Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Compiladores I - DCC053
Trabalho Prático 1
Análise Léxica e Sintática LALR

Hugo Araújo de Sousa
(2013007463)

2º Semestre de 2017

1 Descrição do Problema

Em projetos de compiladores modernos, as etapas iniciais do processo de análise do código fonte são as análises Léxica e Sintática. Durante a análise léxica, o programa fonte é organizado em tokens, que são sequências significativas para a linguagem em questão. Com esses tokens, pode-se contruir a tabela de símbolos, necessária para a fase posterior de análise semântica e de geração de código.

Já a fase de análise sintática é responsável por, a partir dos tokens produzidos inicialmente, impor uma estrutura gramatical sobre o programa fonte.

Nesse trabalho, são implementadas as fases de análise léxica e sintática para a linguagem de programação definida através da gramática mostrada na Figura 1.

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \textit{block} \\
\textit{block} & \rightarrow & \texttt{\{}\ \textit{decls stmts}\ \texttt{\}} \\
\textit{decls} & \rightarrow & \textit{decls decl} \mid \epsilon \\
\textit{decl} & \rightarrow & \textit{type}\ \textbf{id}\ \texttt{;} \\
\textit{type} & \rightarrow & \textbf{int} \mid \textbf{char} \mid \textbf{bool} \mid \textbf{float} \\
\textit{stmts} & \rightarrow & \textit{stmts stmt} \mid \epsilon \\
\textit{stmt} & \rightarrow & \textbf{id}\ \texttt{=}\ \textit{expr}\ \texttt{;} \\
 & \mid & \textbf{if}\ \texttt{(}\ \textit{rel}\ \texttt{)}\ \textit{stmt} \\
 & \mid & \textbf{if}\ \texttt{(}\ \textit{rel}\ \texttt{)}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
 & \mid & \textbf{while}\ \texttt{(}\ \textit{rel}\ \texttt{)}\ \textit{stmt} \\
 & \mid & \textit{block} \\
\textit{rel} & \rightarrow & \textit{expr} \texttt{<} \textit{expr} \mid \textit{expr} \texttt{<=} \textit{expr} \mid \textit{expr} \texttt{>=} \textit{expr} \mid \\
 & & \quad \textit{expr} \texttt{>} \textit{expr} \mid \textit{expr} \\
\textit{expr} & \rightarrow & \textit{expr} \texttt{+} \textit{term} \mid \textit{expr} \texttt{-} \textit{term} \mid \textit{term} \\
\textit{term} & \rightarrow & \textit{term} \texttt{*} \textit{unary} \mid \textit{term} \texttt{/} \textit{unary} \mid \textit{unary} \\
\textit{unary} & \rightarrow & \texttt{-}\ \textit{unary} \mid \textit{factor} \\
\textit{factor} & \rightarrow & \textbf{num} \mid \textbf{real}
\end{array}
$$

Figura 1: Gramática da linguagem para qual a análise léxica e sintática será feita.

2 Metodologia

O trabalho foi realizado utilizando a linguagem de programação Java. Para esta, existem ferramentas que auxiliam no desenvolvimento de analisadores léxico e sintáticos. Nesse trabalho foram utilizadas as ferramentas JLex e Cup, que funcionam de forma coordenada.

Para gerar analisadores léxicos, a ferramenta JLex se mostra de grande utilidade, uma vez que permite que os tokens da linguagem sejam especificados através das expressões regulares correspondentes. Dessa forma, o problema de gerar um analisador léxico é reduzido ao problema de determinar expressões regulares para cada um dos tokens da linguagem.

Já para gerar analisadores sintáticos, a ferramenta CUP funciona de forma coordenada com o JLex. Nela, basta escrever, usando a sintaxe apropriada, as regras da gramática da linguagem. Além disso também pode-se especificar outras ações do analisador de acordo com as necessidades do usuário.

3 Compilação e Uso

Para facilitar a compilação do projeto, foi disponibilizado um arquivo Makefile. Nele, há cinco alvos: **clean**, limpa a compilação do projeto; **lex**, compila o analisador léxico; **cup**, compila o analisador sintático; **all**, compila ambos os analisadores e **run** executa os analisadores.

Além disso, para configurar corretamente o projeto, é necessário incluir os arquivos necessários às ferramentas JLex e CUP, como mostrado na Figura 2.

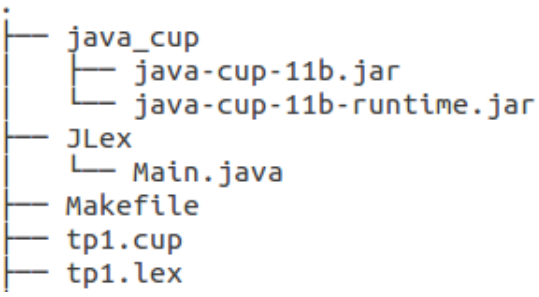

Figura 2: Estrutura do diretório do projeto.

Ao executar os analisadores, o programa fonte passado como entrada é analisado léxica e sintaticamente. Na saída, o programa fonte é impresso até onde o analisador léxico é bem sucedido, dessa forma, ao ocorrer um error léxico, o mesmo é registrado no ponto do programa onde ocorreu.

Caso a análise léxica seja completada sem erros, o analisador sintático imprimirá, em caso de sucesso, todas as produções gramaticais utilizadas no programa e uma mensagem indicando sucesso.

4 Testes e Resultados

A fim de ilustar o uso dos analisadores desenvolvidos, nessa seção são mostrados dois exemplos de teste.

4.1 input1.txt

- Entrada

```
{
        int ab2c;
        float def091;
        def091 = -.90;
        ab2c = 90 * 90;

        if (90 & 2) {
                ab2c = 89 - 2;
        }
}
```

- Saída

```
----------------------
{
        int ab2c;
        float def091;
        def091 = -.90;
```

```
        ab2c = 90 * 90;

        if (90

[ERROR] Illegal character: &
```

4.2 input2.txt

- Entrada

```
{
        {
                {
                        float num;
                        num = +912381.000002E-12;
                }
        }
}
```

- Saída

```
----------------------
{
        {
                {
                        float num;
                        num = +912381.000002E-12;
                }
        }
}
----------------------
program -> block
block -> { decls stmts }
decls ->

stmts -> stmts stmt
stmts ->

stmt -> block
block -> { decls stmts }
decls ->

stmts -> stmts stmt
stmts ->

stmt -> block
block -> { decls stmts }
decls -> decls decl
decls ->

decl -> type id;
type -> float

stmts -> stmts stmt
stmts ->

stmt -> id = expr;
expr -> term
term -> unary
unary -> factor
```

```
40 factor -> real
41
42 Program accepted.
```

5 Código Fonte

5.1 tp1.lex

```
1  package lexsyn;
2
3  import java_cup.runtime.Symbol;
4
5  %%
6  %cup
7  %%
8
9  [ \t\r\n\f] { /* ignore white space. */
10         System.out.print(yytext());
11 }
12
13 "{" {
14         System.out.print(yytext());
15         return new Symbol(sym.LCBRACK);
16 }
17
18 "}" {
19         System.out.print(yytext());
20         return new Symbol(sym.RCBRACK);
21 }
22
23 ";" {
24         System.out.print(yytext());
25         return new Symbol(sym.SEMI);
26 }
27
28 "(" {
29         System.out.print(yytext());
30         return new Symbol(sym.LPAREN);
31 }
32
33 ")" {
34         System.out.print(yytext());
35         return new Symbol(sym.RPAREN);
36 }
37
38 (int) {
39         System.out.print(yytext());
40         return new Symbol(sym.INTT);
41 }
42
43 (char) {
44         System.out.print(yytext());
45         return new Symbol(sym.CHART);
46 }
47
48 (bool) {
49         System.out.print(yytext());
50         return new Symbol(sym.BOOLT);
```

```
51  }
52
53  (float) {
54          System.out.print(yytext());
55          return new Symbol(sym.FLOATT);
56  }
57
58  "=" {
59          System.out.print(yytext());
60          return new Symbol(sym.ASSIGN);
61  }
62
63  (while) {
64          System.out.print(yytext());
65          return new Symbol(sym.WHILE);
66  }
67
68  (if) {
69          System.out.print(yytext());
70          return new Symbol(sym.IF);
71  }
72
73  (else) {
74          return new Symbol(sym.ELSE);
75  }
76
77  [A-Za-z][A-Za-z0-9]* {
78          System.out.print(yytext());
79          return new Symbol(sym.ID);
80  }
81
82  "<" {
83          System.out.print(yytext());
84          return new Symbol(sym.LT);
85  }
86
87  "<=" {
88          System.out.print(yytext());
89          return new Symbol(sym.LE);
90  }
91
92  ">=" {
93          System.out.print(yytext());
94          return new Symbol(sym.GE);
95  }
96
97  ">" {
98          System.out.print(yytext());
99          return new Symbol(sym.GT);
100 }
101
102 "+" {
103         System.out.print(yytext());
104         return new Symbol(sym.PLUS);
105 }
106
107 "-" {
108         System.out.print(yytext());
109         return new Symbol(sym.MINUS);
110 }
```

```
111
112 "*" {
113         System.out.print(yytext());
114         return new Symbol(sym.MUL);
115 }
116
117 "/" {
118         System.out.print(yytext());
119         return new Symbol(sym.DIV);
120 }
121
122 [+|-]?[0-9]*[.][0-9]+([E|e][+|-]?[0-9]+)? {
123         System.out.print(yytext());
124         return new Symbol(sym.REALN, new Double(yytext()));
125 }
126
127 [+|-]?[0-9]+([E|e][+|-]?[0-9]+)? {
128         System.out.print(yytext());
129         return new Symbol(sym.INTN, new Integer(yytext()));
130 }
131
132 . {
133         System.out.println("\n\n[ERROR] Illegal character: "+yytext());
134         System.exit(1);
135 }
```

5.2 tp1.cup

```
1  package lexsyn;
2
3  import java_cup.runtime.*;
4
5  parser code {:
6          public static void main(String args[]) throws Exception {
7                  System.out.println("----------------------");
8                  parser myParser = new parser(new Yylex(System.in));
9                  myParser.parse();
10                 System.out.print("Program accepted.");
11         }
12 :}
13
14 terminal LCBRACK, RCBRACK, SEMI, LPAREN, RPAREN;
15 terminal INTT, CHART, BOOLT, FLOATT;
16 terminal ASSIGN;
17 terminal WHILE, IF, ELSE;
18
19 terminal INTN;
20 terminal REALN;
21
22 terminal ID;
23
24 terminal LT, LE, GT, GE;
25 terminal PLUS, MINUS, MUL, DIV;
26
27 non terminal program, block, decls, decl, type, stmts, stmt, rel, expr, term;
28 non terminal unary, factor;
29
30 precedence left PLUS, MINUS, MUL, DIV, ELSE;
31
```

```
32  program        ::= block:b {:
33                                 System.out.println("\n----------------------");
34                                 String rules = new String("program -> block\n" + b);
35                                 System.out.print(rules);
36                             :}
37          ;
38  block         ::= LCBRACK decls:ds stmts:ss RCBRACK {:
39                                 RESULT = new String("block -> { decls stmts }\n" + ds + ss);
40                             :}
41          ;
42  decls         ::= decls:ds decl:d {:
43                                 RESULT = new String("decls -> decls decl\n" + ds + d);
44                             :}
45          |                   /* empty */ {:
46                                 RESULT = new String("decls -> \n\n");
47                             :}
48          ;
49  decl          ::= type:t ID SEMI {:
50                                 RESULT = new String("decl -> type id;\n" + t);
51                             :}
52          ;
53  type         ::= INTT {:
54                                 RESULT = new String("type -> int\n\n");
55                             :}
56          |                   CHART {:
57                                 RESULT = new String("type -> char\n\n");
58                             :}
59          |                   BOOLT {:
60                                 RESULT = new String("type -> bool\n\n");
61                             :}
62          |                   FLOATT {:
63                                 RESULT = new String("type -> float\n\n");
64                             :}
65          ;
66  stmts         ::= stmts:ss stmt:s {:
67                                 RESULT = new String("stmts -> stmts stmt\n" + ss + s);
68                             :}
69          |                    /* empty */ {:
70                                 RESULT = new String("stmts -> \n\n");
71                             :}
72          ;
73  stmt          ::= ID ASSIGN expr:e SEMI {:
74                                 RESULT = new String("stmt -> id = expr;\n" + e);
75                             :}
76          |                    IF LPAREN rel:r RPAREN stmt:s {:
77                                 RESULT = new String("stmt -> if ( rel ) stmt\n" + r + s);
78                             :}
79          |                    IF LPAREN rel:r RPAREN stmt:s1 ELSE stmt:s2 {:
80                                 RESULT = new String("stmt -> if ( rel ) stmt else stmt\n" +
81                                 r + s1 + s2);
82                             :}
83          |                   WHILE LPAREN rel:r RPAREN stmt:s {:
84                                 RESULT = new String("stmt -> while ( rel ) stmt\n" + r + s);
85                             :}
86          |                   block:b {:
87                                 RESULT = new String("stmt -> block\n" + b);
88                             :}
89          ;
90  rel          ::= expr:e1 LT expr:e2 {:
91                                 RESULT = new String("rel -> expr < expr\n" + e1 + e2);
```

```
92                              :}
93          |                     expr:e1 LE expr:e2 {:
94                                      RESULT = new String("rel -> expr <= expr\n" + e1 + e2);
95                              :}
96          |                     expr:e1 GT expr:e2 {:
97                                      RESULT = new String("rel -> expr > expr\n" + e1 + e2);
98                              :}
99          |                     expr:e1 GE expr:e2 {:
100                                     RESULT = new String("rel -> expr >= expr\n" + e1 + e2);
101                             :}
102         |                    expr:e {:
103                                     RESULT = new String("rel -> expr\n" + e);
104                             :}
105         ;
106 expr          ::= expr:e PLUS term:t {:
107                                     RESULT = new String("expr -> expr + term\n" + e + t);
108                             :}
109         |                     expr:e MINUS term:t {:
110                                     RESULT = new String("expr -> expr - term\n" + e + t);
111                             :}
112         |                     term:t {:
113                                     RESULT = new String("expr -> term\n" + t);
114                             :}
115         ;
116 term          ::= term:t MUL unary:u {:
117                                     RESULT = new String("term -> term * unary\n" + t + u);
118                             :}
119          |                     term:t DIV unary:u {:
120                                     RESULT = new String("term -> term / unary\n" + t + u);
121                             :}
122          |                     unary:u {:
123                                     RESULT = new String("term -> unary\n" + u);
124                             :}
125          ;
126 unary          ::= MINUS unary:u {:
127                                     RESULT = new String("unary -> - unary\n" + u);
128                             :}
129         |                     factor:f {:
130                                     RESULT = new String("unary -> factor\n" + f);
131                             :}
132         ;
133 factor         ::= INTN:n {:
134                                     RESULT = new String("factor -> num\n\n");
135                             :}
136         |                     REALN:n {:
137                                     RESULT = new String("factor -> real\n\n");
138                             :}
139         ;
```

5.3 Makefile

```
1  all: lex cup
2
3  .PHONY: lex
4  lex:
5          @ javac JLex/Main.java
6          @ java JLex.Main *.lex
7          @ mv *.lex.java Ylex.java
8
```

```
9  .PHONY: cup
10 cup:
11         @ java -jar java_cup/java-cup-11b.jar -interface -parser parser *.cup
12         @ javac -d . -cp java_cup/java-cup-11b-runtime.jar *.java
13
14 .PHONY: run
15 run:
16         @ java -cp java_cup/java-cup-11b-runtime.jar:. lexsyn.parser
17
18 .PHONY: clean
19 clean:
20         @ rm -f JLex/*.class *.lex.java
21         @ rm -f *.java
22         @ rm -rf lexsyn
```

6 Referências

- Aho, A.V.; Sethi, R.; Ullman, J.D. Compilers Principles, Techniques, and Tools, Addison Wesley, 1986.