Trabalho Prático 2 de Compiladores

Análise Sintática

Douglas Rodrigues de Almeida
douglasralmeida@live.com

1. Introdução

O objetivo deste trabalho foi a criação de um analisador sintático que será usado no projeto de criação de um compilador durante a disciplina Compiladores I. O analisador foi implementado para a linguagem MLM, Mini Linguagem M, um subconjunto da linguagem Pascal, cuja especificação vem a seguir:

```
program ::= program identifier ";" decl_list compound_stmt

decl_list ::= decl_list " ;" decl
            | decl

decl ::= ident_list ":" type

ident_list ::= ident list " ," identifier
             | identifier

type ::= integer | real | boolean | char

compound stmt ::= begin stmt_list end

stmt_list ::= stmt list ";" stmt

            | stmt

stmt := assign_stmt | if_stmt | loop_stmt | read_stmt | write_stmt | compound_stmt

assign_stmt ::= identifier " :=" expr

if_stmt ::= if cond then stmt | if cond then stmt else stmt

cond ::= expr

loop_stmt ::= stmt_prefix do stmt_list stmt_suffix

stmt_prefix ::= while cond | E

stmt_suffix ::= until cond | end

read_stmt ::= read "(" iden_list ")"

write_stmt ::= write "(" expr_list ")"

expr_list ::= expr | expr_list " ," expr

expr ::= simple_expr | simple_expr RELOP simple_expr

simple_expr ::= term | simple_expr ADDOP term

term := factor_a | term MULOP factor_a

fator a := "-" factor | factor

factor := identifier | constant | " (" expr " )" | NOT factor

constant ::= integer_constant | real_constant | char_constant | boolean_constant

boolean_constant := false | true

unsigned_integer ::= digit digit*

sign ::= + | - | E

scale_factor := "E" sign unsigned_integer

unsigned_real ::= unsigned_integer ( E | "." digit*)( E | scale factor)

integer_constant ::= unsigned_integer

real_constant ::= unsigned_real
```

```
char_constant ::= "'" caractereASCII "'"
letter ::= A | B | ··· Z | a | b | ··· z
digit ::= 0 | 2 | 3 | 4 |5 | 6 | 7 | 8 | 9
identifier ::= letter ( letter | digit )*
RELOP ::= = | < | <= | > | >= | ! = | NOT
ADDOP ::= + | – | or
MULOP ::= ? | / | div | mod | and
```

2. Resumo do projeto

Um analisador sintático é utilizado para verificar se o código-fonte de um programa pode ser gerado pela gramática da linguagem. Ele obtém os tokens produzidos pelo analisador léxico a partir do código e gerando uma árvore gramatical que o representa.

O analisador léxico gerado no Trabalho Prático 1 foi adaptado para ser utilizado pelo analisador sintático. Ele foi implementado na linguagem Lex que é utilizada pela ferramenta JFlex (https://jflex.de/) para geração de analisadores léxicos prontos para serem compilados em Java. Essa ferramenta foi utilizada em detrimento ao JLex por ser compatível com as versões mais modernas do Java. A especificação léxica está no arquivo Yylex.lex que se encontra na subpasta src/especificacoes. A saída deve ser gerada na subpasta src/pacotePrincipal com o nome AnalisadorLexico.java. Uma cópia está disponível nos anexos desse relatório.

O analisador sintático foi implementado através da ferramenta CUP (http://www2.cs.tum.edu/projects/cup/) que gera analisadores sintáticos LALR para a linguagem Java. A especificação sintática está no arquivo MLM.cup que se encontra na subpasta src/especificações. A saída deve ser gerada na subpasta src/pacotePrincipal com o nome AnalisadorSintatico.java. Uma ccópia está disponível nos anexos desse relatório.

A geração dos código dos analisadores podem ser feitos através da linha de comando. A geração do analisador léxico é feita pelo comando:

```
jflex -d src/pacotePrincipal src/especificacoes/Yylex.lex
```

A geração do analisador léxico é feita pelo comando:

```
cups -parser AnalisadorSintatico -symbols sym -interface -destdir src/pacotePrincipal
src/especificacoes/MLM.cup
```

O projeto foi criado usando a IDE Eclipse 2019.06 e OpenJDK 12 e pode ser compilado diretamente da IDE ou pela linha de comando:

```
java -jar D:\eclipse\plugins\org.eclipse.jdt.core_3.18.0.v20190522-0428.jar -d bin/ -p
lib/java_cup.jar src/pacotePrincipal/
```

A linha de comando acima pode mudar dependendo da ferramenta de desenvolvimento utilizada e seu local de instalação.

Se compilado utilizando Java 12, a execução pela linha de comando será feita através da sintaxe:

```
java -p bin/;lib/java_cup.jar --add-reads compilador=java.cup -m
compilador/pacotePrincipal.Compilador <arquivoentrada>
```

O código-fonte está disponível no site do Github: https://github.com/douglasralmeida/comp1

3. Conflitos

A gramática original possui um conflito shift-reduce na produção abaixo:

```
if_stmt ::= if cond then stmt | if cond then stmt else stmt
```

Isto ocorre devido a ambiguidade da produção que pode gerar duas árvores gramaticais diferentes para determinadas sentenças na entrada. Para corrigir esta ambiguidade a produção foi substituída pela produção abaixo:

```
if_stmt_a ::= IF cond THEN if_stmt_a ELSE if_stmt_a

if_stmt_b ::= IF cond THEN stmt  IF cond THEN if_stmt_a ELSE if_stmt_b

if_stmt ::= if_stmt_a | if_stmt_b
```

Esta produção resolve o problema de ambiguidade pois, para cada sentença na entrada, ela irá gerar uma única árvore gramatical.

4. Testes

Foram efetuados dois testes com o analisador sintático. O primeiro com uma entrada correta, o segundo com uma entrada incorreta, onde o analisador deverá exibir uma mensagem de erro na produção *if_stmt*. Em ambos os casos as produções gramaticais utilizadas na entrada são exibidas na tela de saída.

```
1. program teste;
2.
3. b: boolean;
4. c: char;
5. d: integer
6.
7. begin
8.    b := true;
9.    c := 'X';
10.   write(c);
11.   if b = false then
12.     c := 'Y';
13.   write(c);
14.   while B = true do
15.     c := 'Z';
16.     d := d + 2
17.   End
18. end
```

O analisador sintático deverá exibir a seguinte saída:

```
Analisando arquivo test/correto.pas...
ident_list -> identifier
type -> boolean
decl -> ident_list: type
dec_list -> decl
ident_list -> identifier
type -> char
decl -> ident_list: type
dec_list -> dec_list; decl
ident_list -> identifier
type -> integer
decl -> ident_list: type
dec_list -> dec_list; decl
constant -> true
factor -> constant
factor_a -> factor
term -> factor_a
simple_expr -> term
expr -> simple_expr
assign_stmt -> IDENTIFIER := expr
stmt_list -> stmt
constant -> 'X'
factor -> constant
factor_a -> factor
term -> factor_a
simple_expr -> term
expr -> simple_expr
assign_stmt -> IDENTIFIER := expr
stmt_list -> stmt_list; stmt
factor -> IDENTIFIER
factor_a -> factor
term -> factor_a
simple_expr -> term
constant -> false
factor -> constant
factor_a -> factor
term -> factor_a
simple_expr -> term
simple_expr RELOP simple_expr
cond -> expr
constant -> 'Y'
factor -> constant
factor_a -> factor
term -> factor_a
simple_expr -> term
expr -> simple_expr
```

```
assign_stmt -> IDENTIFIER := expr
if_stmt -> IF cond THEN stmt
stmt_list -> stmt_list; stmt
factor -> IDENTIFIER
factor_a -> factor
term -> factor_a
simple_expr -> term
expr -> simple_expr
expr_list -> expr
write_stmt -> WRITE(expr_list)
stmt_list -> stmt_list; stmt
factor -> IDENTIFIER
factor_a -> factor
term -> factor_a
simple_expr -> term
constant -> true
factor -> constant
factor_a -> factor
term -> factor_a
simple_expr -> term
simple_expr RELOP simple_expr
cond -> expr
stmt_prefix -> WHILE cond
constant -> 'Z'
factor -> constant
factor_a -> factor
term -> factor_a
simple_expr -> term
expr -> simple_expr
assign_stmt -> IDENTIFIER := expr
stmt_list -> stmt
factor -> IDENTIFIER
factor_a -> factor
term -> factor_a
simple_expr -> term
constant -> 2
factor -> constant
factor_a -> factor
term -> factor_a
simple_expr ADDOP term
expr -> simple_expr
assign_stmt -> IDENTIFIER := expr
stmt_list -> stmt_list; stmt
stmt_suffix -> e
loop_stmt -> stmt_prefix DO stmt_list
stmt_suffix
stmt_list -> stmt_list; stmt
compound_stmt -> begin stmt_list end
program -> PROGRAM identifier; dec_list
compound_stmt

Aceito.
```

A entrada abaixo deverá gerar um erro de sintaxe no analisador.

```
1. program teste;
2.
3. b: boolean;
4.
5. Begin
6.     b := true;
7.     if b = false integer then
8.       write('X')
9. end
```

Saída do arquivo com a entrada incorreta.

```
Analisando arquivo test/incorreto.pas...

ident_list -> identifier
type -> boolean
decl -> ident_list: type
dec_list -> decl
constant -> true
factor -> constant
factor_a -> factor
term -> factor_a
simple_expr -> term
expr -> simple_expr
assign_stmt -> IDENTIFIER := expr
stmt_list -> stmt
factor -> IDENTIFIER
factor_a -> factor
```

```
term -> factor_a
simple_expr -> term

Um erro de sintaxe foi encontrado com a expressão "integer" na linha 7, coluna 17.
Eram esperadas uma das expressões: [THEN].
Ocorreu um erro ao processar o arquivo: Can't recover from previous error(s).

Rejeitado.
```

5. Bibliografia

AHO, A. V. A.; SETH, R.; ULLMAN, J. D. **Compiladores. Princípios, Técnicas e Ferramentas**. Rio de Janeiro: LTC, 1995.

BERK, E. JLex: A lexical analyzer generator for Java. **JLex**, 2000. Disponivel em: <https://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html>. Acesso em: 10 Setembro 2019.

KLEIN, G.; ROWE, S.; DECAMPS, R. JFlex User's Manual. **JFlex**, 2018. Disponivel em: <https://jflex.de/manual.html>. Acesso em: 10 Setembro 2019.

MAZA, M. M. Ambiguous Grammar. **Compiler Theory:** Syntax Analysis, 2004. Disponivel em: <https://www.csd.uwo.ca/~moreno/CS447/Lectures/Syntax.html/Syntax.html>. Acesso em: 03 Outubro 2019.

PETTER, M.; HUDSON, S. E. CUP User's Manual. **CUP**, 2014. Disponivel em: <http://www2.cs.tum.edu/projects/cup/docs.php>. Acesso em: 03 Outubro 2019.

6. Código-fonte

Arquivo MLM.cup:

```
/* Analisador Sintático para linguagem MLM */

package pacotePrincipal;

import java.lang.reflect.Field;
import java.util.List;
import java.util.LinkedList;
import java_cup.runtime.ComplexSymbolFactory.ComplexSymbol;
import java_cup.runtime.Symbol;

class AnalisadorSintatico;

/* Código Java personalizado para o analisador sintático */
parser code {:
    protected void report_expected_token_ids(){
        List<Integer> ids = expected_token_ids();
        LinkedList<String> list = new LinkedList<String>();

        for (Integer expected: ids) {
            list.add(symbl_name_from_id(expected));
        }
        System.out.println("Eram esperadas uma das expressões: " + list + ".");
    }

    public String symbl_name_from_id(int id){
```

```
    Field[] fields = getSymbolContainer().getFields();

    for (Field f : fields) {
      try {
      if (f.getInt(null)==id)
        return f.getName();
    } catch (IllegalArgumentException e) {
      e.printStackTrace();
    } catch (IllegalAccessException e) {
      e.printStackTrace();
    }
    }

    return "invalid symbol id";
  }

    public void syntax_error(Symbol s) {
        ComplexSymbol cs = (ComplexSymbol)s;

        System.out.println(String.format("Um erro de sintaxe foi encontrado com a expres-
são \"%s\" na linha %d, coluna %d.", cs.getName(), cs.xleft.getLine(), cs.xleft.getColumn()));
        report_expected_token_ids();
    };
:}

/* Terminais (tokens retornados pelo analisador léxico). */
terminal            ADDOP, RELOP, MULOP, PROGRAM, INTEGER, REAL, BOOLEAN, CHAR, BEGIN,
                    END, IF, THEN, ELSE, DO, WHILE, UNTIL, READ, WRITE, IDENTIFIER,
                    EOLCHAR, DECCHAR, ENUMCHAR, ASSIGNCHAR, OPENPARCHAR, CLOSEPARCHAR,
                    MINUSCHAR, NOT, EQUAL, LESS, GREATER, PLUS, TIMES, DIVIDED, LESSEQUAL,
                    GRATEREQUAL, DIFFERENT, OR, AND, MOD;

/* Não terminais */
non terminal        program, dec_list, decl, ident_list, type, compound_stmt,
                    stmt_list, stmt,
                    assign_stmt, if_stmt, cond, loop_stmt, stmt_prefix, stmt_suffix,
                    read_stmt, write_stmt, expr_list, expr, simple_expr, term,
                    factor_a, factor, constant,
                    if_stmt_a, if_stmt_b;

/* Precedência */
precedence left ADDOP;
precedence left MULOP;

/* Ponto de partida */
start with program;

/* Regras gramaticais */
program ::= PROGRAM IDENTIFIER EOLCHAR dec_list compound_stmt {: System.out.println("program -
> PROGRAM identifier; dec_list compound_stmt"); :};
```

```
dec_list ::= dec_list EOLCHAR decl {: System.out.println("dec_list -> dec_list; decl"); :}
           | decl {: System.out.println("dec_list -> decl"); :};

decl ::= ident_list DECCHAR type {: System.out.println("decl -> ident_list: type"); :};

ident_list ::= ident_list ENUMCHAR IDENTIFIER  {: System.out.println("ident_list -
> ident_list, identifier"); :}
             | IDENTIFIER  {: System.out.println("ident_list -> identifier"); :};

type ::= INTEGER {: System.out.println("type -> integer"); :}
       | REAL {: System.out.println("type -> real"); :}
       | BOOLEAN {: System.out.println("type -> boolean"); :}
       | CHAR {: System.out.println("type -> char"); :};

compound_stmt ::= BEGIN stmt_list END {: System.out.println("compound_stmt -> be-
gin stmt_list end"); :};

stmt_list ::= stmt_list EOLCHAR stmt {: System.out.println("stmt_list -> stmt_list; stmt"); :}
            | stmt  {: System.out.println("stmt_list -> stmt"); :};

stmt ::= assign_stmt | if_stmt | loop_stmt | read_stmt | write_stmt | compound_stmt {: Sys-
tem.out.println("stmt -> assign_stmt | if_stmt | loop_stmt | read_stmt | write_stmt | com-
pound_stmt"); :};

assign_stmt ::= IDENTIFIER ASSIGNCHAR expr {: System.out.println("assign_stmt -> IDENTI-
FIER := expr"); :};

/* Conflito shift-reduce:
if_stmt ::= IF cond THEN stmt
          | IF cond THEN stmt ELSE stmt; */

if_stmt_a ::= IF cond THEN if_stmt_a ELSE if_stmt_a;

if_stmt_b ::= IF cond THEN stmt {: System.out.println("if_stmt -> IF cond THEN stmt"); :}
            | IF cond THEN if_stmt_a ELSE if_stmt_b {: System.out.println("if_stmt -
> IF cond THEN stmt ELSE stmt"); :};

if_stmt ::= if_stmt_a | if_stmt_b;

cond ::= expr {: System.out.println("cond -> expr"); :};

loop_stmt ::= stmt_prefix DO stmt_list stmt_suffix {: System.out.println("loop_stmt -> stmt_pre-
fix DO stmt_list stmt_suffix"); :};

stmt_prefix ::= WHILE cond {: System.out.println("stmt_prefix -> WHILE cond"); :}
              | /* vazio */ {: System.out.println("stmt_prefix -> e"); :};

stmt_suffix ::= UNTIL cond {: System.out.println("stmt_suffix -> UNTIL cond"); :}
              | END {: System.out.println("stmt_suffix -> e"); :};
```

```
read_stmt ::= READ OPENPARCHAR ident_list CLOSEPARCHAR {: System.out.println("read_stmt -
> READ(ident_list)"); :};

write_stmt ::= WRITE OPENPARCHAR expr_list CLOSEPARCHAR {: System.out.println("write_stmt -
> WRITE(expr_list)"); :};

expr_list ::= expr {: System.out.println("expr_list -> expr"); :}
            | expr_list ENUMCHAR expr {: System.out.println("expr_list -> expr_list; expr"); :};

expr ::= simple_expr {: System.out.println("expr -> simple_expr"); :}
       | simple_expr RELOP:r simple_expr {: System.out.println("simple_expr RELOP sim-
ple_expr"); :};

simple_expr ::= term {: System.out.println("simple_expr -> term"); :}
              | simple_expr ADDOP:a term {: System.out.println("simple_expr ADDOP term"); :};

term ::= factor_a {: System.out.println("term -> factor_a"); :}
       | term MULOP factor_a {: System.out.println("term MULOP factor_a"); :};

factor_a ::= MINUSCHAR factor {: System.out.println("factor_a -> -factor"); :}
           | factor {: System.out.println("factor_a -> factor"); :};

factor ::= IDENTIFIER {: System.out.println("factor -> IDENTIFIER"); :}
         | constant {: System.out.println("factor -> constant"); :}
         | OPENPARCHAR expr CLOSEPARCHAR {: System.out.println("factor -> (expr)"); :}
         | NOT factor {: System.out.println("factors -> NOT factor"); :};

constant ::= INTEGER:i {: System.out.println("constant -> " + i); :}
           | REAL:r {: System.out.println("constant -> " + r); :}
           | CHAR:c {: System.out.println("constant -> '" + c + "'"); :}
           | BOOLEAN:b {: System.out.println("constant -> " + b); :};
```

Arquivo yylex.lex:

```
/* Analisador Léxico para linguagem MLM */

package pacotePrincipal;

import java_cup.runtime.Symbol;
import java_cup.runtime.ComplexSymbolFactory;
import java_cup.runtime.ComplexSymbolFactory.Location;

%%

/* Procedimentos personalizado para o analisador léxico */
%{
    ComplexSymbolFactory symbolFactory;
    StringBuffer string = new StringBuffer();

    public AnalisadorLexico(java.io.Reader in, ComplexSymbolFactory sf) {
```

```
        this(in);
        symbolFactory = sf;
    }
    private Symbol symbol(String name, int sym) {
        return symbolFactory.newSymbol(name, sym, new Location(yyline+1,yyco-
lumn+1,yychar), new Location(yyline+1,yycolumn+yylength(),yychar+yylength()));
    }
    private Symbol symbol(String name, int sym, Object val) {
        Location left = new Location(yyline+1,yycolumn+1,yychar);
        Location right = new Location(yyline+1,yycolumn+yylength(), yychar+yylength());

        return symbolFactory.newSymbol(name, sym, left, right, val);
    }

    private void exibirErro(String msg) {
        System.out.println("Existe um erro na linha "+(yyline+1)+", column "+(yyco-
lumn+1)+" : " + msg);
    }
%}

%cup
%public
%class AnalisadorLexico
%implements AnalisadorSintaticoSym
%char
%line
%column

/* definicoes regulares */
delimin=\r|\n|\r\n|[ \t\f]
stoken={delimin}+
letter=[a-zA-Z]
digit=[0-9]
identifier={letter}({letter}|{digit})*
unsigned_integer=0|[1-9]{digit}*
sign=[+-]?
scale_factor=E{sign}{unsigned_integer}
unsigned_real={unsigned_integer}(.{digit}*)?({scale_factor})?
integer_constant={unsigned_integer}
real_constant={unsigned_real}
char_constant=\'[^\r\n]\'
boolean_constant=false|true

%% /* regras de traducao */

<YYINITIAL> {
    /* operadores de relação */
    "="      { return symbol("equal", RELOP, Integer.valueOf(EQUAL)); }
    "<"      { return symbol("less", RELOP, Integer.valueOf(LESS)); }
    "<="     { return symbol("lessequal", RELOP, Integer.valueOf(LESSEQUAL)); }
    ">"      { return symbol("greater", RELOP, Integer.valueOf(GREATER)); }
```

```
    ">="     { return symbol("greaterequal", RELOP, Integer.valueOf(GRATEREQUAL)); }
    "!="     { return symbol("different", RELOP, Integer.valueOf(DIFFERENT)); }
    "not"    { return symbol("not", RELOP, Integer.valueOf(NOT)); }

    /* operadores de adição */
    "+"    { return symbol("plus", ADDOP, Integer.valueOf(PLUS)); }
    "-"    { return symbol("minus", ADDOP, Integer.valueOf(ADDOP)); }
    "or"   { return symbol("or", ADDOP, Integer.valueOf(OR)); }

    /* operadores de multiplicação */
    "*"      { return symbol("times", MULOP, Integer.valueOf(TIMES)); }
    "/"      { return symbol("divided", MULOP, Integer.valueOf(DIVIDED)); }
    "div"    { return symbol("divided", MULOP, Integer.valueOf(DIVIDED)); }
    "mod"    { return symbol("mod", MULOP, Integer.valueOf(MOD)); }
    "and"    { return symbol("and", MULOP, Integer.valueOf(AND)); }

    /* palavras reservadas */
    "program" { return symbol("program", PROGRAM); }
    "integer" { return symbol("integer", INTEGER); }
    "real"    { return symbol("real", REAL); }
    "boolean" { return symbol("boolean", BOOLEAN); }
    "char"    { return symbol("char", CHAR); }
    "begin"   { return symbol("begin", BEGIN); }
    "end"     { return symbol("end", END); }
    "if"      { return symbol("if", IF); }
    "then"    { return symbol("then", THEN); }
    "else"    { return symbol("else", ELSE); }
    "do"      { return symbol("do", DO); }
    "while"   { return symbol("while", WHILE); }
    "until"   { return symbol("until", UNTIL); }
    "read"    { return symbol("read", READ); }
    "write"   { return symbol("rite", WRITE); }

    /* sinais */
    ":=" { return symbol("assignchar", ASSIGNCHAR); }
    ";"  { return symbol("eolchar", EOLCHAR); }
    ":"  { return symbol("decchar", DECCHAR); }
    "("  { return symbol("openparchar", OPENPARCHAR); }
    ")"  { return symbol("closeparchar", CLOSEPARCHAR); }
    ","  { return symbol("enumchar", ENUMCHAR); }

    /* constantes */
    {boolean_constant}  { return symbol("bool_const", BOOLEAN, Boolean.valueOf(Boolean.parseBoo-
lean(yytext()))); }
    {integer_constant}  { return symbol("int_const", INTEGER, Integer.valueOf(Integer.par-
seInt(yytext()))); }
    {real_constant}     { return symbol("real_const", REAL, Float.valueOf(Float.parse-
Float(yytext()))); }
    {char_constant}     { return symbol("char_const", CHAR, Character.valueOf(yytext().cha-
rAt(1))); }
```

```
    /* outros */
    {identifier} { return symbol("identifier", IDENTIFIER, yytext()); }

    /* espaço em branco */
    {stoken}     { /* ignora */ }
}

<<EOF>>                          { return symbolFactory.newSymbol("EOF", EOF, new Loca-
tion(yyline+1,yycolumn+1,yychar), new Location(yyline+1,yycolumn+1,yychar+1)); }

/* erro */
[^]         { exibirErro("Caractere não esperado: " + yytext()); }
```

Arquivo Compilador.java:

```
package pacotePrincipal;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.StringReader;
import java_cup.runtime.ComplexSymbolFactory;
import java_cup.runtime.ScannerBuffer;

public class Compilador {
    public static void main(String[] args) throws IOException {
        StringReader sr = null;
        if (args.length < 1) {
            System.out.println("A sintaxe do comando está incorreta.");
            System.out.println("Use: java -p bin\\ -m compilador/pacotePrincipal.Compilador <ar-
quivoentrada>");
            System.exit(1);
        }
        try {
            BufferedReader br = new BufferedReader(new FileReader(args[0]));
            StringBuilder sb = new StringBuilder();
            String line = br.readLine();
            while (line != null) {
                sb.append(line);
                sb.append(System.lineSeparator());
                line = br.readLine();
            }
            br.close();
            sr = new StringReader(sb.toString());
        } catch (IOException ex) {
            System.out.println(String.format("Ocorreu um erro ao carregar o ar-
quivo %s: %s.", args[0], ex.getMessage()));
            System.exit(1);
        }
        try {
            ComplexSymbolFactory csf = new ComplexSymbolFactory();
            AnalisadorLexico al = new AnalisadorLexico(sr, csf);
```

```
            ScannerBuffer scanner = new ScannerBuffer(al);
            AnalisadorSintatico sint = new AnalisadorSintatico(scanner, csf);
            System.out.println("Analisando arquivo " + args[0] + "...");
            sint.parse();
            System.out.println("\nAceito.");
        } catch (Exception ex) {
            System.out.println(String.format("Ocorreu um erro ao processar o ar-
quivo: %s.", ex.getMessage()));
            ex.printStackTrace();
            System.out.println("\nRejeitado.");
            System.exit(1);
        }
        System.exit(0);
    }
}
```