Compilador TIGER:
TP1 – Analisador Léxico

Professora: Marisa A. S. Bigonha
Aluna: Scarlet Gianasi Viana
Matrícula: 2016006891

1. Introdução

O objetivo desse trabalho foi implementar a primeira parte de um compilador para a linguagem **TIGER**: a análise léxica. Para tal, foi utilizado o gerador de analisadores léxicos **Flex**, utilizando a linguagem **C**.

2. Análisador Léxico

Esta parte da compilação de um programa tem como função receber o arquivo fonte como entrada e dividí-lo em um fluxo de **tokens**, que serão posteriormente repassados ao analisador sintático.

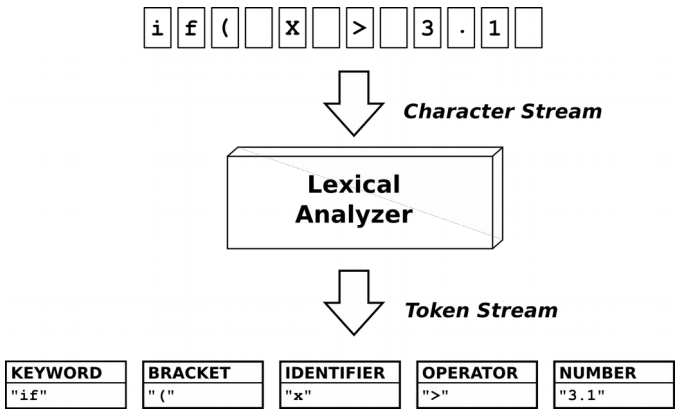

Imagem 1: exemplo ilustrativo da entrada e saída do analisador léxico.

2.1. Tokens

Tokens são pares **<nome, valor>** que contêm os nomes dos tokens da linguagem e seu valor (quando existe). Para cada token existe um ou mais lexemas, que são os padrões que casam com cada token.

2.1.2. Tokens na linguagem TIGER:

Existem 45 tokens na linguagem, descritos abaixo:

- **Palavras-chave:** *array, if, then, else, while, for, to, do, let, in, end, of, break, nil, function, var, type, import, primitive*.
- **Símbolos:** , : ; ( ) [ ] { } . + * / = <> < > <= >= & | :=
- **Tipos:** *número* (digitos de 0 a 9), *string* (quaisquer caracteres e/ou dígitos entre aspas), *identificador* (qualquer caractere seguido por qualquer outro dígito ou caractere).

2.1.2. Outros símbolos

Além dos tokens da linguagem, TIGER também permite o uso de **espaços em branco**, **quebra de linhas** e **comentários aninhados** (quaisquer dígitos ou caracteres entre **/*** e ***/**). Todos estes

símbolos são descartados pelo analisador léxico, pois não são tokens propriamente ditos, mas apenas símbolos que facilitam a escrita e leitura do código pelos programadores.

2.2. LEX

Para gerar o analisador léxico, como previamente descrito, foi utilizado o gerador de analisador léxico **Flex**, que recebe o código **tiger.lex** que possui todas as definições de tokens, expressões regulares e qualquer código C que será executado na análise léxica.

O arquivo **tiger.lex** é composto por três partes divididas por **%%**:

- Definições
- Regras
- Código do usuário

As mesmas serão descritas a seguir.

2.2.1. Definições

Essa seção possui a declaração das definições de todos os tokens da linguagem, descritos na seção 2.1.2 da documentação. Também possui duas declarações, **%option noyywrap** (pois só se usa um arquivo de entrada neste projeto) e **%x C_COMMENT** para definir o estado C_COMMENT, que será utilizado para reconhecer comentários aninhados, na seção de regras.

2.2.2. Regras

Esta é a seção mais importante do arquivo **tiger.lex**. Ela descreve as regras e expressões regulares que identificam os tokens e o código em C que o analisador executará ao reconhecer cada token.

Para este trabalho, a cada token válido encontrado, o mesmo e seu valor são impressos no formato **<nome, valor>**, e então o valor de sua definição (descrito na seção de definições) é retornado. Quando o analisador sintático for implementado, a impressão dos tokens se tornará desnecessária, porém está presente para meios de teste do funcionamento do analisador léxico.

Para cada tipo de token, sua regra foi descrita. Para os tokens palavra-chave, a regra é apenas a sequência de caracteres que formam o nome do token. Ex.: regra do token “array” é **array**. Para os símbolos, sua regra é dada pelo símbolo propriamente dito entre aspas. Ex.: regra de “+” é **“+”**. Para os tipos, outras regras mais complexas foram utilizadas, são descritas abaixo:

- *Regra do ID:* **[a-zA-Z][a-zA-Z0-9_]***
  Significa um caractere maiúsculo ou minúsculo seguido de 0 ou mais caracteres ou dígitos, seguidos ou não de “**_**” (pela definição da linguagem).
- *Regra do número:* **[0-9]+**
  Significa pelo o menos um dígito de 0 a 9 seguido por outros dígitos.
- *Regra da string:* **\"(\\.|[^"\\])*\"**
  Significa qualquer sequência de caracteres ou dígitos ou qualquer outra coisa fechados entre aspas.

Os outros símbolos também possuem regras, mas eles não retornan nada pois não são tokens, mas precisam ter regras definidas para que o analisador consiga identificá-los.

Os comentários têm como regra quaisquer dígitos ou números fechados entre **/*** e ***/** , mas como podem ser aninhados, eles caracterizam um estado, C_COMMENT que permite reconhecer todos os blocos de comentários, e somente depois disso voltar ao estado inicial.

Os espaços são reconhecidos pela regra **“ “**, as quebras de linha por **“\n”** e tabulações por **[\t\f]** , mas não retornam nada, são apenas reconhecidos. Para ajudar na leitura da saída dos analisador, a cada quebra de linhas é impressa uma quebra também, evitando blocos muito extensos de tokens.

Qualquer outro padrão que não foi reconhecido até então é determinado pela regra “**.**” e uma mensagem de erro é impressa.

2.2.3. Código do Usuário

Nesta seção foi definida apenas a função **main()**, que é responsável por realizar as chamadas de **yylex()**, que reconhece um token entre os padrões dados pelas regras. A chamada dessa função é responsabilidade do analisador sintático, mas como o mesmo ainda não foi implementado, o analisador léxico realiza a chamada enquanto houverem caracteres a serem lidos para fins de testes do reconhecimento das expressões.

3. O arquivo tiger.lex

O programa **tiger.lex** em que se gera o analisador léxico implementado se encontra abaixo:

```
%{
#include<stdio.h>
#include<string.h>

/* Definicoes */

// Palavras-chave
#define ARRAY 0
#define IF 1
#define THEN 2
#define ELSE 3
#define WHILE 4
#define FOR 5
#define TO 6
#define DO 7
#define LET 8
#define IN 9
#define END 10
#define OF 11
#define BREAK 12
#define NIL 13
#define FUNCTION 14
#define VAR 15
#define TYPE 16
#define IMPORT 17
```

```
#define PRIMITIVE 18

// Simbolos
#define COMMA 19
#define COLON 20
#define SEMICOLON 21
#define LEFT_PARENT 22
#define RIGHT_PARENT 23
#define LEFT_BRACKET 24
#define RIGHT_BRACKET 25
#define LEFT_BRACE 26
#define RIGHT_BRACE 27
#define DOT 28
#define PLUS 29
#define MINUS 30
#define TIMES 31
#define DIVIDE 32
#define EQUAL 33
#define NOT_EQUAL 34
#define LESS 35
#define LESS_EQUAL 36
#define GREATER 37
#define GREATER_EQUAL 38
#define AND 39
#define OR 40
#define ASSIGN 41

// Tipos
#define NUMBER 42
#define STRING 43
#define ID 44
%}

%option noyywrap
%x C_COMMENT

/* Regras */
%%

array       { printf("<ARRAY, %s> ", yytext); return ARRAY; }
if          { printf("<IF, %s> ", yytext); return IF; }
then        { printf("<THEN, %s> ", yytext); return THEN; }
else        { printf("<ELSE, %s> ", yytext); return ELSE; }
while       { printf("<WHILE, %s> ", yytext); return WHILE; }
for         { printf("<FOR, %s> ", yytext); return FOR; }
to          { printf("<TO, %s> ", yytext); return TO; }
do          { printf("<DO, %s> ", yytext); return DO; }
let         { printf("<LET, %s> ", yytext); return LET; }
in          { printf("<IN, %s> ", yytext); return IN; }
end         { printf("<END, %s> ", yytext); return END; }
of          { printf("<OF, %s> ", yytext); return OF; }
break       { printf("<BREAK, %s> ", yytext); return BREAK; }
nil         { printf("<NIL, %s> ", yytext); return NIL; }
function    { printf("<FUNCTION, %s> ", yytext); return FUNCTION; }
var         { printf("<VAR, %s> ", yytext); return VAR; }
type        { printf("<TYPE, %s> ", yytext); return TYPE; }
import      { printf("<IMPORT, %s> ", yytext); return IMPORT; }
primitive   { printf("<PRIMITIVE, %s> ", yytext); return PRIMITIVE; }
","         { printf("<COMMA, %s> ", yytext); return COMMA; }
```

```
":"         { printf("<COLON, %s> ", yytext); return COLON; }
";"         { printf("<SEMICOLON, %s> ", yytext); return SEMICOLON; }
"("         { printf("<LEFT_PARENT, %s> ", yytext); return LEFT_PARENT; }
")"         { printf("<RIGHT_PARENT, %s> ", yytext); return RIGHT_PARENT; }
"["         { printf("<LEFT_BRACKET, %s> ", yytext); return LEFT_BRACKET; }
"]"         { printf("<RIGHT_BRACKET, %s> ", yytext); return RIGHT_BRACKET; }
"{"         { printf("<LEFT_BRACE, %s> ", yytext); return LEFT_BRACE; }
"}"         { printf("<RIGHT_BRACE, %s> ", yytext); return RIGHT_BRACE; }
"."         { printf("<DOT, %s> ", yytext); return DOT; }
"+"         { printf("<PLUS, %s> ", yytext); return PLUS; }
"-"         { printf("<MINUS, %s> ", yytext); return MINUS; }
"*"         { printf("<TIMES, %s> ", yytext); return TIMES; }
"/"         { printf("<DIVIDE, %s> ", yytext); return DIVIDE; }
"="         { printf("<EQUAL, %s> ", yytext); return EQUAL; }
"<>"        { printf("<NOT_EQUAL, %s> ", yytext); return NOT_EQUAL; }
"<"         { printf("<LESS, %s> ", yytext); return LESS; }
"<="        { printf("<LESS_EQUAL, %s> ", yytext); return LESS_EQUAL; }
">"         { printf("<GREATER, %s> ", yytext); return GREATER; }
">="        { printf("<GREATER_EQUAL, %s> ", yytext); return GREATER_EQUAL; }
"&"         { printf("<AND, %s> ", yytext); return AND; }
"|"         { printf("<OR, %s> ", yytext); return OR; }
":="        { printf("<ASSIGN, %s> ", yytext); return ASSIGN; }

[a-zA-Z][a-zA-Z0-9_]*   { printf("<ID, %s> ", yytext); return ID; }
[0-9]+                  { printf("<NUMBER, %s> ", yytext); return NUMBER; }
\"(\\.|[^"\\])*\"       { printf("<STRING, %s> ", yytext); return STRING; }

"/*"                     { BEGIN(C_COMMENT); }
<C_COMMENT>"*/"          { BEGIN(INITIAL); }
<C_COMMENT>\n            { continue; }
<C_COMMENT>.             { continue; }

" "         { continue; }
[ \t\f]     { continue; }
"\n"        { printf("\n"); continue; }
.           { printf("Syntax error."); continue;}

%%
/* Codigo do usuario */
int main() {
    while(1) {
      yylex();
    };
    return 0;
}
```

3.1. Execução do código

Para realizar a análise léxica de um código, primeiramente o analisador léxico propriamente dito deve ser gerado. No caso, será gerado pelo Flex, então o mesmo deve estar instalado na máquina que for utilizá-lo. Sendo assim, basta gerar o analisador com o comando **lex tiger.lex** no terminal. O analisador léxico gerado terá o nome “lex.yy.c”, e o mesmo então deve ser compilado com o comando **cc lex.yy.c -lfl**. Assim, basta inserir a entrada com **./a.out**, e o analisador léxico imprimirá na tela o resultado da análise.

4. Testes

Foram realizados testes com o analisador gerado e a partir dos resultados obtidos, é possível dizer que ele é capaz de compreender todas as regras da linguagem TIGER especificadas nessa documentação. Como exemplo do funcionamento do programa, seja a entrada o programa que calcula o problema das 8 Rainhas abaixo:

```
/* A program to solve the 8-queens problem */

let
    var N := 8

    type intArray = array of int

    var row := intArray [ N ] of 0
    var col := intArray [ N ] of 0
    var diag1 := intArray [N+N-1] of 0
    var diag2 := intArray [N+N-1] of 0

    function printboard() =
       (for i := 0 to N-1
       do (for j := 0 to N-1
            do print(if col[i]=j then " O" else " .");
           print("\n"));
         print("\n"))

    function try(c:int) =
( /*  for i:= 0 to c do print("."); print("\n"); flush();*/
     if c=N
     then printboard()
     else for r := 0 to N-1
         do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
                 then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
                     col[c]:=r;
                       try(c+1);
                   row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)

)
 in try(0)
end
```

A saída do analisador léxico foi:

```
<LET, let>
<VAR, var> <ID, N> <ASSIGN, :=> <NUMBER, 8>

<TYPE, type> <ID, intArray> <EQUAL, => <ARRAY, array> <OF, of> <ID, int>

<VAR, var> <ID, row> <ASSIGN, :=> <ID, intArray> <LEFT_BRACKET, [> <ID, N>
<RIGHT_BRACKET, ]> <OF, of> <NUMBER, 0>
<VAR, var> <ID, col> <ASSIGN, :=> <ID, intArray> <LEFT_BRACKET, [> <ID, N>
<RIGHT_BRACKET, ]> <OF, of> <NUMBER, 0>
<VAR, var> <ID, diag1> <ASSIGN, :=> <ID, intArray> <LEFT_BRACKET, [> <ID, N>
<PLUS, +> <ID, N> <MINUS, -> <NUMBER, 1> <RIGHT_BRACKET, ]> <OF, of> <NUMBER,
0>
<VAR, var> <ID, diag2> <ASSIGN, :=> <ID, intArray> <LEFT_BRACKET, [> <ID, N>
```

```
<PLUS, +> <ID, N> <MINUS, -> <NUMBER, 1> <RIGHT_BRACKET, ]> <OF, of> <NUMBER,
0>

<FUNCTION, function> <ID, printboard> <LEFT_PARENT, (> <RIGHT_PARENT, )>
<EQUAL, =>
<LEFT_PARENT, (> <FOR, for> <ID, i> <ASSIGN, :=> <NUMBER, 0> <TO, to> <ID, N>
<MINUS, -> <NUMBER, 1>
<DO, do> <LEFT_PARENT, (> <FOR, for> <ID, j> <ASSIGN, :=> <NUMBER, 0> <TO, to>
<ID, N> <MINUS, -> <NUMBER, 1>
<DO, do> <ID, print> <LEFT_PARENT, (> <IF, if> <ID, col> <LEFT_BRACKET, [> <ID,
i> <RIGHT_BRACKET, ]> <EQUAL, => <ID, j> <THEN, then> <STRING, " O"> <ELSE,
else> <STRING, " ."> <RIGHT_PARENT, )> <SEMICOLON, ;>
<ID, print> <LEFT_PARENT, (> <STRING, "\n"> <RIGHT_PARENT, )> <RIGHT_PARENT, )>
<SEMICOLON, ;>
<ID, print> <LEFT_PARENT, (> <STRING, "\n"> <RIGHT_PARENT, )> <RIGHT_PARENT, )>

<FUNCTION, function> <ID, try> <LEFT_PARENT, (> <ID, c> <COLON, :> <ID, int>
<RIGHT_PARENT, )> <EQUAL, =>
<LEFT_PARENT, (>
<IF, if> <ID, c> <EQUAL, => <ID, N>
<THEN, then> <ID, printboard> <LEFT_PARENT, (> <RIGHT_PARENT, )>
<ELSE, else> <FOR, for> <ID, r> <ASSIGN, :=> <NUMBER, 0> <TO, to> <ID, N>
<MINUS, -> <NUMBER, 1>
<DO, do> <IF, if> <ID, row> <LEFT_BRACKET, [> <ID, r> <RIGHT_BRACKET, ]>
<EQUAL, => <NUMBER, 0> <AND, &> <ID, diag1> <LEFT_BRACKET, [> <ID, r> <PLUS, +>
<ID, c> <RIGHT_BRACKET, ]> <EQUAL, => <NUMBER, 0> <AND, &> <ID, diag2>
<LEFT_BRACKET, [> <ID, r> <PLUS, +> <NUMBER, 7> <MINUS, -> <ID, c>
<RIGHT_BRACKET, ]> <EQUAL, => <NUMBER, 0>
<THEN, then> <LEFT_PARENT, (> <ID, row> <LEFT_BRACKET, [> <ID, r>
<RIGHT_BRACKET, ]> <ASSIGN, :=> <NUMBER, 1> <SEMICOLON, ;> <ID, diag1>
<LEFT_BRACKET, [> <ID, r> <PLUS, +> <ID, c> <RIGHT_BRACKET, ]> <ASSIGN, :=>
<NUMBER, 1> <SEMICOLON, ;> <ID, diag2> <LEFT_BRACKET, [> <ID, r> <PLUS, +>
<NUMBER, 7> <MINUS, -> <ID, c> <RIGHT_BRACKET, ]> <ASSIGN, :=> <NUMBER, 1>
<SEMICOLON, ;>
<ID, col> <LEFT_BRACKET, [> <ID, c> <RIGHT_BRACKET, ]> <ASSIGN, :=> <ID, r>
<SEMICOLON, ;>
<ID, try> <LEFT_PARENT, (> <ID, c> <PLUS, +> <NUMBER, 1> <RIGHT_PARENT, )>
<SEMICOLON, ;>
<ID, row> <LEFT_BRACKET, [> <ID, r> <RIGHT_BRACKET, ]> <ASSIGN, :=> <NUMBER, 0>
<SEMICOLON, ;> <ID, diag1> <LEFT_BRACKET, [> <ID, r> <PLUS, +> <ID, c>
<RIGHT_BRACKET, ]> <ASSIGN, :=> <NUMBER, 0> <SEMICOLON, ;> <ID, diag2>
<LEFT_BRACKET, [> <ID, r> <PLUS, +> <NUMBER, 7> <MINUS, -> <ID, c>
<RIGHT_BRACKET, ]> <ASSIGN, :=> <NUMBER, 0> <RIGHT_PARENT, )>

<RIGHT_PARENT, )>
<IN, in> <ID, try> <LEFT_PARENT, (> <NUMBER, 0> <RIGHT_PARENT, )>

<END, end>
```

Como já especificado, as quebras de linha foram mantidas apenas para facilitar a leitura. Neste exemplo é possível perceber que o analisador consegue reconhecer identificadores, símbolos, palavras chaves, strings, e ignora espaços e comentários.

5. Próximos passos

Com o analisador léxico, o próximo passo é implementar o analisador sintático, que poderá utilizar o fluxo de tokens que é possível ser obtido a partir da implementação obtida neste trabalho. Não foi implementado nenhum tipo de tratamento de erro, então qualquer padrão que não esteja definido nas regras é apenas ignorado, mas uma mensagem de erro é transmitida. A posição dos tokens em relação ao código fonte também não está sendo utilizada, pois nesta etapa não é necessária, e na especificação do projeto se pede apenas os tokens, mas para montar a tabela de símbolos que será usada pelo analisador sintático ela será útil.

6. Conclusão

Com este trabalho foi possível implementar a primeira parte do compilador da linguagem TIGER.

Com a primeira etapa pronta e reconhecendo todos os tokens, será possível continuar o projeto e implementar as outras fases do compilador.

7. Referências

- https://www.lrde.epita.fr/~tiger/tiger.html#Tiger-Language-Reference-Manual
- http://dinosaur.compilertools.net/flex/index.html