

# Trabalho Prático 1

Universidade Federal de Minas Gerais  
Departamento de Ciência da Computação  
Compiladores I

João Francisco Barreto da Silva Martins  
<joaofbsm@dcc.ufmg.br>

13 de Setembro de 2017

## 1 Introdução

Este trabalho prático tem como objetivo a implementação de um analisador léxico e um analisador sintático **LALR** (*Look-ahead left-to-right*), ambas partes indispensáveis do *front end* de um compilador.

Para auxiliar a implementação foram utilizadas duas ferramentas escritas em Java: **JFlex** (*Java Fast Lexical Analyzer Generator*) para gerar o analisador léxico e **CUP** (*Construction of Useful Parsers*) para gerar o analisador sintático LALR.

As análises devem ser feitas baseadas na gramática apresentada na figura 1.

## 2 Desenvolvimento

Nesta seção vamos entrar mais a fundo no processo de desenvolvimento dos analisadores usando as ferramentas geradoras desenvolvidas em Java. Os códigos passados para os geradores estão descritos na próxima seção.

### 2.1 Analisador Léxico

A criação do analisador léxico foi feita usando a ferramenta geradora **JFlex**[2]. Diversas outras existem, tanto escritas em Java, como a **JLex**, ou em C, como a **Lex** e a **Flex**. No entanto, nossa escolha foi baseada no fato de a

```

program → block
block → { decls stmts }
decls → decls decl |  $\epsilon$ 
decl → type id ;
type → int | char | bool | float
stmts → stmts stmt |  $\epsilon$ 
stmt → id = expr ;
        | if ( rel ) stmt
        | if ( rel ) stmt else stmt
        | while ( rel ) stmt
        | block
rel → expr < expr | expr <= expr | expr >= expr |
        expr > expr | expr
expr → expr + term | expr - term | term
term → term * unary | term / unary | unary
unary → - unary | factor
factor → num | real

```

Figura 1: Gramática descrita no enunciado do trabalho.

JFlex dar muita maleabilidade ao código e ainda estar sob constante desenvolvimento. Atualmente ela está na versão 1.6.1.

O arquivo `Lexer.flex` contém as instruções para modelagem do analisador léxico. Nele são descritas as expressões regulares e símbolos a serem procurados e extraídos de um dado código fonte. Para tal processo, o analisador é formado por um autômato finito determinístico (AFD). Inicialmente, um autômato finito não-determinístico (AFN) é gerado, este é convertido em um AFD que é então minimizado. Um exemplo de saída criada nesse processo pode ser visto na figura 2.

```

Reading "Lexer.flex"
Constructing NFA : 106 states in NFA
Converting NFA to DFA :
.....
57 states before minimization, 51 states in minimized DFA
Writing code to "Lexer.java"

```

Figura 2: Saída gerada na criação do analisador léxico pelo JFlex.

Para executar o JFlex é necessária a execução da seguinte sequência via linha de comando:

```
java -jar jflex-1.6.1.jar Lexer.flex
```

Com a adição da opção `--dot` ainda é possível obter as máquinas de estados em todas as suas etapas (AFN, AFD, AFD minimizada) no formato `.dot`. Assim, através do uso da biblioteca `graphviz`, é possível converter os arquivos `.dot` para `.png` com o seguinte comando:

```
dot -Tpng afd.dot afd.png
```

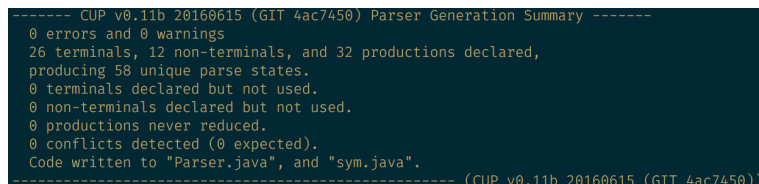
Nenhum autômato finito foi apresentado aqui devido ao tamanho impraticável de suas representações gráficas.

## 2.2 Analisador Sintático

A criação do analisador sintático LALR foi feita usando o gerador **CUP**[1], que atualmente está em sua versão 0.11b e é mantido pela *Technical University of Munich*. O CUP tem integração direta com o JFlex, o que simplifica em muito o processo. O arquivo `Parser.cup` contém as instruções para construção do analisador, como a descrição dos terminais, não-terminais e as regras da gramática. Os símbolos definidos na fase de análise léxica são fundamentais aqui.

A saída do gerador nos fornece alguns dados sobre as dimensões do parser e sua integridade ao longo do processo, como pode ser visto na figura 3. Para executarmos o CUP, é necessário o seguinte comando:

```
java -jar java-cup-11b.jar -interface -parser Parser Parser.cup
```



```
----- CUP v0.11b 20160615 (GIT 4ac7450) Parser Generation Summary -----
0 errors and 0 warnings
26 terminals, 12 non-terminals, and 32 productions declared,
producing 58 unique parse states.
0 terminals declared but not used.
0 non-terminals declared but not used.
0 productions never reduced.
0 conflicts detected (0 expected).
Code written to "Parser.java", and "sym.java".
----- (CUP v0.11b 20160615 (GIT 4ac7450))
```

Figura 3: Saída gerada na criação do analisador sintático pelo CUP.

Houve um conflito do tipo `shift-reduce` na gramática com relação a regra do `if` sozinho ou do `if-else`, acusado pelo CUP com a saída demonstrada na figura 4. Esse conflito foi resolvido ao darmos precedência à esquerda para o símbolo `ELSE`.

```

Warning : *** Shift/Reduce conflict found in state #47
         between stmt ::= IF LEFTPAR rel RIGHTPAR stmt (*)
         and      stmt ::= IF LEFTPAR rel RIGHTPAR stmt (*) ELSE stmt
         under symbol ELSE
         Resolved in favor of shifting.

Error : *** More conflicts encountered than expected -- parser generation aborted
----- CUP v0.11b 20160615 (GIT 4ac7450) Parser Generation Summary -----
      1 error and 1 warning
      26 terminals, 12 non-terminals, and 32 productions declared,
      producing 58 unique parse states.
      0 terminals declared but not used.
      0 non-terminals declared but not used.
      0 productions never reduced.
      1 conflict detected (0 expected).
      No code produced.

```

Figura 4: Saída acusando conflito shift-reduce.

## 2.3 Integração de Analisadores

Para que ambos analisadores funcionem em conjunto, é necessário a execução de uma biblioteca de *runtime* do CUP. Essa execução é feita da seguinte forma:

```
javac -cp java-cup-11b-runtime.jar:. *.java
```

Agora o arquivo `Parser.class` pode ser chamado por linha de comando para execuções das funções de análise, como será demonstrado na seção 4.

## 3 Código

Nesta seção apresentaremos os códigos fontes fornecidos aos geradores.

### Lexer.flex

```

1  import java_cup.runtime.Symbol;
2  import java_cup.runtime.ComplexSymbolFactory;
3  import java_cup.runtime.ComplexSymbolFactory.Location;
4  import java.io.InputStream;
5  import java.io.InputStreamReader;
6
7  %%
8
9  %cup
10 %public
11 %class Lexer
12 %implements sym
13 %line

```

```

14 %column
15
16 %{
17     public Lexer(java.io.Reader in, ComplexSymbolFactory sf){
18         this(in);
19         symbolFactory = sf;
20     }
21
22     ComplexSymbolFactory symbolFactory;
23     StringBuffer string = new StringBuffer();
24
25     private Symbol symbol(String name, int sym) {
26         return symbolFactory.newSymbol(name, sym, new Location(yyline + 1, yycolumn + 1, yychar),
27             new Location(yyline + 1, yycolumn + yylength(), yychar + yylength()));
28     }
29
30     private Symbol symbol(String name, int sym, Object val) {
31         return symbolFactory.newSymbol(name, sym, new Location(yyline + 1, yycolumn + 1, yychar),
32             new Location(yyline + 1, yycolumn + yylength(), yychar + yylength()), val);
33     }
34
35     private void error(String message) {
36         System.out.println("Error at line " + (yyline+1) + ", column " + (yycolumn+1) + " : " + message);
37     }
38 }%
39
40 %eofval{
41     return symbolFactory.newSymbol("EOF", EOF, new Location(yyline + 1, yycolumn + 1, yychar),
42         new Location(yyline + 1, yycolumn + 1, yychar + 1));
43 %eofval}
44
45 identifier = [a-zA-Z_][a-zA-Z0-9_]*
46 num = 0 | [1-9][0-9]*
47 real = [0 | [1-9][0-9]+] + [.] [0 | [1-9][0-9]+]*
48 ignore = [ \r | \n | \r\n | [ \t\f]
49
50 %%
51
52 <YYINITIAL> {
53
54     "if"          {System.out.println("< keyword, if >");
55                   return symbol("if", IF); }
56     "else"        {System.out.println("< keyword, else >");
57                   return symbol("else", ELSE); }
58     "while"       {System.out.println("< keyword, while >");
59                   return symbol("while", WHILE); }
60
61     "{"           {System.out.println("< separator, { >");
62                   return symbol("begin", BEGIN); }

```

```

63     "}"      {System.out.println("< separator, } >");
64               return symbol("end", END); }
65     ";"      {System.out.println("< separator, ; >");
66               return symbol("semicolon", SEMICOLON); }
67     "("      {System.out.println("< separator, ( >");
68               return symbol("rightpar", LEFTPAR); }
69     ")"      {System.out.println("< separator, ) >");
70               return symbol("leftpar", RIGHTPAR); }
71
72     "int"     {System.out.println("< type, int_type >");
73               return symbol("int", INTEGER_TYPE); }
74     "char"    {System.out.println("< type, char_type >");
75               return symbol("char", CHAR_TYPE); }
76     "bool"    {System.out.println("< type, bool_type >");
77               return symbol("bool", BOOL_TYPE); }
78     "float"   {System.out.println("< type, float_type >");
79               return symbol("float", FLOAT_TYPE); }
80
81     "+"       {System.out.println("< arith_op, + >");
82               return symbol("plus", PLUS); }
83     "-"       {System.out.println("< arith_op, - >");
84               return symbol("minus", MINUS); }
85     "*"       {System.out.println("< arith_op, * >");
86               return symbol("times", TIMES); }
87     "/"       {System.out.println("< arith_op, / >");
88               return symbol("div", DIVIDE); }
89     "="       {System.out.println("< attrib, = >");
90               return symbol("attrib", EQ); }
91     "<="      {System.out.println("< comp_op, <= >");
92               return symbol("leq", LEQ); }
93     ">="      {System.out.println("< comp_op, >= >");
94               return symbol("geq", GEQ); }
95     "<"       {System.out.println("< comp_op, < >");
96               return symbol("lt", LT); }
97     ">"       {System.out.println("< comp_op, > >");
98               return symbol("gt", GT); }
99
100    {identifier} {System.out.println("< id, "+ yytext() +" >");
101                  return symbol("id", IDENTIFIER, yytext()); }
102
103    {num}        {System.out.println("< int_const, "+ yytext() +" >");
104                  return symbol("intconst", INTEGER, new Integer(yytext())); }
105    {real}       {System.out.println("< float_const, "+ yytext() +" >");
106                  return symbol("floatconst", FLOAT,
107                                new Float(yytext().substring(0,yylength() - 1))); }
108
109    {ignore}     {/* IGNORE */}
110  }
111

```

```

112  /* ERROR */
113  [^]      {
114          error("Illegal character <"+ yytext() + ">");
115      }

```

---

## Parser.cup

---

```

1  import java.io.*;
2  import java_cup.runtime.ComplexSymbolFactory;
3  import java_cup.runtime.ScannerBuffer;
4
5  parser code {:
6      public Parser(Lexer lex, ComplexSymbolFactory sf) {
7          super(lex,sf);
8      }
9
10     public static void main(String[] args) throws Exception {
11         ComplexSymbolFactory csf = new ComplexSymbolFactory();
12         ScannerBuffer lexer = new ScannerBuffer(new Lexer(
13             new BufferedReader(new FileReader(args[0])),csf));
14         System.out.println("=====\nParsing file "+
15             args[0] + "\n=====\\n");
16         Parser p = new Parser(lexer,csf);
17         Object o = p.parse().value;
18         System.out.println("");
19
20     }
21 :};
22
23 terminal IF, ELSE, WHILE, BEGIN, END, SEMICOLON, LEFTPAR, RIGHTPAR;
24 terminal INTEGER_TYPE, CHAR_TYPE, BOOL_TYPE, FLOAT_TYPE, EQ, LT, LEQ, GT, GEQ;
25 terminal PLUS, MINUS, TIMES, DIVIDE, IDENTIFIER, INTEGER, FLOAT;
26
27 non terminal program, block, decls, decl, type, stmts, stmt, rel, expr, term, unary, factor;
28
29 precedence left ELSE;
30
31 program ::= block {: System.out.println("program → block"); :}
32     ;
33 block ::= BEGIN decls stmts END {: System.out.println("block → { decls stmts }"); :}
34     ;
35 decls ::= decls decl {: System.out.println("decls → decls decl"); :}
36     | {: System.out.println("decls → e"); :} /* EMPTY RULE */
37     ;
38 decl ::= type IDENTIFIER:ide SEMICOLON {: System.out.println("decl → type id("+ ide +")"); :}
39     ;
40 type ::= INTEGER_TYPE {: System.out.println("type → int"); :}

```

```

41     | CHAR_TYPE {: System.out.println("type → char"); :}
42     | BOOL_TYPE {: System.out.println("type → bool"); :}
43     | FLOAT_TYPE {: System.out.println("type → float"); :}
44     ;
45 stmts ::= stmts stmt {: System.out.println("stmts → stmts stmt"); :}
46     | {: System.out.println("stmts → e"); :} /* EMPTY RULE */
47     ;
48 stmt ::= IDENTIFIER:ide EQ expr SEMICOLON
49     | {: System.out.println("stmt → id( "+ ide +" ) = expr"); :}
50     | IF LEFTPAR rel RIGHTPAR stmt
51     | {: System.out.println("stmt → if ( rel ) stmt"); :}
52     | IF LEFTPAR rel RIGHTPAR stmt ELSE stmt
53     | {: System.out.println("stmt → if ( rel ) stmt else stmt"); :}
54     | WHILE LEFTPAR rel RIGHTPAR stmt
55     | {: System.out.println("stmt → while ( rel ) stmt"); :}
56     | block {: System.out.println("stmt → block"); :}
57     ;
58 rel ::= expr LT expr {: System.out.println("rel → expr < expr"); :}
59     | expr LEQ expr {: System.out.println("rel → expr <= expr"); :}
60     | expr GEQ expr {: System.out.println("rel → expr >= expr"); :}
61     | expr GT expr {: System.out.println("rel → expr > expr"); :}
62     | expr {: System.out.println("rel → expr"); :}
63     ;
64 expr ::= expr PLUS term {: System.out.println("expr → expr + term"); :}
65     | expr MINUS term {: System.out.println("expr → expr - term"); :}
66     | term {: System.out.println("expr → term"); :}
67     ;
68 term ::= term TIMES unary {: System.out.println("term → term * unary"); :}
69     | term DIVIDE unary {: System.out.println("term → term / unary"); :}
70     | unary {: System.out.println("term → unary"); :}
71     ;
72 unary ::= MINUS unary {: System.out.println("unary → - unary"); :}
73     | factor {: System.out.println("unary → factor"); :}
74     ;
75 factor ::= INTEGER:inte {: System.out.println("factor → num( "+ inte +" )"); :}
76     | FLOAT:flt {: System.out.println("factor → real( "+ flt +" )"); :}
77     ;

```

---

## 4 Testes

Após a construção do parser com os comandos definidos na seção 2, é possível rodar testes com ele através da linha de comando:

```
java -cp java-cup-11b-runtime.jar:. Parser tests/test1.txt
```

Foram desenvolvidos três arquivos de teste: um totalmente correto, um



com erros léxicos(lexemas inválidos) e um outro sem erros léxicos mas com erros sintáticos. A saída do analisador léxico, contendo os pares <tipo, valor>, só será mostrada para o primeiro teste pois ele já engloba a maioria dos tokens.

#### 4.1 Teste 1 - Correto

```
{
    int i; char c; bool b; float f;
    i = 99;
    f = 3.1416;
    while (1) {
        if (1 * 2 <= 4) {
            int i2;
            i2 = 99 - 1;
        }
        else {
            int i3;
            i3 = 99 + 1;
        }
    }
}
```

#### Saída Analisador Léxico

```
< separator, { >
< type, int_type >
< id, i >
< separator, ; >
< type, char_type >
< id, c >
< separator, ; >
< type, bool_type >
< id, b >
< separator, ; >
< type, float_type >
< id, f >
< separator, ; >
< id, i >
< attrib, = >
```

```
< int_const, 99 >
< separator, ; >
< id, f >
< attrib, = >
< float_const, 3.1416 >
< separator, ; >
< keyword, while >
< separator, ( >
< int_const, 1 >
< separator, ) >
< separator, { >
< keyword, if >
< separator, ( >
< int_const, 1 >
< arith_op, * >
< int_const, 2 >
< comp_op, <= >
< int_const, 4 >
< separator, ) >
< separator, { >
< type, int_type >
< id, i2 >
< separator, ; >
< id, i2 >
< attrib, = >
< int_const, 99 >
< arith_op, - >
< int_const, 1 >
< separator, ; >
< separator, } >
< keyword, else >
< separator, { >
< type, int_type >
< id, i3 >
< separator, ; >
< id, i3 >
< attrib, = >
< int_const, 99 >
< arith_op, + >
< int_const, 1 >
```

```
< separator, ; >
< separator, } >
< separator, } >
< separator, } >
```

### Saída Analisador Sintático

```
=====
Parsing file tests/test1.txt
=====

decls → e
type → int
decl → type id(i)
decls → decls decl
type → char
decl → type id(c)
decls → decls decl
type → bool
decl → type id(b)
decls → decls decl
type → float
decl → type id(f)
decls → decls decl
stmts → e
factor → num(99)
unary → factor
term → unary
expr → term
stmt → id(i) = expr
stmts → stmts stmt
factor → real(3.141)
unary → factor
term → unary
expr → term
stmt → id(f) = expr
stmts → stmts stmt
factor → num(1)
unary → factor
```

```
term → unary
expr → term
rel → expr
decls → e
stmts → e
factor → num(1)
unary → factor
term → unary
factor → num(2)
unary → factor
term → term * unary
expr → term
factor → num(4)
unary → factor
term → unary
expr → term
rel → expr <= expr
decls → e
type → int
decl → type id(i2)
decls → decls decl
stmts → e
factor → num(99)
unary → factor
term → unary
expr → term
factor → num(1)
unary → factor
term → unary
expr → expr - term
stmt → id(i2) = expr
stmts → stmts stmt
block → { decls stmts }
stmt → block
decls → e
type → int
decl → type id(i3)
decls → decls decl
stmts → e
factor → num(99)
```

```

unary → factor
term → unary
expr → term
factor → num(1)
unary → factor
term → unary
expr → expr + term
stmt → id(i3) = expr
stmts → stmts stmt
block → { decls stmts }
stmt → block
stmt → if ( rel ) stmt else stmt
stmts → stmts stmt
block → { decls stmts }
stmt → block
stmt → while ( rel ) stmt
stmts → stmts stmt
block → { decls stmts }
program → block

```

## 4.2 Teste 2 - Erro léxico

```

{
    float f; char c; bool b;
    f = 1.5 * 8.23;
    $%`&#
}

```

O erro aqui acontece porque esses caracteres são ilegais, porém é importante notar que o erro que acontece no analisador léxico não é crítico, e portanto não é passado para frente. Os caracteres são ignorados e o programa segue seu fluxo de execução. Isso vai ser mais restrito a medida que a especificação do compilador se torne mais completa.

```

=====
Parsing file tests/test2.txt
=====

decls → e

```

```

type → float
decl → type id(f)
decls → decls decl
type → char
decl → type id(c)
decls → decls decl
type → bool
decl → type id(b)
decls → decls decl
stmts → e
factor → real(1.5)
unary → factor
term → unary
factor → real(8.2)
unary → factor
term → term * unary
expr → term
Error at line 4, column 5 : Illegal character <$>
Error at line 4, column 6 : Illegal character <%>
Error at line 4, column 7 : Illegal character <`>
Error at line 4, column 8 : Illegal character <&>
Error at line 4, column 9 : Illegal character <#>
stmt → id(f) = expr
stmts → stmts stmt
block → { decls stmts }
program → block

```

### 4.3 Teste 3 - Erro sintático

```

{
    int weight; int group; int charge; int distance;
    distance = 2300;
    weight = 4000;
    if (weight > 60) group = 5;
    else group = weight + 14 / 15;
    charge = 40 + 3 * distance / 1000;
}

```

O erro aqui acontece pois a gramática não permite que um identificador

esteja dentro da condição do `if`. Mesmo se isso fosse permitido, existem erros mais a frente onde, em atribuições, identificadores estão à direita do operador.

```
=====
Parsing file tests/test3.txt
=====

decls → e
type → int
decl → type id(weight)
decls → decls decl
type → int
decl → type id(group)
decls → decls decl
type → int
decl → type id(charge)
decls → decls decl
type → int
decl → type id(distance)
decls → decls decl
stmts → e
factor → num(2300)
unary → factor
term → unary
expr → term
stmt → id(distance) = expr
stmts → stmts stmt
factor → num(4000)
unary → factor
term → unary
expr → term
stmt → id(weight) = expr
stmts → stmts stmt
Syntax error for input symbol "id" spanning from unknown:5/9(0)
to unknown:5/14(6)
instead expected token classes are [MINUS, INTEGER, FLOAT]
Couldn't repair and continue parse for input symbol "id"
spanning from unknown:5/9(0) to unknown:5/14(6)
Exception in thread "main" java.lang.Exception: Can't recover
```

```
from previous error(s)
    at java_cup.runtime.lr_parser.
report_fatal_error(lr_parser.java:392)
    at java_cup.runtime.lr_parser.
unrecovered_syntax_error(lr_parser.java:539)
    at java_cup.runtime.lr_parser.parse(lr_parser.java:731)
    at Parser.main(Parser.java:194)
```

## 5 Conclusão

Com a finalização desse trabalho temos mais da metade do *front end* do compilador pronto, só restando o desenvolvimento do analisador semântico. Os analisadores funcionaram corretamente, tanto na expansão de macros quanto na avaliação de programas de acordo com regras de produção especificadas.

A maior parte do tempo foi gasto entendendo o funcionamento das ferramentas geradoras, que não são nada triviais pois, apesar de serem feitas em Java, possuem sintaxes próprias para grande parte dos arquivos.

## Referências

- [1] C. Ananian, F. Flannery, D. Wang, A. Appel, and M. Petter, “CUP - LALR Parser Generator For Java,” 2017. [Online]. Available: <http://www2.cs.tum.edu/projects/cup/>
- [2] G. Klein, “JFlex - JFlex The Fast Scanner Generator for Java,” 2017. [Online]. Available: <http://jflex.de/>