

UNIVERSIDADE FEDERAL DO PAMPA
CURSO DE ENGENHARIA DA COMPUTAÇÃO
DISCIPLINA DE LINGUAGENS FORMAIS

Trabalho Prático T1

Írio Rafael de Menezes Borges

Bagé, 16 de Julho de 2018

RESUMO

Este trabalho descreve a implementação de um analisador léxico e sintático determinístico de uma linguagem L, realizado para disciplina de Linguagens formais durante o primeiro semestre de 2018.

Palavras-chave: linguagens formais, gramáticas livre de contexto, compiladores, analisador léxico, analisador sintático, flex, bison, Linguagem de programação C.

SUMÁRIO

1 INTRODUÇÃO	4
2 DESENVOLVIMENTO	5
3 CONSIDERAÇÕES FINAIS	16
REFERÊNCIAS	17

1. INTRODUÇÃO

Seja a linguagem de programação L, gerada pela seguinte gramática livre de contexto:

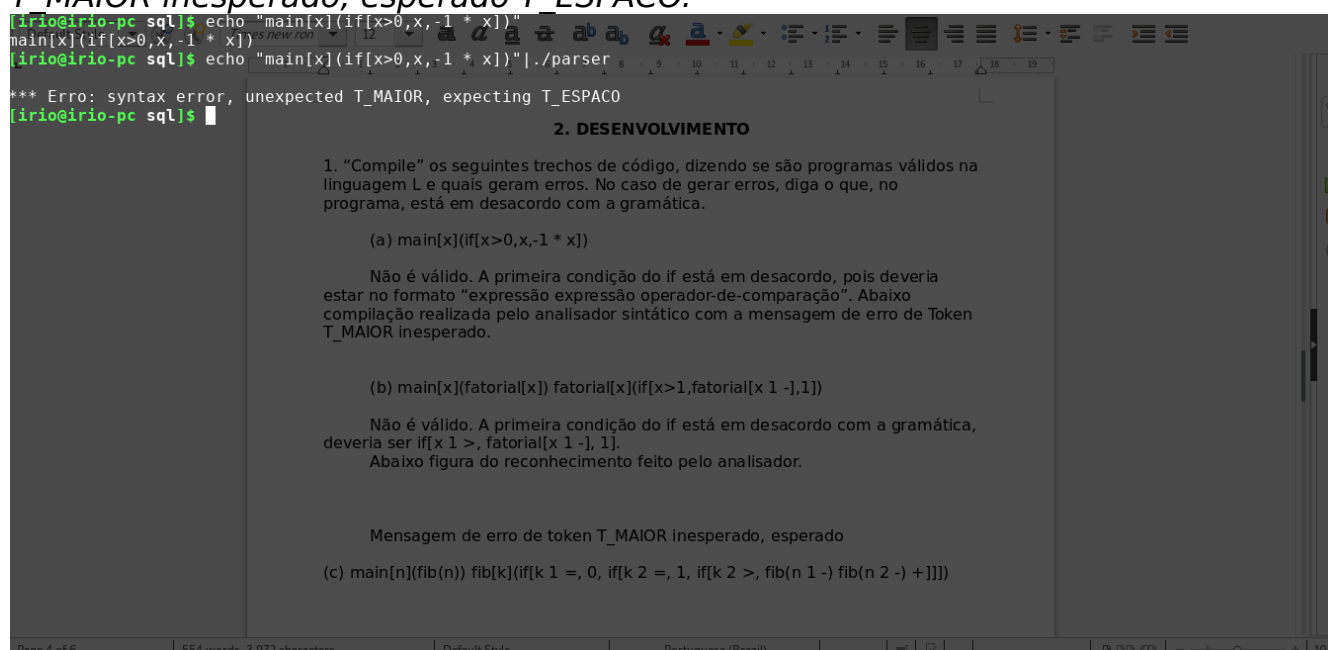
```
<programa> ::= <principal> <lista-de-funções>
<principal> ::= main[<params>](<corpo>)
<lista-de-funções> ::= <função>
<lista-de-funções> ::= <função> <lista-de-funções>
<função> ::= <id>[ ](<corpo>)
<função> ::= <id>[<params>](<corpo>)
<params> ::= <id>,<params>
<params> ::= <id>
<corpo> ::= <id>[ ]
<corpo> ::= <id>[<args>]
<corpo> ::= if[<cond>,<corpo>,<corpo>]
<args> ::= <arg>
<args> ::= <arg>,<args>
<arg> ::= <exp>
<arg> ::= <corpo>
<exp> ::= <num>
<exp> ::= <id>
<exp> ::= <exp> <exp> +
<exp> ::= <exp> <exp> -
<exp> ::= <corpo>
<cond> ::= <exp> <exp> >
<cond> ::= <exp> <exp> <
<cond> ::= <exp> <exp> =
<cond> ::= <exp> <exp> <>
<id> ::= <letra> <seqsimb>
<num> ::= <digito>
<num> ::= <digito> <num>
<letra> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digito> ::= 0|1|2|3|4|5|6|7|8|9
<seqsimb> ::= ε
<seqsimb> ::= <letra> <seqsimb>
<seqsimb> ::= <digito> <seqsimb>
```

2. DESENVOLVIMENTO

1. “Compile” os seguintes trechos de código, dizendo se são programas válidos na linguagem L e quais geram erros. No caso de gerar erros, diga o que, no programa, está em desacordo com a gramática.

(a) `main[x](if[x>0,x,-1 * x])`

Não é válido. A primeira condição do if está em desacordo, pois deveria estar no formato “expressão expressão operador-de-comparação”. Abaixo compilação realizada pelo analisador sintático com a mensagem de erro de Token T_MAIOR inesperado, esperado T_ESPACO.



```
irio@irio-pc sql$ echo "main[x](if[x>0,x,-1 * x])"
main[x](if[x>0,x,-1 * x])
irio@irio-pc sql$ echo "main[x](if[x>0,x,-1 * x])" | ./parser
*** Erro: syntax error, unexpected T_MAIOR, expecting T_ESPACO
irio@irio-pc sql$
```

2. DESENVOLVIMENTO

1. “Compile” os seguintes trechos de código, dizendo se são programas válidos na linguagem L e quais geram erros. No caso de gerar erros, diga o que, no programa, está em desacordo com a gramática.

(a) `main[x](if[x>0,x,-1 * x])`

Não é válido. A primeira condição do if está em desacordo, pois deveria estar no formato “expressão expressão operador-de-comparação”. Abaixo compilação realizada pelo analisador sintático com a mensagem de erro de Token T_MAIOR inesperado.

(b) `main[x](fatorial[x]) fatorial[x](if[x>1,fatorial[x 1 -],1])`

Não é válido. A primeira condição do if está em desacordo com a gramática, deveria ser `if[x 1 >, fatorial[x 1 -], 1]`. Abaixo figura do reconhecimento feito pelo analisador.

Mensagem de erro de token T_MAIOR inesperado, esperado

(c) `main[n](fib(n)) fib[k](if[k 1 =, 0, if[k 2 =, 1, if[k 2 >, fib(n 1 -) fib(n 2 -) +]])`

(b) `main[x](fatorial[x]) fatorial[x](if[x>1,fatorial[x 1 -],1])`

Não é válido. A primeira condição do if está em desacordo com a gramática, deveria ser `if[x 1 >, fatorial[x 1 -], 1]`.

Abaixo figura do reconhecimento feito pelo analisador.

```
irio@irio-pc sql$ echo "main[x](fatorial[x])fatorial[x](if[x>1,fatorial[x-1],1])" | ./parser
*** Erro: syntax error, unexpected T_MAIOR, expecting T_ESPACO
irio@irio-pc sql$
```

(b) main[x](fatorial[x]) fatorial[x](if[x>1,fatorial[x-1],1])

Não é válido. A primeira condição do if está em desacordo com a gramática, deveria ser if[x 1 >, fatorial[x 1 -], 1].

Abaixo figura do reconhecimento feito pelo analisador.

Mensagem de erro de token T_MAIOR inesperado, esperado

(c) main[n](fib(n)) fib[k](if[k 1 =, 0, if[k 2 =, 1, if[k 2 >, fib(n 1 -) fib(n 2 -) +]])

(d) main[a,b,c](fatorial[a b + c *]) fatorial[x](if[x 1 >,fatorial[x 1 -],1])

Mensagem de erro de token T_MAIOR inesperado, esperado T_ESPACO.

(c) main[n](fib(n)) fib[k](if[k 1 =, 0, if[k 2 =, 1, if[k 2 >, fib(n 1 -) fib(n 2 -) +]])

O primeiro erro desta palavra é nos parênteses do "fib(n)", pois esperava T ABRE COLCHETE. Como mostra a figura abaixo:

```
irio@irio-pc sql$ echo "main[n](fib(n))fib[k](if[k 1 =,0,if[k 2 =,1,if[k 2 >,fib(n 1 -)fib(n 2 -) +]])" | ./parser
*** Erro: syntax error, unexpected T_ABRE_PARENTESE, expecting T_ABRE_COLCHETE
irio@irio-pc sql$
```

Mensagem de erro de token T_MAIOR inesperado, esperado

Mensagem de erro de token T_MAIOR inesperado, esperado T_ESPACO.

(c) main[n](fib(n)) fib[k](if[k 1 =, 0, if[k 2 =, 1, if[k 2 >, fib(n 1 -) fib(n 2 -) +]])

(d) main[a,b,c](fatorial[a b + c *]) fatorial[x](if[x 1 >,fatorial[x 1 -],1])

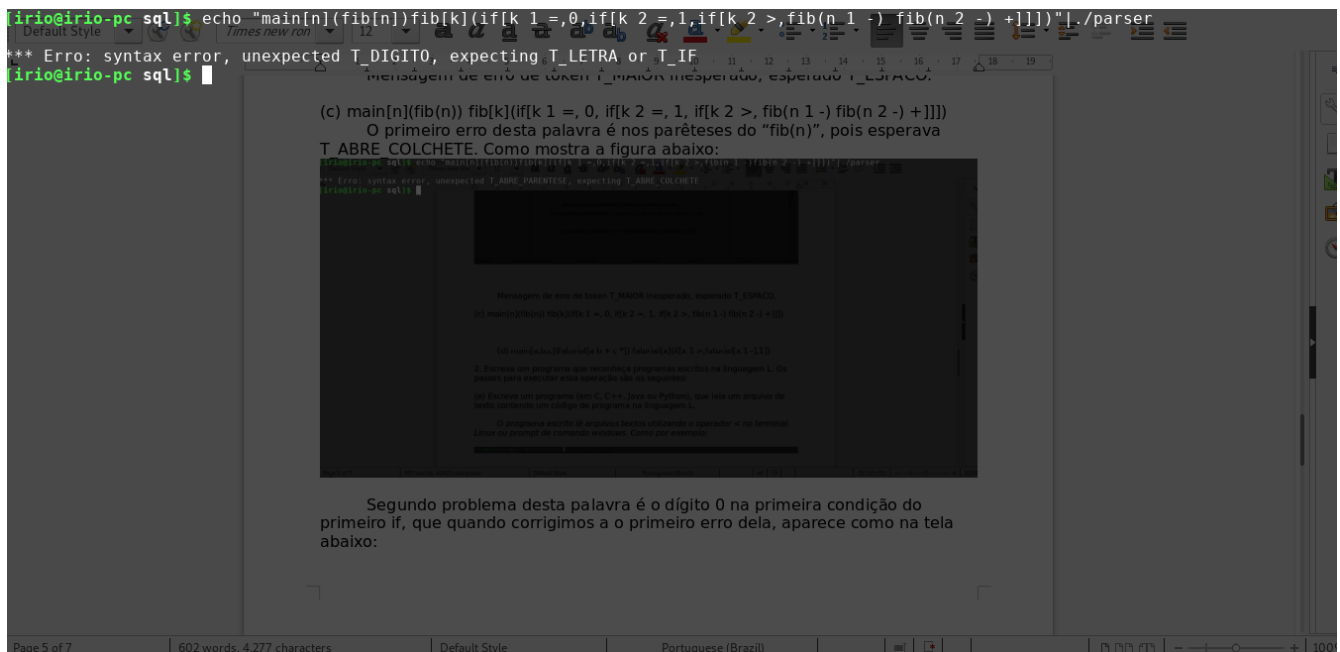
2. Escreva um programa que reconheça programas escritos na linguagem L. Os passos para executar essa operação são os seguintes:

(a) Escreva um programa (em C, C++, Java ou Python), que leia um arquivo de texto contendo um código de programa na linguagem L.

O programa escrito lê arquivos textos utilizando o operador < no terminal Linux ou prompt de comando windows. Como por exemplo:

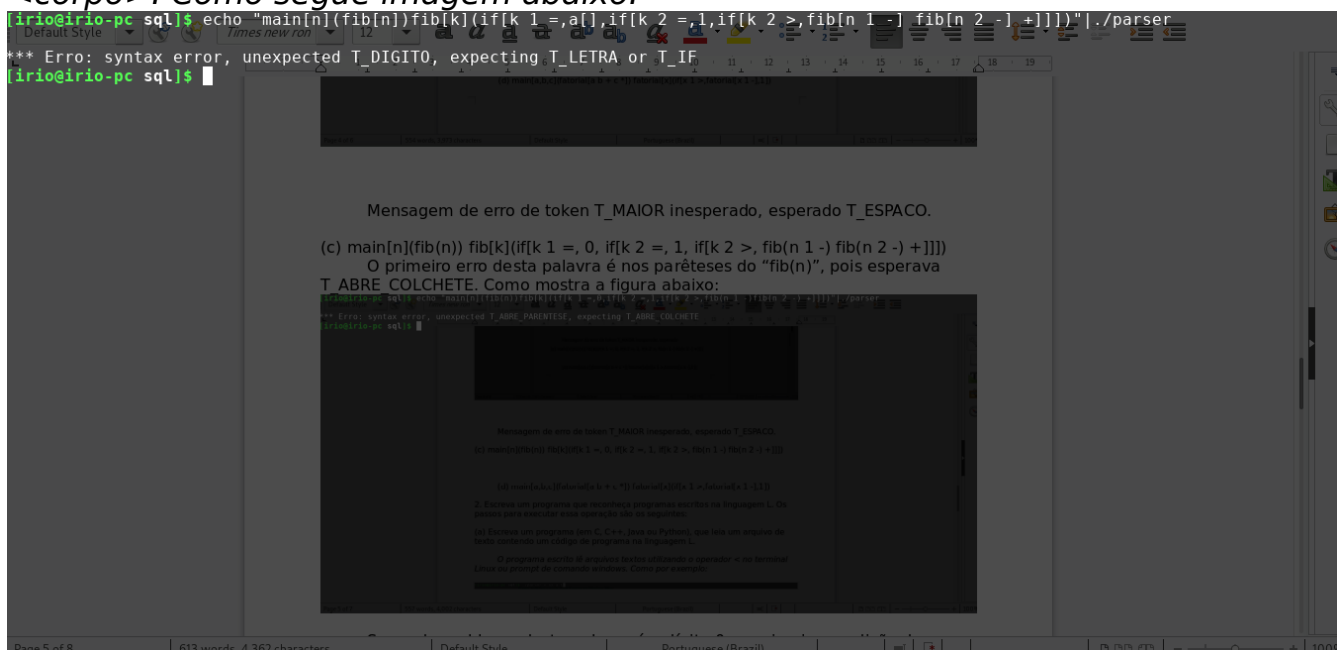
Segundo problema desta palavra é o dígito 0 no primeiro <corpo> do primeiro if, que quando corrigimos o erro anterior, aparece como na tela abaixo:

6

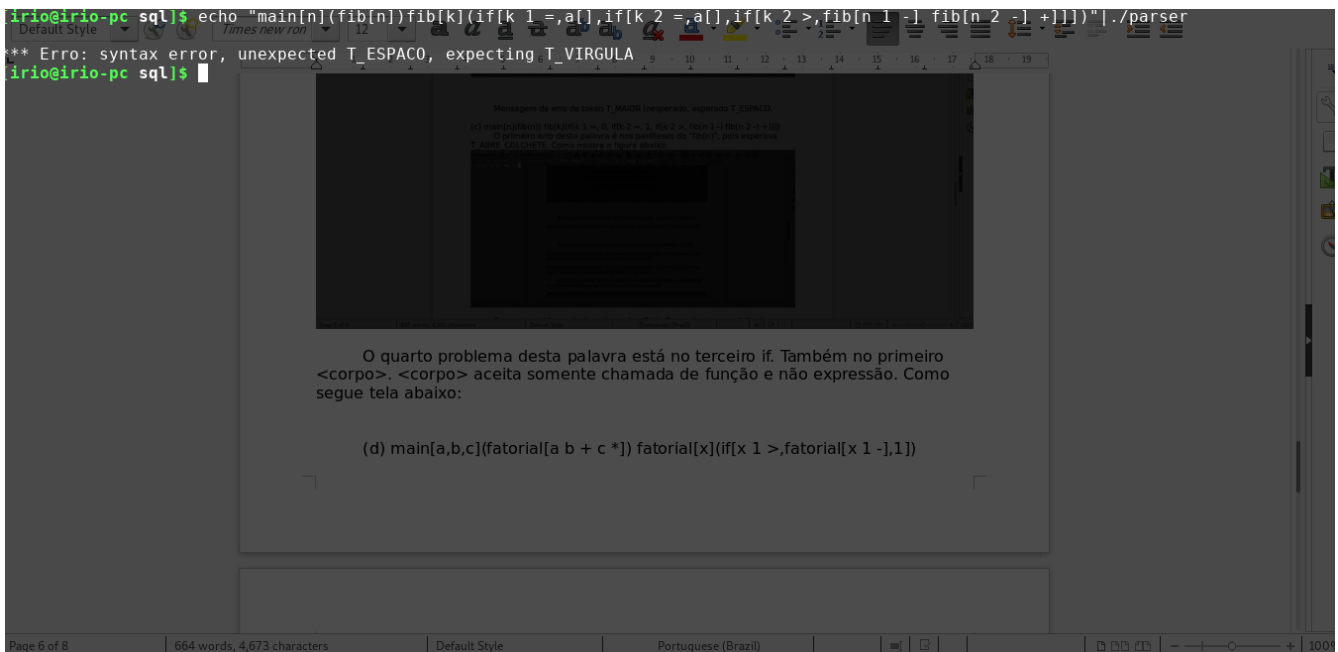


T_DIGITO inesperado, esperado T_LETRA ou T_IF.

O terceiro problema desta palavra, é igual ao anterior, porém no segundo if. Ele tem um dígito na primeira condição, onde deveria ter uma expressão do tipo <corpo>. Como segue imagem abaixo:



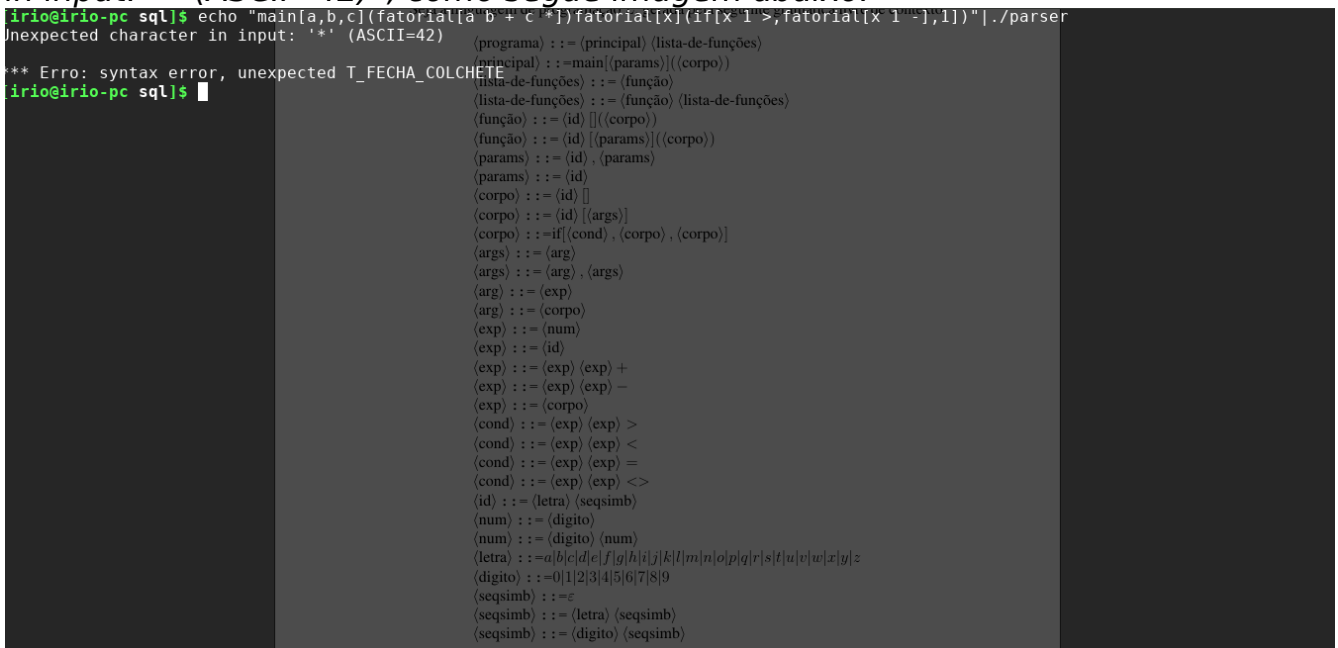
O quarto problema desta palavra está no terceiro if. Também no primeiro <corpo>. <corpo> aceita somente chamada de função e não expressão. Como segue tela abaixo:



T_ESPACO inesperado, esperado T_VIRGULA.

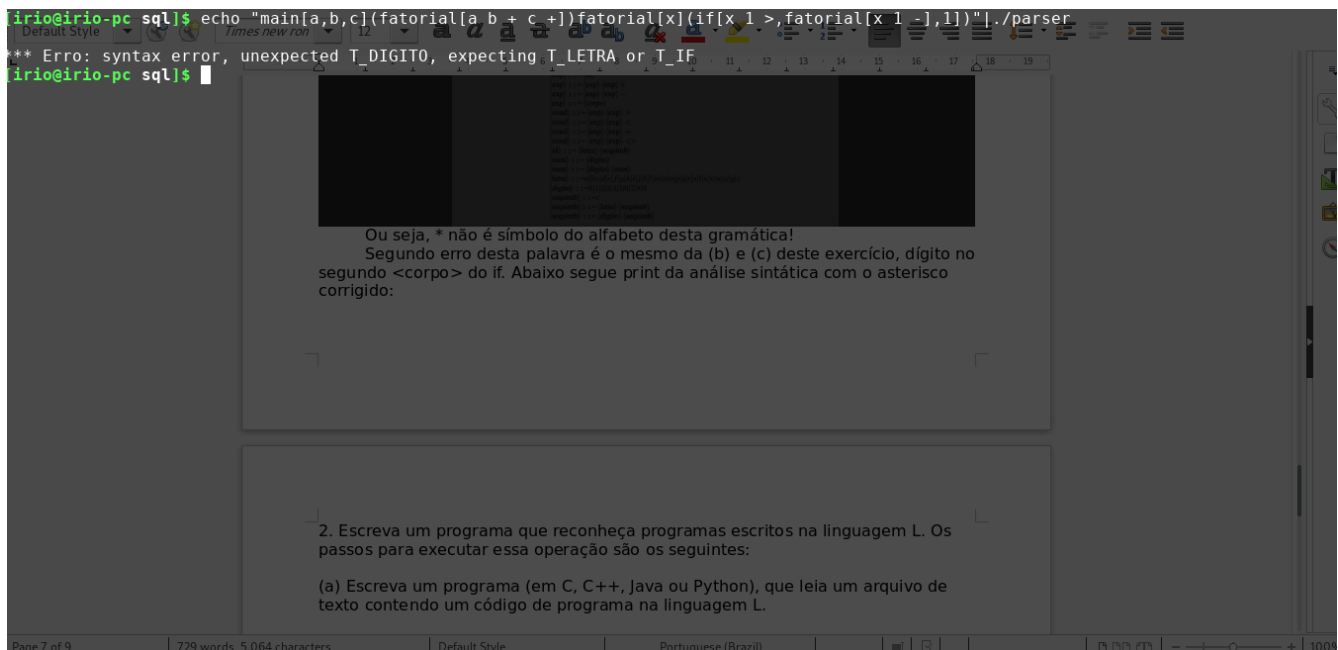
(d) `main[a,b,c](fatorial[a b + c *]) fatorial[x](if[x 1 >,fatorial[x 1 -],1])`

*Primeiro erro desta palavra é o caracter *, que dá erro de “Unexpected char in input: ‘*’ (ASCII=42)”, como segue imagem abaixo:*



*Ou seja, * não é símbolo do alfabeto desta gramática!*

Segundo erro desta palavra é o mesmo da (b) e (c) deste exercício, dígito no segundo <corpo> do if. Abaixo segue print da análise sintática com o asterisco corrigido:



No caso, troquei o “” pelo sinal de “+” para testar o restante da palavra.*

2. Escreva um programa que reconheça programas escritos na linguagem L. Os passos para executar essa operação são os seguintes:

(a) Escreva um programa (em C, C++, Java ou Python), que leia um arquivo de texto contendo um código de programa na linguagem L.

O programa escrito lê arquivos textos utilizando o operador < no terminal Linux ou prompt de comando windows. Como por exemplo:

```
[irio@irio-pc sql]$ ./parser < ex-a.L
```

Ou então usando o comando “echo”, como por exemplo:

```
echo "main[a,b,c](fatorial[a b + c +])fatorial[x](if[x 1 >,fatorial[x 1  
-],1])" | ./parser
```

O programa que lê o arquivo texto é o próprio parser gerado, e o código-fonte dele segue nos itens (b) e (c) deste exercício.

(b) Escreva um analisador léxico (autômato finito determinístico) que separe os itens do programa em identificadores, números, operadores e demais símbolos delimitadores da linguagem, conforme a gramática dada.

Para análise léxica foi utilizado o software flex, versão 2.6.4 para Manjaro Linux. Segue código do arquivo scanner.l que é onde estão os tokens utilizados na gramática.

```

%{
#include "common.h"
#include "parser.h"
}%

/* Definições */
LETRA [a-z]

DIGITO [0-9]
ANY_CHAR .
FIM_LINHA [\n]

%option case-insensitive

%% /* Regras */

"MAIN" { return T_MAIN; }
"[" { return T_ABRE_COLCHETE; }
"]" { return T_FECHA_COLCHETE; }
"(" { return T_ABRE_PARENTESE; }
")" { return T_FECHA_PARENTESE; }
"" { return T_VAZIO; }
{LETRA} { return T_LETRA; }
{DIGITO} { return T_DIGITO; }
", " { return T_VIRGULA; }
"+" { return T_MAIS; }
"-" { return T_MENOS; }
" " { return T_ESPACO; }
">" { return T_MAIOR; }
"<" { return T_MENOR; }
{FIM_LINHA} { return T_FIM; }
"=" { return T_IGUAL; }
"if" { return T_IF; }

{ANY_CHAR} {
    printf("Unexpected character in input: '%c' (ASCII=%d)\n", yytext[0],
yytext[0]);
}

```

No caso foram utilizados 17 tokens: `T_MAIN`, `T_ABRE_COLCHETE`, `T_FECHA_COLCHETE`, `T_ABRE_PARENTESE`, `T_FECHA_PARENTESE`, `T_VAZIO`, `T_LETRA`, `T_DIGITO`, `T_VIRGULA`, `T_MAIS`, `T_MENOS`, `T_ESPACO`, `T_MAIOR`, `T_MENOR`, `T_FIM`, `T_IGUAL` e `T_IF`.

Os arquivos `common.h` e `parser.h` seguem como apêndice.

Com o parâmetro `-o` do `flex`, é gerado um arquivo `scanner.c` a partir deste arquivo `scanner.l`. Depois este arquivo `c` é compilado com o `gcc` utilizando o parâmetro `-c`.

(c) A partir da gramática dada e dos elementos identificados no passo anterior, escreva um analisador sintático determinístico que reconheça programas escritos na linguagem L.

*O programa utilizado para gerar a gramática foi o bison versão 3.05.
Segue conteúdo do arquivo parser.y abaixo:*

```
%{
#include "common.h"
#include <stdio.h>

int deuErro = 0;
int teste;

%}

%token T_LETRA
%token T_DIGITO
%token T_MAIN
%token T_ABRE_COLCHETE
%token T_FECHA_COLCHETE
%token T_ABRE_PARENTESE
%token T_FECHA_PARENTESE
%token T_IF
%token T_IGUAL
%token T_MAIOR
%token T_MENOR
%token T_MAIS
%token T_MENOS
%token T_VIRGULA
%token T_VAZIO
%token T_ESPACO
%token T_PONTO_E_VIRGULA
%token T_FIM

%error-verbose

%%

programa:
    principal lista-de-funcoes T_FIM
;

principal:
    T_MAIN T_ABRE_COLCHETE params T_FECHA_COLCHETE
    T_ABRE_PARENTESE corpo T_FECHA_PARENTESE
;
11
```

lista-de-funcoes:

funcao

;

lista-de-funcoes:

funcao lista-de-funcoes

;

funcao:

id T_ABRE_COLCHETE T_FECHA_COLCHETE T_ABRE_PARENTESE corpo
T_FECHA_PARENTESE

;

funcao:

id T_ABRE_COLCHETE params T_FECHA_COLCHETE T_ABRE_PARENTESE
corpo T_FECHA_PARENTESE

;

params:

id T_VIRGULA params

;

params:

id

;

corpo:

id T_ABRE_COLCHETE T_FECHA_COLCHETE

;

corpo:

id T_ABRE_COLCHETE args T_FECHA_COLCHETE

;

corpo:

T_IF T_ABRE_COLCHETE cond T_VIRGULA corpo T_VIRGULA corpo
T_FECHA_COLCHETE

;

args:

arg

;

args:

arg T_VIRGULA args

;

arg:

12

```

    exp
;
arg:
    corpo
;
exp:
    num
;
exp:
    id
;
exp:
    exp T_ESPACO exp T_ESPACO T_MAIS
;
exp:
    exp T_ESPACO exp T_ESPACO T_MENOS
;
exp:
    corpo
;
cond:
    exp T_ESPACO exp T_ESPACO cond1
;
cond1:
    T_MAIOR
    |T_MENOR
    |T_IGUAL
    |T_MENOR T_MAIOR
;
id:
    T_LETRA seqsimb
;
num:
    T_DIGITO num2
;
num2:
    13

```

```

T_DIGITO num
    |
;

seqsimb:
    T_LETRA seqsimb
    |
;

seqsimb:
    T_DIGITO seqsimb
;

%%

void yyerror(const char* errmsg)
{
    deuErro = 1;
    //Print(root);
    printf("\n*** Erro: %s\n", errmsg);
}

int yywrap(void) { return 1; }

int main(int argc, char** argv)
{
    yyparse();

    if(deuErro == 0){
        printf("\nPalavra ou trecho de código reconhecido pela gramática!!\n");
    }

    return 0;
}

```

Na regra <cond> foi feita uma refatoração a esquerda para eliminar uma ambiguidade que o bison não corrigiu. Os tokens da linguagem também estão presentes neste arquivo. Para a geração do parser.c é utilizado:

bison -d -o parser.c parser.y

Após isso precisamos gerar o parser.o com o gcc, como abaixo:

gcc -c parser.c

(d) No caso do programa apresentado como entrada possuir erros de sintaxe, o analisador sintático deverá ser capaz de reconstruir a derivação usada, mostrando qual foi o erro que ocorreu.

No arquivo `parser.y` existe a função:

```
void yyerror(const char* errmsg)
{
    deuErro = 1;
    //Print(root);
    printf("\n*** Erro: %s\n", errmsg);
}
```

Ela mostra o token inesperado e o que era esperado pela gramática, porém não faz a reconstrução da derivação utilizada.

Durante toda construção do `scanner.l` e do `parser.y` foi utilizado um `makefile`, que se mostrou muito útil, pois economiza tempo compilando arquivo por arquivo, apenas digitamos `make` no terminal e está tudo gerado. Segue o código dele abaixo:

```
parser: parser.o scanner.o
gcc -o parser parser.o scanner.o
parser.o: parser.c
gcc -c parser.c
scanner.o: scanner.c
gcc -c scanner.c
scanner.c:
flex -o scanner.c scanner.l
parser.c: parser.y
bison -d -o parser.c parser.y
```

3. CONSIDERAÇÕES FINAIS

Foi realizado neste trabalho duas etapas para a implementação de um compilador. É visto que é muito mais eficiente utilizar ferramentas já desenvolvidas para análise léxica e sintática do que implementar tudo do começo em alguma linguagem de programação. Neste caso também foi utilizada a técnica de refatoração para eliminar ambiguidades da gramática, ou seja, não é simplesmente “jogar” a gramática no bison que ela está pronta.

Referências bibliográficas:

https://pt.wikipedia.org/wiki/Gram%C3%A1tica_livre_de_contexto

<http://bughunter.tecland.com.br/criando-um-simples-parser-usando-bison-e-flex>