TP1/TP2 - Analisador Sintático

Felipe Cadar Chamone - 2016006417
João Pedro Pinto Coelho - 2015125161

Outubro de 2019

1 Introdução

Esse trabalho visa desenvolver um compilador para a linguagem MLM (Mini Linguagem M) apresentada na especificação. A linguagem MLM é uma versão reduzida de Pascal. Essa documentação descreve nosso desenvolvimento das primeiras duas etapas do compilador.

A primeira etapa é elaborar o analisador léxico, para agrupar as sequências de caracteres em *tokens*, a partir das convenções léxicas fornecidas.

A segunda etapa é o analisador sintático. Usando os *tokens* gerados na etapa anterior, o analisador sintático gera frases gramaticais para descrever a linguagem.

2 Analisador Léxico

O analisador léxico deve separar todo o programa em *tokens* que serão usados na próxima etapa do compilador. Para fazer esse analisador usamos uma ferramenta que gera analisadores léxicos a partir de definições de expressões regulares, o FLEX.

Segue o código de definição:

```
%{
#include "tab.h"
%}

DIGIT               [0-9]
LETTER              [A-zA-Z]
IDENTIFIER          {LETTER}({LETTER}|{DIGIT})*

RELOP               "="|"<"|"<="|">"|">="|"!="
ADDOP               "+"|"-"|"or"
MULOP               "*"|"/"|"div"|"mod"|"and"
```

```
SIGN                [+-]?

UNSIGNED_INTEGER    {DIGIT}+
SCALE_FACTOR        "E"{SIGN}{UNSIGNED_INTEGER}
UNSIGNED_REAL       {UNSIGNED_INTEGER}(("."{DIGIT}({DIGIT})*)?)(({SCALE_FACTOR})?)
INTEGER_CONSTANT    UNSIGNED_INTEGER
REAL_CONSTANT       UNSIGNED_REAL
CHAR_CONSTANT       "'"{LETTER}"'"

%%


program             {return PROGRAM;}
integer             {return INTEGER;}
real                {return REAL;}
boolean             {return BOOLEAN;}
char                {return CHAR;}
PROCEDURE           {return PROCEDURE;}
value               {return VALUE;}
reference           {return REFERENCE;}
begin               {return BEGIN;}
end                 {return END;}
if                  {return IF;}
then                {return THEN;}
else                {return ELSE;}
repeat              {return REPEAT;}
until               {return UNTIL;}
read                {return READ;}
write               {return WRITE;}
not                 {return NOT;}
false               {return TFALSE;}
true                {return TTRUE;}
[,]                 {return COLON;}
[;]                 {return SEMICOLON;}
[:]                 {return TWOPOINTS;}
[)]                 {return RPARENTHESES;}
[(]                 {return LPARENTHESES;}
":="                {return ATRIB;}
NOT                 {return NOT;}
{RELOP}             {return RELOP;}
{ADDOP}             {return ADDOP;}
{MULOP}             {return MULOP;}
{IDENTIFIER}        {return IDENTIFIER;}
{UNSIGNED_INTEGER}  {return UNSIGNED_INTEGER;}
{UNSIGNED_REAL}     {return UNSIGNED_REAL;}
{SIGN}              {return SIGN;}
```

```
{SCALE_FACTOR}      {return SCALE_FACTOR;}
{INTEGER_CONSTANT}  {return INTEGER_CONSTANT;}
{REAL_CONSTANT}     {return REAL_CONSTANT;}
{CHAR_CONSTANT}     {return CHAR_CONSTANT;}
\n                  {}
[ \t]+              {}

%%
void yyerror(char const *error){
    printf("FAIL:%s\n",error);
}
```

Nele primeiro definimos as expressões regulares que reconhecem cada *token* da linguagem, e em seguida definimos ações a serem feitas sempre que reconhecemos uma dessas *token* durante a leitura do programa de entrada.

3 Analisador Sintático

Nessa etapa, usamos os tokens gerados pelo analisador léxico para construir a gramática que descreve a linguagem.

Aqui, usamos o YACC (*Yet Another Compiler Compiler*), ferramenta que gera o analisador sintatico a partir de uma descrição da gramática. Como entrada, ele depende da saída do Flex gerada anteriormente.

```
%{ /* Declarações */
#include <stdio.h>
%}
%token PROGRAM
%token INTEGER
%token REAL
%token BOOLEAN
%token CHAR
%token PROCEDURE

%token VALUE
%token REFERENCE
%token BEGIN
%token END
%token IF
%token THEN
%token ELSE
%token REPEAT
%token UNTIL
%token READ
%token WRITE
%token TFALSE
```

```
%token TTRUE
%token SEMICOLON
%token LPARENTHESES
%token RPARENTHESES
%token TWOPOINTS
%token COLON
%token MINUS
%token ASSIGNOP
%token NOT
%token RELOP
%token ADDOP
%token MULOP
%token LETTER
%token DIGIT
%token IDENTIFIER
%token UNSIGNED_INTEGER
%token SIGN
%token SCALE_FACTOR
%token UNSIGNED_REAL
%token INTEGER_CONSTANT
%token REAL_CONSTANT
%token CHAR_CONSTANT

%%
program : PROGRAM IDENTIFIER SEMICOLON decl_list
compound_stmt
;
decl_list : decl_list SEMICOLON decl
| decl
;

decl : dcl_var
| dcl_proc
;
dcl_var : ident_list TWOPOINTS type
;
ident_list : ident_list COLON IDENTIFIER
| IDENTIFIER
;
type : INTEGER
| REAL
| BOOLEAN
| CHAR
;
dcl_proc : tipo_retornado PROCEDURE IDENTIFIER
espec_parametros corpo
```

```
;
tipo_retornado : INTEGER
| REAL
| BOOLEAN
| CHAR
| %empty/* empty */
;
corpo : TWOPOINTS decl_list SEMICOLON
compound_stmt id_return
;
id_return : IDENTIFIER
| %empty/* empty */
;

espec_parametros : LPARENTHESES lista_de_parametros RPARENTHESES
;
lista_de_parametros : parametro
| lista_de_parametros COLON parametro
;
parametro : modo type TWOPOINTS IDENTIFIER
;

modo : VALUE
| REFERENCE
;
compound_stmt : BEGIN stmt_list END
;
stmt_list : stmt_list SEMICOLON stmt
| stmt
;
stmt : assign_stmt
| if_stmt
| repeat_stmt
| read_stmt
| write_stmt
| compound_stmt
| function_ref_par
;
assign_stmt : IDENTIFIER ASSIGNOP expr
;
if_stmt : IF cond THEN stmt
| IF cond THEN stmt ELSE stmt
;
cond : expr
;
```

```
repeat_stmt : REPEAT stmt_list UNTIL expr
;
read_stmt : READ LPARENTHESES ident_list RPARENTHESES
;
write_stmt : WRITE LPARENTHESES expr_list RPARENTHESES
;
expr_list : expr
| expr_list COLON expr
;
expr : Simple_expr
| Simple_expr RELOP Simple_expr
;

Simple_expr : term
| Simple_expr ADDOP term
;
term : factor_a
| term MULOP term
;
factor_a : MINUS factor
| factor
;
factor : IDENTIFIER
| constant
| LPARENTHESES expr RPARENTHESES
| NOT factor
| function_ref_par
;
function_ref_par : variable LPARENTHESES expr_list RPARENTHESES
;
variable : Simple_variable_or_proc
;
Simple_variable_or_proc : IDENTIFIER
;

constant : INTEGER_CONSTANT
| REAL_CONSTANT
| CHAR_CONSTANT
| boolean_constant
;
boolean_constant : TFALSE
| TTRUE
;
%%
int yywrap(){
return 1;
```

```
}
int main(){
yyparse();

return 1;
}
```

4 Como usar as ferramentas

A instalação das ferramentas no linux pode ser feita com:

```
apt-get install flex # para o analisador léxico
apt-get install bison # para o analisador sintático
```

Para gerar os analisadores primeiros temos que considerar os arquivos "Ylex.lex" com as definiçoes léxicas e o "Grm.yacc" com as definiçoes sintáticas.

```
yacc -d Grm.yacc
flex Ylex.lex
gcc lex.yy.c y.tab.c -o analisador.out
```

E para usar simplesmente executamos

```
./analisador.out < input_program.mlm
```

5 Conclusão

Essas etapas iniciais mostram a importância dos compiladores. A partir de uma sequência de carateres, foi possível gerar unidades de significado, os *tokens*, e usá-los para especificar uma gramática.

A tarefa de fazer o computador "compreender" o texto que os programadores escrevem é fundamental para a computação moderna, e usando as estruturas e técnicas apresentadas em sala conseguimos passar de apenas um texto para uma gramática que começa a tomar uma forma que pode ser eficientemente processada e utilizada.