’(int|float|String|char)identifier(=<for\_statement>|:<foreach\_statement>)

(c++ based. I think he said that we could use the math expression code from previous assignment to fill in the “Boolean” expressions since c++ accepts a number and doesn’t force a Boolean expression like Java does) so the middle expression of the for loop and the condition expression for the While and If statements just call Expression()

(c++ based)

**<for\_statement> 🡪**

<expression>;<expression>;identifier=<expression> ‘)’<block>

(c++ based)

**<foreach\_statement> 🡪**

identifier ‘)’<block>

(c++ based)

**<while\_statement> 🡪**

‘(’<expression>‘)’<block>

(c++ based)

**<dowhile\_statement> 🡪**

<block>while ‘(’<expression>‘)’

(c++ based)

**<if\_statement> 🡪**

‘(’<expression>‘)’<block>

(java based)

**<assignment\_statement> 🡪**

=<expression>

(java based)

**<return\_statement> 🡪**

<expression>

(the math ebnf used in homework 3)

**<expression> 🡪**

<term>{(+|-)}<term>

**<term> 🡪**

<factor>{(\*|/|%)}<factor>

**<factor> 🡪**

INT\_LIT | FLOAT\_LIT | IDENTIFIER | ‘(’<expression>‘)’

also I added an error function that is called whenever a syntax error happens and it prints what method the error occurred in. Each method passes a string identifying itself when calling error(). Then system.exit(1) stops the program:

public static void error(String methodName) {

System.out.println("\nERROR, NOT VALID INPUT. ERROR OCCURRED IN: " + methodName + "()");

System.exit(1);

}

I have prepared a test text file for every type of statement and will upload them when submitting. I also included a test file called “test.txt” that has some different nested statements and more than 1 statement in a block statement to show that my code works fine.

If you would like to see the output for the different txt files for yourself when running the program comment all of those “lexAnalyzer = …” lines and uncomment 1 of them in the main method:

Text

Description automatically generated

here is the input for test.txt:

*VOID MAIN(){*

*if(3+2){*

*variable1 = 1;*

*while(3+2\*(variable1/3)){*

*variable2 = 2;*

*};*

*};*

*return 5+variable2;*

*}*

the Main block contains an if statement and a return statement. The if statement block contains an assignment statement and a while statement. The while statement contains an assignment statement. Here is the output below that traces the lexemes and non terminal function calls. It is pretty long, but it’s easier to look over this 1 text file output than look at all the different text file outputs:

Next token is: {token value: 38, token name: VOID\_CODE}, Next lexeme is VOID

Enter <program>

Next token is: {token value: 39, token name: MAIN\_CODE}, Next lexeme is MAIN

Next token is: {token value: 25, token name: LEFT\_PAREN}, Next lexeme is (

Next token is: {token value: 26, token name: RIGHT\_PAREN}, Next lexeme is )

Enter <block>

Next token is: {token value: 5, token name: LEFT\_CURLY}, Next lexeme is {

Next token is: {token value: 31, token name: IF\_CODE}, Next lexeme is if

Enter <statement>

Enter <if\_statement>

Next token is: {token value: 25, token name: LEFT\_PAREN}, Next lexeme is (

Enter <Expression>

Enter <Term>

Enter <Factor>

Next token is: {token value: 10, token name: INT\_LIT}, Next lexeme is 3

Next token is: {token value: 21, token name: ADD\_OP}, Next lexeme is +

Exit <Factor>

Exit <Term>

Enter <Term>

Enter <Factor>

Next token is: {token value: 10, token name: INT\_LIT}, Next lexeme is 2

Next token is: {token value: 26, token name: RIGHT\_PAREN}, Next lexeme is )

Exit <Factor>

Exit <Term>

Exit <Expression>

Enter <block>

Next token is: {token value: 5, token name: LEFT\_CURLY}, Next lexeme is {

Next token is: {token value: 11, token name: IDENT}, Next lexeme is variable1

Enter <statement>

Enter <assignment\_statement>

Next token is: {token value: 20, token name: ASSIGN\_OP}, Next lexeme is =

Enter <Expression>

Enter <Term>

Enter <Factor>

Next token is: {token value: 10, token name: INT\_LIT}, Next lexeme is 1

Next token is: {token value: 6, token name: SEMICOLON}, Next lexeme is ;

Exit <Factor>

Exit <Term>

Exit <Expression>

Exit <assignment\_statement>

Exit <statement>

Next token is: {token value: 33, token name: WHILE\_CODE}, Next lexeme is while

Enter <statement>

Enter <while\_statement>

Next token is: {token value: 25, token name: LEFT\_PAREN}, Next lexeme is (

Enter <Expression>

Enter <Term>

Enter <Factor>

Next token is: {token value: 10, token name: INT\_LIT}, Next lexeme is 3

Next token is: {token value: 21, token name: ADD\_OP}, Next lexeme is +

Exit <Factor>

Exit <Term>

Enter <Term>

Enter <Factor>

Next token is: {token value: 10, token name: INT\_LIT}, Next lexeme is 2

Next token is: {token value: 23, token name: MULT\_OP}, Next lexeme is \*

Exit <Factor>

Enter <Factor>

Next token is: {token value: 25, token name: LEFT\_PAREN}, Next lexeme is (

Enter <Expression>

Enter <Term>

Enter <Factor>

Next token is: {token value: 11, token name: IDENT}, Next lexeme is variable1

Next token is: {token value: 24, token name: DIV\_OP}, Next lexeme is /

Exit <Factor>

Enter <Factor>

Next token is: {token value: 10, token name: INT\_LIT}, Next lexeme is 3

Next token is: {token value: 26, token name: RIGHT\_PAREN}, Next lexeme is )

Exit <Factor>

Exit <Term>

Exit <Expression>

Next token is: {token value: 26, token name: RIGHT\_PAREN}, Next lexeme is )

Exit <Factor>

Exit <Term>

Exit <Expression>

Enter <block>

Next token is: {token value: 5, token name: LEFT\_CURLY}, Next lexeme is {

Next token is: {token value: 11, token name: IDENT}, Next lexeme is variable2

Enter <statement>

Enter <assignment\_statement>

Next token is: {token value: 20, token name: ASSIGN\_OP}, Next lexeme is =

Enter <Expression>

Enter <Term>

Enter <Factor>

Next token is: {token value: 10, token name: INT\_LIT}, Next lexeme is 2

Next token is: {token value: 6, token name: SEMICOLON}, Next lexeme is ;

Exit <Factor>

Exit <Term>

Exit <Expression>

Exit <assignment\_statement>

Exit <statement>

Next token is: {token value: 4, token name: RIGHT\_CURLY}, Next lexeme is }

Exit <block>

Next token is: {token value: 6, token name: SEMICOLON}, Next lexeme is ;

Exit <while\_statement>

Exit <statement>

Next token is: {token value: 4, token name: RIGHT\_CURLY}, Next lexeme is }

Exit <block>

Next token is: {token value: 6, token name: SEMICOLON}, Next lexeme is ;

Exit <if\_statement>

Exit <statement>

Next token is: {token value: 40, token name: RETURN\_CODE}, Next lexeme is return

Enter <statement>

Enter <return\_statement>

Enter <Expression>

Enter <Term>

Enter <Factor>

Next token is: {token value: 10, token name: INT\_LIT}, Next lexeme is 5

Next token is: {token value: 21, token name: ADD\_OP}, Next lexeme is +

Exit <Factor>

Exit <Term>

Enter <Term>

Enter <Factor>

Next token is: {token value: 11, token name: IDENT}, Next lexeme is variable2

Next token is: {token value: 6, token name: SEMICOLON}, Next lexeme is ;

Exit <Factor>

Exit <Term>

Exit <Expression>

Exit <return\_statement>

Exit <statement>

Next token is: {token value: 4, token name: RIGHT\_CURLY}, Next lexeme is }

Exit <block>

Exit <program>

**1. 25 points**

**Create an LR Parsing table for the following grammar (10 Points) and show the steps to**

**solve the following problems**

**S -> a C | A C**

**A -> a B b | A a**

**B -> b B | c C**

**C -> c C | d**

**a. abbbccd**

**b. accd**

**c. acdbaacd**

**d. acdbd**

**e. abcdbad**

I am only going to show the LR Parsing table for problem 1 above and not do a,b,c,d,e manually. Instead I will do problem 2 which is worth 50 points, and then I will show the steps to solve a,b,c,d,e but it will be done programmatically so I don’t know if you will count that as solving a,b,c,d,e in problem 1 or not.

In class he said we could use the link on icollege to make a LR(1) parsing table:

<http://jsmachines.sourceforge.net/machines/lr1.html>

Here is the input I wrote for the website which is just the grammar described above without the

“ | ”. And also it seems that on the website you must have some beginning nonterminal that only leads to the next non terminal(E->S). Without it the steps generated and the LR parsing table seem incomplete.

A picture containing text

Description automatically generated

Here it is if you would like to copy and paste it on the website:

E -> S

S -> a C

S -> A C

A -> a B b

A -> A a

B -> b B

B -> c C

C -> c C

C -> d

Here is the table generated by the website:

Calendar

Description automatically generated

**2. (50 points) Given the grammar from the previous problem, programmatically show**

**where each string fails the grammar or create a parse tree showing they are in the**

**language.**

This problem is similar to the extra credit problem from Homework 3, so I used that to start this problem. I first altered it to work with this new table and grammar rules shown above. Once I got that working, I needed a way to make a tree data structure storing the LHS and RHS of the grammar rules used during reduction so I could have a parse tree at the end to traverse. One problem was that my original code simply pushed Strings onto the stack. I made a node class with variables: value(contains the string that was used in my original code), isState(Boolean showing if this node is just a State value on the stack or not, I didn’t really use this variable but it could be useful to make sure a State never has children), and then child1(another node), child2, and child3. I chose 3 children because rule 3 of the grammar above has 3 items on the RHS so at most a LHS node will have 3 children. Then I modified my code some to now access the “value” variable of the node since now my code works with node objects. Whenever a reduction is being done, I temporarily stored the nodes being popped from the stack into an array, and before pushing the LHS node onto the stack, I set its child variables to those nodes that were just popped. At the end of the parsing(assuming the input was valid in the language produced by the grammar), the stack should contain nodes with values [0,S,1]. The node with S should now be the root node of the parse tree. I then popped the node with value 1(which is no longer needed), and popped again the one with S and stored it as a node variable called root. This root node was then passed to a function that does pre-order traversal of a tree. I used pre-order traversal because on Discord, the teacher told us to use pre-order traversal to show the parse tree.

Here is an example when the input was not valid and the output will show

where the string fails the grammar as the problem 2 instructions says it should. I will also show the websites steps so you can see that my code shows the same steps

input(a from problem 1):

*abbbccd*

website steps:

Calendar

Description automatically generated with medium confidence

My program’s steps:

Stack: [0]

Input: [$, d, c, c, b, b, b, a]

Action: S,2

Stack: [0, a, 2]

Input: [$, d, c, c, b, b, b]

Action: S,8

Stack: [0, a, 2, b, 8]

Input: [$, d, c, c, b, b]

Action: S,8

Stack: [0, a, 2, b, 8, b, 8]

Input: [$, d, c, c, b]

Action: S,8

Stack: [0, a, 2, b, 8, b, 8, b, 8]

Input: [$, d, c, c]

Action: S,17

Stack: [0, a, 2, b, 8, b, 8, b, 8, c, 17]

Input: [$, d, c]

Action: S,21

Stack: [0, a, 2, b, 8, b, 8, b, 8, c, 17, c, 21]

Input: [$, d]

Action: S,22

Stack: [0, a, 2, b, 8, b, 8, b, 8, c, 17, c, 21, d, 22]

Input: [$]

Action: null

Input WAS NOT accepted.

Reason: Cell in the Action Table corresponding to state 22 and symbol $ contained no action.

Now for input that is valid for the language:

input(c from problem 1):

*acdbaacd*

website steps:

Diagram

Description automatically generated with low confidence

My program’s steps are below. At the end it shows the pre-order traversal of the parse tree. If you look at the tree on the right in the image of the website above you can see that my code’s pre order traversal output is correct for the tree:

Stack: [0]

Input: [$, d, c, a, a, b, d, c, a]

Action: S,2

Stack: [0, a, 2]

Input: [$, d, c, a, a, b, d, c]

Action: S,6

Stack: [0, a, 2, c, 6]

Input: [$, d, c, a, a, b, d]

Action: S,15

Stack: [0, a, 2, c, 6, d, 15]

Input: [$, d, c, a, a, b]

Action: R,8 (use GOTO[6, C])

Stack: [0, a, 2, c, 6, C, 13]

Input: [$, d, c, a, a, b]

Action: R,6 (use GOTO[2, B])

Stack: [0, a, 2, B, 5]

Input: [$, d, c, a, a, b]

Action: S,12

Stack: [0, a, 2, B, 5, b, 12]

Input: [$, d, c, a, a]

Action: R,3 (use GOTO[0, A])

Stack: [0, A, 3]

Input: [$, d, c, a, a]

Action: S,10

Stack: [0, A, 3, a, 10]

Input: [$, d, c, a]

Action: R,4 (use GOTO[0, A])

Stack: [0, A, 3]

Input: [$, d, c, a]

Action: S,10

Stack: [0, A, 3, a, 10]

Input: [$, d, c]

Action: R,4 (use GOTO[0, A])

Stack: [0, A, 3]

Input: [$, d, c]

Action: S,11

Stack: [0, A, 3, c, 11]

Input: [$, d]

Action: S,7

Stack: [0, A, 3, c, 11, d, 7]

Input: [$]

Action: R,8 (use GOTO[11, C])

Stack: [0, A, 3, c, 11, C, 18]

Input: [$]

Action: R,7 (use GOTO[3, C])

Stack: [0, A, 3, C, 9]

Input: [$]

Action: R,2 (use GOTO[0, S])

Stack: [0, S, 1]

Input: [$]

Action: accept

Input WAS accepted.

Pre-order traversal of parse tree generated:

S A A A a B c C d b a a C c C d

If you want to run the code yourself and see the output, in the main method I have the inputs of a,b,c,d,e from problem 1.

A picture containing shape

Description automatically generated

Just uncomment 1 of the “String[] inputString = …” lines you want to see and comment the rest.

And in case I can get credit for a,b,c,d,e with this program output and not doing it manually, here are the outputs for b,d,e which I haven’t shown yet:

input(b):

*accd*

output:

Stack: [0]

Input: [$, d, c, c, a]

Action: S,2

Stack: [0, a, 2]

Input: [$, d, c, c]

Action: S,6

Stack: [0, a, 2, c, 6]

Input: [$, d, c]

Action: S,14

Stack: [0, a, 2, c, 6, c, 14]

Input: [$, d]

Action: S,15

Stack: [0, a, 2, c, 6, c, 14, d, 15]

Input: [$]

Action: R,8 (use GOTO[14, C])

Stack: [0, a, 2, c, 6, c, 14, C, 19]

Input: [$]

Action: R,7 (use GOTO[6, C])

Stack: [0, a, 2, c, 6, C, 13]

Input: [$]

Action: R,7 (use GOTO[2, C])

Stack: [0, a, 2, C, 4]

Input: [$]

Action: R,1 (use GOTO[0, S])

Stack: [0, S, 1]

Input: [$]

Action: accept

Input WAS accepted.

Pre-order traversal of parse tree generated:

S a C c C c C d

input(d):

*acdbd*

output:

Stack: [0]

Input: [$, d, b, d, c, a]

Action: S,2

Stack: [0, a, 2]

Input: [$, d, b, d, c]

Action: S,6

Stack: [0, a, 2, c, 6]

Input: [$, d, b, d]

Action: S,15

Stack: [0, a, 2, c, 6, d, 15]

Input: [$, d, b]

Action: R,8 (use GOTO[6, C])

Stack: [0, a, 2, c, 6, C, 13]

Input: [$, d, b]

Action: R,6 (use GOTO[2, B])

Stack: [0, a, 2, B, 5]

Input: [$, d, b]

Action: S,12

Stack: [0, a, 2, B, 5, b, 12]

Input: [$, d]

Action: R,3 (use GOTO[0, A])

Stack: [0, A, 3]

Input: [$, d]

Action: S,7

Stack: [0, A, 3, d, 7]

Input: [$]

Action: R,8 (use GOTO[3, C])

Stack: [0, A, 3, C, 9]

Input: [$]

Action: R,2 (use GOTO[0, S])

Stack: [0, S, 1]

Input: [$]

Action: accept

Input WAS accepted.

Pre-order traversal of parse tree generated:

S A a B c C d b C d

input(e):

*abcdbad*

output:

Stack: [0]

Input: [$, d, a, b, d, c, b, a]

Action: S,2

Stack: [0, a, 2]

Input: [$, d, a, b, d, c, b]

Action: S,8

Stack: [0, a, 2, b, 8]

Input: [$, d, a, b, d, c]

Action: S,17

Stack: [0, a, 2, b, 8, c, 17]

Input: [$, d, a, b, d]

Action: S,22

Stack: [0, a, 2, b, 8, c, 17, d, 22]

Input: [$, d, a, b]

Action: R,8 (use GOTO[17, C])

Stack: [0, a, 2, b, 8, c, 17, C, 20]

Input: [$, d, a, b]

Action: R,6 (use GOTO[8, B])

Stack: [0, a, 2, b, 8, B, 16]

Input: [$, d, a, b]

Action: R,5 (use GOTO[2, B])

Stack: [0, a, 2, B, 5]

Input: [$, d, a, b]

Action: S,12

Stack: [0, a, 2, B, 5, b, 12]

Input: [$, d, a]

Action: R,3 (use GOTO[0, A])

Stack: [0, A, 3]

Input: [$, d, a]

Action: S,10

Stack: [0, A, 3, a, 10]

Input: [$, d]

Action: R,4 (use GOTO[0, A])

Stack: [0, A, 3]

Input: [$, d]

Action: S,7

Stack: [0, A, 3, d, 7]

Input: [$]

Action: R,8 (use GOTO[3, C])

Stack: [0, A, 3, C, 9]

Input: [$]

Action: R,2 (use GOTO[0, S])

Stack: [0, S, 1]

Input: [$]

Action: accept

Input WAS accepted.

Pre-order traversal of parse tree generated:

S A A a B b B c C d b a C d