UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
EURISTENEDE VANUEL FRANCISCO DAS NEVES SANTOS
ROBERTO ROCHA

COMPILADORES

Implementação de uma calculadora avançada com Flex e Bison

Ponta Grossa

2021

Euristenede Vanuel Francisco das Neves Santos

Roberto Rocha

COMPILADORES

Implementação de uma calculadora avançada com Flex e Bison

Trabalho apresentado na disciplina de Compiladores, como parte dos requisitos necessários para obtenção do título de Bacharel em Ciência da Computação. Orientado pelo Prof. Me. Gleifer Vaz Alves.

Lista de Figuras

1. Uma breve apresentação sobre as linguagens de programação e compiladores

O estudo das linguagens de programação está entre uma das principais áreas da Ciência da Computação. Pois sendo uma ferramenta presente atualmente em diversas atividades, no computador são desenvolvidas linguagens de programação com metas diferentes. Ou seja, a linguagem de programação veio para facilitar a vida das pessoas com a tecnologia.

O compilador é o programa responsável para traduzir estas linguagens de programação. Basicamente, um compilador pode ser visto como um programa que recebe um código fonte de entrada e traduz para um programa equivalente em outra linguagem (LOUREN, 2004). Ou seja, é recebido um programa que processa instruções escritas em linguagem de programação específicas e traduz em linguagem de máquina ou código para o processador poder ler. Para fazer esta tradução são usadas várias etapas.

Este trabalho faz o uso da linguagem de programação C, para escrever a tabela de símbolos e as funções que implementa a estrutura AST. Também foi utilizado a ferramenta Flex, que tem por objetivo escrever o analisador léxico, e a ferramenta Bison que escreve o analisador sintático.

2. Apresentação das linguagens utilizada no trabalho

- **Linguagem C**

O C é uma linguagem de programação desenvolvida pelo cientista da computação Dennis Ritchie, em 1972, tendo como base duas outras linguagens de programação: a BCLP e a Algol 68, (NOLETO, 2020).

Noleto, 2020 descreve em um artigo que a linguagem de programação C ainda é uma das mais populares, devido às diversas vantagens que apresenta, tal como: portabilidade, geração de código eficiente, simplicidade, confiabilidade, facilidade de uso

e regularidade. Sendo assim torna quase que obrigatório ter essa linguagem no currículo de uma pessoa que trabalho com desenvolvimento de software.

- **Ferramenta Flex**

No compilador se utiliza de regras com formatação de elementos de caracteres e frases válidas de uma linguagem que são expressadas na gramatica da linguagem. Para tal aplicação é feito o reconhecimento das sentenças por grupo de sequencias de caracteres em palavras que deverão ser agrupados e identificados como palavras reservadas da linguagem, constantes e identificadores, sendo a análise léxica. Onde os elementos reconhecidos nessa primeira fase são denominados de itens léxicos ou *tokens*. Estes tokens é passado para a seguinte fase, análise sintática.

E para este trabalho o gerador léxico é a ferramenta Flex para a análise léxica. Onde o papel é obter um conjunto de expressões regulares e produzir uma rotina em C que irá executar a análise léxica identificando os *tokens.* Ou seja, as entradas são feitas por expressões regulares e comandos em linguagem de programação, geralmente em C, onde o analisador é gerado quando uma expressão é reconhecida, sendo executados através dos comandos associados a ela.

Para poder rodar ele é necessário ter o Flex instalado assim como um compilador C qualquer. Primeiro é executado o comando que irá executar um ficheiro *lex.yy.c* que deve ser executado em linha de comando para gerar o analisador léxico. Sempre por um terminal de linha de comandos.

- **Ferramenta Bison**

Como já informado anteriormente, em compiladores a segunda etapa é a analise sintática. Onde a função é validar a estrutura gramatical do programa, identificando se está de acordo com as regras da linguagem da gramatica fornecida. Tais regras, são construções da linguagem descrita através de regras que produzem ou geram elementos incluindo símbolos terminais e não terminais. Neste trabalho é apresentado utilizando a notação BNF as produções. Conforme a varredura da cadeia de *tokens* fornecida pela analise léxica da primeira fase e a aplicação das regras de produção da gramatica, são obtidos as sequencias de símbolos que formam a estrutura sintática, como os comandos e as expressões.

Para fazer a análise sintática neste trabalho é usado a fragmenta bison, onde o papel dela é resumir uma gramatica e produzir uma rotina C que irá executar a análise sintática. Sendo uma ferramenta complementar da Flex, descreve uma gramatica de livre de contexto, e ela possui elementos que são: léxicos ou *tokens*, símbolos terminais; elementos sintáticos, símbolos não terminais; regras de produção, definidos pelos símbolos não terminais em termos de sequência de terminais e não terminais; e uma regra *start*, que reduz todos os elementos da gramatica para uma regra.

Para poder rodar, a partir de um terminal de comando do arquivo do flex e depois do bison, *arquivo.y*, para gerar um *arquivo.h* e *arquivo.tab.c*. Depois deve se abrir tal arquivo para gerar o executável.

3. Apresentação das regras de análise léxica e diagramas de transição

Abaixo está ilustrado as figuras com as regras léxicas e seus respectivos diagramas de transição.

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

Figura 1 - tokem EXP

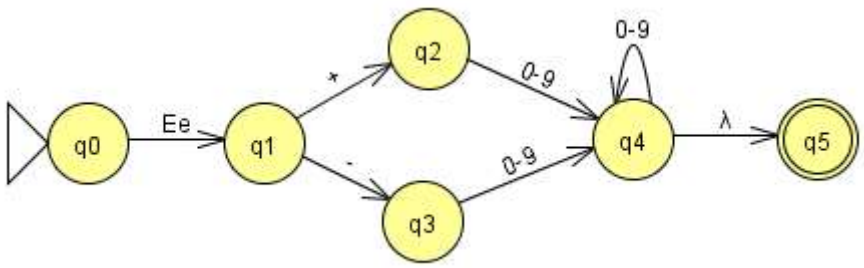


Figura 2 Diagrama de Transição tokem EXP

```
%%
"+" |    /* operadores de caracter unico */
"-" |
"*" |
"/" |
"=" |
"|" |
"," |
";" |
"(" |
")" { return yytext[0]; }
```

Figura 3 - Operadores e caractere único

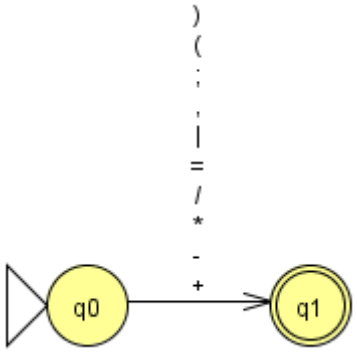


Figura 4 - Diagrama de transição Operadores e caractere único

```
">"  { 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; }
```

Figura 5 - tokens de comparação CMP

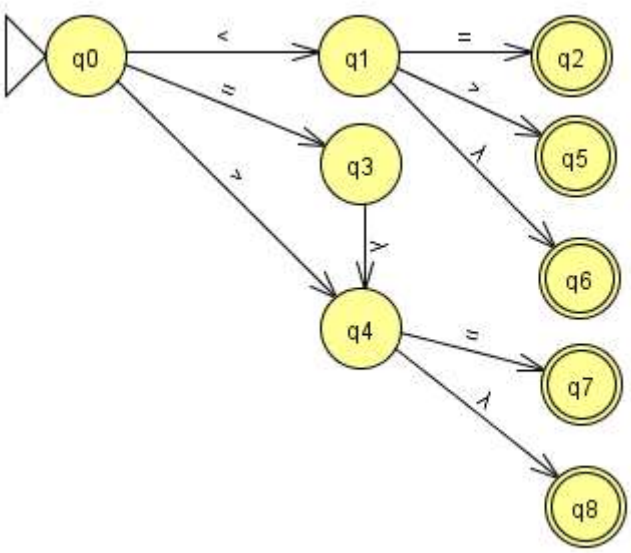


Figura 6 - Diagrama de transição tokens de comparação CMP

```
"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; }
```

Figura 7 - Funções pré-definidas

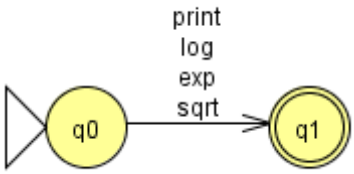


Figura 8 - Diagrama de transição Funções pré-definidas

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

"//".*
[ /t] /* ignora espaço em branco */

\\\n { printf("c> "); } /* ignora continuação de linha */

\n { return EOL; }

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

Figura 9 - tokens para nomes, números e delimitadores

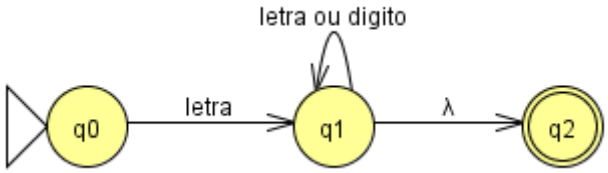


Figura 10 - Diagrama de transição para tokens de nomes

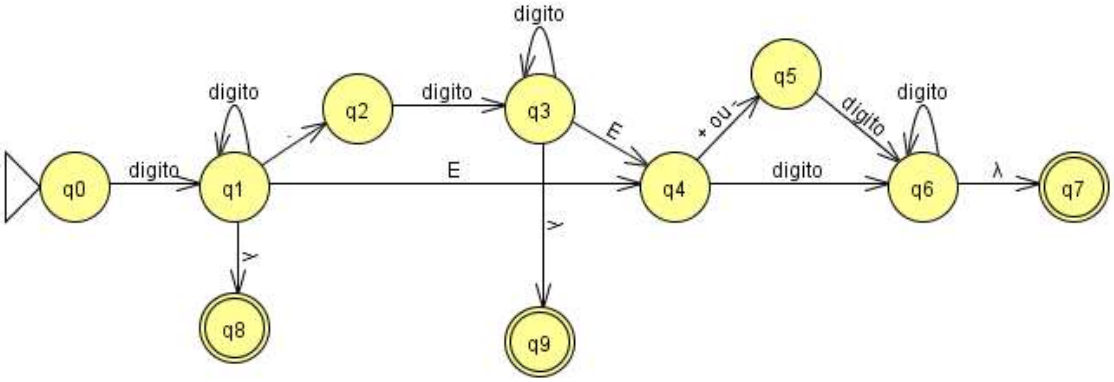


Figura 11 - Diagrama de transição para tokens de números

Figura 12 - Diagrama de transição para tokens de delimitadores

4. Descrição do analisador léxico conforme implementação no Flex

Primeiro foi incluído os arquivos de cabeçalhos e a tabela de símbolo que fica no arquivo calculadora.h, segue na figura abaixo:

```
%{
#include "calculadora.h"
#include "calculadora.tab.h"
%}
```

Figura 13 - inclusão dos cabeçalhos

Em seguida foi definido o tokem EXP, um expoente flutuante

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

Figura 14 - tokem EXP

Operadores de adição, subtração, multiplicação e outros mostrado na figura abaixo, são operadores de caractere único, então o analisador léxico verifica a existência de cada um e retorna o mesmo.

```
%%
"+" |   /* operadores de caracter unico */
"-" |
"*" |
"/" |
"=" |
"|" |
"," |
";" |
"(" |
")" { return yytext[0]; }
```

Figura 15 - tokens de operadores de caractere único

Em seguida foi definido os operadores responsáveis por realizar comparações, então o analisador léxico verifica cada um e retorna um tokem CMP.

```
">"  { 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; }
```

Figura 16 - tokens de comparação CMP

A próxima etapa foi definir algumas funções pré-definidas.

```
"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; }
```

Figura 17 - tokens de funções pré-definidas

Por último foi definido os tokens responsável por nomes, números, espaços em branco, continuação de linhas, fim de linhas e o desconhecimento de qualquer outro caractere não definido no analisador léxico.

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

"//".*
[ /t] /* ignora espaço em branco */

\\\n { printf("c> "); } /* ignora continuação de linha */

\n { return EOL; }

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

Figura 18 - tokens para nomes, números e delimitadores

5. Descrição da Tabela de Símbolos (TS)

A tabela de símbolos é um arquivo com extensão .h, na qual contém o cabeçalho e estrutura de cada função que é implementada no arquivo .c. abaixo segue uma breve descrição sobre cada símbolo, estrutura e cabeçalho de função. Na figura 19 é apresentado uma tabela de símbolos, onde cada um pode ser uma variável ou uma função definida pelo usuário. A variável *value* é uma variável que guarda o valor do símbolo, o ponteiro *func* aponta para uma estrutura AST (Arvore Sintática Abstrata) e o ponteiro *syms* aponta para uma lista de argumentos fictício.

```
/* tabela de simbolos */
struct symbol { /* um nome de variável */
 char *name;
 double value;
 struct ast *func; /* stmt para função */
 struct symlist *syms; /* lista de argumentos */
};
```

Figura 19 - Tabela de símbolos

Na figura 20 é apresentado a estrutura que define o tamanho do símbolo, como uma lista de símbolos de 9997 posições.

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

struct symbol *lookup(char*);
```

Figura 20 - Símbolo de tamanho fixo

Já na figura 21 é apresentado duas funções que tem por objetivo criar e deletar uma lista de símbolos.

```
struct symlist *newsymlist(struct symbol *sym, struct symlist *next);
void symlistfree(struct symlist *sl);
```

Figura 21 - Criar e deletar lista de símbolos

Na figura 22 apresenta a definição de símbolos para cada tipo de nó da AST.

```
/* tipos de nós
 * + - * / |
 * 0-7 operadores de comparação, 04 igual, 02 menor que, 01 maior que
 * L expressão ou lista de comandos
 * I comando IF
 * W comando WHILE
 * N simbolo de referência
 * = atribuiao
 * S lista de simbolos
 * F chamada de função pre-definida
 * C chamada de função def. p/ usuario
 * O comando FOR
 */
```

Figura 22 - Definição de símbolos

A figura 23 apresenta 4 funções pré-definidas

```
enum bifs {
 B_sqrt = 1,
 B_exp,
 B_log,
 B_print
};
```

Figura 23 - Funções pré-definidas

A figura 24 apresenta a estrutura da AST

```
struct ast {
  int nodetype;
  struct ast *l;
  struct ast *r;
};
```

Figura 24 - Definição da AST

A figura 25 apresenta uma estrutura com chamada para as funções pré-definidas

```
struct fncall { /* funções pré-definidas */
  int nodetype; /* tipo F */
  struct ast *l;
  enum bifs functype;
};
```

Figura 25 - Funções pré-definidas

A figura 26 apresenta uma estrutura com chamadas para funções definidas pelo usuário

```
struct ufncall { /* funções usuário */
  int nodetype; /* tipo C */
  struct ast *l; /* lista de arguments */
  struct symbol *s;
};
```

Figura 26 - Funções definidas pelo usuário

A figura 27 apresenta duas estruturas para controlar as expressões *if then else*, *while do* e o *for*.

```
struct flow {
  int nodetype; /* tipo I ou W */
  struct ast *cond; /* condição */
  struct ast *tl; /* ramo "then" ou lista "do" */
  struct ast *el; /* ramo opcional "else" */
};

struct foor {
  int nodetype; /* tipo O */
  struct ast *init; /* inicialização */
  struct ast *cond; /* condição */
  struct ast *inc; /* incremento */
  struct ast *tl; /* ramo "then" ou lista "do" */
};
```

Figura 27 - Controle de expressões

A figura 28 apresenta 3 estruturas de constantes e de referências para tabela de simbolos

```
struct numval {
  int nodetype; /* tipo K */
  double number;
};

struct symref {
  int nodetype; /* tipo N */
  struct symbol *s;
};

struct symasgn {
  int nodetype; /* tipo = */
  struct symbol *s;
  struct ast *v; /* valor a ser atribuido */
};
```

Figura 28 - Constantes e Referências

A figura 29 apresenta a construção da AST

```
/* construção de uma AST */
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(int functype, struct ast *l);
struct ast *newcall(struct symbol *s, struct ast *l);
struct ast *newref(struct symbol *s);
struct ast *newasgn(struct symbol *s, struct ast *v);
struct ast *newnum(double d);
struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *tr);
struct ast *newfor(int nodetype, struct ast *init, struct ast *cond, struct ast *inc, struct ast *tl);
```

Figura 29 - Construção da AST

E por último a figura 30 apresenta a definição de uma função, avaliação da AST e a liberação da AST

```
/* definição de uma função */
void dodef(struct symbol *name, struct symlist *syms, struct ast *stmts);

/* avaliação de uma AST */
double eval(struct ast *);

/* deletar e liberar uma AST */
void treefree(struct ast *);
```

Figura 30 - Definição de função

6. Definição das regras de análise sintática e diagramas de sintaxe

Abaixo segue uma lista de figuras apresentando a gramática no formado BNF que foi utilizada para gerar os diagramas de sintáxe na ferramenta Railroad Diagram (http://bottlecaps.de/rr/ui).

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

stmt::= IF exp THEN list
    | IF exp THEN list ELSE list
    | WHILE exp DO list
    | FOR '(' exp ';' exp ';' exp ')' list
    | exp

list::=
    | stmt ';' list

exp::= exp CMP exp
   | exp '+' exp
   | exp '-' exp
   | exp '*' exp
   | exp '/' exp
   | '|' exp
   | '(' exp ')'
   | '-' exp UMINUS
   | NUMBER
   | NAME
   | NAME '=' exp
   | FUNC '(' explist ')'
   | NAME '(' explist ')'

explist::= exp
       | exp ',' explist

symlist::= NAME
       | NAME ',' symlist
```

Figura 31 - Gramática utilizada para gerar os diagramas de sintaxe

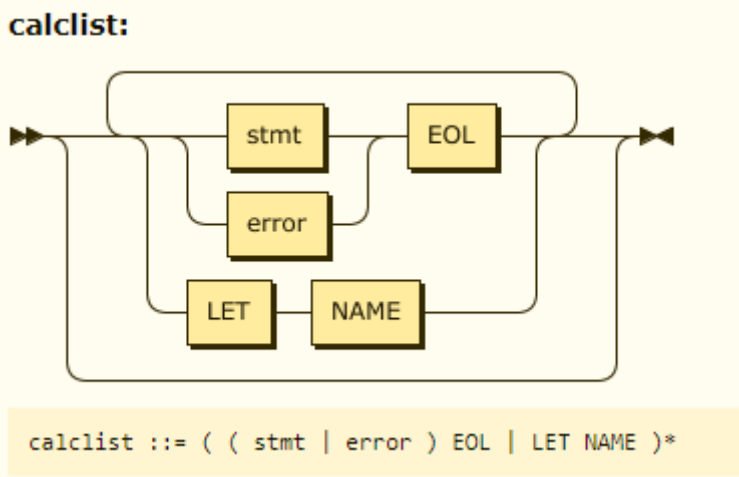


Figura 32 - Diagrama do calclist

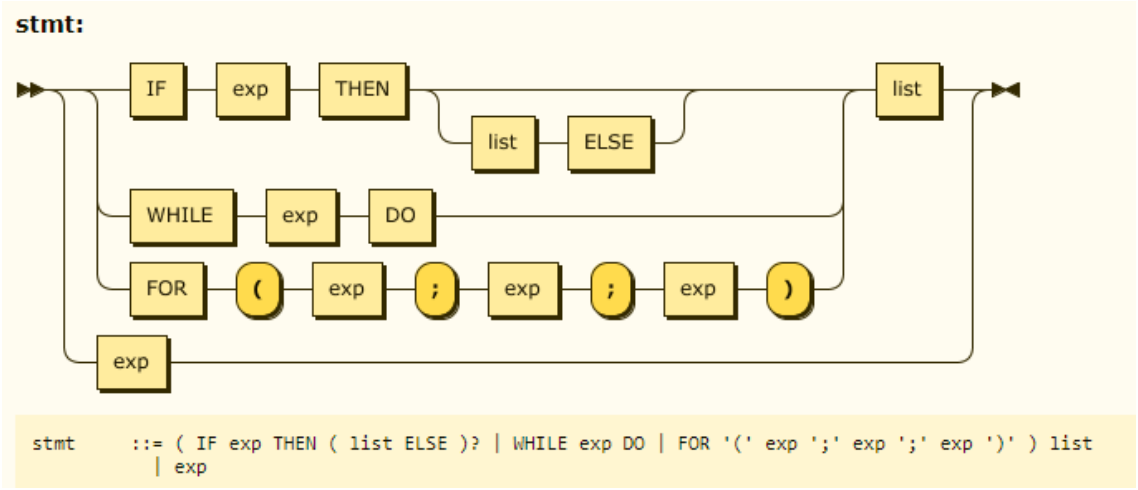


Figura 33 - Diagrama do stmt

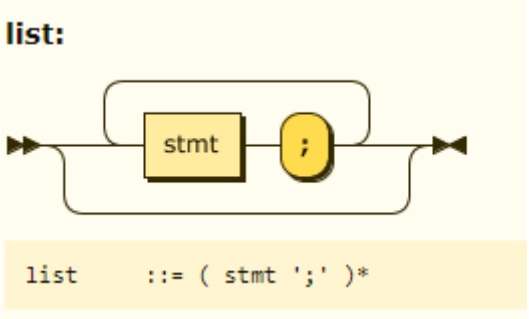
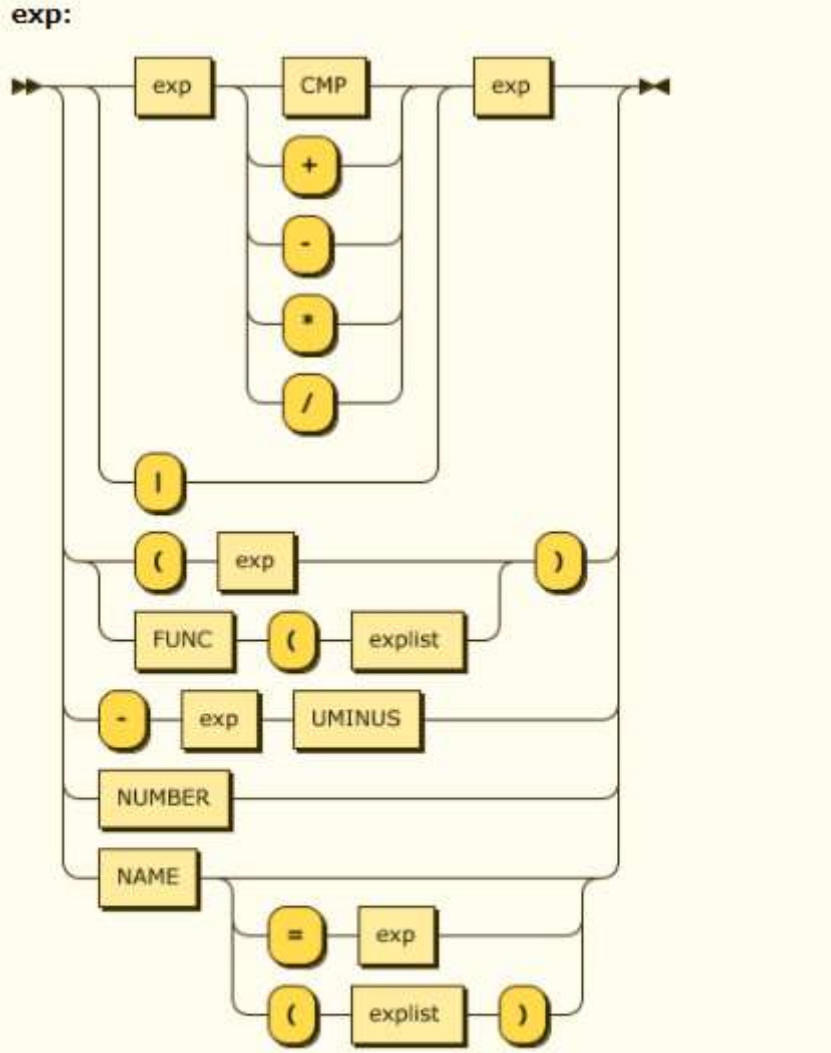
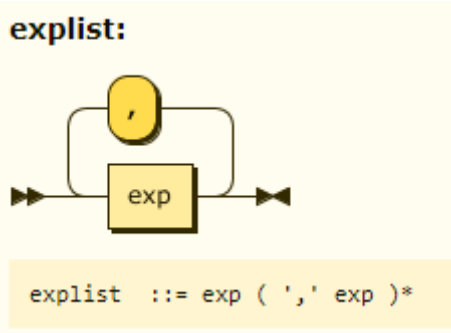


Figura 34 - Diagrama do list




```
exp      ::= ( exp ( CMP | '+' | '-' | '*' | '/' ) | '|' ) exp
           | ( '(' exp | FUNC '(' explist ) ')'
           | '-' exp UMINUS
           | NUMBER
           | NAME ( '=' exp | '(' explist ')' )?
```

Figura 35 - Diagrama do exp




```
explist  ::= exp ( ',' exp )*
```

Figura 36 - Diagrama do explist

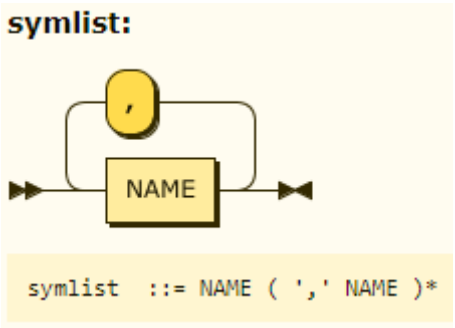


Figura 37 - Diagrama do symlist

7. Descrição do analisador sintático conforme implementação no Bison

No analisador sintático é incluso algumas bibliotecas do C e a tabela de símbolo calculadora.h, em seguida é feito uma estrutura na qual possui um ponteiro para AST, uma variável double, outra variável inteira para controlar o tipo de função e dois ponteiros para tabela de símbolos. Segue a figura 38.

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

%union {
    struct ast *a;
    double d;
    struct symbol *s;    /* qual o simbolo? */
    struct symlist *sl;
    int fn;              /* qual a função? */
}
```

Figura 38 - Bibliotecas e Estrutura de união

Em seguida é realizado a declaração dos tokens, e a tipagem, segue a ilustração na figura 39.

```
/* declaração de tokens */
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL

%token IF THEN ELSE WHILE DO LET FOR /*Adicionando o comando FOR*/

%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS

%type <a> exp stmt list explist
%type <sl> symlist
```

Figura 39 - Declaração de tokens

A gramática se distingue entre as declarações (stmt) e as expressões (exp). O stmt faz o controle de fluxo dos comandos (IF/THEN/ELSE, WHILE/DO e o adicional FOR). Já as expressões exp possui uma lista de regras, sendo que cada regra corresponde a uma instrução e chama uma rotina para construir um nó na AST. As duas regras podem ser analisadas na figura 40.

```
stmt: IF exp THEN list                      { $$ = newflow('I', $2, $4, NULL); }
    | IF exp THEN list ELSE list            { $$ = newflow('I', $2, $4, $6); }
    | WHILE exp DO list                     { $$ = newflow('W', $2, $4, NULL); }
    | FOR '(' exp ';' exp ';' exp ')' list  { $$ = newfor('O', $3, $5, $7, $9); }
    | exp
    ;

exp: exp CMP exp { $$ = newcmp($2, $1, $3); }
   | exp '+' exp { $$ = newast('+', $1,$3); }
   | exp '-' exp { $$ = newast('-', $1,$3);}
   | exp '*' exp { $$ = newast('*', $1,$3); }
   | exp '/' exp { $$ = newast('/', $1,$3); }
   | '|' exp { $$ = newast('|', $2, NULL); }
   | '(' exp ')' { $$ = $2; }
   | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
   | NUMBER { $$ = newnum($1); }
   | NAME { $$ = newref($1); }
   | NAME '=' exp { $$ = newasgn($1, $3); }
   | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
   | NAME '(' explist ')' { $$ = newcall($1, $3); }
   ;
```

Figura 40 - regras para declarações e expressões

A definição list, é recursiva a direita, criando uma lista de declarações e ou expressões, sempre que tiver uma lista de declaração ou expressão um novo nó é adicionado na AST. O mesmo pode ser visto na figura 41.

```
list: {$$ = NULL} /*vazio*/
    | stmt ';' list { if ($3 == NULL)
                        $$ = $1;
                      else
                        $$ = newast('L', $1, $3);
                    }
    ;
```

Figura 41 - Lista de declarações e ou expressões

A definição explist é recursiva a direita e cria uma lista de expressões e constrói uma AST das listas, já o symlist cria uma lista de símbolos fictícios para definição de funções. Ilustrado na figura 42.

```
explist: exp
       | exp ',' explist { $$ = newast('L', $1, $3); }
       ;

symlist: NAME { $$ = newsymlist($1, NULL); }
       | NAME ',' symlist { $$ = newsymlist($1, $3); }
       ;
```

Figura 42 - Lista de expressão e de símbolos

O calclist é a regra que reconhece uma lista de instruções e declarações de funções, a mesma faz a avaliação da AST, imprime o resultado, e libera a estrutura AST. Pode se dizer que é a regra principal da gramática.

```
calclist: /* vazio */
        | calclist stmt EOL {
            printf("= %4.4g\n> ", eval($2));
            treefree($2);
          }
        | calclist LET NAME '(' symlist ')' '=' list EOL {
            dodef($3, $5, $8);
            printf("Defined %s\n> ", $3->name);
          }
        | calclist error EOL { yyerror; printf("> "); }
        ;
```

Figura 43 - Principal função da gramática

8. Conjunto de testes

Abaixo segue um conjunto de testes da calculadora avançada que foi desenvolvida, a descrição de cada figura apresenta do que se trata o teste.

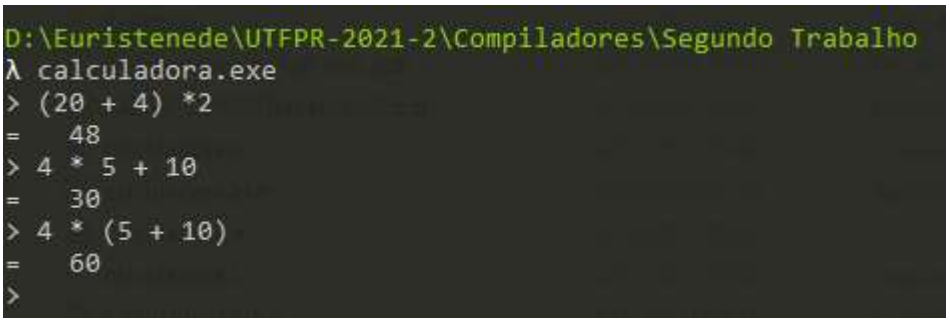


Figura 44 - Expressões matemáticas

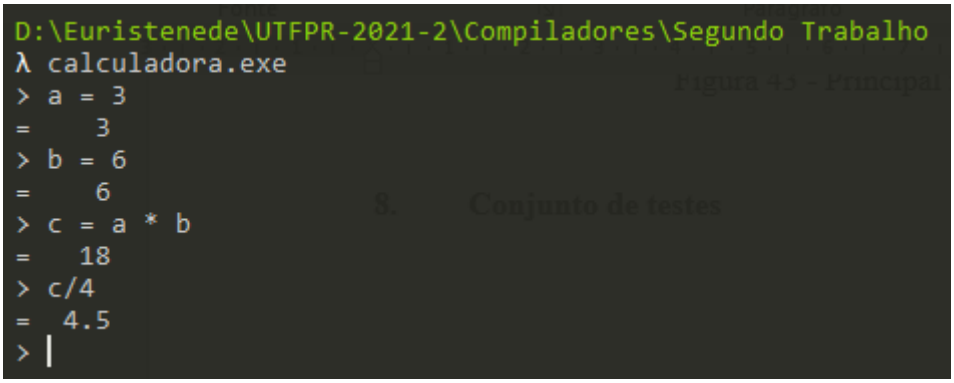


Figura 45 - Expressões matemáticas, continuação

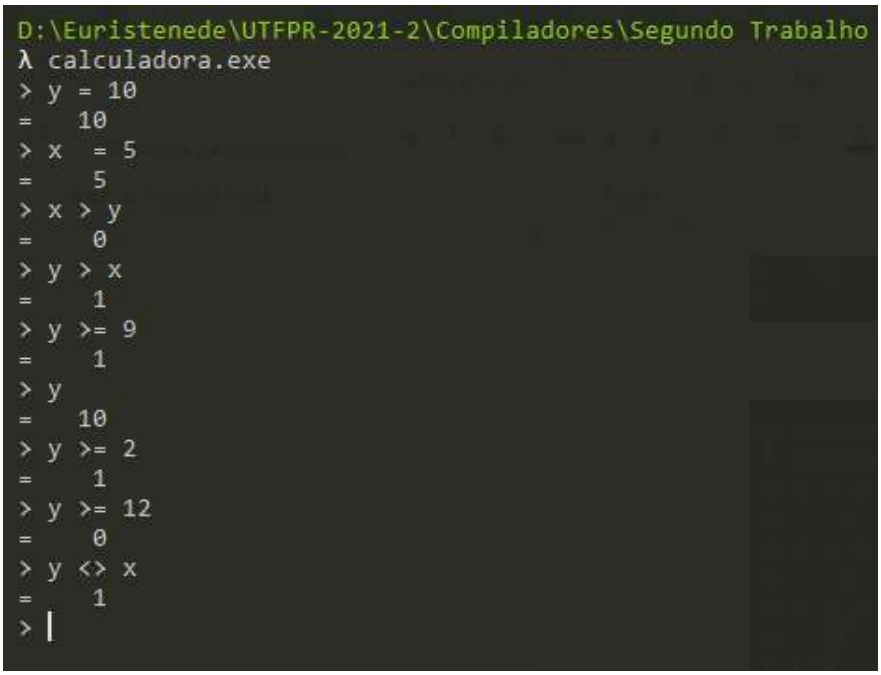


Figura 46 - Operadores de comparação

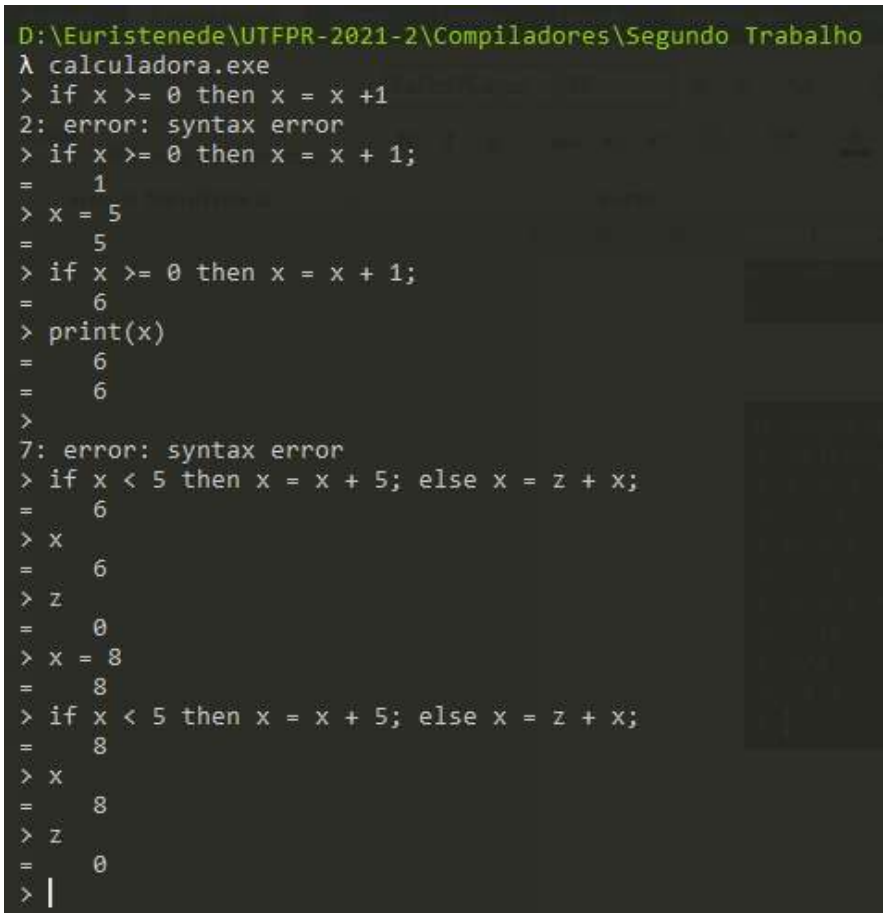


Figura 47 - Comando if/then/else

```
D:\Euristenede\UTFPR-2021-2\Compiladores\Segundo Trabalho
λ calculadora.exe
> y = 10
=   10
> while y > 0 do y = y - 1; printf(y);
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
=    0
> y = 10
=   10
> while y > 0 do y = y - 1; print(y);
=    9
=    8
=    7
=    6
=    5
=    4
=    3
=    2
=    1
=    0
=    0
> while y <= 36 do y = print(exp(y));
=    1
= 2.718
= 15.15
= 3.814e+006
= 3.814e+006
>
```

Figura 48 - Comando while/do

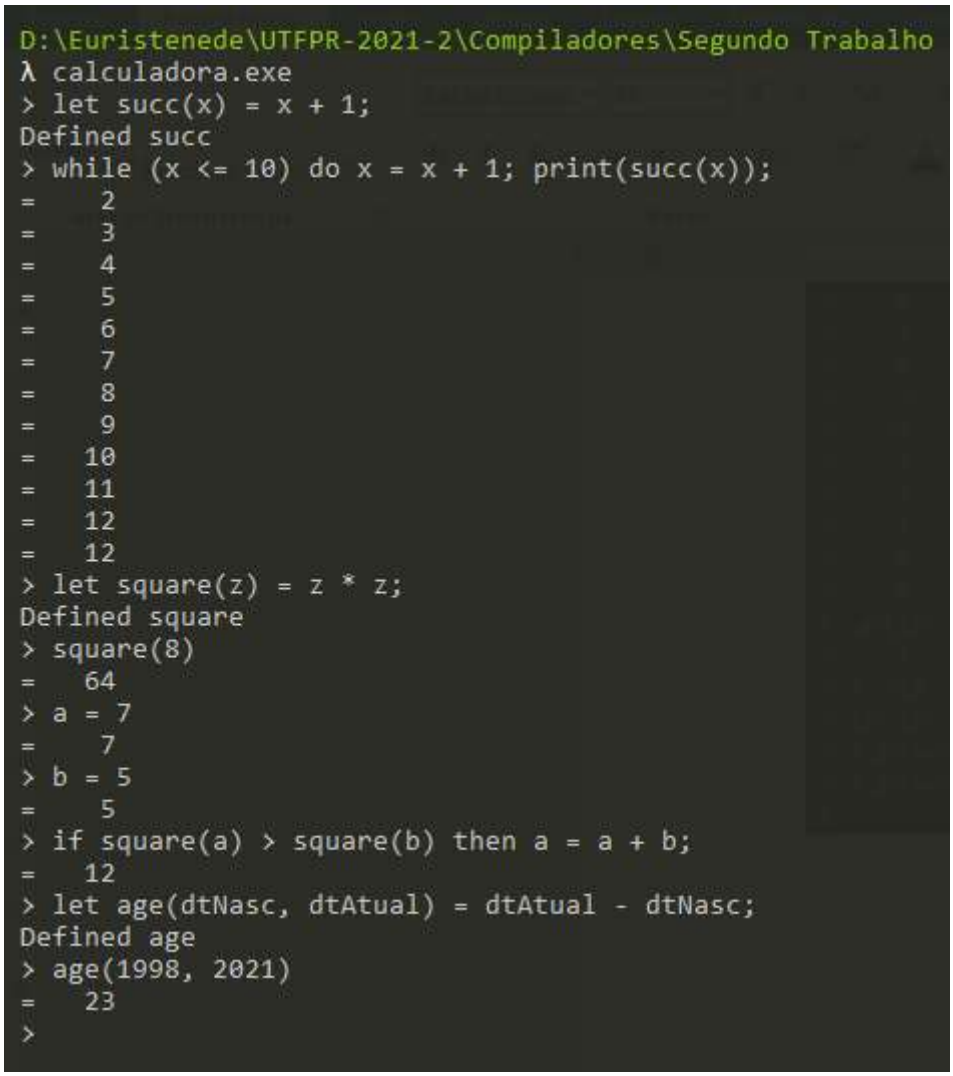


Figura 49 - Definindo funções de usuário

```
D:\Euristenede\UTFPR-2021-2\Compiladores\Segundo Trabalho
λ calculadora.exe
> let sum(x, y) = while x <= y do print(x); x = x + 1;
2: error: syntax error
> let sum(x, y) = while x <= y do print(x); x = x + 1; ;
Defined sum
> let sum2(x, y) = while x <= y do x = x + 1; ; print(x);
Defined sum2
> sum(1, 10)
=    1
=    2
=    3
=    4
=    5
=    6
=    7
=    8
=    9
=   10
=   11
> sum2(1, 10)
=   11
=   11
> i = 0
=    0
> j = 6
=    6
> while(i < j) do sum2(i, j); i = i + 1; j = j - 1;
=    7
=    6
=    5
=    3
>
```

Figura 50 - Funções de usuário

```
D:\Euristenede\UTFPR-2021-2\Compiladores\Segundo Trabalho
λ calculadora.exe
> let sq(n)=e=1; while |((t=n/e)-e)>.001 do e=avg(e,t);;
Defined sq
> let avg(a,b)=(a+b)/2;
Defined avg
> sq(10)
= 3.162
> sqrt(10)
= 3.162
> sq(10)-sqrt(10)
= 0.000178
>
```

Figura 51 - Utilizando funções pré-definidas

```
D:\Euristenede\UTFPR-2021-2\Compiladores\Segundo Trabalho
λ calculadora.exe
> for(i=0; i > 10;i = i + 1) print(i);
```

Figura 52 - For Adicionado

```
=    0
=    0
=    0
=    0
=    0
=    0
=    0
=    0
=    0
=    0
=    0
^C
```

Figura 53 - Acabou dando erro no for, e entrando em loop infinito.

REFERENCIAS

LOUDEN, Kenneth C. **Compiladores: princípios e prática**. 2. ed. São Paulo: Cengage Learning, 2004. xiv, 569 p. ISBN 9788522104222.

NOLETO, Cairo. **Linguagem C: o que é e quais os principais fundamentos!** 2020. Disponível em: https://blog.betrybe.com/linguagem-de-programacao/linguagem-c/. Acesso em: 14 dez. 2021.