Analisador Léxico e Sintático de uma Linguagem Básica
Flex & Bison

Gabriel A. Posonski[1], **Helena Rentschler**[1]

[1]Departamento de Informática (DAINF)
Universidade Tecnológica Federal do Paraná – Ponta Grossa, PR – Brazil

{gabrielalessi,helenarentschler}@alunos.utfpr.edu.br

Abstract. *This article describes the operation of a lexical and syntactical analyzer program for a basic programming language presented in the Compilers discipline of the Computer Science course at UTFPR-PG. For this purpose, the Flex and Bison programs were used, in addition to auxiliary functions in C language.*

Resumo. *Este artigo descreve o funcionamento de um programa analisador léxico e sintático para uma linguagem de programação básica apresentada na disciplina de Compiladores, do curso de Ciência da Computação da UTFPR-PG. Para tal, foi utilizado os programas Flex e Bison, além de funções auxiliares em linguagem C.*

1. Introdução

Sem dúvidas, as primeiras etapas de um processo de Compilação são umas das mais importantes, pois é nela que está contida a lógica léxica e sintática do código-fonte que define qualquer programa de computador. Uma excelente forma de compreender tal processo, é através de uma simples implementação utilizando algumas das diversas ferramentas disponíveis atualmente. Para esse trabalho acadêmico, será utilizado o Flex para a análise léxica, Bison para análise sintática, além de funções auxiliares em linguagem C para manipulação de árvores sintáticas e tabelas de símbolos. Como a propósta do trabalho era a implementação e adição de algumas funcionalidades em uma linguagem básica apresentada no material do profº Dr. Gleifer Vaz Alves, foi-se reaproveitado grande parte do código-fonte contido nas apostilas, fazendo-se as alterações necessárias para o correto funcionamento do analisador.

2. Gramática

Um dos primeiros passos na construção de um compilador, é a definição da gramática que irá compor os códigos implementados da linguagem. Para defini-la, foi utilizado o formalismo EBNF [CMU]:

2.1. EBNF

```
calclist ::= ( (stmt
             | 'let' NAME '(' symlist ')' '=' list
             | error ) '\n' )*
    stmt ::= ('if' exp 'then' (list 'else')?
             | 'while' exp 'do'
             | 'for' '(' exp ';' exp ';' exp ')' )list
             | exp
    list ::= ( stmt ';' )*
     exp ::= exp ( CMP | LOGOP | '+' | '-' | '*' | '/' ) exp
             | ( '(' exp | FUNC '(' explist ) ')'
             | NUMBER
             | NAME ( '=' exp | '(' explist ')' )?
 explist ::= exp ( ',' exp )*
 symlist ::= NAME ( ',' symlist )*
  NUMBER ::= [0-9]+ ( '.'  [0-9]* )?  ( [Ee] [+-]?  [0-9]+ )?
    NAME ::= [a-zA-Z][a-zA-Z0-9]*
    FUNC ::= 'sqrt' | 'exp' | 'log' | 'print'
   LOGOP ::= '&&' | '||'
     CMP ::= '<' | '>' | '<>' | '==' | '>=' | '<='
```

Como alguns *tokens* retornavam apenas o próprio nome como valor, converteu-se os mesmos para literais nas regras de produção. o Simbolo não-terminal 'error' possui uma particularidade: não possui regras de produção, pois é um *token* utilizado pelo *Bison* para retorno de erros dentro do programa. Para essa definição, considera-se que 'error' produza qualquer mensagem de erro possível na implementação e, dessa forma, consiga continuar a execução para análise de outras expressões.

2.2. Diagramas de Sintaxe

Para a construção dos diagramas, foi utilizado o site Railroad Diagram [Railroad] .

2.2.1. calclist

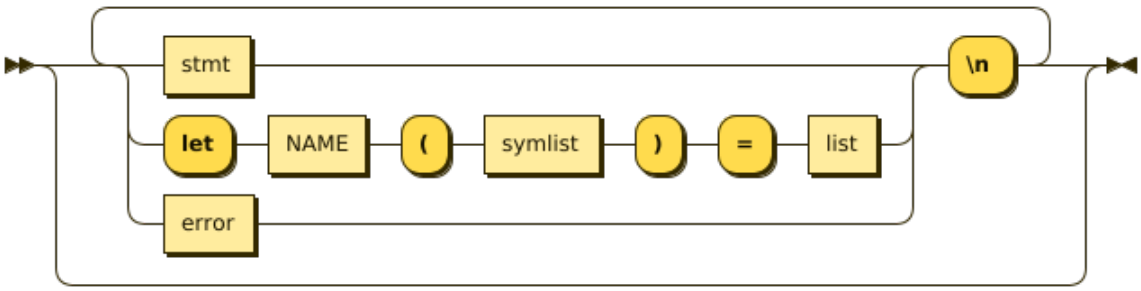

Figure 1. Diagrama calclist

2.2.2. stmt

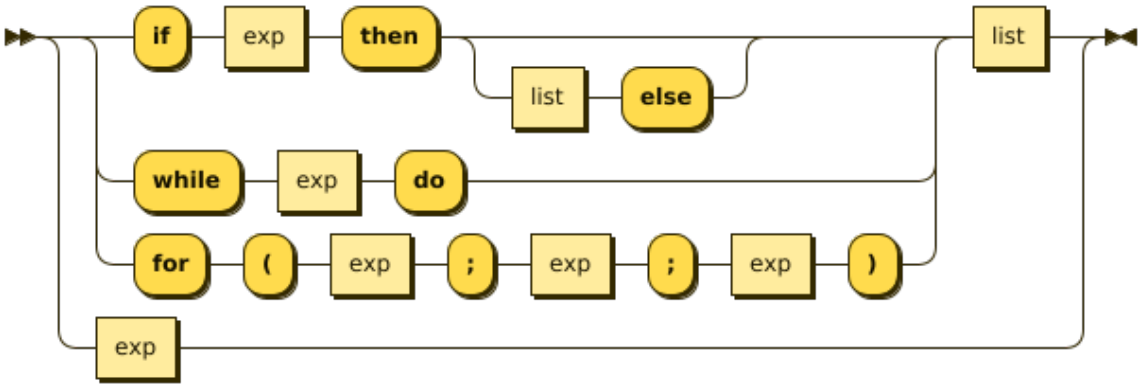

Figure 2. Diagrama stmt

2.2.3. list

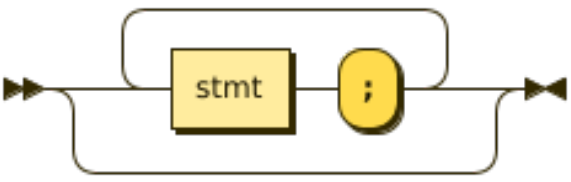

Figure 3. Diagrama list

2.2.4. exp

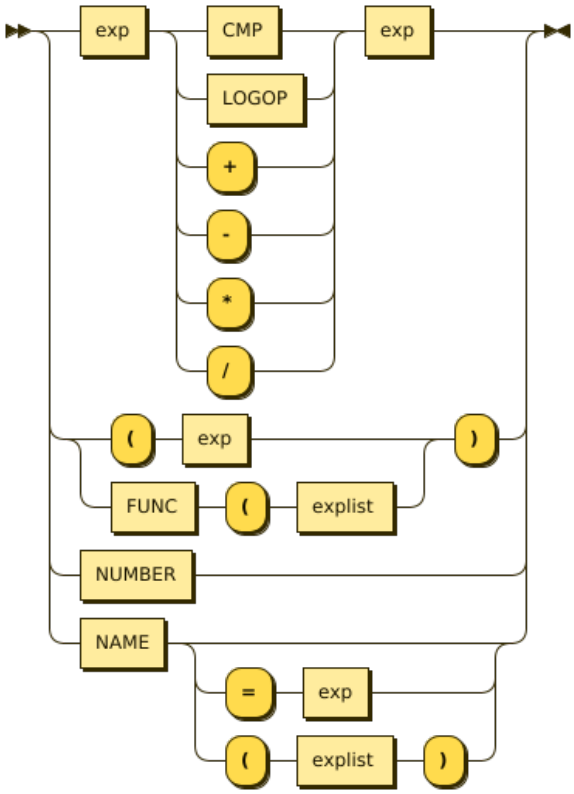

Figure 4. Diagrama exp

2.2.5. explist

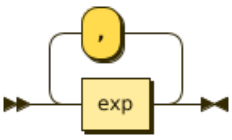

Figure 5. Diagrama explist

2.2.6. symlist

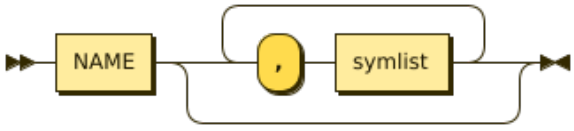

Figure 6. Diagrama symlist

2.2.7. NUMBER

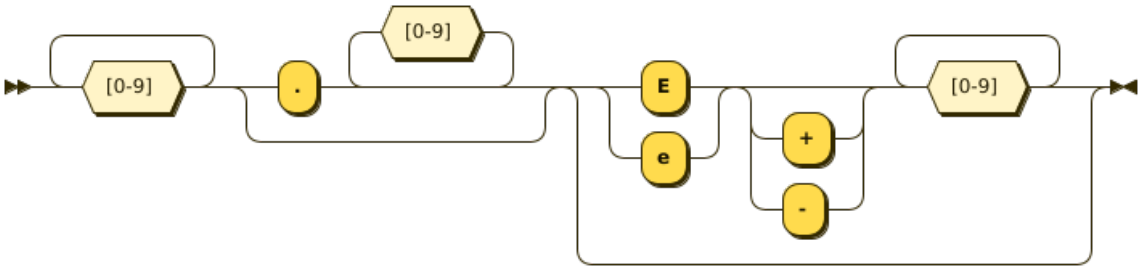

Figure 7. Diagrama NUMBER

2.2.8. NAME

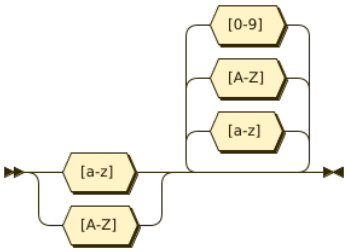

Figure 8. Diagrama NAME

2.2.9. FUNC

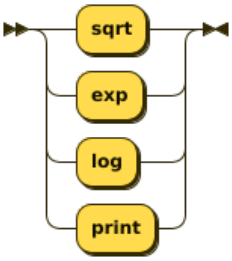

Figure 9. Diagrama FUNC

2.2.10. LOGOP

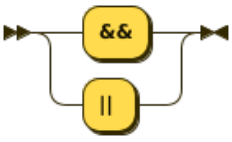

Figure 10. Diagrama LOGOP

2.2.11. CMP

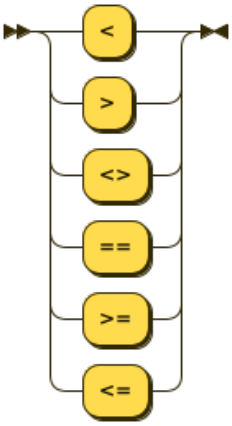

Figure 11. Diagrama CMP

3. Arquivo Lexer

Iniciando na Análise léxica, tem-se o arquivo `lexer.l`:

```
%option noyywrap nodefault yylineno
%{
    #include "basicLang.h"
    #include "parser.tab.h"
%}

EXP ([Ee][+-]?[0-9]+)

%%
"+" |
"-" |
"*" |
"/" |
"=" |
"," |
";" |
"(" |
")"     { return yytext[0]; }

">"     { yylval.fn = 1; return CMP; }
"<"     { yylval.fn = 2; return CMP; }
"<>"    { yylval.fn = 3; return CMP; }
"=="    { yylval.fn = 4; return CMP; }
">="    { yylval.fn = 5; return CMP; }
"<="    { yylval.fn = 6; return CMP; }

"&&"    { yylval.fn = 'A'; return LOGOP; }
```

```
"||"    { yylval.fn = 'O'; return LOGOP; }

"if"    { return IF; }
"then"  { return THEN; }
"else"  { return ELSE; }
"while" { return WHILE; }
"do"    { return DO; }
"for"   { return FOR; }
"let"   { return LET; }

"sqrt"  { yylval.fn = B_sqrt; return FUNC; }
"exp"   { yylval.fn = B_exp; return FUNC; }
"log"   { yylval.fn = B_log; return FUNC; }
"print" { yylval.fn = B_print; return FUNC; }

[a-zA-Z][a-zA-Z0-9]*    { yylval.s = lookup(yytext); return NAME; }
[0-9]+"."[0-9]*{EXP}? |
"."?[0-9]+{EXP}?        { yylval.d = atof(yytext); return NUMBER; }

"//".*
[ \t]

\\\n    { printf("c> "); }

\n      { return EOL; }

.       { yyerror("Caracter desconhecido %c\n", *yytext); }
%%
```

Dos requisitos do trabalho, a única mudança realizada nesse arquivo foi a adição dos lexemas '`&&`' e '`||`' que agem juntamente com o *token* 'LOGOP' para operadores lógicos *and* e *or*; e de '`for`' para a implementação do laço *for*. O *Lexer* é responsável por fracionar o código-fonte em trechos de caracteres (lexemas) que tenham certa significância dentro da gramática implementada. Este trabalha em conjunto com o *Bison* - que será abordado na próxima seção - auxiliando-o na análise sintática retornando os tokens no processo de leitura do arquivo.

4. Arquivo Parser

No processo de análise sintática, temos as definições do arquivo `parser.y`:

```
    %{
    #include <stdio.h>
    #include <stdlib.h>
    #include "basicLang.h"
%}

%union{
```

```
8       ast *a;
9       double d;
10      symbol *s;
11      symList *sl;
12      int fn;
13  }
14
15  %token <d> NUMBER
16  %token <s> NAME
17  %token <fn> FUNC
18  %token EOL
19
20  %token IF THEN ELSE WHILE DO LET FOR LOGOP
21
22  %right '='
23  %left <fn> LOGOP
24  %nonassoc <fn> CMP
25  %left '+' '-'
26  %left '*' '/'
27
28  %type <a> exp stmt list explist
29  %type <sl> symlist
30
31  %start calclist
32
33  %%
34  stmt: IF exp THEN list  { $$ = newFlow('I', $2, $4, NULL, NULL); }
35      | IF exp THEN list ELSE list  { $$ = newFlow('I', $2, $4, $6,
        ↪  NULL); }
36      | WHILE exp DO list  { $$ = newFlow('W', $2, $4, NULL, NULL); }
37      | FOR '('exp';'exp';'exp')' list { $$ = newFlow('R', $5, $9, $3,
        ↪  $7); }
38      | exp
39      ;
40
41  list:                { $$ = NULL;}
42      | stmt ';' list { if ($3 == NULL)
43                $$ = $1;
44          else
45                $$ = newAst('L',$1, $3); }
46      ;
47
48  exp: exp LOGOP exp { $$ = newLogOp($2, $1, $3); }
49      | exp CMP exp    { $$ = newCmp($2,$1,$3); }
50      | exp '+' exp   { $$ = newAst('+',$1, $3); }
51      | exp '-' exp   { $$ = newAst('-',$1, $3); }
```

```
52      | exp '*' exp   { $$ = newAst('*',$1, $3); }
53      | exp '/' exp   { $$ = newAst('/', $1, $3); }
54      | '(' exp ')'   { $$ = $2; }
55      | NUMBER        { $$ = newNum($1); }
56      | NAME          { $$ = newRef($1); }
57      | NAME '=' exp  { $$ = newAssign($1, $3); }
58      | FUNC '(' explist ')' { $$ = newFunc($1, $3); }
59      | NAME '(' explist ')' { $$ = newCall($1, $3); }
60      ;
61
62  explist: exp
63      | exp ',' explist  { $$ = newAst('L', $1, $3); }
64      ;
65
66  symlist: NAME { $$ = newSymList($1, NULL); }
67      | NAME ',' symlist { $$ = newSymList($1, $3); }
68      ;
69
70  calclist:
71      | calclist stmt EOL {
72          printf(" = %4.4g\n> ", eval($2));
73          freeTree($2);
74      }
75      | calclist LET NAME '(' symlist ')' '=' list EOL {
76          doDef($3, $5, $8);
77          printf("Defined %s\n>", $3->name); }
78      | calclist error EOL { yyerrok; printf("> "); }
79  %%
```

Nota-se inicialmente uma grande similaridade do arquivo do *Bison* com a da definição da gramática, pois é aqui onde se define as regras de produção. Das principais mudanças, tem-se: adição dos *tokens* 'FOR' e 'LOGOP' e suas regras de produção; e alteração da ordem de precedência dos operadores. No modelo apresentado na apostila, ao realizar uma operação de comparação, e.g.: `a = 5 > 3`, o analisador atribuía o valor `'5'` ao símbolo `'a'` ao invés de `'1'`, o que não é um comportamento esperado, já que espera-se um valor booleano para expressões como essa. Então, movendo a expressão "`%right '='`" para o topo do trecho, garantimos que operações de atribuição serão as últimas a serem validadas. Os operadores lógicos sucedem os operadores comparativos na ordem de precedência, possibilitando expressões do tipo: `a > b && c <= d` , onde temos que verificar mais de uma comparação, muito comuns em condicionais de mudança de fluxo de execução.

5. Funções Auxiliares

Nesta seção, iremos analisar as funções auxiliares que promovem a estruturação da análise sintática, utilizando tanto a tabela de símbolos, quanto a análise sintática e avaliação de operações. Primeiro, apresenta-se o arquivo de declarações, `basicLang.h`

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>

/*interface lexer*/
extern int yylineno;
void yyerror(char *s, ...);


/*tabela de simbolos*/
typedef struct symbol
{
    char *name;
    double value;
    struct ast *func;
    struct symList *syms;
} symbol;

/*tabela de simbolos com tamanho fixo*/
#define NHASH 9997
extern symbol symTab[NHASH];

symbol *lookup(char *);

/*lista de simbolos para lista de argumentos*/
typedef struct symList
{
    symbol *sym;
    struct symList *next;
} symList;

struct symList *newSymList(symbol *sym, symList *next);
void freeSymList(symList *sl);

/*************************************************************
 *tipos de nos
 * + - * /
 * 0-7 op comparacao
 * L expressao ou lista de comandos
 * I comando IF
 * W comando while
 * R comando FOR
 * N symbol de ref
 * = atribuicao
```

```
 * S lista de simbolos
 * F chamada de funcao pre-definida
 * C chamada de funcao def. por usuario
 * A and
 * O or
 *********************************************************/

enum bifs /*funcoes pre definidas*/
{
    B_sqrt = 1,
    B_exp,
    B_log,
    B_print
};

/*nos na AST*/
typedef struct ast
{
    int nodetype;
    struct ast *l;
    struct ast *r;
} ast;

typedef struct fnCall /*pre-definida*/
{
    int nodetype; /*tipo F*/
    struct ast *l;
    enum bifs functype;
} fnCall;

typedef struct ufnCall /*usuario*/
{
    int nodetype; /*tipo C*/
    struct ast *l;
    symbol *s;
} ufnCall;

typedef struct flow
{
    int nodetype; /*tipo I, W ou R*/
    struct ast *cond; /*condicao*/
    struct ast *tl;     /*ramo then ou lista do*/
    struct ast *el;    /*ramo opcional else*/
    ast *posCmd; /*comando apos iteracao para for*/
} flow;

```

```
93  typedef struct numVal
94  {
95      int nodetype;    /*tipo K*/
96      double number;
97  } numVal;
98
99  typedef struct symRef
100 {
101     int nodetype;    /*tipo N*/
102     symbol *s;
103 } symRef;
104
105 typedef struct symAssign
106 {
107     int nodetype;    /*tipo =*/
108     symbol *s;
109     ast *v;          /*valor a ser atribuido*/
110 } symAssign;
111
112 /*construcao da AST*/
113 ast *newAst(int nodetype, ast *l, ast *r);
114 ast *newCmp(int cmptype, ast *l, ast *r);
115 ast *newFunc(int functype, ast *l);
116 ast *newCall(symbol *s, ast *l);
117 ast *newRef(symbol *s);
118 ast *newAssign(symbol *s, ast *v);
119 ast *newNum(double number);
120 ast *newFlow(int nodetype, ast *cond, ast *tl, ast *tr, ast
    ↪  *posCond);
121 ast *newLogOp(int logicalType, ast *l, ast *r);
122 /*definicao de uma funcao*/
123 void doDef(symbol *name, symList *syms, ast *stmts);
124
125 /*avaliacao da AST*/
126 double eval(ast *);
127
128 /*destruir AST*/
129 void freeTree(ast *);
130
```

Consequentemente, tem-se o arquivo `basicLang.c` com a implementação das funções previamente declaradas, além da função `main()`, na qual se inicia o programa:

```
1  #include "basicLang.h"
2  symbol symTab[NHASH];
3
4  static unsigned symHash(char *sym)
5  {
```

```
6       unsigned int hash = 0;
7       unsigned c;
8       while (c = *sym++)
9           hash = hash * 9 ^ c;
10      return hash;
11  }
12
13  symbol *lookup(char *sym)
14  {
15      symbol *sp = &symTab[symHash(sym) % NHASH];
16      int scount = NHASH;
17      while (--scount >= 0)
18      {
19          if (sp->name && !strcasecmp(sp->name, sym))
20              return sp;
21          if (!sp->name)
22          {
23              sp->name = strdup(sym);
24              sp->value = 0;
25              sp->func = NULL;
26              sp->syms = NULL;
27              return sp;
28          }
29
30          if (++sp >= symTab + NHASH)
31              sp = symTab;
32      }
33      yyerror("overflow na tabela de simbolos\n");
34      abort();
35  }
36
37  ast *newAst(int nodetype, ast *l, ast *r)
38  {
39      ast *a = malloc(sizeof(ast));
40      if (!a)
41      {
42          yyerror("sem espaco");
43          exit(1);
44      }
45      a->nodetype = nodetype;
46      a->l = l;
47      a->r = r;
48      return a;
49  }
50
51  ast *newNum(double d)
```

```
{
    numVal *a = malloc(sizeof(numVal));
    if (!a)
    {
        yyerror("sem espaco");
        exit(1);
    }
    a->nodetype = 'K';
    a->number = d;
    return (ast *)a;
}

ast *newCmp(int cmpType, ast *l, ast *r)
{
    ast *a = malloc(sizeof(ast));
    if (!a)
    {
        yyerror("sem espaco");
        exit(1);
    }
    a->nodetype = '0' + cmpType;
    a->l = l;
    a->r = r;
    return a;
}

ast *newLogOp(int logicalType, ast *l, ast *r)
{
    ast *a = malloc(sizeof(ast));
    if (!a)
    {
        yyerror("sem espaco");
        exit(1);
    }
    a->nodetype = logicalType;
    a->l = l;
    a->r = r;
    return a;
}
ast *newFunc(int funcType, ast *l)
{
    fnCall *a = malloc(sizeof(fnCall));
    if (!a)
    {
        yyerror("sem espaco");
        exit(1);
```

```
98      }
99      a->nodetype = 'F';
100     a->l = l;
101     a->functype = funcType;
102     return (ast *)a;
103 }
104
105 ast *newCall(symbol *s, ast *l)
106 {
107     ufnCall *a = malloc(sizeof(ufnCall));
108     if (!a)
109     {
110         yyerror("sem espaco");
111         exit(1);
112     }
113     a->nodetype = 'C';
114     a->l = l;
115     a->s = s;
116     return (ast *)a;
117 }
118
119 ast *newRef(symbol *s)
120 {
121     symRef *a = malloc(sizeof(symRef));
122     if (!a)
123     {
124         yyerror("sem espaco");
125         exit(1);
126     }
127     a->nodetype = 'N';
128     a->s = s;
129     return (ast *)a;
130 }
131
132 ast *newAssign(symbol *s, ast *v)
133 {
134     symAssign *a = malloc(sizeof(symAssign));
135     if (!a)
136     {
137         yyerror("sem espaco");
138         exit(1);
139     }
140     a->nodetype = '=';
141     a->s = s;
142     a->v = v;
143     return (ast *)a;
```

```
144 }
145
146 ast *newFlow(int nodetype, ast *cond, ast *tl, ast *el, ast
    ↪  *posCmd)
147 {
148     flow *a = malloc(sizeof(flow));
149     if (!a)
150     {
151         yyerror("sem espaco");
152         exit(1);
153     }
154     a->nodetype = nodetype;
155     a->cond = cond;
156     a->tl = tl;
157     a->el = el;
158     a->posCmd = posCmd;
159     return (ast *)a;
160 }
161
162 void freeTree(ast *a)
163 {
164     switch (a->nodetype)
165     {
166     case '+':
167     case '-':
168     case '*':
169     case '/':
170     case '1':
171     case '2':
172     case '3':
173     case '4':
174     case '5':
175     case '6':
176     case 'A':
177     case 'O':
178     case 'L':
179         freeTree(a->r);
180     case 'C':
181     case 'F':
182         freeTree(a->l);
183     case 'K':
184     case 'N':
185         break;
186     case '=':
187         free(((symAssign *)a)->v);
188         break;
```

```
189     case 'R':
190         free(((flow *)a)->posCmd);
191     case 'I':
192     case 'W':
193         free(((flow *)a)->cond);
194         if (((flow *)a)->tl)
195             freeTree(((flow *)a)->tl);
196         if (((flow *)a)->el)
197             freeTree(((flow *)a)->el);
198         break;
199     default:
200         printf("erro interno: free bad node %c \n", a->nodetype);
201     }
202     free(a);
203 }
204 
205 symList *newSymList(symbol *sym, symList *next)
206 {
207     symList *sl = malloc(sizeof(symList));
208     if (!sl)
209     {
210         yyerror("sem espaco");
211         exit(1);
212     }
213     sl->sym = sym;
214     sl->next = next;
215     return sl;
216 }
217 
218 void freeSymList(symList *sl)
219 {
220     symList *nsl;
221     while (sl)
222     {
223         nsl = sl->next;
224         free(sl);
225         sl = nsl;
226     }
227 }
228 
229 static double callBuiltIn(fnCall *);
230 static double callUser(ufnCall *);
231 
232 double eval(ast *a)
233 {
234     double v;
```

```
235     if (!a)
236     {
237         yyerror("erro interno, null eval");
238         return 0.0;
239     }
240 
241     switch (a->nodetype)
242     {
243     case 'K':
244         v = ((numVal *)a)->number;
245         break;
246     case 'N':
247         v = ((symRef *)a)->s->value;
248         break;
249     case '=':
250         v = ((symAssign *)a)->s->value = eval(((symAssign *)a)->v);
251         break;
252     case '+':
253         v = eval(a->l) + eval(a->r);
254         break;
255     case '-':
256         v = eval(a->l) - eval(a->r);
257         break;
258     case '*':
259         v = eval(a->l) * eval(a->r);
260         break;
261     case '/':
262         if (eval(a->r) == 0.0)
263         {
264             yyerror("Divisao por zero");
265             return 0.0;
266         }
267         v = eval(a->l) / eval(a->r);
268         break;
269     case 'A':
270         v = (eval(a->l) && eval(a->r));
271         break;
272     case 'O':
273         v = (eval(a->l) || eval(a->r));
274         break;
275     case '1':
276         v = (eval(a->l) > eval(a->r)) ? 1 : 0;
277         break;
278     case '2':
279         v = (eval(a->l) < eval(a->r)) ? 1 : 0;
280         break;
```

```
    case '3':
        v = (eval(a->l) != eval(a->r)) ? 1 : 0;
        break;
    case '4':
        v = (eval(a->l) == eval(a->r)) ? 1 : 0;
        break;
    case '5':
        v = (eval(a->l) >= eval(a->r)) ? 1 : 0;
        break;
    case '6':
        v = (eval(a->l) <= eval(a->r)) ? 1 : 0;
        break;
    case 'I':
        if (eval(((flow *)a)->cond) != 0)
        {
            if (((flow *)a)->tl)
            {
                v = eval(((flow *)a)->tl);
            }
            else
            {
                v = 0.0;
            }
        }
        else
        {
            if (((flow *)a)->el)
            {
                v = eval(((flow *)a)->el);
            }
            else
            {
                v = 0.0;
            }
        }
        break;
    case 'W':
        v = 0.0;
        if (((flow *)a)->tl)
        {
            while (eval(((flow *)a)->cond) != 0)
                v = eval(((flow *)a)->tl);
        }
        break;
    case 'R':
        v = 0.0;
```

```
327         if (((flow *)a)->tl)
328         {
329             for (eval(((flow *)a)->el); eval(((flow *)a)->cond);
                ↪  eval(((flow *)a)->posCmd))
330             {
331 
332                 v = eval(((flow *)a)->tl);
333             }
334             break;
335         }
336     case 'L':
337         eval(a->l);
338         v = eval(a->r);
339         break;
340     case 'F':
341         v = callBuiltIn((fnCall *)a);
342         break;
343     case 'C':
344         v = callUser((ufnCall *)a);
345         break;
346     default:
347         printf("erro interno: bad node %c\n", a->nodetype);
348     }
349     return v;
350 }
351 
352 static double callBuiltIn(fnCall *f)
353 {
354     enum bifs funcType = f->functype;
355     double v = eval(f->l);
356     switch (funcType)
357     {
358     case B_sqrt:
359         return sqrt(v);
360     case B_exp:
361         return exp(v);
362     case B_log:
363         return log(v);
364     case B_print:
365         printf(" = %4.4g\n", v);
366         return v;
367     default:
368         printf("Funcao desconhecida: %d\n", funcType);
369         return 0.0;
370     }
371 }
```

```
372
373 void doDef(symbol *name, symList *syms, ast *func)
374 {
375     if (name->syms)
376         freeSymList(name->syms);
377     if (name->func)
378         freeTree(name->func);
379     name->syms = syms;
380     name->func = func;
381 }
382
383 static double callUser(ufnCall *f)
384 {
385     symbol *fn = f->s;
386     symList *sl;
387     ast *args = f->l;
388     double *oldVal, *newVal;
389     double v;
390     int nargs;
391     int i;
392     if (!fn->func)
393     {
394         printf("Funcao nao definida: %s\n", fn->name);
395         return 0.0;
396     }
397     sl = fn->syms;
398     for (nargs = 0; sl; sl = sl->next)
399         nargs++;
400     oldVal = (double *)malloc(nargs * sizeof(double));
401     newVal = (double *)malloc(nargs * sizeof(double));
402     if (!oldVal || !newVal)
403     {
404         yyerror("sem espaco em %s", fn->name);
405         return 0.0;
406     }
407     for (i = 0; i < nargs; i++)
408     {
409         if (!args)
410         {
411             yyerror("poucos argumentos na chamada da funcao %s",
                 ↪  fn->name);
412             free(oldVal);
413             free(newVal);
414             return 0.0;
415         }
416
```

```
417         if (args->nodetype == 'L')
418         {
419             newVal[i] = eval(args->l);
420             args = args->r;
421         }
422         else
423         {
424             newVal[i] = eval(args);
425             args = NULL;
426         }
427     }
428 
429     sl = fn->syms;
430     for (i = 0; i < nargs; i++)
431     {
432         symbol *s = sl->sym;
433         oldVal[i] = s->value;
434         s->value = newVal[i];
435         sl = sl->next;
436     }
437     free(newVal);
438     v = eval(fn->func);
439     sl = fn->syms;
440     for (i = 0; i < nargs; i++)
441     {
442         symbol *s = sl->sym;
443         s->value = oldVal[i];
444         sl = sl->next;
445     }
446     free(oldVal);
447     return v;
448 }
449 
450 void yyerror(char *s, ...)
451 {
452     va_list ap;
453     va_start(ap, s);
454 
455     fprintf(stderr, "Erro na linha %d: ", yylineno);
456     vfprintf(stderr, s, ap);
457     fprintf(stderr, "\n");
458 }
459 
460 int main(int argc, char *argv[]) {
461 
462     if(argc == 1){
```

```
463         printf("> ");
464         yyparse();
465         return 0;
466     }
467 
468     if (argc < 3) {
469         fprintf(stderr, "Uso: %s <arquivo_entrada>
        ↪  <arquivo_saida>\n", argv[0]);
470         return 1;
471     }
472 
473     FILE *inputFile = fopen(argv[1], "r");
474     if (!inputFile) {
475         perror("Erro ao abrir arquivo de entrada");
476         return 1;
477     }
478 
479     FILE *outputFile = fopen(argv[2], "w");
480     if (!outputFile) {
481         perror("Erro ao abrir arquivo de saída");
482         fclose(inputFile);
483         return 1;
484     }
485 
486     if (freopen(argv[2], "w", stdout) == NULL) {
487         perror("Erro ao redirecionar stdout");
488         fclose(inputFile);
489         fclose(outputFile);
490         return 1;
491     }
492 
493     yyin = inputFile;
494     printf("> ");
495     yyparse();
496 
497     fclose(inputFile);
498     fclose(outputFile);
499 
500     return 0;
501 }
```

A estrutura de dados utilizada para a tabela de símbolos foi uma tabela *hash* com sondagem linear, que faz o *hashing* utilizando operações de multiplicação e `xor` bit a bit para gerar o valor da chave. Para o auxílio do *parser*, são utilizadas Árvores Sintáticas Abstratas - ASTs, que possibilitam a eficiente avaliação das expressões lidas pelo analisador.

A função `main()` foi alterada para possibilitar a leitura de arquivos, a fim de simular um compilador, que recebe um arquivo de entrada e gera outro como saída. A partir do valor de `argc`, o programa sabe qual rotina executar: o programa executa com interface pelo terminal se não houver argumentos adicionais; exibe um erro caso seja fornecido um arquivo de entrada e não um de saída ou vice-versa; ou executa simulando um compilador se for fornecido os dois arquivos necessários para leitura e escrita.

5.1. Funcionalidade FOR

Para a implementação do laço `for`, foi-se reaproveitado o mesmo TAD das outras produções de alteração de fluxo, a struct `flow`, na qual foi inserido um ponteiro adicional do tipo `ast` com o nome `"posCmd"`. Dos ponteiros presentes, `"el"` armazena o *init* do laço, `"cond"` contém a condição de execução, `"posCmd"` a expressão a ser executada após a execução de uma iteração e `"tl"` a lista de comandos do laço. A AST é feita com a chamada da função `newFlow()` dentro do trecho de código C no arquivo `parser.y` ao passo em que o *parser* verifica uma expressão casada com a produção correspondente a um *for*. Uma *flag* `'R'` (escolhido arbitrariamente na indisponibilidade da opção `'F'`) para a variável `"nodetype"`, que, será utilizada em duas rotinas seguintes: a de avaliação e liberação de memória.

Após a redução da árvore para as produções de `"calclist"`, a função `eval()` é chamada passando o endereço de memória do nó `stmt` como parâmetro, que irá recursivamente percorrer a árvore até alcançar as folhas, que serão terminais e poderão ser utilizados para a avaliação da expressão. A principal estrutura da rotina é uma função `switch()` que verifica qual é o valor armazenado em `"nodetype"` do nó no contexto atual da execução. No caso do `for()` implementado, será o valor decimal do literal `'R'`, o qual entrará em um fluxo onde é executado um `for()` da linguagem C, chamando a função `eval()` para cada ramo do nodo dentro de cada campo correspondente da estrutura de repetição. Ao final do laço, é retornado o valor gerado pela execução do mesmo.

5.2. Funcionalidade AND e OR

Para a construção de operadores lógicos, a struct `ast` já se faz suficiente, visto que é a mesma utilizada nas expressões de comparação e expressões numéricas. Aqui, o nodo recebe a *flag* com valor `'A'` ou `'O'`, de *and* e `or`, os ponteiros `"l"` e `"r"` recebem os lados esquerdo e direito da expressão, muito similar com as demais produções de `exp`.

Quando `eval()` é chamado para avaliar as expressões lógicas, o programa executa a estrutura `switch()` anteriormente descrita, executando o caso de acordo com o valor de `"nodetype"`, onde, será realizado uma comparação *and* ou *or* da linguagem C com os 2 lados da expressão e armazenando o resultado na variável de retorno.

6. Liberação de Memória

A última modificação necessária para o correto funcionamento foi a alteração da função `freeTree()`, que percorre a árvore sintática recursivamente, até atingir os

nodos folha, que são liberados com `free()` e então o nodo-pai também é liberado. As modificações feitas foram a adição da liberação do ponteiro `"posCmd"` para o caso da funcionalidade `for` e organização dos casos de `"nodetype"` para as três funcionalidades, garantindo a efetiva liberação da memória após a avaliação de uma AST.

7. Conclusão

Com o desenvolvimento desse trabalho, foi possível compreender melhor os elementos iniciais do processo de Compilação, bem como a construção desses mecanismos. Pode-se afirmar que analisadores léxicos e sintáticos são ferramentas poderosas para diversas aplicações computacionais.

References

CMU. EBNF: A Notation to Describe Syntax, Carnegie Mellon School of Computer Science. `https://www.cs.cmu.edu/~pattis/misc/ebnf2.pdf`. Acesso em: 27/11/2024.

Railroad. Railroad Diagram Generator. `https://rr.red-dove.com/ui`. Acesso em: 27/11/2024.