**COVER  PAGE**

**CS323 Programming Assignments**

**Fill out all entries 1 - 7.    If not, there will be deductions!**

1.   Names [ 1. Tejas Ghalsasi                    ]

2. Assignment Number  [ 2 ]

3. Due Dates         **Softcopy**     [ 9th April ], **Hardcopy** [     9th April   ]

4. Turn-In  Dates **Softcopy**     [ 9th April ], **Hardcopy**  [    9th April ]

5. Executable FileName [  topDownParse                                              ]

  (**A file that can be executed without compilation by the instructor**)

6. LabRoom                    [                  ]

**(Execute your program in a lab in the CS building before submission)**

7. Operating System       [Windows      ]

https://docs.google.com/drawings/d/sRm_Glaiw3a6YndaGE_zZqA/image?w=600&h=1&rev=1&ac=1

**To be filled out by the Instructor:**

GRADE:

COMMENTS:

**CPSC 323 Project Documentation**

About 2-3 pages

1. **Problem Statement**
2. **CS323**
3. **Assignment 2**
4. Due (Soft copy: **Monday  4-9**, 11:59 pm,
5. Hard Copy: **Monday  4-9**, 5.30 pm)
7. The second assignment is to write a syntax analyzer using a top-down parser. You may use any top-down parser such as a RDP,  a predictive recursive descent parser or a table driven predictive parser.
8. **Hence, your assignment consists of the following tasks**:
9. 1. Rewrite the grammar Rat18S to remove any left recursion and backtracking. Use left factorization if necessary.
10. 2. Use the **lexer()**  generated in the assignment 1 to get the tokens
11. 3. **The parser should print to an output file the tokens, lexemes and the production rules  used;**
12. That is, **first, write the token and lexeme found**
13. Then, **print out all productions rules** used for analyzing this token
14. Note:  - a simple way to do it is to have a “print statement” at the beginning of each function that will print
15. the production rule.
16. - It would be a good idea to have a “switch” with the “print statement” so that you can turn it on or off.
17. 4. **Error handling:** if a syntax error occurs, your parser should generate a meaningful error message, such as token, lexeme, line number, and error type etc.
18. Then, your program may exit or you may continue for further analysis.
19. The bottom line is that your program must be able to parse the entire program if it is syntactically correct.
20. **5. Turn in your assignment according to the specifications given in the project outline**
21. **Example**
22. Assume we have the following statement
23. …..
24. a = b + c;
25. …….
26. ***One possible output would be as follows*:**
27. **Token: Identifier          Lexeme: a**
28. <Statement> -> <Assign>
29. <Assign> ->  <Identifier> = <Expression> ;
30. **Token: Operator          Lexeme: =**
31. **Token: Identifier          Lexeme: b**
32. <Expression> -> <Term> <Expression Prime>
33. <Term> -> <Factor> <Term Prime>
34. <Factor> -> <Identifier>
35. **Token:  Operator          Lexeme: +**
36. <Term Prime> -> ε
37. <Expression Prime> -> + <Term> <Expression Prime>
38. **Token:  Identifier           Lexeme: c**
39. <Term>  -> <Factor> <Term Prime>
40. <Factor> -> <Identifier>
41. **Token: Separator           Lexeme: ;**
42. <Term Prime> -> ε
43. <Expression Prime> -> ε
44. **How to use your program**

Enter the following commands:

javac \*.java

java topDownParse

1. **Design of your program**

I re-wrote my first assignment because we need a lexer input which sends the output forward. And my first assignment wasn’t really working correctly

The main method first calls the lexer and then calls the parser. I have created small static void functions which determine each parse stage

In the topdownParse program I have used 2 Queues in the form of linkedList. This is used coupled with a list of methods.

A method takes control to Go to < (<Opt Function Definitions>), @>

B method takes control to Go to <Opt Function Definitions>

C method takes control to Go to <Opt Declaration List>

D method takes control to Go to <Statement List>

The program is compiling and Running .

1. **Any Limitation**

The parser is giving the same output due to some reason.

1. **Any shortcomings**

As I rewrote my code entirely there were atleast a 150 errors which I resolved . But now the parser is giving the same output for almost all inputs.

topDownParse.java

//TEJAS GHALSASI

import java.util.\*;

import java.io.\*;

public class topDownParse

{

static String lexeme="";

static String token="";

static Queue <String> tokenQueue= new LinkedList<String>();

static Queue <String> lexemeQueue=new LinkedList<String>();

// <Rat18S> -> <Opt Function Definitions> %% <Opt Declaration List> <Statement List>

static void A()

{

    // If lexeme = First(<Opt Function Definitions>), @

    if (lexeme.compareToIgnoreCase("@") == 0)

    {

        System.out.println("<RAT18S> -> <Opt Function Definitions> %% <Opt Declaration List> <Statement List>");

        B();    // Go to <Opt Function Definitions>

    }

    else    // Error Message

    {

        System.out.println("ERROR at '" + lexeme + "'! @ is expected! <RAT18S>");

        // "<RAT18S> -> <Opt Function Definitions> %% <Opt Declaration List> <Statement List>" );

    }

    // If lexeme = Follow(<Opt Function Definitions>), %%

    if (lexeme.compareToIgnoreCase("%%") == 0)  // Read in %%

    {

        lexerQueue();   // Move to the next item in the queue.

        C();    // Go to <Opt Declaration List>

        D();    // Go to <Statement List>

    }

    else    // Error Message

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting %%. <RAT18S>");

    }

}

// <Opt Function Definitions> -> <Function Definitions> | <Empty>

static void B()

{

    // If lexeme = First(<Opt Function Definitions>), <Function Definitions>-> @

    if (lexeme.compareToIgnoreCase("@") == 0)

    {

        System.out.println("<Opt Function Definitions> -> <Function Definitions>");

        E();    // <Function Definitions>

    }

    // If lexeme = Follow(<Function Definitions>), %%

    else if (lexeme.compareToIgnoreCase("%%") == 0 )

    {

        System.out.println("<Opt Function Definitions> -> <Empty>" );

        F();    // <Empty>

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting @ or %%! <Opt Function Definitions>");

    }

}

// <Function Definitions> -> <Function> | <Function> <Function Definitions>

static void E()

{

    // If lexeme = First(<Function>), @

    if (lexeme.compareToIgnoreCase("@") == 0)

    {

        System.out.println("<Function Definitions> -> <Function>" );

        G();    // <Function>

        if (lexeme.compareToIgnoreCase("@") == 0)

        {

            System.out.println("<Function Definitions> -> <Function> <Function Definitions>" );

            E();    // <Function Definition>

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expected @. <Function Definitions>" );

    }

}

// <Function> -> @<Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>

static void G()

{

    if (lexeme.compareToIgnoreCase("@") == 0)

    {

        System.out.println("<Function> -> @<Identifier> (<Opt Declaration List>) <Opt Declaration List> <Body>" );

        lexerQueue();

        //H();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting @. <Function>" );

    }

    if (lexeme.compareToIgnoreCase("identifier") == 0)

    {

        lexerQueue();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier! <Function>" );

    }

    if (lexeme.compareToIgnoreCase("(") == 0)

    {

        lexerQueue();

        I();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting '('. <Function>" );

    }

    if (lexeme.compareToIgnoreCase(")") == 0)

    {

        lexerQueue();

        C();

        J();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting ')'. <Function>" );

    }

}

// <Opt Parameter List> -> <Parameter List> | <Empty>

static void I()

{

    // If lexeme is an identifier. First(K) = L-> M-> H(identifier)

    if (token.compareToIgnoreCase("identifier") == 0)

    {

        System.out.println("<Opt Parameter List> -> <Parameter List>" );

        K();

    }

    // If lexeme is part of Follow(I) = ")"

    else if (lexeme.compareToIgnoreCase(")") == 0)

    {

        System.out.println("<Opt Parameter List> -> <Empty>" );

        F();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier or ). <Opt Parameter List>" );

    }

}

// <Parameter List> -> <Parameter> | <Parameter>, <Parameter List>

static void K()

{

    // If lexeme is an identifier. First(L) = M-> H(identifier)

    if (token.compareToIgnoreCase("identifier") == 0)

    {

        System.out.println("<Parameter List> -> <Parameter>" );

        L();

        if (lexeme.compareToIgnoreCase(",") == 0)

        {

            System.out.println("<Parameter List> -> <Parameter>, <Parameter List>" );

            lexerQueue();

            K();

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier! <Parameter List>" );

    }

}

// <Parameter> -> <IDs> : <Qualifier>

static void L()

{

    // If lexeme is an identifier. First(M) = H(identifier)

    if (token.compareToIgnoreCase("identifier") == 0)

    {

        System.out.println("<Parameters> -> <IDs> : <Qualifier>" );

        M();

        if (lexeme.compareToIgnoreCase(":") == 0)

        {

            lexerQueue();

            N();

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier! <Parameter>" );

    }

}

// <Qualifier> -> "integer" | "boolean" | "floating"

static void N()

{

    if (lexeme.compareToIgnoreCase("integer") == 0 || lexeme.compareToIgnoreCase("true") == 0 ||

        lexeme.compareToIgnoreCase("false") == 0 || lexeme.compareToIgnoreCase("floating") == 0)

    {

        System.out.println("<Qualifier> -> " + lexeme );

        lexerQueue();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting either: integer, boolean, or floating." );

    }

}

// <Body> -> { <Statement List> }

static void J()

{

    if (lexeme.compareToIgnoreCase("{") == 0)

    {

        System.out.println("<Body> -> { <IDs> }" );

        lexerQueue();

        D();

        if(lexeme.compareToIgnoreCase("}") != 0)

        {

            System.out.println("ERROR at '" + lexeme + "'! } is expected! <Body>" );

        }

        else

        {

            lexerQueue();

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! { is expected! <Body>" );

    }

}

// <Opt Declaration List> -> <Declaration List> | <Empty>

static void C()

{

    // If lexeme is First(<Declaration List>) = int, bool, or float

    if (lexeme.compareToIgnoreCase("integer") == 0 || lexeme.compareToIgnoreCase("true") == 0 ||

        lexeme.compareToIgnoreCase("false") == 0 || lexeme.compareToIgnoreCase("floating") == 0)

    {

        System.out.println("<Opt Declaration List> -> <Declaration List>" );

        O();

    }

    // if lexeme is part of the Follow(<Opt Declaration List>)

    else if (lexeme.compareToIgnoreCase("{") == 0 || token.compareToIgnoreCase("identifier") == 0 ||

        lexeme.compareToIgnoreCase("if") == 0 || lexeme.compareToIgnoreCase("return") == 0 ||

        lexeme.compareToIgnoreCase("write") == 0 || lexeme.compareToIgnoreCase("read") == 0 ||

        lexeme.compareToIgnoreCase("while") == 0)

    {

        System.out.println("<Opt Declaration List> -> <Empty>" );

        F();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting integer, boolean, floating, {, if, while, read, write, true, or false! <Opt Declaration List>" );

    }

}

// <Declaration List> -> <Declaration>; | <Declaration>; <Declaration List>

static void O()

{

    if (lexeme.compareToIgnoreCase("integer") == 0 || lexeme.compareToIgnoreCase("true") == 0 ||

        lexeme.compareToIgnoreCase("false") == 0 || lexeme.compareToIgnoreCase("floating") == 0)

    {

        System.out.println("<Declaration List> -> <Declaration>;" );

        P();

        if (lexeme.compareToIgnoreCase(";") == 0)

        {

            System.out.println("<Declaration List> -> <Declaration>; <Declaration List>" );

            lexerQueue();

            O();

        }

    }

}

// <Declaration> -> <Qualifier> <IDs>

static void P()

{

    if (lexeme.compareToIgnoreCase("integer") == 0 || lexeme.compareToIgnoreCase("boolean") == 0 ||

        lexeme.compareToIgnoreCase("floating") == 0)

    {

        System.out.println("<Declaration> -> <Qualifier> <IDs>" );

        N();

        M();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting integer, boolean, or floating! <Declaration>" );

    }

}

// <IDs> -> <Identifier> | <Identifier>, <IDs>

static void M()

{

    if (token.compareToIgnoreCase("identifier") == 0)

    {

        System.out.println("<IDs> -> <Identifier>" );

        lexerQueue();

        if (lexeme.compareToIgnoreCase(",") == 0)

        {

            System.out.println("<IDs> -> <Identifier>, <IDs>" );

            lexerQueue();

            M();

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier! <IDs>" );

    }

}

// <Statement List> -> <Statement> | <Statement> <Statement List>

static void D()

{

    if (lexeme.compareToIgnoreCase("{") == 0 || token.compareToIgnoreCase("identifier") == 0 ||

        lexeme.compareToIgnoreCase("if") == 0 || lexeme.compareToIgnoreCase("return") == 0 ||

        lexeme.compareToIgnoreCase("write") == 0 || lexeme.compareToIgnoreCase("read") == 0 ||

        lexeme.compareToIgnoreCase("while") == 0)

    {

        System.out.println("<Statement List> -> <Statement>" );

        Q();

        if (lexeme.compareToIgnoreCase("{") == 0 || token.compareToIgnoreCase("identifier") == 0 ||

            lexeme.compareToIgnoreCase("if") == 0 || lexeme.compareToIgnoreCase("return") == 0 ||

            lexeme.compareToIgnoreCase("write") == 0 || lexeme.compareToIgnoreCase("read") == 0 ||

            lexeme.compareToIgnoreCase("while") == 0)

        {

            System.out.println("<Statement List> -> <Statement> <Statement List>" );

            D();

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifer or either: {, if, return, write, read, or while! <Statement List>" );

    }

}

// <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>

static void Q()

{

    if (lexeme.compareToIgnoreCase("{") == 0 || token.compareToIgnoreCase("identifier") == 0 ||

        lexeme.compareToIgnoreCase("if") == 0 || lexeme.compareToIgnoreCase("return") == 0 ||

        lexeme.compareToIgnoreCase("write") == 0 || lexeme.compareToIgnoreCase("read") == 0 ||

        lexeme.compareToIgnoreCase("while") == 0)

    {

        if (lexeme.compareToIgnoreCase("{") == 0)

        {

            System.out.println("<Statement> -> <Compound>" );

            R();

        }

        else if (token.compareToIgnoreCase("identifier") == 0)

        {

            System.out.println("<Statement> -> <Assign>" );

            S();

        }

        else if (lexeme.compareToIgnoreCase("if") == 0)

        {

            System.out.println("<Statement> -> <If>" );

            T();

        }

        else if (lexeme.compareToIgnoreCase("return") == 0)

        {

            System.out.println("<Statement> -> <Return>" );

            U();

        }

        else if (lexeme.compareToIgnoreCase("write") == 0)

        {

            System.out.println("<Statement> -> <Write>" );

            V();

        }

        else if (lexeme.compareToIgnoreCase("while") == 0)

        {

            System.out.println("<Statement> -> <While>" );

            X();

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier or either: {, if, return, write, or while! <Statement>" );

    }

}

// <Compound> -> { <Statement List> }

static void R()

{

    if (lexeme.compareToIgnoreCase("{") == 0)

    {

        System.out.println("<Compound> -> { <Statement List> }" );

        lexerQueue();

        D();

        if (lexeme.compareToIgnoreCase("}") != 0)

        {

            System.out.println("ERROR at '" + lexeme + "'! Expecting a }. <Compound>" );

        }

        else

        {

            lexerQueue();

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting a {. <Compound>" );

    }

}

// <Assign> -> <Identifier> := <Expression> ;

static void S()

{

    if (token.compareToIgnoreCase("identifier") == 0)

    {

        System.out.println("<Assign> -> <Identifier>" );

        lexerQueue();

        if (lexeme.compareToIgnoreCase(":=") == 0)

        {

            lexerQueue();

            Z();

            if (lexeme.compareToIgnoreCase(";") != 0)

            {

                System.out.println("ERROR at " + lexeme + "! Expecting ; <Assign>" );

            }

            else

            {

                lexerQueue();

            }

        }

        else

        {

            System.out.println("ERROR at " + lexeme + "! Expecting := <Assign>" );

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier. <Assign>" );

    }

}

// <If> -> if (<Condition>) <Statement> fi | if (<Condition>) <Statement> else <Statement> fi

static void T()

{

    if (lexeme.compareToIgnoreCase("if") == 0)

    {

        System.out.println("<If> -> if (<Condition>) <Statement>" );

        lexerQueue();

        if (lexeme.compareToIgnoreCase("(") == 0)

        {

            lexerQueue();

            Y();

            if (lexeme.compareToIgnoreCase(")") == 0)

            {

                lexerQueue();

                Q();

                if (lexeme.compareToIgnoreCase("fi") == 0)

                {

                    System.out.println("<ID> -> if (<Condition>) <Statement> fi" );

                    lexerQueue();

                }

                else if (lexeme.compareToIgnoreCase("else") == 0)

                {

                    System.out.println("if (<Condition>) <Statement> else <Statement> fi" );

                    lexerQueue();

                    Q();

                    if (lexeme.compareToIgnoreCase("fi") == 0)

                    {

                        lexerQueue();

                    }

                    else

                    {

                        System.out.println("ERROR at '" + lexeme + "'! Expecting fi. <If>" );

                    }

                }

                else

                {

                    System.out.println("ERROR at '" + lexeme + "'! Expecting fi or else. <If>" );

                }

            }

            else

            {

                System.out.println("ERROR at '" + lexeme + "'! Expecting an ). <If>" );

            }

        }

        else

        {

            System.out.println("ERROR at '" + lexeme + "'! Expecting an (. <If>" );

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an if. <If>" );

    }

}

// <Return> -> return; | return <Expression>;

static void U()

{

    lexerQueue();

    if (lexeme.compareToIgnoreCase(";") == 0)

    {

        System.out.println("<Return> -> return;" );

        lexerQueue();

    }

    else

    {

        System.out.println("<Return> -> return <Expression>;" );

        Z();

        if (lexeme.compareToIgnoreCase(";") != 0)

        {

            System.out.println("ERROR at '" + lexeme + "'! Expecting ';'. <Return>" );

        }

        else

        {

            lexerQueue();

        }

    }

}

// <Write> -> write (<Expression>);

static void V()

{

    if (lexeme.compareToIgnoreCase("write") == 0)

    {

        System.out.println("<Write> -> write (<Expression>);" );

        lexerQueue();

        if (lexeme.compareToIgnoreCase("(") == 0)

        {

            Z();

            if (lexeme.compareToIgnoreCase(")") != 0)

            {

                System.out.println("ERROR at '" + lexeme + "'! Expecting a ) statement! <Write>" );

            }

            else

            {

                lexerQueue();

                if (lexeme.compareToIgnoreCase(";") != 0)

                {

                    System.out.println("ERROR at '" + lexeme + "'! Expecting a ; ! <Write>" );

                }

                else

                {

                    lexerQueue();

                }

            }

        }

        else

        {

            System.out.println("ERROR at '" + lexeme + "'! Expecting a ( ! <Write>" );

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting a write statement! <Write>" );

    }

}

// <Read> -> read (<IDs>)

static void W()

{

    if (lexeme.compareToIgnoreCase("read") == 0)

    {

        System.out.println("<Read> -> read (<IDs>)" );

    lexerQueue();

        if (lexeme.compareToIgnoreCase("(")!= 0)

        {

            lexerQueue();

            M();

            if (lexeme.compareToIgnoreCase(")") != 0)

            {

                System.out.println("ERROR at '" + lexeme + "'! Expecting ) ! <Read>" );

            }

            else

            {

                lexerQueue();

            }

        }

        else

        {

            System.out.println("ERROR at '" + lexeme + "'! Expecting ( ! <Read>" );

        }

    }

    else

    {

        System.out.println("ERROR at " + lexeme + "! Expecting a read statement! <Read>" );

        }

}

// <While> -> while (<Condition>) <Statement>

static void X()

{

    if (lexeme.compareToIgnoreCase("while") == 0)

    {

        System.out.println("<While> -> while (<Condition>) <Statement>" );

        lexerQueue();

        if (lexeme.compareToIgnoreCase("(") == 0)

        {

            lexerQueue();

            Y();

            if(lexeme.compareToIgnoreCase(")") == 0)

            {

                lexerQueue();

                Q();

            }

            else

            {

                System.out.println("ERROR at '" + lexeme + "'! Expecting a ')' ! <While>" );

            }

        }

        else

        {

            System.out.println("ERROR at '" + lexeme + "'! Expecting a '(' ! <While>" );

        }

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting a while statement! <While>" );

    }

}

// <Condition> -> <Expression> <Relop> <Expression>

static void Y()

{

    if (token.compareToIgnoreCase("identifier") == 0 || token.compareToIgnoreCase( "integer") == 0 ||

        token.compareToIgnoreCase("(") == 0 || token.compareToIgnoreCase("real") == 0 ||

        lexeme.compareToIgnoreCase("true") == 0 ||lexeme.compareToIgnoreCase("false") == 0)

    {

        System.out.println("<Condition> -> <Expression> <Relop> <Expression>" );

        Z();

        a();

        Z();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier, integer, real, or statements: integer, (, true, or false! <Condition>" );

    }

}

// <Relop> -> := | '=' | '/=' | '>' | '<' | '=>' | '<='

static void a()

{

    if (lexeme.compareToIgnoreCase(":=") == 0 || lexeme.compareToIgnoreCase("/=") == 0 ||

        lexeme.compareToIgnoreCase(">") == 0 || lexeme.compareToIgnoreCase("<") == 0 ||

        lexeme.compareToIgnoreCase("=>") == 0 || lexeme.compareToIgnoreCase("<=") == 0)

    {

        System.out.println("<Relop> -> " + lexeme );

        lexerQueue();

    }

    else

    {

        System.out.println("ERROR at " + lexeme + "'! Expecting either: :=, /=, >, <, =>, or <=. <Relop>" );

    }

}

/\* <Expression> -> <Expression> + <Term> | <Expression> - <Term> | <Term>

<Expression> -> <Term> <Expression Prime> \*/

static void Z()

{

    if (token.compareToIgnoreCase("identifier") == 0 || token.compareToIgnoreCase("integer") == 0 ||

        lexeme.compareToIgnoreCase("(") == 0 || lexeme.compareToIgnoreCase("real") == 0 ||

        lexeme.compareToIgnoreCase("true") == 0 || lexeme.compareToIgnoreCase("false") == 0)

    {

        System.out.println("<Expression> -> <Term> <Expression Prime>" );

        b();

        Zp();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier, integer, real, or either: (, true, or false. <Expression>" );

    }

}

// <Expression Prime> -> + <Term> <Expresion Prime> | - <Term> <Expression Prime> | Є

static void Zp()

{

    if (lexeme.compareToIgnoreCase("+") == 0 || lexeme.compareToIgnoreCase("-") == 0)

    {

        System.out.println("<Expression Prime> -> " + lexeme + " <Term> <Expression Prime>" );

        lexerQueue();

        b();

        Zp();

    }

    else if (lexeme.compareToIgnoreCase(";") == 0 || lexeme.compareToIgnoreCase(")") == 0 ||

        lexeme.compareToIgnoreCase(":=") == 0 || lexeme.compareToIgnoreCase("/=") == 0 ||

        lexeme.compareToIgnoreCase(">") == 0 || lexeme.compareToIgnoreCase("<") == 0 ||

        lexeme.compareToIgnoreCase("=>") == 0 || lexeme.compareToIgnoreCase("<=") == 0)

    {

        System.out.println("<Expression Prime> -> <Empty>" );

        F();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting +, -, ;, ), or a! <Expression Prime>" );

    }

}

/\* <Term> -> <Term> \* <Factor> | <Term> / <Factor> | <Factor>

<Term> -> <Factor> <Term Prime> \*/

static void b()

{

    if (token.compareToIgnoreCase("identifier") == 0 || token.compareToIgnoreCase("integer") == 0 ||

        lexeme.compareToIgnoreCase("(") == 0 || lexeme.compareToIgnoreCase("real") == 0 ||

        lexeme.compareToIgnoreCase("true") == 0 || lexeme.compareToIgnoreCase("false") == 0)

    {

        System.out.println("<Term> -> <Factor> <Term Prime>" );

        c();

        bp();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting an identifier, integer, real, or either: (, true, or false! <Term>" );

    }

}

// <Term Prime> -> \* <Factor> <Term Prime> | / <Factor> <Term Prime> | Є

static void bp()

{

    if (lexeme.compareToIgnoreCase("\*") == 0 || lexeme.compareToIgnoreCase("/") == 0)

    {

        System.out.println("<Expression Prime> -> " + lexeme + " <Term> <Expression Prime>" );

        lexerQueue();

    }

    else if (lexeme.compareToIgnoreCase("+") == 0 || lexeme.compareToIgnoreCase("-") == 0 ||

        lexeme.compareToIgnoreCase(":=") == 0 || lexeme.compareToIgnoreCase("/=") == 0 ||

        lexeme.compareToIgnoreCase(">") == 0 || lexeme.compareToIgnoreCase("<") == 0 ||

        lexeme.compareToIgnoreCase("=>") == 0 || lexeme.compareToIgnoreCase("<=") == 0 ||

        lexeme.compareToIgnoreCase(")") == 0 || lexeme.compareToIgnoreCase(";") == 0)

    {

        System.out.println("<Term Prime> -> <Empty>" );

        F();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting either \* or /. <Term Prime>" );

    }

}

// <Factor> -> - <Primary> | <Primary>

static void c()

{

    if (lexeme.compareToIgnoreCase("-") == 0)

    {

        System.out.println("<Factor> -> - <Primary>" );

        lexerQueue();

        d();

    }

    else if(token.compareToIgnoreCase("identifier") == 0 || token.compareToIgnoreCase("integer") == 0 ||

        lexeme.compareToIgnoreCase("(") == 0 || lexeme.compareToIgnoreCase("real") == 0 ||

        lexeme.compareToIgnoreCase("true") == 0 || lexeme.compareToIgnoreCase("false") == 0)

    {

        System.out.println("<Factor> -> <Primary>" );

        d();

    }

    else

    {

        System.out.println("ERROR at '" + lexeme + "'! Expecting either an identifier, real, or either: -, (, integer, true, or false! <Factor>" );

    }

}

// <Primary> -> <Identifier> | <Integer> | <Identifier>[<IDs>] | (<Expression>) | <Real> | "true" | "false"

static void d()

{

    if (token.compareToIgnoreCase("identifier") == 0)

    {

        System.out.println("<Primary> -> <Identifier>" );

        lexerQueue();

        if (lexeme.compareToIgnoreCase("[") == 0)

        {

            System.out.println("<Primary> -> <Identifier> [ <IDs> ]" );

            lexerQueue();

            M();

            if (lexeme.compareToIgnoreCase("]") != 0)

            {

                System.out.println("ERROR at " + lexeme + "! Expecting ]. <Primary>" );

            }

            else

            {

                lexerQueue();

            }

        }

    }

    else if (lexeme.compareToIgnoreCase("(") == 0)

    {

        System.out.println("<Primary> -> (<Expression>)" );

        Z();

        // Checking for ending paranthesis.

        if (lexeme.compareToIgnoreCase(")") != 0)

        {

            System.out.println("ERROR at '" + lexeme + "'! Expecting a ). <Primary>" );

        }

        else

        {

            lexerQueue();

        }

    }

    else if (token.compareToIgnoreCase("integer") == 0)

    {

        System.out.println("<Primary> -> <Integer>" );

        lexerQueue();

    }

    else if (lexeme.compareToIgnoreCase("true") == 0)

    {

        System.out.println("<Primary> -> true" );

        lexerQueue();

    }

    else if (lexeme.compareToIgnoreCase("false") == 0)

    {

        System.out.println("<Primary> -> false" );

        lexerQueue();

    }

    else

    {

        System.out.println("ERROR at " + lexeme + "! Expecting identifier or integer or either: (, true, or false! <Primary>" );

    }

}

// <Empty> -> Є

static void F()

{

    System.out.println("<Empty> -> Epsilon" );

}

static void lexerQueue()

{

    if (!tokenQueue.isEmpty() && !lexemeQueue.isEmpty())

    {

        token = tokenQueue.peek();

        lexeme = lexemeQueue.peek();

        tokenQueue.poll();

        lexemeQueue.poll();

        if (token.compareToIgnoreCase("keyword") == 0 || token.compareToIgnoreCase("operator") == 0 ||

            token.compareToIgnoreCase("integer") == 0)

        {

            System.out.println("\nTOKEN: " + token + "\t\tLEXEME: " + lexeme );

        }

        else

        {

            System.out.println("\nTOKEN: " + token + "\tLEXEME: " + lexeme );

        }

    }

}

static void parse()

{

Scanner sc=null;

try

        {

sc = new Scanner( new File("abc.txt"));

}

        catch(Exception e)

        {

            e.printStackTrace();

        }

    lexerQueue();

    A();

    sc.close();

}

public static void main(String args[])

{

    lexicalAnalyzer a=new lexicalAnalyzer();

    a.lexer();

    parse();

}

}

lexicalAnalyzer.java

import java.util.\*;

import java.io.\*;

/\*

Tejas Ghalsasi

This class is a lexical analyzer i have re written.

\*/

public class lexicalAnalyzer

{

Queue <String> tokenQueue= new LinkedList<String>();

Queue <String> lexemeQueue=new LinkedList<String>();

public static BufferedReader inStream;

public static PrintWriter outStream;

    public static int a;

    public static char c;

    public lexicalAnalyzer()

    {

        setIO("abc.txt","xyz.txt");

    }

    public static int getNextChar()

    {

        int ret=0;

        try

        {

            return inStream.read();

        }

        catch(IOException e)

        {

            e.printStackTrace();

        }

        return ret;

    }

    public static int getChar()

    {

        int i = getNextChar();

        while ( Character.isWhitespace((char) i) )

            i = getNextChar();

        return i;

    }

    public static void display(String s)

    {

        setIO("abc.txt","xyz.txt");

        outStream.print(s);

    }

    public static void displayln(String s)

    {

        setIO("abc.txt","xyz.txt");

        outStream.println(s);

    }

    public static void setIO(String inFile, String outFile)

    {

        try

        {

            inStream = new BufferedReader( new FileReader(inFile) );

            outStream = new PrintWriter( new FileOutputStream(outFile) );

            a = inStream.read();

        }

        catch(FileNotFoundException e)

        {

            e.printStackTrace();

        }

        catch(IOException e)

        {

            e.printStackTrace();

        }

    }

    public static void closeIO()

    {

        try

        {

            inStream.close();

            outStream.close();

        }

        catch(IOException e)

        {

            e.printStackTrace();

        }

    }

boolean seperator(char input[], int length)

{

    int numSep = 10;

String temp= "();:,{}[]@%";

    char seperators[]=temp.toCharArray();

    boolean sep = false;

    if (length < 2)

    {

        for (int i = 0; i < numSep; ++i)

        {

            if (seperators[i] == input[0])

            {

                sep = true;

                break;

            }

        }

    }

    else

    {

        if (input[0] == '%' && input[1] == '%')

        {

            sep = true;

        }

    }

    return sep;

}

boolean operators(char input[], int length)

{

    int numOps = 7;

    char operators[] = { '<', '>', '+', '-', '\*', '/', '=' };

    boolean oper = false;

    if (length == 2)

    {

        if ((input[0] == ':' && input[1] == '=') ||

            (input[0] == '=' && input[1] == '=') ||

            (input[0] == '+' && input[1] == '+') ||

            (input[0] == '=' && input[1] == '>') ||

            (input[0] == '<' && input[1] == '=') ||

            (input[0] == '/' && input[1] == '='))

        {

            oper = true;

        }

    }

    else

    {

        for (int i = 0; i < numOps; ++i)

        {

            if (operators[i] == input[0])

            {

                oper = true;

                break;

            }

        }

    }

    return oper;

}

boolean keyword(char input[])

{

     int numKey = 13;

    String keywords[] = { "integer", "if", "else", "fi", "while",

        "return", "read", "write", "for", "true", "false", "Boolean",

        "floating" };

    boolean key = false;    // Initial state

String inputString=new String(input);

    for (int i = 0; i < numKey; ++i)

    {

        if (keywords[i].equals(inputString))

        {

            key = true;

            break;

        }

    }

    return key;

}

boolean idFSM(/\*String input\*/ char input[], int length)

{

    int state = 0;

    int col;

    int table[][] =new int[][] {    // q0 = 1; F: {1, 3, 4}

                        // l #

                    /\*0\*/{ 1, 2 },

                    /\*1\*/{ 1, 3 },

                    /\*2\*/{ 2, 2 },

                    /\*3\*/{ 4, 2 },

                    /\*4\*/{ 4, 5 },

                    /\*5\*/{ 4, 2 } };

    for (int i = 0; i < length; ++i)

    {

        if (Character.isLetter(input[i]))

        {

            col = 0;

            state = table[state][col];

        }

        else if (input[i] == '#')

        {

            col = 1;

            state = table[state][col];

        }

        else

        {

            state = 2;

        }

    }

    if (state == 1 || state == 3 || state == 4)

    {

        return true;

    }

    else

    {

        return false;

    }

}

int numFSM(char input[], int length)

{

    int state = 1;

    int col;

    int table[][] = new int[][]{    // q0 = 1; F: {1(int), 4(real)}

                        // d .

                    /\*0\*/{ 1, 2 },

                 /\*1\*/{ 1, 3 },

                 /\*2\*/{ 2, 2 },  // Invalid State row

                    /\*3\*/{ 4, 2 },

                    /\*4\*/{ 4, 2 } };

    for (int i = 0; i < length; ++i)

    {

        if (Character.isDigit(input[i]))

        {

            col = 0;

            state = table[state][col];

        }

        else if (input[i] == '.')

        {

            col = 1;

            state = table[state][col];

        }

        else

        {

            state = 2;

        }

    }

    return state;

}

void lexer()

{

    char c=' ';

    char input[]=new char[1000];

    //char seperators[] = "();:,";

    boolean firstSep = false;

    String firstSeperator="";

    boolean lastSep = false;

    String lastSeperator="";

    int i = 0;

    File inFile = new File("abc.txt");

Scanner sc=null;

    displayln("TOKEN\t\t\tLEXEME");

try

        {

sc = new Scanner( inFile);

        }

        catch(IOException e)

        {

            e.printStackTrace();

        }

while (sc.hasNextLine())

    {

        c +=sc.next().charAt(0);

        if (c == ' ' || c == '\n' || c == '\t')

        {

            input[i] = '\0';

            int inputLength = i;

            i = 0;

            if (inputLength > 0)

            {

                if (firstSep == true)

                {

                    displayln("seperator \t\t"+firstSeperator);

                    System.out.println( "seperator \t\t"+firstSeperator);

                    for (int j = 0; j < inputLength; ++j)

                    {

                        input[j] = input[j + 1];

                    }

                    input[inputLength] = '\0';

                    inputLength--;

                    input=(firstSeperator).toCharArray();

                    tokenQueue.add("separator");

                    lexemeQueue.add(""+input);

                    firstSep = false;

                }

                if (input[inputLength - 1] == ')' ||

                    input[inputLength - 1] == ';' ||

                    input[inputLength - 1] == ',')

                {

                    lastSeperator =""+input[inputLength - 1];

                    input[inputLength - 1] = '\0';

                    inputLength--;

                    lastSep = true;

                }

                if (seperator(input, inputLength) == true)

                {

                    displayln("seperator \t\t");

                    System.out.print( "seperator \t\t");

                    for (int j = 0; j < inputLength; ++j)

                    {

                        System.out.print( input[j]);

                    }

                    System.out.println();

                    displayln(" ");

                    String inp=new String(input);

                    tokenQueue.add("separator");

                    lexemeQueue.add(inp);

                }

                // check if operator.

                else if (operators(input, inputLength) == true)

                {

                    displayln("operator \t\t");

                    System.out.print( "operator \t\t");

                    for (int j = 0; j < inputLength; ++j)

                    {

                        System.out.print( input[j]);

                        //display(input[j]);

                    }

                    System.out.println();

                    displayln("");

                    String inp=new String(input);

                    tokenQueue.add("operator");

                    lexemeQueue.add(inp);

                }

                // check if keyword.

                else if (keyword(input) == true)

                {

                    System.out.println( input + " is a keyword");

                    displayln("keyword \t\t");

                    System.out.print( "keyword \t\t");

                    for (int j = 0; j < inputLength; ++j)

                    {

                        System.out.print( input[j]);

                    //  display(input[j]);

                    }

                    displayln("");

                    System.out.println();

                    String inp=new String(input);

                    tokenQueue.add("keyword");

                    lexemeQueue.add(inp);

                    if (lastSep == true)

                    {

                        displayln("seperator \t\t"+lastSeperator);

                        System.out.println("seperator \t\t"+lastSeperator);

                        lastSep = false;

                        inp=(lastSeperator);

                        tokenQueue.add("separator");

                        lexemeQueue.add(inp);

                    }

                }

                // check if identifier

                else if (idFSM(input, inputLength) == true)

                {

                        System.out.print( "identifier \t\t");

                    for (int j = 0; j < inputLength; ++j)

                    {

                        System.out.print( input[j]);

                    }

                    System.out.println();

                    displayln("");

                    String inp=new String(input);

                    tokenQueue.add("identifier");

                    lexemeQueue.add(inp);

                    if (lastSep == true)

                    {

                        displayln("seperator \t\t"+lastSeperator);

                        System.out.print( "seperator \t\t"+lastSeperator);

                        lastSep = false;

                        inp=(lastSeperator);

                        tokenQueue.add("separator");

                        lexemeQueue.add(inp);

                    }

                }

                // check if int

                else if (numFSM(input, inputLength) == 1)

                {

                    display("integer \t\t");

                    System.out.print( "integer \t\t");

                    for (int j = 0; j < inputLength; ++j)

                    {

                        System.out.print( input[j]);

                        //display(input[j]);

                    }

                    displayln("");

                    System.out.println();

                    String inp=new String(input);

                    tokenQueue.add("integer");

                    lexemeQueue.add(inp);

                    if (lastSep == true)

                    {

                        displayln("seperator \t\t"+ lastSeperator);

                        System.out.println( "seperator \t\t"+ lastSeperator);

                        lastSep = false;

                        inp=(lastSeperator);

                        tokenQueue.add("separator");

                        lexemeQueue.add(inp);

                    }

                }

                // check if real

                else if (numFSM(input, inputLength) == 4)

                {

                    display("real number \t\t");

                    System.out.print( "real number \t\t");

                    for (int j = 0; j < inputLength; ++j)

                    {

                        System.out.print( input[j]);

                        //display(input[j]);

                    }

                    displayln("");

                    System.out.println();

                    String inp=new String(input);

                    tokenQueue.add("real");

                    lexemeQueue.add(inp);

                    if (lastSep == true)

                    {

                        displayln("separator \t\t");

                        System.out.println( "separator \t\t");

                        inp=(lastSeperator);

                        tokenQueue.add("seperator");

                        lexemeQueue.add(inp);

                        lastSep = false;

                    }

                }

                else

                {

                    display("invalid \t\t");

                    System.out.print("invalid \t\t");

                    for (int j = 0; j < inputLength; ++j)

                    {

                        System.out.print(input[j]);

                    }

displayln(" ");

                    System.out.println();

                    String inp=""+(input);

                    tokenQueue.add("invalid");

                    lexemeQueue.add(inp);

                }

            }

            // Empty input char array.

            for (int j = 0; j < inputLength; ++j)

            {

                input[j] = '\0';

            }

        }

        else

        {

            // Put the chars c is getting into the char array c.

            input[i] = c;

            i++;

            // Check if there is a '(' or '[' at the start of input.

            if (input[0] == '(')

            {

                // If true, set firstSep to true and save it to firstSeperator for later.

                firstSep = true;

                firstSeperator =""+input[0];

            }

        }

    }

}

}