Trabalho 1 Compiladores

Alunos

Lucas Stadler Karau
UTFPR-PG
https://github.com/zskarau

Thales Janisch Santos
UTFPR-PG
https://github.com/Thales3006

1. Introdução

Este relatório tem como objetivo demonstrar a implementação da Calculadora Avançada em *Flex & Bison* vista nas aulas de Compiladores, ministradas pelo Prof. Dr. Gleifer Vaz Alves. Para demonstrá-la, serão apresentados e detalhados: linguagem utilizada, análise léxica e sintática, tabela de símbolos, descrição da análise sintática em BNF e um conjunto de testes significativo para ilustrar ofuncionamento da calculadora. Em maiores detalhes, serão vistas as implementações novas pedidas na especificação do trabalho: o laço de repetição FOR, e os operadores lógicos AND e OR. Para os diagramas de sintaxe e de transição, foram utilizadas as ferramentas sugeridas: *Railroad Diagram* e *Debuggex*.

2. Desenvolvimento

Nesse tópico será explicada a construção da calculadora (versão estendida) vista em aula. Sendo que, em cada parte do desenvolvimento serão destrinchadas as etapas que ocorrem para o funcionamento da linguagem da calculadora, a qual apresenta estruturas condicionais, laços de repetição, declaração de função informada pelo usuário, função para imprimir, operadores lógicos AND e OR adicionados, uma AST (Árvore Sintática Abstrata) para controlar o fluxo de código que será dado como entrada, além de controlar as funções definidas pelo usuário. A AST com as funções novas será detalhada abaixo, na Seção 2.1.

2.1. Linguagem

A calculadora possui uma AST que, ao ser avaliada a um valor, cai em um dos casos possíveis do programa, a seguir temos o trecho que demonstra os casos do operador AND e OR e a estrutura de repetição FOR.

```
1   case Or:
2       v = eval(a->l) || eval(a->r);
3       break;
4   case And:
5       v = eval(a->l) && eval(a->r);
6       break;
7   case For: {
8       struct flow *f = (struct flow *)a;
9       v = 0.0;
10       if (f->tl) {
11           for (eval(f->init); eval(f->cond) != 0;
eval(f->el),v = eval(f->tl));
12       }
13       break;
14     }
15   struct ast *newflow(evaluation nodetype, struct
ast *cond, struct ast *tl, struct ast *el, struct
ast *init) {
16     struct flow *a = malloc(sizeof(struct flow));
17     if (!a) {
18         yyerror("sem espaco");
19         exit(0);
20     }
21     a->nodetype = nodetype;
22     a->cond = cond;
23     a->tl = tl;
24     a->el = el;
25     a->init = init;
26     return (struct ast *)a;
27 }
```

Foi adicionado o parâmetro init, que vai se referir à condição inicial do FOR.

2.2. Análise Léxica

A análise léxica possui a função de, basicamente, receber o que virá na entrada do programa, seja por leitura de um arquivo de texto (especificação do trabalho) ou por leitura da entrada do usuário pelo terminal, identificar lexemas e relacioná-los com o token que eles (lexemas) se encaixam. As regras léxicas para identificação de tokens são dadas por meio de **expressões regulares**, utilizando de operadores como: + (uma ou mais instâncias), * (zero ou mais instâncias), ? (zero ou uma instância), [a-z] (letra ‘a’ até ‘z’ minúsculas), cadeias de caracteres [a-zA-Z] (letra ‘a’ até ‘z’ minúsculas **ou** letra ‘A’ até ‘Z’ maiúsculas). As regras léxicas seguem na tabela a seguir:

Tokens	Regras Léxicas
IF	“if”
THEN	“then”
ELSE	“else”
WHILE	“while”
DO	“do”
FOR	“for”
LET	“let”
FUNC	“sqrt” | “exp” | “log” | “print”
CMP	“<” | “<=” | “>” | “>=” | “==” | “<>”
AND	“&&”
OR	“||”
NAME	[a-zA-Z_][a-zA-Z0-9_]*
EXP	([Ee][+-]?[0-9]+)
NUMBER	[0-9]+“.”[0-9]*{EXP}? | “.”?[0-9]+{EXP}?
EOL	\n
-	qualquer *whitespace*
-	“+” | “-” | “*” | “/” | “=” | “;” | “;” | “(“ | “)”

Tabela 1: Regras Léxicas

Obs: a última regra retorna *yytext[0]* (função que aponta para o primeiro caracter da entrada), ou seja, simplesmente retorna o caracter inserido.

2.2.1. Diagramas de Transição

Agora, de acordo com as regras léxicas apresentadas na Tabela 1, serão apresentados os diagramas de transição para as regras NUMBER (juntamente com EXP) e NAME, as quais possuem uma complexidade maior na definição da expressão regular. Nessa etapa, foi utilizada a ferramenta *Debuggex* (DEBUGGEX, 2013).

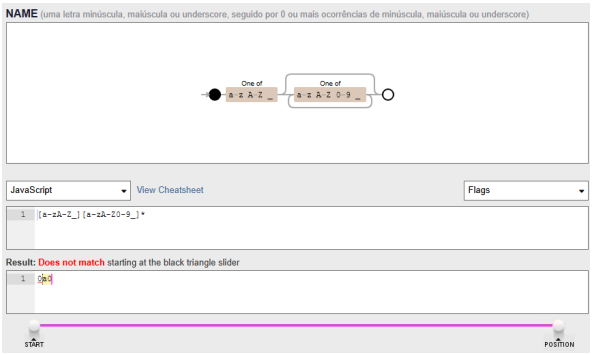

Figura 1: Exemplo NAME

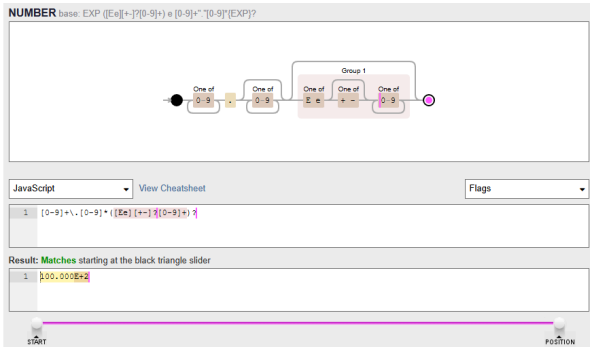

Figura 2: Exemplo NUMBER

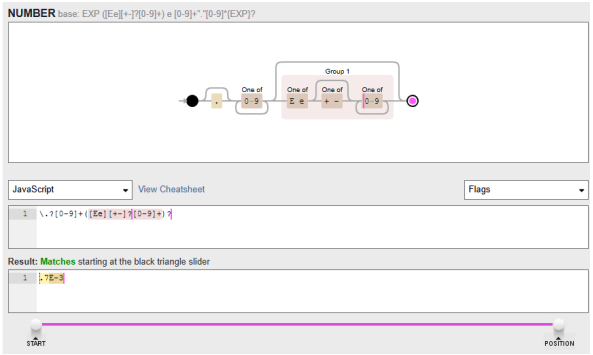

Figura 3: Exemplo NUMBER

Pode-se observar que, na Figura 1 e na Figura 2 estão sendo testadas as expressões regulares para definição de nomes e números da linguagem, além disso, temos a expressão regular EXP inclusa diretamente na expressão de números. A Figura 2 representa uma das possibilidades de EXP, e a Figura 3, a outra possibilidade (as duas separadas por **ou** na Tabela 1).

2.2.2. Flex

A implementação do analisador léxico com suas expressões regulares foi feito utilizando da linguagem *Flex*. Em *Flex*, o código é estruturado em **3 seções**:

1. Declarações
2. Regras de Tradução
3. Funções Auxiliares

2.2.2.1. Declarações

A seguir um trecho de código do *Flex* com as declarações usadas.

```
1 %option noyywrap nodefault yylineno
2
3 %{
4     #include "ast.h"
5     #include "parser.tab.h"
6 %}
```

Na primeira linha do trecho acima, observa-se um conjunto de opções para funcionamento do *Flex* (em *Windows*) (PAXSON; ESTES; MILLAWAY, 2015). A opção *noyywrap* deve ser inserida ao estar em ambiente *Windows*. A função *yywrap()* está relacionada

com a a indicação de um *end of file* recebido pelo *YY_INPUT*. Quando a função *yywrap()* retorna falso (zero), assume-se que a função seguiu adiante e o escaneamento continua. Caso contrário, assume-se que a função acabou e o *scanner* é finalizado. Ao usar *noyywrap*, o *scanner* assume que a função *yywrap()* retorna sempre 1 (verdadeiro). A opção *nodefault* suprime a regra *default*. Isso é útil para encontrar erros nas regras léxicas, pois não existe caso padrão, e ao escanear uma entrada que não corresponde a nenhuma regra, o programa tem como saída uma mensagem de erro (pois não há caso padrão). A opção *yylineno* basicamente direciona o *Flex* para criar um scanner que mantém o número da linha que está sendo lida na variável global *yylineno*. Adiante, tem-se a inclusão das bibliotecas "ast.h" e "parser.tab.h" (que serão descritas na seção da Análise Sintática). Por fim, há a declaração da expressão regular do expoente float, que será utilizada pela definição de número na seção de Regras de Tradução.

2.2.2.2. Regras de Tradução

Nessa seção, apresenta-se as regras de tradução, as quais vão identificar os lexemas e seus respectivos tokens (e se necessário, valor de atributo). Basicamente, a forma das regras é:

Padrão {Ação}

A seguir a o trecho do *Flex* que define como tratar cada lexema que o analisador léxico encontra com o padrão e a ação devida a cada um.

```
1 EXP ([Ee][+-]?[0-9]+)
2 %%
3 "+"|"-"|"*"|"/"|"="|","|";"|"("|")" {
4   return yytext[0];
5 }
6
7 "&&"    { return AND; }
8 "||"    { return OR; }
9
10 ">"  { yylval.fn = Greater; return CMP; }
11 "<"  { yylval.fn = Less; return CMP; }
12 "<>" { yylval.fn = Not_equal; return CMP; }
13 "==" { yylval.fn = Equal; return CMP; }
14 ">=" { yylval.fn = Greater_equal; return CMP; }
15 "<=" { yylval.fn = Less_equal; return CMP; }
16
17 "if"    { return IF; }
18 "then"  { return THEN; }
19 "else"  { return ELSE; }
20 "while" { return WHILE; }
21 "for"   { return FOR; }
22 "do"    { return DO; }
23 "let"   { return LET; }
24
25 "sqrt"  { yylval.fn = B_sqrt; return FUNC; }
26 "exp"   { yylval.fn = B_exp; return FUNC; }
27 "log"   { yylval.fn = B_log; return FUNC; }
28 "print" { yylval.fn = B_print; return FUNC; }
29
30
31 [a-zA-Z_][a-zA-Z0-9_]* {
32     yylval.s = lookup(yytext);
33     return NAME;
34 }
35
36 [0-9]+"."[0-9]*{EXP}? {
37     yylval.d = atof(yytext);
38     return NUMBER;
39 }
40
41 "."?[0-9]+{EXP}? {
42     yylval.d = atof(yytext);
43     return NUMBER;
44 }
45
46 [ \t]+ ;
47
48 \\\n             { printf("> "); }
49
50 \n               { return EOL; }
51
52
53 .  { yyerror("Caractere desconhecido: %c\n",
*yytext); }
54 %%
```

Cada padrão é uma expressão regular, que pode utilizar das declarações anteriores (como NUMBER utilizando EXP). As ações são fragmentos de código em C. Os operadores de comparação (CMP) possuem valores de atributos diferentes, que são associados à variável *yylval* (a qual guarda qual é o tipo do token CMP encontrado). De certa forma, os tokens FUNC funcionam similarmente ao CMP. Na linha 31, é definida a expressão para NAME, e nas linhas 36 e 41 para NUMBER. Em ambas as definições de NUMBER, a função *yylval* recebe a entrada lida por *yytext* (string) transformada para número flutuante com a função *atof*.

Na linha 46, observa-se a expressão: "Espaço ou tabulação 1 ou mais vezes", a qual não tem nenhuma função de retorno ou ação. Isso significa que, ao escanear espaço(s) ou tabulação(ões), o programa ignora-os e segue adiante. A linha 48 representa: "ao ler \n (apenas \n)", imprima "> ". Isso é executado quando não se escreve nada na entrada e se aperta *Enter* (\n). Agora, quando o analisador léxico lê uma entrada qualquer (após o usuário pressionar *Enter*), ao chegar no final da entrada, a qual possui um \n por conta do *Enter*, é retornado o valor de EOL (*end of line*). Na linha 53 há caracter " . ", o qual em *flex* significa **qualquer coisa**. Ao ler uma entrada que possua um caractere (ou cadeia de caracteres) que não corresponde a nenhum dos tokens especificados anteriormente, o caractere cai na regra do " . ", a qual imprime **Caracter nao reconhecido: %c**, onde %c significa o caracter que foi inserido na entrada.

A funcionalidade nova FOR é similar a outras estruturas condicionais e de repetição. Ao escanear a entrada literal (lexema) 'for', o lexer retorna o token FOR. O mesmo ocorre com o AND e o OR.

2.2.2.3. Funções Auxiliares

Em alguns casos, a função *main* é inserida nesse campo, porém na linguagem da calculadora as funções em C estão inclusas na AST.

2.3. Tabela de Símbolos

A tabela de símbolos no arquivo *ast.c* é usada para armazenar variáveis e funções identificadas por nome durante a execução do interpretador. Ela é implementada como um vetor fixo chamado *symtab*, com tamanho definido por *NHASH*, e cada entrada é uma estrutura *symbol* que armazena o nome, valor e, quando aplicável, os parâmetros e o corpo de uma função.

A função lookup é responsável por buscar ou inserir um símbolo na tabela. Para isso, utiliza uma função de hash (*symhash*) que calcula um índice a partir do nome do símbolo. Em caso de colisões, a função avança linearmente até encontrar uma posição livre ou o símbolo desejado. Se a tabela estiver cheia, o programa retorna um erro.

Durante a execução, variáveis têm seus valores armazenados diretamente no campo *value* da estrutura, enquanto funções definidas pelo usuário guardam a lista de parâmetros e a árvore de execução (AST) correspondente. A tabela permite que esses símbolos sejam reutilizados e atualizados conforme o interpretador avalia as expressões.

2.4. Análise Sintática

Agora, na análise sintática, são apresentadas as regras sintáticas utilizando a gramática na forma de *Backus-Naus*, conhecida como BNF. As regras de análise sintática com a gramática na forma BNF são feitas da seguinte forma: primeiro, declara-se alguns dos terminais (como CMP e NAME). Segundo, tem-se a seção de expressões, e.g "<stmt>", que possui 5 possibilidades de escolha. Além disso, observa-se que as variáveis são delimitadas por "<>", e os terminais são escritos sem delimitação.

```
1  CMP  ::=  '>' | '<' | '>=' | '<=' | '<>' | '=='
2  EXP  ::=  ([Ee][+-]?[0-9]+)
3  NUMBER ::=  [0-9]+'.'[0-9]*(EXP)? | '.'?
[0-9]+(EXP)?
4  NAME  ::=  [a-zA-Z_][a-zA-Z0-9_]*
5  FUNC  ::=  'sqrt' | 'exp' | 'log' | 'print'
6  WS  ::=  [ ]+
7  EOL  ::=  '\n'
8  IF  ::=  'if'
9  THEN  ::=  'then'
10 ELSE  ::=  'else'
11 WHILE  ::=  'while'
12 DO  ::=  'do'
13 LET  ::=  'let'
14 FOR  ::=  'for'
15
16 <stmt>  ::=  IF WS <exp> WS THEN <list>
17     | IF <exp> THEN <list> ELSE <list>
18     | WHILE <exp> DO <list>
19     | FOR ( <init> ; <cond> ; <inc> )
20     | <exp>
21
22 <init>  ::=  NAME  =  <exp>
23 <cond>  ::=  <exp>  CMP  <exp>
24 <inc>  ::=  NAME  =  <exp>
25
26 <list>  ::= ''
27     | <stmt> ; <list>
28
29 <exp>  ::=  <exp> CMP <exp>
30     | <exp> + <exp>
31     | <exp> - <exp>
32     | <exp> * <exp>
33     | <exp> / <exp>
34     | ( <exp> )
35     | NUMBER
36     | NAME
37     | NAME = <exp>
38     | FUNC ( <explist> )
39     | NAME ( <explist> )
40
41 <explist> ::= <exp>
42     | <exp> , <explist>
43
44 <symlist> ::=  NAME
45     | NAME , <symlist>
46
47 <calclist>  ::=  ''
48     | <calclist> <stmt> EOL
49     | <calclist> LET NAME ( <symlist> ) = <list>
EOL
50     | <calclist> error EOL
```

2.4.1. Diagramas de Sintaxe

Os diagramas de sintaxe são apresentados para representar visualmente a gramática. Para criação desses diagramas, foi utilizada a ferramenta *Railroad Diagram*, a qual é capaz de representar os diagramas sintáticos por meio da gramática na forma BNF (*Backus-Naus Form*). Como a gramática descrita na seção anterior já se encontra no formato BNF, são necessários somente alguns ajustes para gerar os diagramas na ferramenta *Railroad* (RADEMACHER, 2015), que estão descritos na Figura 4.

2.4.2. Bison

A implementação em Bison funciona em três partes, similiar ao *Flex*. As três são:

1. Declarações
2. Regras de Tradução
3. Rotinas de Suporte em C

Na parte de declarações existem duas seções, ambas opcionais. A primeira se refere a declarações comuns em C, delimitadas por "%(" **e** "%)". A segunda está relacionada com a declaração de tokens que poderão ser utilizados na próxima seção que define as regras da gramática.

Após as declarações, são inseridas as regras de tradução após o primeiro par de "%%". Cada regra possui uma produção da gramática e uma ação semântica correspondente. A cabeça da regra é definida como $$, e as partes do corpo da regra são definidas como $$_i$$. Em geral, a ação semântica padrão é { $$ = $1; } ($1 sendo o primeiro elemento do corpo).

Funcionalidades novas AND e OR possuem:

- Associatividade à esquerda;
- Prioridade maior que a atribuição e os operadores comparativos;
- Prioridade menor que os operadores aritméticos;
- Estrutura "**exp: exp AND exp**" e "**exp: exp OR exp**";
- Quando ocorrer o caso de AND e OR, a cabeça da regra "exp" tem a função *newast* atribuída com os parâmetros (And, $1, $3) ou (Or, $1, $3).

Funcionalidade nova FOR possui:

- Estrutura "**FOR '(' init ';' cond ';' inc ')' list { $$ = newflow(For, $5, $9, $7, $3); }**"
- $$ se refere à cabeça da regra (stmt);
- É enviada à função *newflow* os parâmetros (For, $5, $9, $7, $3)
- $5 se refere à "cond";
- $9 se refere à "list";
- $7 se refere à "inc";
- $3 se refere à "init".

Por fim, temos a parte de rotinas de suporte em C. Na gramática da calculadora, não há rotinas, acabando nas regras de tradução.

```
1   %{
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "ast.h"
5 %}
6
7
8 %union {
9     struct ast *a;
10     long double d;
11     struct symbol *s;    /* qual símbolo? */
12     struct symlist *sl;
13     int fn;              /* qual função? */
```

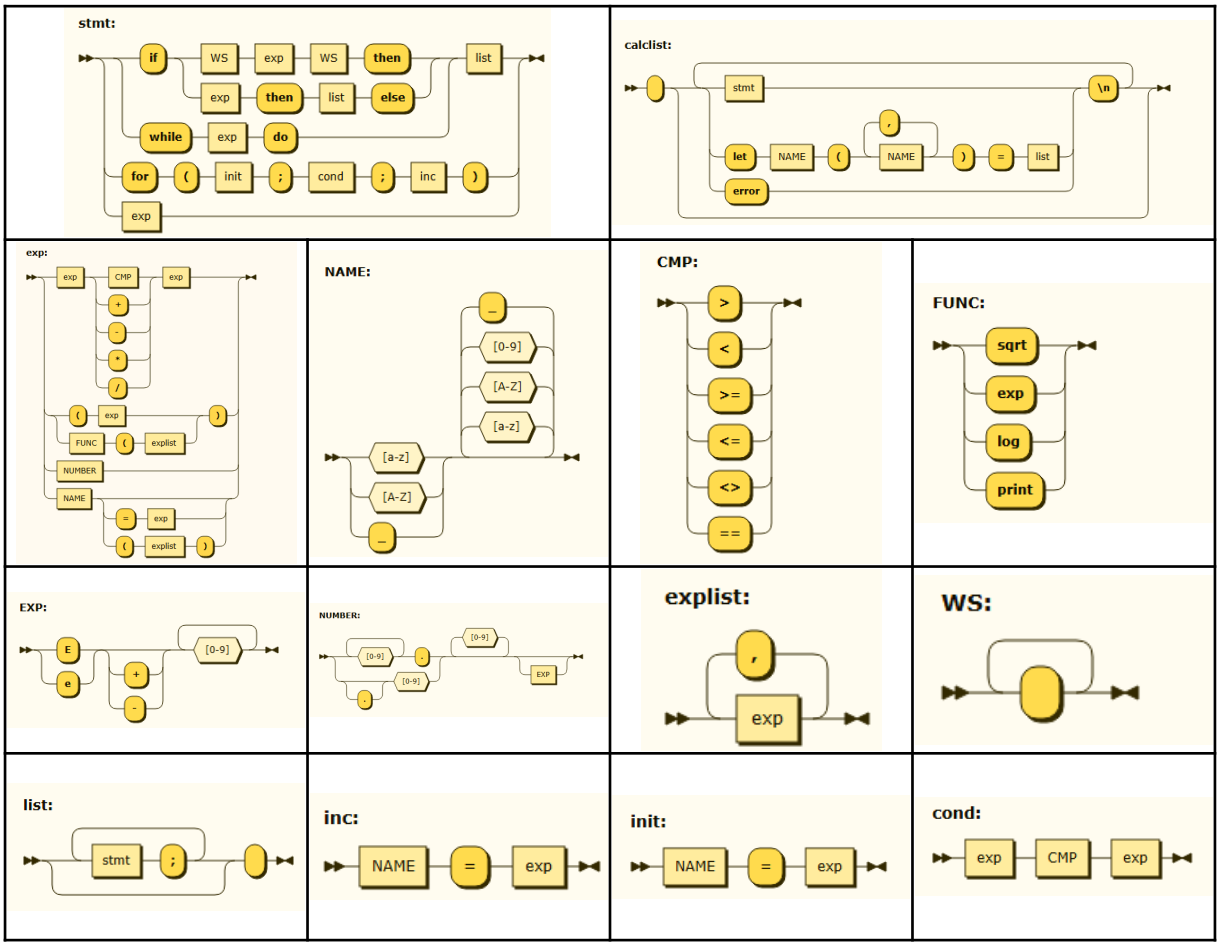

Figura 4: Diagramas de Sintaxe

```
14 }
15
16 /* declaração de tokens */
17 %token <d> NUMBER
18 %token <s> NAME
19 %token <fn> FUNC
20 %token EOL
21
22 %token IF THEN ELSE WHILE DO LET FOR AND OR
23
24 %nonassoc <fn> CMP
25 %right '='
26 %left AND OR
27 %left '+' '-'
28 %left '*' '/'
29
30 %type <a> exp stmt list explist init cond inc
31 %type <sl> symlist
32
33 %start calclist
34 %%
35
36 stmt:
37       IF exp THEN list                { $$ =
newflow(If_else, $2, $4, NULL, NULL); }
38     | IF exp THEN list ELSE list      { $$ =
newflow(If_else, $2, $4, $6, NULL); }
39     | WHILE exp DO list               { $$ =
newflow(While, $2, $4, NULL, NULL); }
40     | FOR '(' init  ';'cond ';' inc ')' list { $$
= newflow(For, $5, $9, $7, $3); }
41     | exp
42 ;
43
44 init: NAME '=' exp      { $$ = newasgn($1,
$3); };
45 cond:  exp CMP exp      { $$ = newcmp($2, $1,
$3); };
46 inc: NAME '=' exp       { $$ = newasgn($1,
$3); };
47
48 list:
49       /* vazio! */                  { $$ =
NULL; }
50     | stmt ';' list {
51           if ($3 == NULL)
52               $$ = $1;
53           else
54               $$ = newast(Statement, $1, $3);
55       }
56 ;
57
58 exp:
59       exp CMP exp        { $$ = newcmp($2, $1,
$3); }
60     | exp '+' exp        { $$ = newast(Addition,
$1, $3); }
61     | exp '-' exp        { $$ =
newast(Subtraction, $1, $3); }
62     | exp '*' exp        { $$ =
newast(Multiplication, $1, $3); }
63     | exp '/' exp        { $$ = newast(Division,
$1, $3); }
64     | exp AND exp       { $$ = newast(And, $1,
$3); }
65     | exp OR exp       { $$ = newast(Or, $1,
$3); }
66     | '(' exp ')'        { $$ = $2; }
67     | NUMBER             { $$ = newnum($1); }
68     | NAME               { $$ = newref($1); }
69     | NAME '=' exp       { $$ = newasgn($1,
$3); }
70     | FUNC '(' explist ')' { $$ = newfunc($1,
$3); }
71     | NAME '(' explist ')' { $$ = newcall($1,
$3); }
72 ;
73
74 explist:
75       exp                { $$ = $1; }
76     | exp ',' explist    { $$ = newast(Statement,
$1, $3); }
77 ;
78
79 symlist:
80       NAME               { $$ = newsymlist($1,
NULL); }
81     | NAME ',' symlist   { $$ = newsymlist($1,
$3); }
82 ;
83
84 calclist:
85       /* vazio! */
86     | calclist stmt EOL {
87           printf("= %g\n", (double)eval($2));
88           treefree($2);
89       }
90     | calclist LET NAME '(' symlist ')' '=' list
EOL {
91           dodef($3, $5, $8);
92           printf("Defined %s\n", $3->name);
93       }
94     | calclist error EOL { yyerrok;
printf(">\n"); }
95 ;
96
```

3. Testes

- **Teste 1**

```
1 2+(4-1)\*2
2 3/2 + 1/3 + .7E2
3 3.0e2 - 500
4
```

- **Teste 2**

```
1 let succ(x) = x+1;
2 succ(1)
3 let max(x,y) = if x >= y then x; else y;;
4 max(5,11.52)
5
```

- **Teste 3**

```
1 let func(x, y) = while(x < y) do x = x + y; y =
y + 10; print(x);;
2 func(1, 100)
3 while(x < 100) do print(x); x = x + 1;
4
```

- **Teste 4**

```
1 x = 1
2 y = 10
```

```
3 for(i = 0; i < y; i = i + 1) x = i * i;
print(x);
4
```

- **Teste 5**

```
1 x = 10
2 print(x)
3
```

- **Teste 6**

```
1 x = 1
2 y = 0
3 if(x && y) then x = x + 1;
4 if(x || y) then x = x + 10;
5
```

- **Teste 7**

```
1 let max(x, y) = if(x > y) then print(x); else
print(y);;
2 max(1, 10)
3 max(1, 0)
4 max(1)
5 print()
6 let min(x, y) = if(x < y) then print(x); else
print(y);
7
```

- **Teste 8**

```
1 if x == 0 || y == 0 then 1; else x;
2 let fac(x) = if (x == 0) || (x == 1) then 1;
else fac(x-1) * x; ;
3 fac(0)
4 fac(1)
5 fac(2)
6 fac(3)
7 fac(4)
8 let fib(x) = if (x == 0) then 0; else if x == 1
then 1; else fib(x-2)+fib(x-1); ; ;
9 fib(0)
10 fib(1)
11 fib(2)
12 fib(3)
13 fib(4)
14 fib(5)
15 fib(6)
16 fib(7)
17 fib(8)
18 fib(9)
19
```

- **Teste 9**

```
1 let fib(x) = if (x == 0) then 0; else if x == 1
then 1; else fib(x-2)+fib(x-1); ;
2 x = 1;
3 while x < 10 x = x + 1; print(x);
4
5
6 ç
7 ¬
8 🐳
9 £
10 ³
11 ²
12 ¹
```

4. Conclusão

Por fim, a linguagem da Calculadora Avançada permite o usuário testar estruturas condicionais, operadores comparitivos, estruturas de repetição, operadores aritméticos, definir funções, usar as funções como logaritmo, raiz quadrada, entre outros. Ela utiliza uma AST, arquivos de texto para realização de testes e apresenta diagramas léxicos e sintáticos com a gramática na forma BNF. Nesse trabalho foi utilizado a LLM (Large Language Model) *ChatGPT* (OPENAI, 2025) para transformar o PDF em texto para utilizar a linguagem em código e para ajudar a identificar erros nos testes, além de explicar algumas partes do código.

Referências

DEBUGGEX. **Debuggex: Testador visual de expressões regulares**. Disponível em: <https://www.debuggex.com/>. Acesso em: 21 mai. 2025.

OPENAI. **ChatGPT**. , 2025.

PAXSON, V.; ESTES, W.; MILLAWAY, J. **Lexical Analysis with Flex**. Berkeley, CA, nov. 2015.

RADEMACHER, G. **RR: Railroad Diagram Generator**. Disponível em: <https://bottlecaps.de/rr/ui>. Acesso em: 21 mai. 2025.