Nguyen, Thomas
COEN 171
Professor Vu
October 9th, 2016

Homework #3

Problem Set

4.2 Perform the Pairwise Disjointness Test for the following grammar rules

a) S -> aSb | bAA

FIRST sets for the RHSs are {a}, {b}
This set is disjoint

b) A -> b{aB} | a

FIRST sets for the RHSs are {b}, {a}
This set is disjoint

c) B -> aB | a

FIRST sets for the RHSs are {a}, {a}
This set is not disjoint

4.4 Show a trace of the recursive descent parser given in Section 4.4.1 for the string a * (b + c).

I used the following Grammar:

<expr> -> <term> { (+ | -) <term> }
<term> -> <factor> { (* | /) <factor> |
<factor> -> <id> | int_constant | <expr>

Trace:

Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11
Next lexeme is a

Exit <factor>
Next token is: 23
Next lexeme is *

Enter <factor>
Next token is: 25
Next lexeme is (

Enter <expr>

Enter <term>
Enter <factor>
Next token is: 11
Next lexeme is b

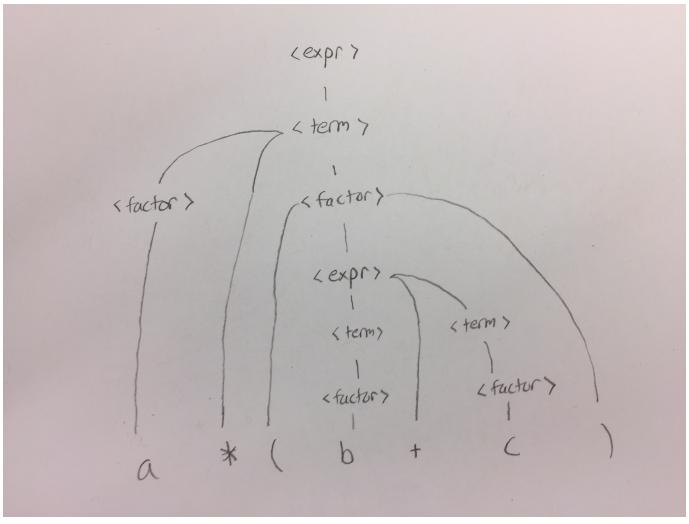
Exit <factor>
Exit <term>
Next token is: 21
Next lexeme is +

Enter <term>
Enter <factor>
Next token is: 11
Next lexeme is c

Exit <factor>
Exit <term>
Exit <expr>
Next token is: 23
Next lexeme is)

Exit <factor>
Exit <term>
Exit <expr>
Next token is: -1
Next lexeme is EOF

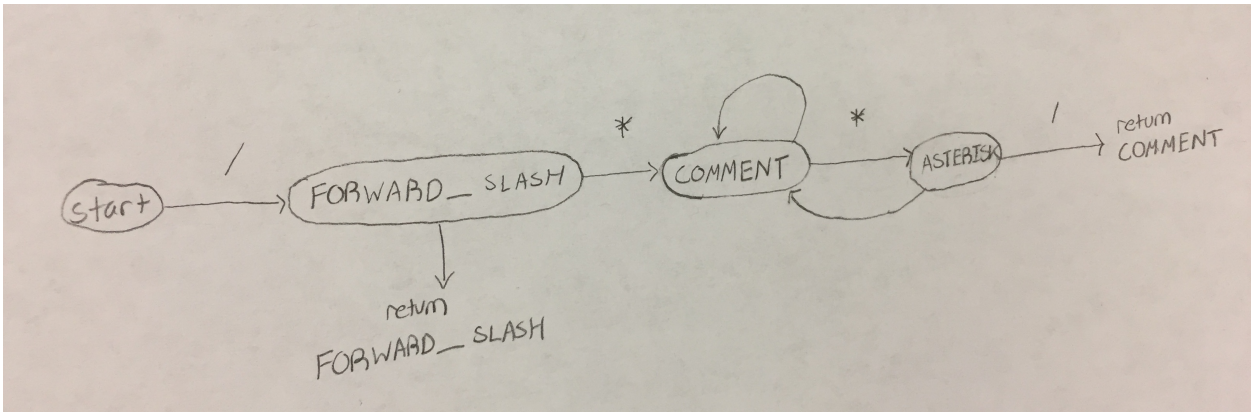
Parse Tree:



Programming Exercises

4.1 Design a state diagram to recognize one form of the comments of the C-based programming languages, those that begin with /* and end with */

This design state diagram assumes that on each arc, the program will call *getchar* and read in the next lexeme. The symbol on top of each arc represents the expected lexeme. If there is no expected lexeme then there is no symbol in correlation with the arc.



4.3 Write and test the code to implement the state diagram of Problem 1

```
// This function assumes that you will run this function after seeing a FORWARD_SLASH
int getComment() {
	getChar();

	// If not actually a comment, return that it was just a slash
	if(charClass != ASTERISK) {
		return FORWARD_SLASH;
	}
	else {
		// If it is a comment, traverse through the comment
		do {
			getChar();
			// Continue traversing until we see next asterisk
			while (charClass != ASTERISK) {
				getChar();
			}
			getChar();
		// If we see a slash, continue traversing … else we can stop traversing and return
		} while (charClass != FORWARD_SLASH);

		return COMMENT;
	}
}
```

4.6 Convert the lexical analyzer (which is written in C) given in Section 4.2 to Java

Received help working on this problem by referencing https://goo.gl/XgjlVA

```
public class front {

/* Global declarations */

/* variables */
   static int charClass;
   static String lexeme;
   static char nextChar;
   static int lexLen;
   static int token;
   static int nextToken;
   static File in_fp;
   static FileInputStream fis;

/* Character classes */
   public static final int LETTER = 0;
   public static final int DIGIT = 1;
   public static final int UNKNOWN = 99;

/* Token codes */
   public static final int INT_LIT = 10;
   public static final int IDENT = 11;
   public static final int ASSIGN_OP = 20;
   public static final int ADD_OP = 21;
   public static final int SUB_OP = 22;
   public static final int MULT_OP = 23;
   public static final int DIV_OP = 24;
   public static final int LEFT_PAREN = 25;
   public static final int RIGHT_PAREN = 26;


/*****************************************************************************************/


/* main driver */
   public void main() throws FileNotFoundException {

/* Open the input data file and process its contents */
      in_fp = new File("front.in");
      fis = new FileInputStream(in_fp);
      if (!(in_fp.exists())) {
         System.out.println("ERROR - cannot open front.in");
      } else {
         getChar();
         do {
            lex();
         } while (nextToken != -1);
      }
   }
```

```
/******************************************************************************************/

/* lookup - a function to lookup operators and parentheses and return the token */
   int lookup(char ch) {
      switch (ch) {
         case '(':
            addChar();
            nextToken = LEFT_PAREN;
            break;

         case ')':
            addChar();
            nextToken = RIGHT_PAREN;
            break;

         case '+':
            addChar();
            nextToken = ADD_OP;
            break;

         case '-':
            addChar();
            nextToken = SUB_OP;
            break;

         case '*':
            addChar();
            nextToken = MULT_OP;
            break;

         case '/':
            addChar();
            nextToken = DIV_OP;
            break;

         default:
            addChar();
            nextToken = -1;
            break;
      }
      return nextToken;
   }

/******************************************************************************************/

/* addChar - a function to add nextChar to lexeme */
   void addChar() {
      if (lexLen <= 98) {
```

```
        lexeme[lexLen++] = nextChar;
        lexeme[lexLen] = 0;
      } else
        System.out.println("Error - lexeme is too long");
   }

/*****************************************************************************************/

/* getChar - a function to get the next character of input and determine its character class */
   void getChar() {
      try {
        nextChar = (char) fis.read();
      } catch (IOException e) {
        e.printStackTrace();
      }
      if (nextChar != -1) {
        if (Character.isAlphabetic(nextChar))
          charClass = LETTER;
        else if (Character.isDigit(nextChar))
          charClass = DIGIT;
        else charClass = UNKNOWN;
      } else
        charClass = -1;
   }

/*****************************************************************************************/

/* getNonBlank - a function to call getChar until it returns a non-whitespace character */
void getNonBlank() {
      while (Character.isSpaceChar(nextChar))
        getChar();
   }


   /* lex - a simple lexical analyzer for arithmetic expressions */
   int lex() {
      lexLen = 0;
      getNonBlank();
      switch (charClass) {

/* Parse identifiers */
        case LETTER:
          addChar();
          getChar();
          while (charClass == LETTER || charClass == DIGIT) {
            addChar();
            getChar();
          }
          nextToken = IDENT;
```

```
            break;

/* Parse integer literals */
        case DIGIT:
            addChar();
            getChar();
            while (charClass == DIGIT) {
               addChar();
               getChar();
            }
            nextToken = INT_LIT;
            break;

/* Parentheses and operators */
        case UNKNOWN:
            lookup(nextChar);
            getChar();
            break;

/* null */
        case -1:
            nextToken = -1;
            lexeme[0] = 'E';
            lexeme[1] = 'O';
            lexeme[2] = 'F';
            lexeme[3] = 0;
            break;
      } /* End of switch */
      System.out.println("Next token is: "+nextToken+", Next lexeme is "+lexeme);
      return nextToken;
   } /* End of function lex */
}
```