

Compilador para a Linguagem *MLM*

Analizador sintático

Fábio Pereira e Henrique Pessoa

2 de Outubro de 2019

1 Introdução

Este trabalho tem como propósito a aplicação dos conceitos aprendidos ao decorrer da disciplina de Compiladores 2019-2. Para manter o conhecimento pratico em decorrência do modelo teórico apresentado pela disciplina, o trabalho foi dividido em várias etapas.

Para esta segunda entrega, foram corrigidos os problemas apontados no analisador léxico (Sessão **2**) e desenvolvido o **analisador sintático** (Sessão **3**) da linguagem **MLM** descrita na especificação do trabalho. Para isso, utilizamos a ferramenta de geração de análise léxica em C conhecida como *Flex* e o gerador de análise sintática *Yacc*.

2 Análise Léxica

Para utilizar o *Gerador de Analisador Léxico* em questão, precisamos definir todas as expressões necessárias para reconhecer os padrões da linguagem e quais serão as saídas para cada padrão reconhecido pelo analisador para futura utilização na tabela de símbolos.

A estrutura geral do *Flex* é definida da seguinte forma:

```
definicoes /* opcional */
%%
regras
%%
rotinas de usuario /* opcional */
```

Para nossa implementação, utilizamos todas as três possibilidades de módulos do *Flex*, em *definições* foram agrupadas todas as expressões regulares que interpretam as sequências lidas; em *regras* agrupamos todas as formas de retorno para a criação dos **tokens** na tabela de símbolos; e em *rotinas de usuário* nessa primeira etapa apenas da análise léxica, deixamos a função **main()** e a função **yywrap()** e posteriormente para a análise semântica adicionaremos outras funcionalidades.

A Figura 1 a seguir lista todas as definições adotadas para o reconhecimento das expressões da linguagem **MLM**.

```

sint > lex.l
1  %option noinput nounput
2  %{
3      /* Definition section */
4      #include <string.h>
5      #include <stdlib.h>
6      #include "y.tab.h"
7
8      void extern yyerror(char*);
9  %}
10
11  linefeed      \n
12  whitespace    [ \t\r\v\f]
13
14  addop         "+"|"-"|"or"
15  relop         "="|"<"|"<="|">"|>="|"!="|"NOT"
16  mulop         "*"|"/"|"div"|"mod"|"and"
17
18  digit         [0-9]
19  letter        [a-zA-Z]
20  identifier    {letter}{letter}|{digit}*
21
22  unsigned_integer {digit}+
23  sign            [+]?
24  scale_factor    "E"{sign}{unsigned_integer}
25  unsigned_real   {unsigned_integer}("."{digit})*?{scale_factor}?
26  integer_constant {unsigned_integer}
27  real_constant   {unsigned_real}
28  ascii           [\40-\176]
29  char_constant   "''{ascii}''"
30  boolean_constant "true"|"false"
31
32  /* Rule Section */
33  %%

```

Figura 1: Definições do analisador léxico para a linguagem *MLM*.

Feito isso, transcrevemos quais eram as regras para a linguagem **MLM** da especificação da documentação para o nosso analisador léxico utilizando as seguintes expressões, Figura 2. As saídas do analisar léxico foram corrigidas para retornar o par (*chave, valor*) como esperado para a tabela de símbolos do compilador.

```

34  /* Rule Section */
35  %%
36  {addop}      {yyval.strVal = strdup(yytext); return ADDOP;}
37  {relop}      {yyval.strVal = strdup(yytext); return RELOP;}
38  {mulop}      {yyval.strVal = strdup(yytext); return MULOP;}
39
40  "program"    {yyval.strVal = strdup(yytext); return PROGRAM;}
41  "integer"    {yyval.strVal = strdup(yytext); return INTEGER_C;}
42  "real"       {yyval.strVal = strdup(yytext); return REAL_C;}
43  "char"       {yyval.strVal = strdup(yytext); return CHAR_C;}
44  "boolean"    {yyval.strVal = strdup(yytext); return BOOL_C;}
45
46  "read"       {yyval.strVal = strdup(yytext); return READ;}
47  "write"      {yyval.strVal = strdup(yytext); return WRITE;}
48  "if"         {yyval.strVal = strdup(yytext); return IF;}
49  "then"       {yyval.strVal = strdup(yytext); return THEN;}
50  "else"       {yyval.strVal = strdup(yytext); return ELSE;}
51  "do"         {yyval.strVal = strdup(yytext); return DO;}
52  "begin"      {yyval.strVal = strdup(yytext); return BEGIN;}
53  "end"        {yyval.strVal = strdup(yytext); return END;}
54  "while"      {yyval.strVal = strdup(yytext); return WHILE;}
55  "until"      {yyval.strVal = strdup(yytext); return UNTIL;}
56
57  "=="         {return SINGLE_QUOTES;}
58  "=="         {return COMMA;}
59  "=="         {return SINGLE_COLON;}
60  "=="         {return COLON;}
61  "=="         {return ASSIGN;}
62  "=="         {return BRACKET_OPEN;}
63  "=="         {return BRACKET_CLOSE;}
64  "=="         {return CURLY_BRACE_OPEN;}
65  "=="         {return CURLY_BRACE_CLOSE;}
66  "=="         {return BIG_BRACKET_OPEN;}
67  "=="         {return BIG_BRACKET_CLOSE;}
68
69  {integer_constant} {yyval.intVal = atoi(yytext); return INTEGER_C;}
70  {real_constant}   {yyval.realVal = atof(yytext); return REAL_C;}
71  {char_constant}   {yyval.charVal = yytext[1]; return CHAR_C;}
72  {boolean_constant} {yyval.strVal = strdup(yytext); return BOOL_C;}
73  {identifier}      {yyval.strVal = strdup(yytext); return IDENTIFIER;}
74
75  {linefeed}        {yylineno++;}
76  {whitespace}      ;
77  .                 {yyerror(yytext);}
78  %%

```

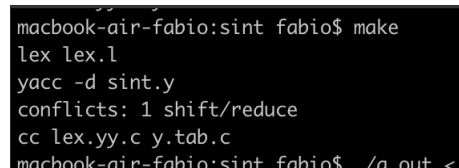
Figura 2: Regras do analisador léxico para a linguagem *MLM* retornando o par (*chave, valor*)

Por fim, para a fase de implementação apenas do analisador léxico a rotina de usuário é responsável apenas por conter a função **main()** e o retorno padrão da linguagem. Posteriormente na fase de análise semântica adicionaremos, também, as formas de busca e geração da árvore de derivação para o código lido.

3 Análise Sintática

A segunda parte do trabalho é referente à construção e validação da gramática da linguagem *MLM*. Para validar a implementação, consideramos o primeiro erro encontrado como fatal e a execução é encerrada exibindo em qual linha do código o erro foi encontrado. O analisador sintático tem como entrada para a análise do **Yacc** a saída correspondente do **Lex**, ou seja, utilizamos os **tokens** criados na primeira etapa do trabalho para a derivação da gramática. Para isso, primeiramente corrigimos a inserção dos tokens feitos na análise léxica e, definimos os *tokens* utilizados de forma a referenciar aqueles gerados no *Lex*, tomando como base o cabeçalho (*y.tab.h*) gerado pelo *Yacc* para definir os valores que são utilizados como retorno pelo *Lex* no momento do reconhecimento de um lexema, e por isso o incluímos no código do *Lex* apresentado anteriormente na Figura 1. Isso é importante pois é esta ligação que garante que os *tokens* definidos anteriormente serão associados corretamente na derivação da gramática.

Além disso, é importante ressaltar que foi necessário efetuar leves alterações na gramática. A principal alteração está relacionada a um problema de ambiguidade denominado *Dangling Else*. Em suma, trata-se de uma cláusula *else* em um *if-then-else* que resulta em condicionais aninhadas ambíguas, fazendo com que o *Yacc* acuse *Shift/Reduce* como podemos ver na compilação descrita na Figura 3 abaixo. Para resolver este problema, utilizamos a diretiva *nonassoc* para a definição dos *tokens* *THEN* e *ELSE*.



```
macbook-air-fabio:sint fabio$ make
lex lex.l
yacc -d sint.y
conflicts: 1 shift/reduce
cc lex.yy.c y.tab.c
macbook-air-fabio:sint fabio$ ./a.out <
```

Figura 3: Conflito shift/reduce durante a compilação do analisador sintático.

Após essa correção, descrevemos abaixo a implementação do nosso analisador sintático **sint.y**:

```
%{
    /* Definition section */
    #include<stdio.h>
    #include<stdlib.h>

    extern void yyerror();
    extern int yylineno;
    extern int yylex();
}%

%union {
    int intVal;
    char* dataType;
    char* strVal;
    float realVal;
    char charVal;
}

%token PROGRAM
%token COMMA SINGLE_QUOTES SEMI_COLON ASSIGN
%token BRACKET_OPEN BRACKET_CLOSE CURLY_BRACE_OPEN
    CURLY_BRACE_CLOSE BIG_BRACKET_OPEN BIG_BRACKET_CLOSE
%token COLON
```

```

%token <strVal> ADDOP
%token <strVal> RELOP
%token <strVal> MULOP

%token <strVal> READ
%token <strVal> WRITE
%token <strVal> IF
%token <strVal> DO
%token <strVal> BEGIN_
%token <strVal> END
%token <strVal> WHILE
%token <strVal> UNTIL

%token <strVal> IDENTIFIER
%token <intVal> INTEGER_C
%token <realVal> REAL_C
%token <charVal> CHAR_C
%token <strVal> BOOL_C

%nonassoc <strVal> THEN
%nonassoc <strVal> ELSE

/* Rule Section */
%%

program : PROGRAM IDENTIFIER SEMLCOLON decl_list compound_stmt
        { printf("done\n"); exit(0); }
        ;

decl_list : decl_list SEMLCOLON decl
          | decl
          ;

decl : ident_list COLON type

type : INTEGER_C
     | REAL_C
     | BOOL_C
     | CHAR_C ;

ident_list : ident_list COMMA IDENTIFIER
           | IDENTIFIER
           ;

compound_stmt : BEGIN_ stmt_list END
              ;

stmt_list : stmt_list SEMLCOLON stmt
          | stmt
          ;

stmt : assign_stmt
     | if_stmt
     | loop_stmt

```

```

        | read_stmt
        | write_stmt
        | compound_stmt
    ;

assign_stmt  :  IDENTIFIER ASSIGN 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
              | /* empty */
              ;

stmt_suffix :  UNTIL cond
              | END
              ;

read_stmt   :  READ BRACKET_OPEN ident_list BRACKET_CLOSE
              ;

write_stmt  :  WRITE BRACKET_OPEN expr_list BRACKET_CLOSE
              ;

expr_list   :  expr
              | expr_list COMMA expr
              ;

expr        :  simple_expr {printf("simple_expr\n");}
              | simple_expr RELOP simple_expr
                {printf("simple_expr_RELOP_simple_expr\n");}
              ;

simple_expr  :  term {printf("term\n");}
              | simple_expr ADDOP term
                {printf("simple_expr_ADDOP_term\n");}
              ;

term        :  factor_a {printf("factor_a\n");}
              | term MULOP factor_a {printf("term_MULOP_factor_a\n");}
              ;

factor_a    :  factor {printf("factor\n");}
              ;

factor      :  IDENTIFIER {printf("identifier\n");}
              | constant {printf("constant\n");}

```

```

        | BRACKET_OPEN expr BRACKET_CLOSE { printf("(expr)\n"); }
    ;

constant : INTEGER_C { printf("integer_constant\n"); }
        | REAL_C      { printf("real_constant\n"); }
        | CHAR_C      { printf("char_constant\n"); }
        | BOOL_C      { printf("bool_constant\n"); }
    ;

%%

int main()
{
    yyparse();
    printf("Accepted\n");
    return 0;
}

```

Após a descrição do código implementado, podemos notar uma segunda alteração necessária na definição dos tokens que foi a alteração do nome do token **BEGIN** para **BEGIN_** já que essa palavra deu conflito com outra definição reservada do **Yacc**.

4 Execução

4.1 Analisador Léxico

O fluxo de execução do *Flex* é dividido em duas etapas. Primeiro executamos o comando *flex* seguido do nosso arquivo que contém a definição para o gerador de analisador léxico e, posteriormente, executamos o comando *cc* seguido do novo arquivo gerado no formato *.yy.c*. A sequência de passos a ser executados foram agrupados no seguinte **Makefile**:

```

all:
    flex MM.lex
    cc lex.yy.c -lfl
clean:
    rm a.out lex.yy.c

```

Este fluxo é melhor visualizado no diagrama abaixo, Figura 4.

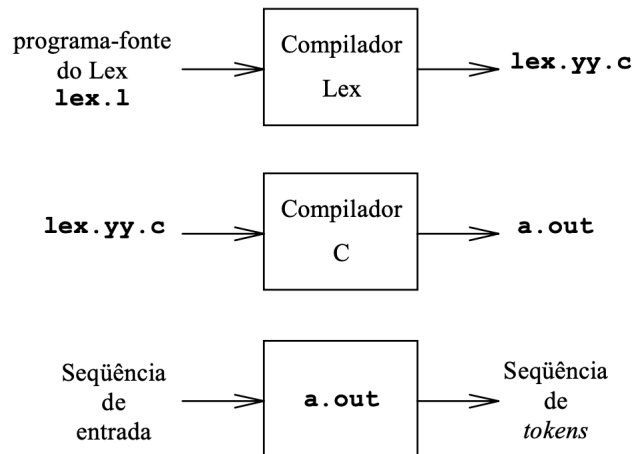


Figura 4: Fluxo de execução de um programa em *lex*.

Após a execução do *Makefile* para iniciar a análise da entrada basta utilizar o comando `./a.out` seguido de uma seqüência de comandos ou um arquivo contendo o código desejado, como no exemplo abaixo.

```
./a.out < codigo.mlm
```

4.2 Analisador Sintático

A execução sintática tem como entrada também um código escrito na linguagem *MLM* porém, sua compilação considera a tabela de símbolos, *tokens*, criados pelo analisador léxico. Para isso, criamos um novo arquivo *MakeFile* para simplificar o processo de compilação do nosso código sintático.

```
all :
    lex lex.l
    yacc -d sint.y
    cc lex.yy.c y.tab.c

free :
    rm a.out* lex.yy.c
```

Após a execução do *Makefile* acima, para iniciar a análise sintática da entrada basta utilizar o comando `./a.out` seguido de uma seqüência de comandos ou um arquivo contendo o código desejado escrito na linguagem *MLM*, como no exemplo abaixo.

```
./a.out < codigo.mlm
```

5 Resultados

5.1 Analisador Léxico

A avaliação dos resultados do analisador léxico desenvolvido foi feita a partir de alguns exemplos de código fonte, na linguagem **MLM**, escritos para esse propósito. Alguns desses códigos serão exibidos abaixo, assim como o resultado gerado.

5.1.1 Exemplo

Nesse exemplo, já considerando a correção da saída do analisador, foi criado um código simples com único intuito demonstrativo. Nele pode-se reconhecer exemplos de quase todos tipos de constantes e operadores de relação, assim como as estruturas de condição e *loop*. Na Figura 5 temos o código usado como entrada do analisador e na Figura 8 temos o retorno da análise já com o mapeamento dos tokens.

```

≡ example_input_lex1.mlm
1  begin
2      b : boolean
3      x : real
4
5      b := true
6      x := 0.5E-1
7
8      if x * 3 > 1 then x := x * 3
9
10     while b := true do
11         x := x-1
12         if x < 2 then b := false
13     end
14 end

```

Figura 5: Exemplo de código **MLM** e entrada do analisador léxico.

```

≡ example_output_lex1.mlm
1  <BEGIN, begin>
2
3      <IDENTIFIER, b> <COLON, :=> <TYPE, boolean>
4
5      <IDENTIFIER, x> <COLON, :=> <TYPE, real>
6
7
8      <IDENTIFIER, b> <ASSIGN, :=> <BOOLEAN_CONSTANT, true>
9
10     <IDENTIFIER, x> <ASSIGN, :=> <REAL_CONSTANT, 0.5E-1>
11
12     <IF, if> <IDENTIFIER, x> <MULOP, *> <INTEGER_CONSTANT, 3> <RELOP, >> <INTEGER_CONSTANT, 1> <THEN, then>
13         <IDENTIFIER, x> <ASSIGN, :=> <IDENTIFIER, x> <MULOP, *> <INTEGER_CONSTANT, 3>
14
15
16     <WHILE, while> <IDENTIFIER, b> <ASSIGN, :=> <BOOLEAN_CONSTANT, true> <DO, do>
17         <IDENTIFIER, x> <ASSIGN, :=> <IDENTIFIER, x> <ADDOP, -> <INTEGER_CONSTANT, 1>
18
19         <IF, if> <IDENTIFIER, x> <RELOP, <> <INTEGER_CONSTANT, 2> <THEN, then>
20             <IDENTIFIER, b> <ASSIGN, :=> <BOOLEAN_CONSTANT, false>
21     <END, end>
22
23 <END, end>
24

```

Figura 6: Exemplo de código **MLM** e saída do analisador léxico.

5.2 Analisador Sintático

5.2.1 Exemplo 1: Soma de dois valores constantes

O primeiro exemplo desenvolvido para o analisador sintático é um simples trecho de código na linguagem **MLM** para demonstrar a sua árvore de derivação sendo gerada como saída de sua execução. Esse código consiste simplesmente de algumas declarações e operações a partir de atribuições e é exibido abaixo, com sua saída na Figura 7.

Entrada:

```

program SOMA2;
valueA, valueB, sumAB: integer
begin
    valueA := 10;
    valueB := 20;
    sumAB := valueA + valueB

```



```
end
```

Saída:

```
henrique@pc:~/Documents/UFMG/Compiladores/TP/Linguagem-MLM/sint$ ./a.out < example_input.mlm
ident_list
ident_list
ident_list
type
decl
decl_list
constant
factor
factor_a
term
simple_expr
expr
assign_stmt
stmt
stmt_list
constant
factor
factor_a
term
simple_expr
expr
assign_stmt
stmt
stmt_list
factor
factor_a
term
simple_expr
factor
factor_a
term
simple_expr
expr
assign_stmt
stmt
stmt_list
compound_stmt
program
Accepted
henrique@pc:~/Documents/UFMG/Compiladores/TP/Linguagem-MLM/sint$
```

Figura 7: Exemplo de saída do analisador sintático com a impressão do passo a passo da árvore de análise.

5.2.2 Exemplo 2: Fluxo de repetição com erro

Foi desenvolvido ainda um segundo exemplo de código **MLM**, no qual o propósito era mostrar a detecção de erros do analisador sintático desenvolvido. Nesse código, há um programa inicialmente bem definido, que realiza algumas declarações e definições, um *if statement* e por fim um *loop statement*, usando o comando *while* de maneira errada, com uma atribuição ao invés de uma condição. Dessa forma, era de se esperar a detecção desse erro na linha em que aparece (linha 10), com a leitura da árvore de derivação ocorrendo normalmente até então. Como exibido na Figura 8, é exatamente isso que acontece.

Entrada:

```
program SOMA2;
x: real;
b: boolean
begin
    b := true;
    x := 0.5E-1;

    if x * 3 > 1 then x := x * 3

    while b := true do
        x := x-1
        if x < 2 then b := false
    end
end
```

Saída:

```
henrique@pc:~/Documents/UFG/Compiladores/TP/linguagem-mlm/sint$ ./a.out < example_erro_input.mlm
ident_list
type
decl
decl_list
ident_list
type
decl
decl_list
constant
factor
factor_a
term
simple_expr
expr
assign_stmt
stmt
stmt_list
constant
factor
factor_a
term
simple_expr
expr
assign_stmt
stmt
stmt_list
factor
factor_a
term
constant
factor
factor_a
term
simple_expr
constant
factor
factor_a
term
simple_expr
expr
cond
factor
factor_a
term
constant
factor
factor_a
term
simple_expr
expr
assign_stmt
stmt
if stmt
stmt
stmt_list
ERROR ON LINE 10 :
syntax error
henrique@pc:~/Documents/UFG/Compiladores/TP/linguagem-mlm/sint$
```

Figura 8: Exemplo de saída do analisador sintático com erro sintático na linha 10 do código.

6 Conclusão

Nessa segunda etapa do trabalho foi possível ter uma visão melhor de todo o processo da qual um compilador é feito. Nele, conseguimos entender como é feita a ligação entre o processo léxico e o sintático por meio da tabela de símbolos, **tokens**, e como aplicar esses conceitos junto ao **Yacc**.

Tivemos algumas dificuldades iniciais nesse processo de referencia entre os dois processos e, também, na sintaxe correta do analisador sintático. Além disso, foi possível identificar os erros iniciais do analisador léxico e corrigi-los nessa segunda entrega para não comprometer a entrega final do trabalho.

Por fim, essa segunda etapa foi de grande importância para o entendimento da matéria dada em sala de aula assim como, para aplicar esses conceitos de forma mais prática diante de um cenário real de uma linguagem seguindo sua definição formal.

Referências

Compiladores - JFlex. Fabio Mascarenhas – 2018.1. Disponível em: <https://dcc.ufrj.br/~fabiom/comp/04JFlex.pdf>. Acesso em 05 de setembro de 2019.

Lex - A Lexical Analyzer Generator. M. E. Lesk and E. Schmidt. Disponível em: <http://dinosaur.compilertools.net/lex/>

Lex and Yacc: A Brisk Tutorial. Saumya K. Debray. Department of Computer Science The University of Arizona Tucson, AZ 85721. Disponível em: <https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/tutorial-large.pdf>

Compilador para a linguagem MLM. Bigonha, Mariza A. S. Disponível em: <https://homepages.dcc.ufmg.br/~mariza/Cursos/CompiladoresI/Comp2019-2/Pagina/Dia-a-Dia/comp2019-2.html>