```
options {
        STATIC = false;
}

PARSER_BEGIN(Analisador)
package compilador.analisadorsintatico;

import java.io.*;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import java.lang.reflect.*;
import compilador.recovery.*;

public class Analisador {
        private static StringBuilder resultado = new StringBuilder();
        private static Boolean houveErro = false;
        private static Boolean houveErroSintatico = false;

        public StringBuilder syntacticErrors = new StringBuilder();
        boolean debug_recovery = true;
        boolean eof;
        Token lastError = null;
        public int contParseError = 0;

        public static void main(String args[]) throws ParseException {
                Analisador parser = new Analisador(System.in);
                resultado = new StringBuilder();

                parser.iniciarCompilador(parser, args, "");
        }

        public Boolean checkCategoriaNome(String categoriaNome) {
                return categoriaNome.equals("DEFAULT") ||
categoriaNome.equals("COMENTARIODELINHA") ||
categoriaNome.equals("COMENTARIODEBLOCO");
        }

        public String getLexemaCategoriaNome(int categoriaNumero) {
                String returnValue = "INVALIDO";
                Field f[] =
AnalisadorConstants.class.getDeclaredFields();

                for (int i = 0; i < f.length; i++) {
                        try {
                                if
(f[i].get(f[i]).equals(categoriaNumero)) {
                                        String categoriaNome =
f[i].getName();
```

```
                                        if
(checkCategoriaNome(categoriaNome)) {
                                                continue;

                                        } else {
                                                returnValue =
f[i].getName();
                                        }
                                }
                        } catch (IllegalAccessException e) {}
                }

                return returnValue;
        }

        public void PrintLexema(Token t) {
                String categoriaNome = getLexemaCategoriaNome(t.kind);

                if (categoriaNome.indexOf("PALAVRA_RESERVADA") > -1) {
                        categoriaNome = "PALAVRA_RESERVADA";
                } else if (categoriaNome.indexOf("SIMBOLO_ESPECIAL") >
-1) {
                        categoriaNome = "SIMBOLO_ESPECIAL";
                }

                switch (categoriaNome) {
                        case "ENTRADA_INVALIDA":
                                printar("ERRO DE LEXEMA: | " + t + " | na
linha " + t.beginLine + " e na coluna " +
                                        t.beginColumn + ". Entrada
invalida, tente novamente.");
                                    setHouveErroLexico();
                                break;
                        case "EOF":
                                printar("Lexema <EOF> indicando o fim do
arquivo encontrado na linha " +
                                        t.beginLine + " e na coluna " +
t.beginColumn);
                                break;
                        default:
                                printar("Lexema: | " + t + " | encontrado
na linha " + t.beginLine + " e na coluna " + t.beginColumn + " da
categoria " + categoriaNome + " com o numero " + t.kind);
                                break;
                }
        }

        public void printar(String mensagem) {
                resultado.append(mensagem + "\n");
```

```
        }

        public static void printarStatic(String mensagem) {
                resultado.append(mensagem + "\n");
        }

        public static void resetHouveErroLexico() {
        Analisador.houveErro = false;
    }

        public static void setHouveErroLexico() {
        Analisador.houveErro = true;
        }

        public static void setHouveErroSintatico(Boolean valor) {
            Analisador.houveErroSintatico = valor;
        }

        public void analisadorLexico() {
                try {
                        Token t = null;
                        t = getNextToken();

//                      if (t.kind == EOF) {
//                              printar("Insira algum lexema para
compilar.");
//                              return;
//                      }

                        while (t.kind != EOF) {
                                String nomeToken = tokenImage[t.kind];
//                              PrintLexema(t);
                                t = getNextToken();
                        }

//                      PrintLexema(t);
                } catch (Error e) {
                        System.out.println(e.getMessage());
                        analisadorLexico();
                }
        }

    static public String im(int x) {
        int k;
        String s;
        s = tokenImage[x];
        k = s.lastIndexOf("\"");

        try {
```

```
            s = s.substring(1,k);
        } catch (StringIndexOutOfBoundsException e)
            {}
        return s;
    }

    public void consumeUntil(RecoverySet g, ParseException e, String
errorMsg, String met)
        throws ParseEOFException, ParseException {

        Token tok;

       if (debug_recovery) {
           System.out.println("met -> " + met);
           System.out.println("Recovery Set: " + g);
       }

       if (g == null) throw e;

       tok = getToken(1);

       while (!eof) {
            if (g.contains(tok.kind)) {
                if ( debug_recovery) {
                   System.out.println("Encontrou o token: " +
im(tok.kind));
                }
                break;
            }

            if (debug_recovery) {
                System.out.println("Ignorando token: " + im(tok.kind));
            }

            getNextToken();
            tok = getToken(1);

            if (tok.kind == EOF && ! g.contains(EOF) ) {
                eof = true;
            }
       }

       if (tok != lastError) {
           String output = String.format(
                   "\n | ERRO SINTATICO | " +
                   "\n Mensagem: %s"+
//                   "\n Encontrado na linha: %s e na coluna: %s : %s " +
                   "\n Esperava: %s \n",
                   errorMsg,
```

```
//                    tok.beginLine,
//                    tok.beginColumn,
//                    tok,
                   expectedTokens(e)
            );

            System.out.println(output);
            syntacticErrors.append(output);
            lastError = tok;
            contParseError++;
        }
        if ( eof ) throw new ParseEOFException("EOF encontrado de modo
prematuro.");
    }

    public static List<String> expectedTokens(ParseException e) {
        List<String> output = new ArrayList<String>();
        for (int[] group : e.expectedTokenSequences) {
            for (int tokenConst : group) {
                output.add(AnalisadorConstants.tokenImage[tokenConst]);
            }
        }
        return output;
    }

        public String iniciarCompilador(Analisador parser, String args[],
String inputTexto) throws ParseException {
                Analisador analisador;
                Analisador analisadorLexico;

                resultado = new StringBuilder();

                if (args.length == 0) {
                        InputStream edicaoInputStream = new
ByteArrayInputStream(inputTexto.getBytes());
                        analisador = new Analisador(edicaoInputStream);
                        analisadorLexico = new
Analisador(edicaoInputStream);
                } else if (args.length == 1) {
                        try {
                                analisador = new Analisador(new
java.io.FileInputStream(args[0]));
                                analisadorLexico = new Analisador(new
java.io.FileInputStream(args[0]));
                        } catch (java.io.FileNotFoundException e) {
                                System.err.println(args[0] + " was not
found.");
                                System.err.println(e);
                                return args[0] + " was not found.";
```

```
                        }
                } else {
                        return "Erro";
                }

        analisador.resetHouveErroLexico();
                analisador.setHouveErroSintatico(false);

        analisadorLexico.analisadorLexico();

        if(!analisadorLexico.houveErro) {
            try {
                int analisarSintaxe = parser.analisadorSintatico();

                if(contParseError > 0) {
                    printar(String.format("Houve %s erros sintaticos: \n
%s", contParseError, parser.syntacticErrors));
                } else
                if(analisarSintaxe == 0) {
                    printar("Sintaxe correta");
                }

                if(parser.houveErroSintatico) {
                    printar("ERRO Sintatico!");
                }
            }
            catch (ParseException e) {
                printar("ERRO Sintatico!");
                printar(e.getMessage());
            }
            catch (ParseEOFException e) {
                printar("ERRO Sintatico!");
                printar(e.getMessage());
            }
        }


        return resultado.toString();
        }
}

PARSER_END(Analisador)

TOKEN [ IGNORE_CASE ]:{
        <PALAVRA_RESERVADA_PROGRAM : "program">
        |       <PALAVRA_RESERVADA_DEFINE : "define">
        |       <PALAVRA_RESERVADA_NOT : "not">
        |       <PALAVRA_RESERVADA_VARIABLE : "variable">
        |       <PALAVRA_RESERVADA_IS : "is">
```

```
        |       <PALAVRA_RESERVADA_NATURAL : "natural">
        |       <PALAVRA_RESERVADA_REAL : "real">
        |       <PALAVRA_RESERVADA_CHAR : "char">
        |       <PALAVRA_RESERVADA_BOOLEAN : "boolean">
        |       <PALAVRA_RESERVADA_EXECUTE : "execute">
        |       <PALAVRA_RESERVADA_SET : "set">
        |       <PALAVRA_RESERVADA_TO : "to">
        |       <PALAVRA_RESERVADA_GET : "get">
        |       <PALAVRA_RESERVADA_PUT : "put">
        |       <PALAVRA_RESERVADA_LOOP : "loop">
        |       <PALAVRA_RESERVADA_WHILE : "while">
        |       <PALAVRA_RESERVADA_DO : "do">
        |       <PALAVRA_RESERVADA_TRUE : "true">
        |       <PALAVRA_RESERVADA_FALSE : "false">
        |       <PALAVRA_RESERVADA_VERIFY : "verify">
}

TOKEN: {
        <#LETRAS : ["a"-"z","A"-"Z"]>
        |   <#DIGITOS : ["0"-"9"]>
        |       <#UNDERLINE : "_">

        |     <IDENTIFICADORES:
                        (
                                (
                                        (
                        (
                                <LETRAS>
                                (<UNDERLINE>)?
                        )
                        |
                        (
                                <UNDERLINE>
                                (<LETRAS>)?
                        )
                                        )
                                        (<DIGITOS>)
                                )
                                | ( <UNDERLINE> <LETRAS> )
                                | ( <LETRAS> )
                        )*
                (
                                (<UNDERLINE>)
                                | (<LETRAS>)
                                | (<LETRAS> <UNDERLINE>)
                                | (<UNDERLINE> <LETRAS>)
                        )
                >
```

```
        |       <CONSTANTE_NUMERICA_INTEIRA:
                <DIGITOS> (<DIGITOS>)? (<DIGITOS>)?
        >

        |       <CONSTANTE_NUMERICA_REAL:
                <DIGITOS> (<DIGITOS>)? (<DIGITOS>)? (<DIGITOS>)?
(<DIGITOS>)? "." <DIGITOS> (<DIGITOS>)?
        >

        |       <CONSTANTE_LITERAL:
                (
                "'"
                (~["'","\n","\r"])*
                "'"
                        )
                        |
                (
                "\""
                (~["\"","\n","\r"])*
                "\""
                        )
                >

        |   <SIMBOLO_ESPECIAL_PONTO: ".">
        |       <SIMBOLO_ESPECIAL_VIRGULA: ",">
        |       <SIMBOLO_ESPECIAL_ABRECHAVES: "{">
        |       <SIMBOLO_ESPECIAL_FECHACHAVES: "}">
        |       <SIMBOLO_ESPECIAL_ABRECOLCHETE: "[">
        |       <SIMBOLO_ESPECIAL_FECHACOLCHETE: "]">
    |   <SIMBOLO_ESPECIAL_ABREPARENTESES: "(">
    |   <SIMBOLO_ESPECIAL_FECHAPARENTESES: ")">
        |       <SIMBOLO_ESPECIAL_OPERADOR_MAIS : "+">
        |       <SIMBOLO_ESPECIAL_OPERADOR_MENOS : "-">
        |       <SIMBOLO_ESPECIAL_OPERADOR_DIVISAO : "/">
        |       <SIMBOLO_ESPECIAL_OPERADOR_MULTIPLICACAO : "*">
        |       <SIMBOLO_ESPECIAL_OPERADOR_POTENTIACAO : "**">
        |       <SIMBOLO_ESPECIAL_OPERADOR_DIVISAO_INTEIRA : "%">
        |       <SIMBOLO_ESPECIAL_OPERADOR_DIVISAO_RESTO : "%%">
        |       <SIMBOLO_ESPECIAL_OPERADOR_IGUALIGUAL : "==">
        |       <SIMBOLO_ESPECIAL_OPERADOR_DIFERENTE : "!=">
        |       <SIMBOLO_ESPECIAL_OPERADOR_MENOR : "<">
        |       <SIMBOLO_ESPECIAL_OPERADOR_MAIOR : ">">
        |       <SIMBOLO_ESPECIAL_OPERADOR_MENORIGUAL : "<=">
        |       <SIMBOLO_ESPECIAL_OPERADOR_MAIORIGUAL : ">=">
        |       <SIMBOLO_ESPECIAL_OPERADOR_E : "&">
        |       <SIMBOLO_ESPECIAL_OPERADOR_OU : "|">
        |       <SIMBOLO_ESPECIAL_OPERADOR_NAO : "!">
}
```

```
SKIP: {
    " "
        |   "\n"
        |   "\r"
        |   "\r\n"
        |   "\f"
        |   "\t"

        |       ":-": COMENTARIODELINHA
        |   "//": COMENTARIODELINHA

        |   "/*": COMENTARIODEBLOCO
}

<COMENTARIODELINHA> SKIP: {
        < ["\n", "\r"]>: DEFAULT
        | < ~[] >
}

<COMENTARIODEBLOCO> SKIP: {
        "*/": DEFAULT
        | < ~[] >
}

SPECIAL_TOKEN: {
    <CABECALHO_INVALIDO : [":"]([" "])+["-"] | [":"]> {
        Analisador.printarStatic("ERRO DE LEXEMA. Voce tentou fazer um
cabecalho? Use :- constanteLiteral");
        Analisador.setHouveErroLexico();
    }

        |    <ATRIBUICAO_INVALIDA : "=" | "==="> {
                Analisador.setHouveErroLexico();
                        Analisador.printarStatic("ERRO DE LEXEMA : | " +
image + " | na linha " + matchedToken.beginLine + " e na coluna " +
matchedToken.beginColumn + " \nVoce tentou fazer uma atribuicao? Use
==");
                }

        |   <CONSTANTE_LITERAL_INVALIDA:
            "'"
            |
            "\""
        > {
        Analisador.setHouveErroLexico();
        Analisador.printarStatic("Erro de lexema ao tentar declarar uma
CONSTANTE_LITERAL na linha " + matchedToken.beginLine + " e na coluna " +
        matchedToken.beginColumn + ". \nVoce tentou declarar uma
constante literal? Lembre de iniciar e finalizar com apostrofos ou aspas
```

```
na mesma linha e de repetir o mesmo simbolo no inicio e no fim.");
    }
}

<*> TOKEN : {
    <EOF> {
        if(curLexState == COMENTARIODEBLOCO) {
            Analisador.printarStatic("ERRO DE LEXEMA. Voce esqueceu de
fechar o comentario de bloco? Lembre-se de fechar com */");
            Analisador.setHouveErroLexico();
            curLexState = DEFAULT;
        }
    }
        |   <ENTRADA_INVALIDA: ~[]>
}

int analisadorSintatico() throws ParseEOFException : {
     RecoverySet g = First.analisadorSintatico;
} {
    try {
        iniciarPrograma(g) <EOF> { return 0; }
    } catch (ParseException e) {
        consumeUntil(g, e, "Erro ao iniciar programa",
"analisadorSintatico");
        return 1;
    }
}

void iniciarPrograma(RecoverySet g) throws ParseEOFException: {} {
    <PALAVRA_RESERVADA_PROGRAM> <SIMBOLO_ESPECIAL_ABRECHAVES>
        declaracaoDeConstantesEVariaveis(g)
        corpoDoPrograma(g)
    <SIMBOLO_ESPECIAL_FECHACHAVES>
    [ <IDENTIFICADORES> ]
}

void declaracaoDeConstantesEVariaveis(RecoverySet g) throws
ParseEOFException: {
    RecoverySet recoveryDeclaracaoDeConstantesEVariaveis =
First.declaracaoDeConstantesEVariaveis.union(g);
 } {
    try{
        [
            <PALAVRA_RESERVADA_DEFINE> <SIMBOLO_ESPECIAL_ABRECHAVES>
                iniciarDeclaracoes()
            <SIMBOLO_ESPECIAL_FECHACHAVES>
        ]
    } catch (ParseException e) {
        consumeUntil(recoveryDeclaracaoDeConstantesEVariaveis, e, "Erro
```

```
ao declarar constantes e variaveis", "declaracaoDeConstantesEVariaveis");
    }
}

void iniciarDeclaracoes(): {} {
    iniciarDeclaracaoConstante() [ iniciarDeclaracaoVariavel() ]
    | iniciarDeclaracaoVariavel() [ iniciarDeclaracaoConstante() ]
}

void iniciarDeclaracaoConstante(): {} {
    <PALAVRA_RESERVADA_NOT> <PALAVRA_RESERVADA_VARIABLE>
        declaraConstante()
}

void iniciarDeclaracaoVariavel(): {} {
    <PALAVRA_RESERVADA_VARIABLE>
        declaraVariavel()
}

void declaraConstante(): {} {
    tipoVariavel() <PALAVRA_RESERVADA_IS> listaIdentificadores() valor()
<SIMBOLO_ESPECIAL_PONTO> [ declaraConstante() ]
}

void declaraVariavel(): {} {
    tipoVariavel() <PALAVRA_RESERVADA_IS> listaIdentificadores()
<SIMBOLO_ESPECIAL_PONTO> [ declaraVariavel() ]
}

void tipoVariavel() : {} {
    <PALAVRA_RESERVADA_NATURAL>
    | <PALAVRA_RESERVADA_REAL>
    | <PALAVRA_RESERVADA_CHAR>
    | <PALAVRA_RESERVADA_BOOLEAN>
}

void listaIdentificadores() : {}
{
    <IDENTIFICADORES> [ <SIMBOLO_ESPECIAL_VIRGULA> listaIdentificadores()
]
}

void listaDeIdentificadoresEOuConstantes() : {} {
    <IDENTIFICADORES> [ <SIMBOLO_ESPECIAL_VIRGULA>
listaDeIdentificadoresEOuConstantes() ]
    | constantes() [ <SIMBOLO_ESPECIAL_VIRGULA>
listaDeIdentificadoresEOuConstantes() ]
}
```

```
void valor() : {} {
    constantes() | <PALAVRA_RESERVADA_TRUE> | <PALAVRA_RESERVADA_FALSE>
}

void constantes() : {} {
    <CONSTANTE_NUMERICA_INTEIRA> | <CONSTANTE_NUMERICA_REAL> |
<CONSTANTE_LITERAL>
}

void corpoDoPrograma(RecoverySet g) throws ParseEOFException: {
 RecoverySet recoveryCorpoDoPrograma = First.corpoDoPrograma.union(g);
} {
    try {
        <PALAVRA_RESERVADA_EXECUTE> <SIMBOLO_ESPECIAL_ABRECHAVES>
            listaDeComandos()
        <SIMBOLO_ESPECIAL_FECHACHAVES>
    } catch (ParseException e) {
        consumeUntil(recoveryCorpoDoPrograma, e, "Erro ao declarar corpo
do programa", "corpoDoPrograma");
    }
}

void listaDeComandos(): {} {
    comandos() <SIMBOLO_ESPECIAL_PONTO> [ listaDeComandos() ]
}

void comandos(): {} {
    comandoAtribuicaoSet()
    | comandoEntradaDeDadosGet()
    | comandoSaidaDeDadosPut()
    | comandoSelecaoVerify()
    | comandoRepeticaoLoop()
}

void comandoAtribuicaoSet(): {} {
    <PALAVRA_RESERVADA_SET> expressao() <PALAVRA_RESERVADA_TO>
listaIdentificadores()
}

void comandoEntradaDeDadosGet(): {} {
    <PALAVRA_RESERVADA_GET> <SIMBOLO_ESPECIAL_ABRECHAVES>
        listaIdentificadores()
    <SIMBOLO_ESPECIAL_FECHACHAVES>
}

void comandoSaidaDeDadosPut(): {} {
    <PALAVRA_RESERVADA_PUT> <SIMBOLO_ESPECIAL_ABRECHAVES>
        listaDeIdentificadoresEOuConstantes()
    <SIMBOLO_ESPECIAL_FECHACHAVES>
```

```
}

void comandoSelecaoVerify(): {} {
    <PALAVRA_RESERVADA_VERIFY> expressao()
        iniciarVerify()
}

void iniciarVerify(): {} {
   LOOKAHEAD(2) verifyTrue() [ verifyFalse() ]
    | verifyFalse() [ verifyTrue() ]
}

void verifyTrue():{} {
    <PALAVRA_RESERVADA_IS> <PALAVRA_RESERVADA_TRUE>
<SIMBOLO_ESPECIAL_ABRECHAVES>
        listaDeComandos()
    <SIMBOLO_ESPECIAL_FECHACHAVES>
}

void verifyFalse():{} {
    <PALAVRA_RESERVADA_IS> <PALAVRA_RESERVADA_FALSE>
<SIMBOLO_ESPECIAL_ABRECHAVES>
        listaDeComandos()
    <SIMBOLO_ESPECIAL_FECHACHAVES>
}

void comandoRepeticaoLoop(): {} {
    whileLoop() | loopDoWhile()
}

void loopDoWhile():{}{
    <PALAVRA_RESERVADA_LOOP> <SIMBOLO_ESPECIAL_ABRECHAVES>
        listaDeComandos()
    <SIMBOLO_ESPECIAL_FECHACHAVES>
    <PALAVRA_RESERVADA_WHILE> expressao() <PALAVRA_RESERVADA_IS>
<PALAVRA_RESERVADA_TRUE>
}

void whileLoop():{}{
    <PALAVRA_RESERVADA_WHILE> expressao() <PALAVRA_RESERVADA_IS>
<PALAVRA_RESERVADA_TRUE> <PALAVRA_RESERVADA_DO>
    <SIMBOLO_ESPECIAL_ABRECHAVES> listaDeComandos()
<SIMBOLO_ESPECIAL_FECHACHAVES>
}

void expressao(): {} {
    expressaoAritmeticaOuLogica() expressaoLinha()
}
```

```
void expressaoLinha(): {} {
    (
        <SIMBOLO_ESPECIAL_OPERADOR_IGUALIGUAL>
expressaoAritmeticaOuLogica()
        | <SIMBOLO_ESPECIAL_OPERADOR_DIFERENTE>
expressaoAritmeticaOuLogica()
        | <SIMBOLO_ESPECIAL_OPERADOR_MENOR> expressaoAritmeticaOuLogica()
        | <SIMBOLO_ESPECIAL_OPERADOR_MAIOR> expressaoAritmeticaOuLogica()
        | <SIMBOLO_ESPECIAL_OPERADOR_MENORIGUAL>
expressaoAritmeticaOuLogica()
        | <SIMBOLO_ESPECIAL_OPERADOR_MAIORIGUAL>
expressaoAritmeticaOuLogica()
    )?
}

void expressaoAritmeticaOuLogica(): {} {
        termo2() menorPrioridade()
}

void menorPrioridade(): {} {
    (
        <SIMBOLO_ESPECIAL_OPERADOR_MAIS> termo2() menorPrioridade()
        | <SIMBOLO_ESPECIAL_OPERADOR_MENOS> termo2() menorPrioridade()
        | <SIMBOLO_ESPECIAL_OPERADOR_OU> termo2() menorPrioridade()
    )?
}

void termo2(): {} {
    termo1() mediaPrioridade()
}

void mediaPrioridade(): {} {
    (
        <SIMBOLO_ESPECIAL_OPERADOR_MULTIPLICACAO> termo1()
mediaPrioridade()
        | <SIMBOLO_ESPECIAL_OPERADOR_DIVISAO> termo1() mediaPrioridade()
        | <SIMBOLO_ESPECIAL_OPERADOR_DIVISAO_INTEIRA> termo1()
mediaPrioridade()
        | <SIMBOLO_ESPECIAL_OPERADOR_DIVISAO_RESTO> termo1()
mediaPrioridade()
        | <SIMBOLO_ESPECIAL_OPERADOR_E> termo1() mediaPrioridade()
    )?
}

void termo1(): {} {
    elemento() maiorPrioridade()
}

void maiorPrioridade(): {} {
```

```
    (<SIMBOLO_ESPECIAL_OPERADOR_POTENTIACAO> elemento()
maiorPrioridade())?
}

void elemento(): {} {
    <IDENTIFICADORES> indice()
    | <CONSTANTE_NUMERICA_INTEIRA>
    | <CONSTANTE_NUMERICA_REAL>
    | <CONSTANTE_LITERAL>
    | <PALAVRA_RESERVADA_TRUE>
    | <PALAVRA_RESERVADA_FALSE>
    | <SIMBOLO_ESPECIAL_ABREPARENTESES> expressao()
<SIMBOLO_ESPECIAL_FECHAPARENTESES>
    |  <SIMBOLO_ESPECIAL_OPERADOR_NAO> <SIMBOLO_ESPECIAL_ABREPARENTESES>
expressao() <SIMBOLO_ESPECIAL_FECHAPARENTESES>
}

void indice(): {} {
    (<SIMBOLO_ESPECIAL_ABRECOLCHETE> <CONSTANTE_NUMERICA_INTEIRA>
<SIMBOLO_ESPECIAL_FECHACOLCHETE>)?
}
```