

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ  
CÂMPUS PONTA GROSSA  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS MAGALHÃES DOMINGUES

ATIVIDADE PRÁTICA: IMPLEMENTAÇÃO DAS ETAPAS DE ANÁLISE LÉXICA E  
SINTÁTICA PARA A LINGUAGEM DE PROGRAMAÇÃO BÁSICA

Atividade Prática Supervisionada,  
apresentado à disciplina Compiladores  
do curso de Ciência da Computação da  
Universidade Tecnológica Federal do  
Paraná – UTFPR.

PONTA GROSSA  
2021

## SUMÁRIO

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>INTRODUÇÃO</b> .....                         | <b>3</b>  |
| <b>2</b> | <b>PROPOSTA</b> .....                           | <b>4</b>  |
| <b>3</b> | <b>METODOLOGIA</b> .....                        | <b>5</b>  |
| <b>4</b> | <b>DESENVOLVIMENTO</b> .....                    | <b>6</b>  |
| 4.1      | Apresentação e representação da gramática ..... | 6         |
| 4.2      | Implementação do analisador léxico .....        | 8         |
| 4.3      | Implementação do analisador sintático .....     | 10        |
| 4.4      | Compilação, entrada e saída.....                | 18        |
| <b>5</b> | <b>CONCLUSÃO</b> .....                          | <b>19</b> |
| <b>6</b> | <b>REFERÊNCIAS</b> .....                        | <b>20</b> |

## **1 INTRODUÇÃO**

O trabalho proposto é relacionado à atividade prática da disciplina de Compiladores, o objetivo do trabalho é implementar as etapas de análise léxica e análise sintática para a linguagem de programação básica.

## **2 PROPOSTA**

A proposta do trabalho é desenvolver uma BNF e diagramas de sintaxe para depois implementar o analisador léxico, respeitando os diagramas de transição, e o analisador sintático através da gramática BNF. Gerenciar a entrada e saída de arquivos também faz parte do trabalho, bem como desenvolver a tabela de símbolos.

### **3 METODOLOGIA**

Foram utilizadas as ferramentas Flex e Bison para a criação do analisador léxico e do analisador sintático respectivamente. Além da plataforma Debuggex[1] para representar os diagramas de transições, e da plataforma Railroad Diagram Generator[2] para representar os diagramas de sintaxe.

## 4 DESENVOLVIMENTO

Para o desenvolvimento do trabalho, é apresentado a gramática BNF, os diagramas de sintaxe, a designação de tokens, os diagramas de transição, o desenvolvimento do código .l em Flex para análise léxica, a tabela de símbolos, o desenvolvimento do código .y em Bison para análise sintática, a utilização de um arquivo .txt para leitura, e a criação de uma saída .txt.

### 4.1 APRESENTAÇÃO E REPRESENTAÇÃO DA GRAMÁTICA

Através da Forma de Backus-Naur, uma notação natural para descrever sintaxe[3], foi possível representar a gramática para o desenvolvimento do analisador sintático, como mostra a Figura 1, e Figura 2. A Figura 3 demonstra os diagramas de sintaxe, e a Figura 4 mostra os tokens.

```
BNF
<stmt>::= if <exp> then <list>
| if <exp> then <list> else <list>
| if <exp> AND <exp> then <list>
| if <exp> AND <exp> then <list> else <list>
| if <exp> OR <exp> then <list>
| if <exp> OR <exp> then <list> else <list>
| while <exp> do <list>
| for <exp> <exp> <list>
| <exp>
<list>::=
| <stmt> ; <list>
<exp>::= <exp> CMP <exp>
| <exp> + <exp>
| <exp> - <exp>
| <exp> * <exp>
| <exp> / <exp>
| ( <exp> )
| NUMBER
| NAME
| NAME = <exp>
| FUNC ( <explist> )
| NAME ( <explist> )
<explist>::= <exp>
| <exp> , <explist>
```

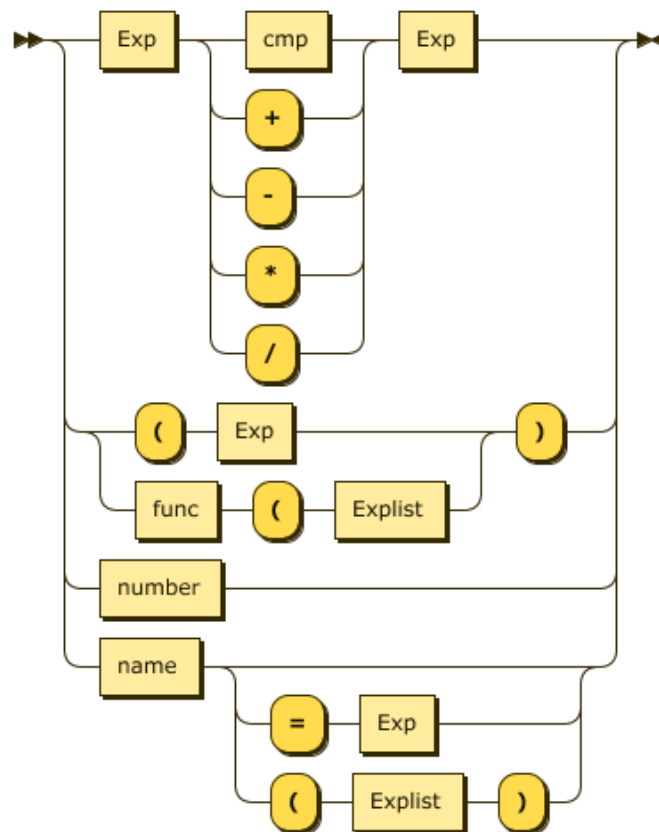
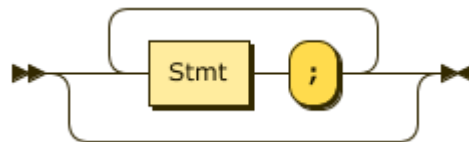
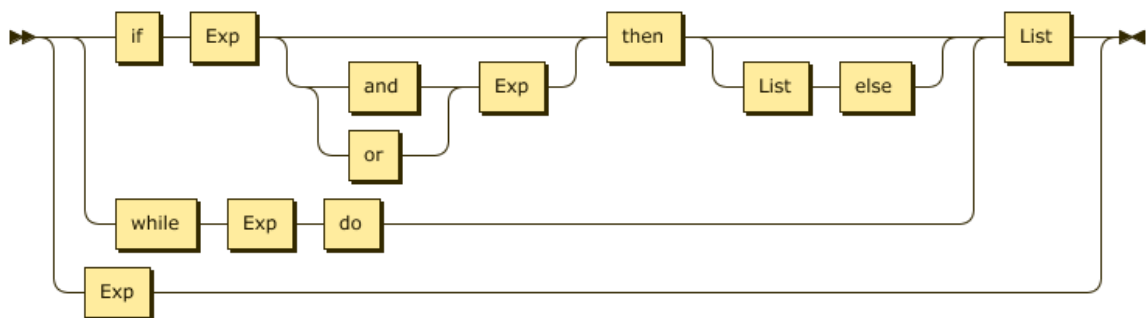
Figura 1 – Gramática BNF

```

<symlist> ::= NAME
            | NAME, <symlist>

<calclist> ::=
            | <calclist> <stmt> EOL
            | <calclist> LET NAME
            | <calclist> error EOL
  
```

Figura 2 – Gramática BNF



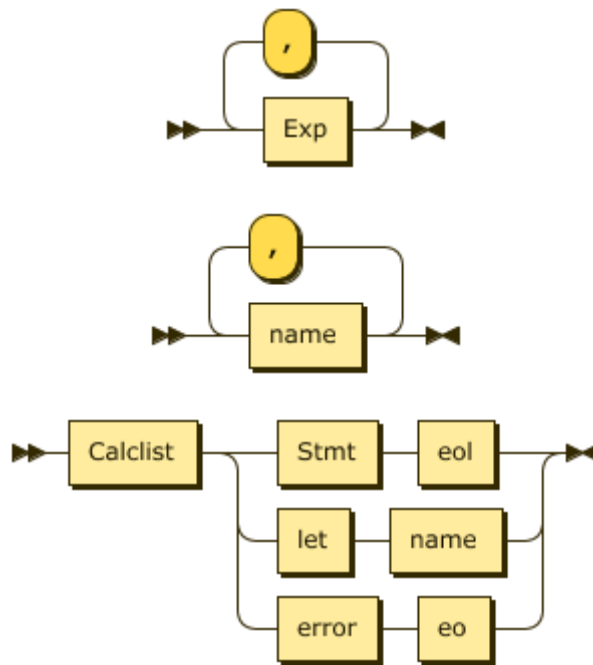


Figura 3 – Diagramas de Sintaxe

```

450 %token <d> NUMBER
451 %token <s> NAME
452 %token <fn> FUNC
453 %token EOL
454
455 %token IF THEN ELSE WHILE DO LET FOR AND OR

```

Figura 4 – Declaração de Tokens

## 4.2 IMPLEMENTAÇÃO DO ANALISADOR LÉXICO

Respondendo às chamadas do Parser, o analisador léxico em Flex identifica os lexemas lidos no arquivo Input.txt para retornar novamente ao analisador sintático. Na Figura 5, é mostrado o código .l em Flex. Na figura 6, os diagramas de transição são apresentados.



```

1 /*
2  *      Lexer para uma calculadora avancada
3  */
4
5 /* reconhecimento de tokens para a calculadora */
6
7 %option noyywrap nodefault yylineno
8 %{
9 # include "bison-calc.h"
10 # include "bison-calc.tab.h"
11 # include <math.h>
12 %}
13
14 /* expoente float */
15 EXP ([Ee][+-]?[0-9]+)
16
17 %%
18 "+" | /* operadores de caracter unico */
19 "-" |
20 "*" |
21 "/" |
22 "=" |
23 ";" |
24 "," |
25 "(" |
26 ")" | { return yytext[0]; }
27 ">" | { yylval.fn = 1; return CMP; } /* operadores de comparacao, todos sao token CMP*/
28 "<" | { yylval.fn = 2; return CMP; }
29 "<=" | { yylval.fn = 3; return CMP; }
30 "==" | { yylval.fn = 4; return CMP; }
31 ">=" | { yylval.fn = 5; return CMP; }
32 "<=" | { yylval.fn = 6; return CMP; }
33 "if" | { return IF; } /* palavras-chave */
34 "then" | { return THEN; }
35 "else" | { return ELSE; }
36 "while" | { return WHILE; }
37 "do" | { return DO; }
38 "let" | { return LET; }
39 "for" | { return FOR; }
40 "AND" | { return AND; }
41 "OR" | { return OR; }
42 "sqrt" | { yylval.fn = B_sqrt; return FUNC; } /* funcoes pre-definidas */
43 "exp" | { yylval.fn = B_exp; return FUNC; }
44 "log" | { yylval.fn = B_log; return FUNC; }
45 "print" | { yylval.fn = B_print; return FUNC; }
46
47 [a-zA-Z][a-zA-Z0-9]* { yylval.s = lookup(yytext); return NAME; } /* nomes */
48
49 [0-9]+ "." [0-9]* { yylval.d = atof(yytext); return NUMBER; }
50 ". "[0-9]+ {EXP}?
51
52 "/*" .*
53 [\t] /* ignora espaco em branco */
54
55 \\n
56 " " { printf("<> "); } /* ignora continuacao de linha*/
57 \n {return EOL; }
58
59 . { yyerror("Caracter desconhecido %c\n",*yytext); }
60 %%

```

Figura 5 – Construção do analisador léxico

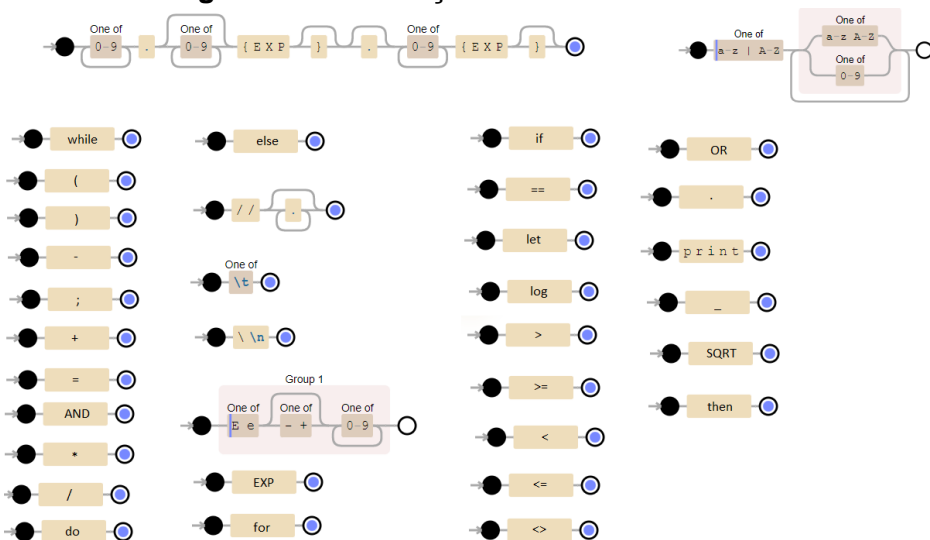


Figura 6 – Diagramas de transição

### 4.3 IMPLEMENTAÇÃO DO ANALISADOR SINTÁTICO

A tabela de símbolos é apresentada na figura 7.

O código header é apresentado na figura 8.

O código yacc em Bison é apresentado na figura 9.

```
/* tab. de simbolos */
struct symbol {          /* um nome de variavel */
char *name;
double value;
struct ast *func;      /* stmt para funcao */
struct symlist *syms;  /* lista de argumentos */
};

/* tab. de simbolos de tamanho fixo*/
#define NHASH 9997
struct symbol symtab[NHASH];

struct symbol *lookup(char*);

/* lista de simbolos, para uma lista de argumentos */
struct symlist{
    struct symbol *sym;
    struct symlist *next;
};
```

Figura 7 – Tabela de Símbolos



```
1 /*
2 *      Declarações para um calculadora avancada
3 */
4
5 /* interface com o lexer */
6 extern int yylineno;
7 void yyerror(char *s, ...);
8
9 /* tab. de simbolos */
10 struct symbol {          /* um nome de variavel */
11 char *name;
12 double value;
13 struct ast *func;      /* stmt para funcao */
14 struct symlist *syms;  /* lista de argumentos */
15 };
16
17 /* tab. de simbolos de tamanho fixo*/
18 #define NHASH 9997
19 struct symbol symtab[NHASH];
20
21 struct symbol *lookup(char*);
22
23 /* lista de simbolos, para uma lista de argumentos */
24 struct symlist{
25     struct symbol *sym;
26     struct symlist *next;
27 };
28
29 struct symlist *newsymlist(struct symbol *sym, struct symlist *next);
30 void symlistfree(struct symlist *sl);
31
32 /* tipos de nos
33 *      + - * /
34 *      0-7 operadores de comparacao, 04 igual, 02 menor que, 01 maior que
35 *      L expressao ou lista de comandos
36 *      I comando IF
37 *      W comando WHILE
```

```

Abrir  bison-calc.h
74 struct flow{
75     int nodetype; /* tipo I ou W ou P */
76     struct ast *cond; /* condicao */
77     struct ast *cond2;
78     struct ast *tl; /* ramo "then" ou lista "do" */
79     struct ast *el; /* ramo opcional "else" */
80     struct ast *para;
81 };
82
83 struct numval {
84     int nodetype; /* tipo K */
85     double number;
86 };
87
88 struct symref {
89     int nodetype; /* tipo N */
90     struct symbol *s;
91 };
92
93 struct symasgn {
94     int nodetype; /* tipo = */
95     struct symbol *s;
96     struct ast *v; /* valor a ser atribuido */
97 };
98
99 /* construcao de uma AST */
100
101 struct ast *newast(int nodetype, struct ast *l, struct ast *r);
102 struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
103 struct ast *newfunc(int functype, struct ast *l);
104 struct ast *newcall(struct symbol *s, struct ast *l);
105 struct ast *newasgn(struct symbol *s, struct ast *v);
106 struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *tr);
107 struct ast *segundoflow(int nodetype, struct ast *cond, struct ast *cond2, struct ast *tl, struct ast *tr);
108
109 /* definicao de uma funcao */
110 void dodef(struct symbol *name, struct symlist *syms, struct ast *stmts);

```

Figura 8 – Código header

```

Abrir  bison-calc.y
1 /*
2  *      Parser para uma calculadora avancada
3  */
4
5 %{
6 # include "bison-calc.h"
7 # include <stdio.h>
8 # include <stdlib.h>
9 # include <stdarg.h>
10 # include <string.h>
11 # include <math.h>
12
13 FILE *yyout;
14 /*
15  * Funcoes Auxiliares para uma calculadora avancada
16  */
17
18
19 /* funcoes em C para TS */
20 /*funcao hashing */
21 static unsigned symhash(char *sym)
22 {
23     unsigned int hash=0;
24     unsigned c;
25     while(c=*sym++)
26         hash=hash*9^c;
27 }
28
29 struct symbol *lookup(char* sym)
30 {
31     struct symbol *sp=&syntab[symhash(sym)%NHASH];
32     int scount=NHASH;
33
34     while(--scount >= 0){
35         if (sp->name && !strcasecmp(sp->name, sym))
36             return sp;
37         if (!sp->name) { /* nova entrada na TS */

```

```
Abrir  [+] bison-calc.y
72 }
73     a->nodetype='K';
74     a->number=d;
75     return (struct ast*)a;
76 }
77
78 struct ast *newcmp(int cmptype, struct ast *l, struct ast *r)
79 {
80     struct ast*a = malloc(sizeof(struct ast));
81     if (!a){
82         yyerror("sem espaco");
83         exit(0);
84     }
85     a->nodetype = '0' + cmptype;
86     a->l=l;
87     a->r=r;
88     return a;
89 }
90
91 struct ast * newfunc(int functype, struct ast *l)
92 {
93     struct fncall *a = malloc(sizeof(struct fncall));
94     if (!a){
95         yyerror("sem espaco");
96         exit (0);
97     }
98     a->nodetype='F';
99     a->l=l;
100     a->functype=functype;
101     return (struct ast *)a;
102 }
103
104 struct ast * newcall(struct symbol *s, struct ast *l)
105 {
106     struct ufncall *a = malloc(sizeof(struct ufncall));
107     if (!a){
108         yyerror("sem espaco");
109         exit(0);
110     }
111     a->nodetype='C';
112     a->l=l;
113     a->s=s;
114     return (struct ast *)a;
115 }
```

```
Abrir  [+] bison-calc.y
37     if (!sp->name) { /* nova entrada na TS */
38         sp->name = strdup(sym);
39         sp->value = 0;
40         sp->func = NULL;
41         sp->syms = NULL;
42         return sp;
43     }
44
45     if (++sp >= symtab+NHASH)
46         sp=symtab; /* tenta a prox entrada */
47
48     yyerror("overflow na tab. simbolos\n");
49     abort(); /*tabela esta cheia*/
50 }
51
52 struct ast * newast(int nodetype, struct ast *l, struct ast *r)
53 {
54     struct ast *a=malloc(sizeof(struct ast));
55
56     if (!a){
57         yyerror("sem espaco");
58         exit(0);
59     }
60     a->nodetype = nodetype;
61     a->l=l;
62     a->r=r;
63     return a;
64 }
65
66 struct ast*newnum(double d)
67 {
68     struct numval *a = malloc(sizeof(struct numval));
69     if (!a){
70         yyerror("sem espaco");
71         exit(0);
72     }
73     a->nodetype='K';
74     a->number=d;
75 }
```

```
bison-calc.y
109         exit(0);
110     }
111     a->nodetype='C';
112     a->l=l;
113     a->s=s;
114     return (struct ast *)a;
115 }
116
117 struct ast * newref(struct symbol *s)
118 {
119     struct symref *a=malloc(sizeof(struct symref));
120     if (!a){
121         yyerror("sem espaco");
122         exit(0);
123     }
124     a->nodetype='N';
125     a->s=s;
126     return (struct ast *)a;
127 }
128
129 struct ast * newasgn(struct symbol *s, struct ast *v)
130 {
131     struct symasgn *a=malloc(sizeof(struct symasgn));
132     if (!a){
133         yyerror("sem espaco");
134         exit(0);
135     }
136     a->nodetype = '=';
137     a->s=s;
138     a->v=v;
139     return (struct ast *)a;
140 }
141
142 struct ast * segundoflow(int nodetype, struct ast *cond, struct ast *cond2, struct ast *tl, struct ast *el)
143 {
144     struct flow *a = malloc(sizeof(struct flow));
145     if (!a){
146         yyerror("sem espaco");
147     }
148     a->nodetype=nodetype;
149     a->cond=cond;
150     a->cond2=cond2;
151     a->tl=tl;
152     a->el=el;
153     return (struct ast *)a;
154 }
155
156 struct ast * newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *el)
157 {
158     struct flow *a = malloc(sizeof(struct flow));
159     if (!a){
160         yyerror("sem espaco");
161         exit (0);
162     }
163     a->nodetype=nodetype;
164     a->cond=cond;
165     a->tl=tl;
166     a->el=el;
167     return (struct ast *)a;
168 }
169
170 /* libera uma arvore de AST */
171
172 void treefree(struct ast *a)
173 {
174     switch(a->nodetype){
175         /* duas subárvores */
176         case '+':
177         case '-':
178         case '*':
179         case '/':
180         case '1': case '2': case '3': case '4': case '5': case '6':
181         case '!':
```

Yacc ▾ Largura da tabulação

```
bison-calc.y
144     if (!a){
145         yyerror("sem espaco");
146         exit (0);
147     }
148     a->nodetype=nodetype;
149     a->cond=cond;
150     a->cond2=cond2;
151     a->tl=tl;
152     a->el=el;
153     return (struct ast *)a;
154 }
155
156 struct ast * newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *el)
157 {
158     struct flow *a = malloc(sizeof(struct flow));
159     if (!a){
160         yyerror("sem espaco");
161         exit (0);
162     }
163     a->nodetype=nodetype;
164     a->cond=cond;
165     a->tl=tl;
166     a->el=el;
167     return (struct ast *)a;
168 }
169
170 /* libera uma arvore de AST */
171
172 void treefree(struct ast *a)
173 {
174     switch(a->nodetype){
175         /* duas subárvores */
176         case '+':
177         case '-':
178         case '*':
179         case '/':
180         case '1': case '2': case '3': case '4': case '5': case '6':
181         case '!':
```



```
Abrir  bison-calc.y
180     case '1': case '2': case '3': case '4': case '5': case '6':
181     case 'L':
182         treefree(a->r);
183
184     /* uma subarvore */
185     case 'C': case 'F':
186         treefree(a->l);
187     /* sem subarvore */
188     case 'K': case 'N':
189         break;
190
191     case '=':
192         free (((struct symasgn *)a)->v);
193         break;
194
195     /* acima de 3 subarvorea */
196     case 'I': case 'W': case 'P': case 'A': case 'O':
197         free(((struct flow *)a)->cond);
198         if(((struct flow *)a)->tl) treefree(((struct flow *)a)->tl);
199         if(((struct flow *)a)->el) treefree(((struct flow *)a)->el);
200         break;
201     default: fprintf(yyout,"erro interno: free bad node %c\n",a->nodetype);
202 }
203 free(a); /* sempre libera o proprio no */
204 }
205 struct symlist * newsymlist(struct symbol *sym, struct symlist *next)
206 {
207     struct symlist *sl = malloc(sizeof(struct symlist));
208
209     if (!sl){
210         yyerror("sem espaco");
211         exit (0);
212     }
213     sl->sym=sym;
214     sl->next=next;
215     return sl;
216 }
```

```
Abrir  bison-calc.y
216 }
217
218 /*libera uma lista de simbolos */
219 void symlistfree(struct symlist *sl)
220 {
221     struct symlist *nsl;
222     while(sl){
223         nsl=sl->next;
224         free(sl);
225         sl=nsl;
226     }
227 }
228
229 /*etapa principal >> avalicao de expressoes, comandos, funcoes */
230
231 static double callbuiltin(struct fncall *);
232 static double calluser(struct ufncall *);
233
234 double eval(struct ast *a)
235 {
236     double v;
237
238     if (!a){
239         yyerror ("erro interno, null eval");
240         return 0.0;
241     }
242     switch(a->nodetype){
243     /*constante*/
244     case 'K': v = ((struct numval *)a)->number; break;
245     /*referencia de nome*/
246     case 'N': v = ((struct symref *)a)->s->value; break;
247     /*atribuicao*/
248     case '=': v=((struct symasgn *)a)->s->value = eval(((struct symasgn *)a)->v); break;
249     /*expressoes*/
250     case '+': v=eval(a->l) + eval(a->r); break;
251     case '-': v=eval(a->l) - eval(a->r); break;
252     case '*': v=eval(a->l) * eval(a->r); break;
```

Abrir

bison-calc.y

```
251 case ':': v=eval(a->l) - eval(a->r); break;
252 case '*': v=eval(a->l) * eval(a->r); break;
253 case '/': v=eval(a->l) / eval(a->r); break;
254
255 /*comparacoes*/
256 case '1': v=(eval(a->l) > eval(a->r))? 1 : 0; break;
257 case '2': v=(eval(a->l) < eval(a->r))? 1 : 0; break;
258 case '3': v=(eval(a->l) != eval(a->r))? 1 : 0; break;
259 case '4': v=(eval(a->l) == eval(a->r))? 1 : 0; break;
260 case '5': v=(eval(a->l) >= eval(a->r))? 1 : 0; break;
261 case '6': v=(eval(a->l) <= eval(a->r))? 1 : 0; break;
262
263 /*controle de fluxo*/
264 /*gramatica permite expressoes vazias, entao devem ser verificadas*/
265 /* if/then/else*/
266 case 'I':
267     if(eval(((struct flow *)a)->cond) !=0) { /*verifica condicao*/
268         if (((struct flow *)a)->tl) { /*ramo verdadeiro*/
269             v=eval(((struct flow *)a)->tl);
270         }else
271             v = 0.0; /*valor default */
272         }else{
273             if (((struct flow *)a)->el){ /*ramo falso*/
274                 v=eval(((struct flow *)a)->el);
275             }else
276                 v=0.0; /*valor default */
277         }
278     break;
279 case 'O':
280     fprintf(yyout,"comando OR\n");
281     if(eval(((struct flow *)a)->cond) !=0 || eval(((struct flow *)a)->cond2)) { /*verifica condicao*/
282         if (((struct flow *)a)->tl) { /*ramo verdadeiro*/
283             v=eval(((struct flow *)a)->tl);
284         }else
285             v = 0.0; /*valor default */
286         }else{
287             if (((struct flow *)a)->el){ /*ramo falso*/
288                 v=eval(((struct flow *)a)->el);
```

Abrir

bison-calc.y

```
287         if (((struct flow *)a)->el){ /*ramo falso*/
288             v=eval(((struct flow *)a)->el);
289         }else
290             v=0.0; /*valor default */
291     }
292
293     break;
294 case 'A':
295     fprintf(yyout,"comando AND\n");
296     if(eval(((struct flow *)a)->cond) !=0 && eval(((struct flow *)a)->cond2)) { /*verifica condicao*/
297         if (((struct flow *)a)->tl) { /*ramo verdadeiro*/
298             v=eval(((struct flow *)a)->tl);
299         }else
300             v = 0.0; /*valor default */
301         }else{
302             if (((struct flow *)a)->el){ /*ramo falso*/
303                 v=eval(((struct flow *)a)->el);
304             }else
305                 v=0.0; /*valor default */
306         }
307     }
308     break;
309 /*while/do*/
310 case 'W':
311     v=0.0;
312     if (((struct flow *)a)->tl){ /* testa se lista de comandos nao eh vazia*/
313         while(eval(((struct flow *)a)->cond) != 0) /* avalia a condicao */
314             v=eval(((struct flow *)a)->tl); /* avalia comandos */
315     }
316     break;
317 /* for */
318 case 'P':
319     v=0.0;
320     fprintf(yyout,"comando for\n");
321     if (((struct flow *)a)->tl){ /* testa se lista de comandos nao eh vazia*/
322         while(eval(((struct flow *)a)->cond) != 0) /* avalia a condicao */
323             v=eval(((struct flow *)a)->tl); /* avalia comandos */
```

```
Abrir  bison-calc.y
317 /* for */
318 case 'P':
319 v=0.0;
320 fprintf(yyout,"comando for\n");
321 if (((struct flow *)a)->tl){/* testa se lista de comandos nao eh vazia*/
322 while(eval(((struct flow *)a)->cond) != 0) /* avalia a condicao */
323 v=eval(((struct flow *)a)->tl);/* avalia comandos */
324 }
325 break;
326 /* lista de comandos*/
327 case 'L': eval(a->l) ; v=eval(a->r); break;
328 case 'F': v = callbuiltin((struct fncall *)a); break;
329 case 'C': v= calluser((struct ufncall *)a); break;
330 default: fprintf(yyout,"erro interno: bad node %c\n",a->nodetype);
331 }
332 return v;
333 }
334
335 static double callbuiltin(struct fncall *f)
336 {
337     enum bifs functype = f->functype;
338     double v=eval(f->l);
339     switch(functype){
340     case B_sqrt:
341         return sqrt(v);
342     case B_exp:
343         return exp(v);
344     case B_log:
345         return log(v);
346     case B_print:
347         fprintf(yyout," =%4.4g\n", v);
348         return v;
349     default:
350         yyerror("Funcao pre definida %d desconhecida\n",functype);
351         return 0.0;
352     }
353 }
```

```
Abrir  bison-calc.y
354
355 /*funcao definida por usuario*/
356 void dodef(struct symbol *name, struct symlist *syms, struct ast *func)
357 {
358     if (name->syms) symlistfree(name->syms);
359     if (name->func) treefree(name->func);
360     name->syms=syms;
361     name->func=func;
362 }
363
364 static double calluser(struct ufncall *f)
365 {
366     struct symbol *fn=f->s; /*nome da funcao*/
367     struct symlist *sl; /* argumentos (originais)*/
368     struct ast *args = f->l; /*argumentos (usados) na funcao */
369     double *oldval, *newval; /*salvar valores de argumentos*/
370     double v;
371     int nargs;
372     int i;
373
374     if (!fn->func){
375         yyerror("chamada para funcao %s indefinida",fn->name);
376         return 0;
377     }
378     /*contar argumentos*/
379     sl=fn->syms;
380     for (nargs=0; sl; sl=sl->next)
381         nargs++;
382     /*prepara o para salvar argumentos*/
383     oldval=(double *)malloc(nargs *sizeof(double));
384     newval=(double *)malloc(nargs * sizeof(double));
385     if (!oldval || !newval){
386         yyerror("Sem espaco em %s", fn->name);
387         return 0.0;
388     }
389
390 /*avaliacao de argumentos*/
```



```

392 }
393     yyerror("poucos argumentos na chamada da funcao %s\n",fn->name);
394     free(oldval);
395     free(newval);
396     return 0.0;
397 }
398
399 if (args->nodetype == 'L') { /* se eh uma lista de nos */
400     newval[i]=eval(args->l);
401     args=args->r;
402 } else { /* se eh o final da lista */
403     newval[i] = eval(args);
404     args=NULL;
405 }
406 }
407
408 /* salvar valores (originais) dos argumentos, atribuir novos valores */
409 sl=fn->syms;
410 for(i=0;i<nargs;i++){
411     struct symbol *s=sl->sym;
412     oldval[i]=s->value;
413     s->value=newval[i];
414     sl=sl->next;
415 }
416 free(newval);
417 /* avaliacao da funcao */
418 v=eval(fn->func);
419 /* recolocar os valores (originais) da funcao */
420 sl=fn->syms;
421 for (i=0;i<nargs;i++){
422     struct symbol *s=sl->sym;
423     s->value = oldval[i];
424     sl=sl->next;
425 }
426
427 free(oldval);
428 return v;
429 }

```

```

Abrir  bison-calc.y
428 return v;
429 }
430
431 void yyerror(char *s, ...)
432 {
433     va_list ap;
434     va_start(ap,s);
435     fprintf(stderr, "%d: error: ",yylineno);
436     vfprintf(stderr, s, ap);
437     fprintf(stderr, "\n");
438 }
439 %}
440
441 %union {
442     struct ast *a;
443     double d;
444     struct symbol *s; /* qual simbolo? */
445     struct symlist *sl;
446     int fn; /* qual funcao? */
447 }
448
449 /* declaracao de tokens */
450 %token <d> NUMBER
451 %token <s> NAME
452 %token <fn> FUNC
453 %token EOL
454
455 %token IF THEN ELSE WHILE DO LET FOR AND OR
456
457 %nonassoc <fn> CMP
458 %right '='
459 %left '+' '-'
460 %left '*' '/'
461
462 %type <a> exp stmt list explist
463 %type <sl> symlist
464
465 %start calclist

```

```

Abrir  bison-calc.y
466 %%
467
468 stmt: IF exp THEN list { $$ = newflow('I', $2, $4, NULL); }
469      | IF exp THEN list ELSE list { $$ = newflow('I', $2, $4, $6); }
470      | IF exp AND exp THEN list { $$ = segundoflow('A', $2, $4, $6, NULL); }
471      | IF exp AND exp THEN list ELSE list { $$ = segundoflow('A', $2, $4, $6, $8); }
472      | IF exp OR exp THEN list { $$ = segundoflow('O', $2, $4, $6, NULL); }
473      | IF exp OR exp THEN list ELSE list { $$ = segundoflow('O', $2, $4, $6, $8); }
474      | WHILE exp DO list { $$ = newflow('W', $2, $4, NULL); }
475      | FOR exp exp list { $$ = newflow('P', $3, $4, NULL); }
476      | exp
477      ;
478
479 list: /* vazio! */ { $$ = NULL; }
480      | stmt ';' list { if ($3 == NULL)
481                      $$ = $1;
482                      else
483                      $$ = newast('L', $1, $3);
484                      }
485      ;
486
487 exp: exp CMP exp { $$ = newcmp($2, $1, $3); }
488     | exp '+' exp { $$ = newast('+', $1, $3); }
489     | exp '-' exp { $$ = newast('-', $1, $3); }
490     | exp '*' exp { $$ = newast('*', $1, $3); }
491     | exp '/' exp { $$ = newast('/', $1, $3); }
492     | '(' exp ')' { $$ = $2; }
493     | NUMBER { $$ = newnum($1); }
494     | NAME { $$ = newref($1); }
495     | NAME '=' exp { $$ = newasgn($1, $3); }
496     | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
497     | NAME '(' explist ')' { $$ = newcall($1, $3); }
498     ;
499
500 explist: exp
501         | exp ',' explist { $$ = newast('L', $1, $3); }
502         ;

```

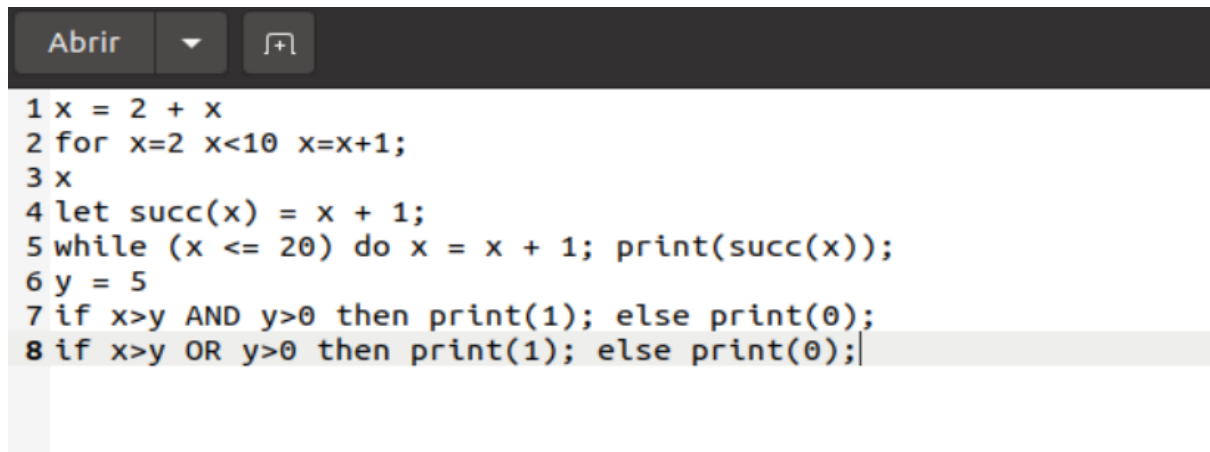
```

500 explist: exp
501         | exp ',' explist { $$ = newast('L', $1, $3); }
502         ;
503
504 symlist: NAME { $$ = newsymlist($1, NULL); }
505         | NAME ',' symlist { $$ = newsymlist($1, $3); }
506         ;
507
508 calclist: /* vazio! */
509         | calclist stmt EOL {
510             fprintf(yyout, "= %4.4g\n> ", eval($2));
511             treefree($2);
512         }
513         | calclist LET NAME '(' symlist ')' '=' list EOL {
514             dodef($3, $5, $8);
515             fprintf(yyout, "Defined %s\n> ", $3->name); }
516         | calclist error EOL { yyerrok; fprintf(yyout, "> "); }
517         ;
518
519 %%
520
521
522 int main(int argc, char *argv[]){
523     extern FILE *yyin, *yyout;
524     yyin = fopen("Input.txt", "r");
525     yyout = fopen("Output.txt", "w");
526     return yyparse();
527 }
528
529

```

#### 4.4 COMPILAÇÃO, ENTRADA E SAÍDA

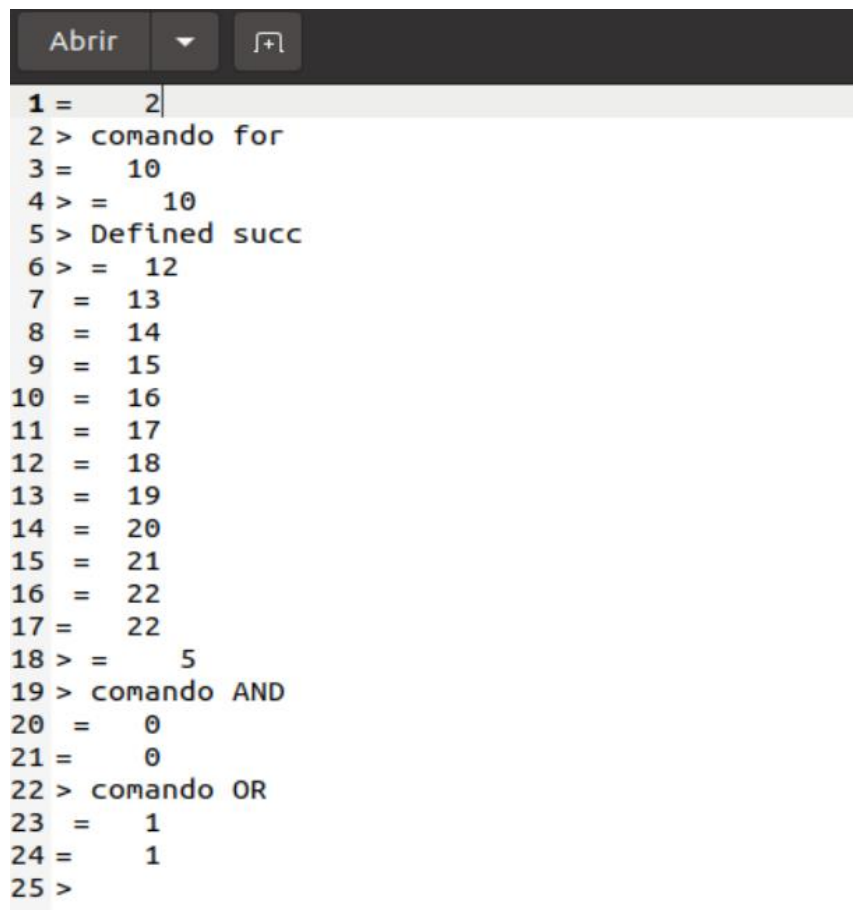
As figuras 10 e 11 demonstram a entrada a ser lida e a saída que foi gerada.



```
Abrir ▼ [+]
```

```
1 x = 2 + x
2 for x=2 x<10 x=x+1;
3 x
4 let succ(x) = x + 1;
5 while (x <= 20) do x = x + 1; print(succ(x));
6 y = 5
7 if x>y AND y>0 then print(1); else print(0);
8 if x>y OR y>0 then print(1); else print(0);
```

Figura 10 – Arquivo de entrada



```
Abrir ▼ [+]
```

```
1 = 2
2 > comando for
3 = 10
4 > = 10
5 > Defined succ
6 > = 12
7 = 13
8 = 14
9 = 15
10 = 16
11 = 17
12 = 18
13 = 19
14 = 20
15 = 21
16 = 22
17 = 22
18 > = 5
19 > comando AND
20 = 0
21 = 0
22 > comando OR
23 = 1
24 = 1
25 >
```

Figura 11 – Arquivo de saída em Markdown

## **5 CONCLUSÃO**

Neste trabalho, foi desenvolvido a implementação do laço for e dos operadores lógicos AND e OR. Além da abstração da gramática BNF, os diagramas de sintaxe foram elaborados. Junto com os diagramas de transição que representam as regras léxicas.

## 6 REFERÊNCIAS

[1] Link do Debuggex:

<https://www.debuggex.com/>

[2] Link do **Railroad Diagram Generator**:

<https://bottlecaps.de/rr/ui>

[3] INGERMAN, P. Z. Panini-Backus Form Suggested. Communications ACM, Vol. 10, No. 3. p. 137. 1967.