

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ  
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL FELIPE TASAKA DE MELO

**RELATÓRIO: INTERPRETADOR DA LINGUAGEM C-**

RELATÓRIO

PONTA GROSSA  
2019

RAFAEL FELIPE TASAKA DE MELO

**RELATÓRIO: INTERPRETADOR DA LINGUAGEM C-**

Relatório apresentado como requisito para o segundo trabalho da matéria de Compiladores.

Orientador: Gleifer Vaz Alves

PONTA GROSSA  
2019

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO .....</b>	<b>3</b>
<b>2</b>	<b>DESENVOLVIMENTO .....</b>	<b>4</b>
2.1	A GRAMÁTICA .....	4
2.2	O CÓDIGO .....	6
2.2.1	Analizador léxico .....	6
2.2.2	Analizador sintático .....	7
<b>3</b>	<b>TABELA DE SÍMBOLOS .....</b>	<b>9</b>
<b>4</b>	<b>AST .....</b>	<b>10</b>
<b>5</b>	<b>TESTES E RESULTADOS.....</b>	<b>11</b>
5.1	COMPILANDO O INTERPRETADOR .....	11
5.2	ARQUIVOS BEM-SUCEDIDOS .....	11
5.3	RESULTADOS .....	12
5.4	CASOS EM QUE O INTERPRETADOR FOI MAL-SUCEDID.....	12
<b>6</b>	<b>CONCLUSÃO .....</b>	<b>13</b>

## 1 INTRODUÇÃO

A linguagem C- é um tipo de C com recursos a menos - podendo assim ser mais otimizado para programações em baixo nível.

Este trabalho tem como objetivo construir um interpretador de C- utilizando a ferramenta Flex e Bison

## 2 DESENVOLVIMENTO

Esta seção apresentará partes do desenvolvimento do projeto com trechos do código e breves explicações sobre o funcionamento do analisadores.

### 2.1 A GRAMÁTICA

Precisaremos de dois itens para fazer o interpretador: um analisador léxico e um analisador sintático. Uma vez com a gramática em mãos podemos definir os tokens.

A gramática está definida a seguir em notação BNF e ela foi fortemente baseada (assim como suas funções) na calculadora avançada(LEVINE, 2009):

$$\begin{aligned} \langle calclist \rangle &::= \emptyset \\ &| \langle calclist \rangle \text{ TYPE NAME ' ( ' } \langle symlist \rangle \text{ ' ) ' ' { ' } \langle list \rangle \text{ ' } \end{aligned}$$

$$\begin{aligned} \langle stmt \rangle &::= \text{ IF ' ( ' } \langle exp \rangle \text{ ' ) ' ' { ' } \langle list \rangle \text{ ' } \text{ ' } \\ &| \text{ IF ' ( ' } \langle exp \rangle \text{ ' ) ' ' { ' } \langle list \rangle \text{ ' } \text{ ' ELSE ' { ' } \langle list \rangle \text{ ' } \text{ ' } \\ &| \text{ WHILE ' ( ' } \langle exp \rangle \text{ ' ) ' ' { ' } \langle list \rangle \text{ ' } \text{ ' } \\ &| \text{ DO ' { ' } \langle list \rangle \text{ ' } \text{ ' WHILE ' ( ' } \langle exp \rangle \text{ ' ) ' } \\ &| \langle exp \rangle \end{aligned}$$

$$\begin{aligned} \langle list \rangle &::= \emptyset \\ &| \langle stmt \rangle \text{ ' ; ' } \langle list \rangle \end{aligned}$$

$$\begin{aligned} \langle exp \rangle &::= \langle exp \rangle \text{ CMP } \langle exp \rangle \\ &| \langle exp \rangle \text{ ' + ' } \langle exp \rangle \\ &| \langle exp \rangle \text{ ' - ' } \langle exp \rangle \\ &| \langle exp \rangle \text{ ' * ' } \langle exp \rangle \\ &| \langle exp \rangle \text{ ' / ' } \langle exp \rangle \\ &| \text{ ' ( ' } \langle exp \rangle \text{ ' ) ' } \\ &| \text{ NUMBER} \\ &| \text{ FUNC ' ( ' } \langle explist \rangle \text{ ' ) ' } \\ &| \text{ NAME} \\ &| \text{ TYPE NAME} \\ &| \text{ TYPE NAME ' = ' } \langle exp \rangle \\ &| \text{ NAME ' = ' } \langle exp \rangle \end{aligned}$$

| NAME '(' *explist* ')'

*explist* ::= *exp*

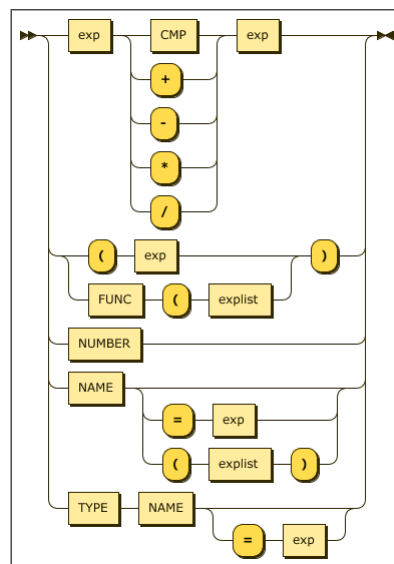
| *exp* ',' *explist* *symlist* ::= NAME

| NAME ',' *symlist*

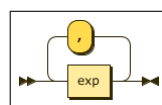
|  $\emptyset$

Onde os tokens '{', '}', '+', '-', '\*', '/', '(', ')', ',', NAME, TYPE, FUNC, NUMBER, CMP, IF, ELSE, DO, WHILE são tokens terminais.

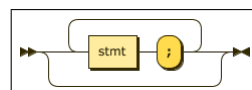
Segue, também, os diagramas de sintaxe:



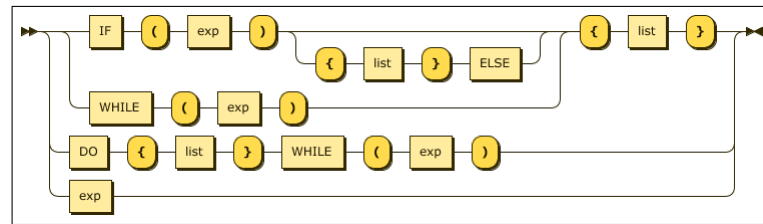
**Figura 1 – Regra exp**



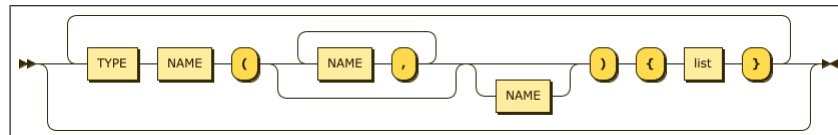
**Figura 2 – Regra explist**



**Figura 3 – Regra list**



**Figura 4 – Regra stmt**



**Figura 5 – Regra calclist**

## 2.2 O CÓDIGO

Nesta seção será apresentado trechos do código - tanto do analisador léxico quanto do analisador sintático.

### 2.2.1 Analisador léxico

O analisador léxico pegará toda a entrada e analisará letras de maneira individual em tokens (MOGENSEN, 2010). Sua principal função é facilitar a análise para o analisador sintático. A ferramenta que fará isso será o Flex.

Uma vez esses tokens separados, o analisador sintático os recombinará - estruturando assim o corpo do texto.

```

"+" |
"-" |
"*" |
"/" |
"=" |
"|" |
"," |
";" |
"(" |
")" |
"{" |
"}" { 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; }

"if" { return IF; }
"then" { return THEN; }
"else" { return ELSE; }
"while" { return WHILE; }
"do" { return DO; }

"int" |
"float" |
"double" { return TYPE; }

"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]* |
".*[0-9]+ { yylval.d = atof(yytext); return NUMBER; }

"//".*
[ \n\t]
\\n
. { yyerror("Mystery character %c\n", *yytext); }

```

**Figura 6 – O analisador léxico**

Não há grandes complicações nas expressões. Temos palavras reservadas para identificar IF, ELSE, DO, entre outras. Qualquer coisa diferente dessas palavras chaves será um texto - que é definido como NAME (uma vez que esse será o nome da variável). Além disso qualquer número é definido como NUMBER. Qualquer outra coisa será considerado um caractere desconhecido.

Uma vez que uma das regras é satisfeita o analisador lexico retorna ao analisador sintático qual token foi encontrado. Com o token em mãos o analisador sintático realizará o código C que será apresentado mais adiante.

### 2.2.2 Analisador sintático

O analisador sintático é mais complexo - uma vez que ele trabalha com a gramática, a interpretação em si e a função main do programa.



Primeiramente o analisador sintático é dividido em três partes (assim como o analisador léxico)(LEVINE, 2009): declarações prévias, a gramática em si e as funções posteriores

Nas declarações prévias são declaradas algumas variáveis que serão utilizadas e a inclusão de bibliotecas como a `stdio.h` e a `stdlib.h`. Além disso também são declarados todos os tokens que serão utilizados.

A segunda parte do programa é a mais essencial para o funcionamento do tradutor. A medida que o analisador léxico retorna os tokens encontrados no arquivo de entrada, o analisador sintático os trata e faz alguns comandos em C para realizar a interpretação em si.

Como o código é relativamente grande não serão tratadas todas as partes da gramática, mas alguns pontos principais serão explicados.

Diferentemente do primeiro trabalho, coisas mais complexas serão necessárias de se implementar - como uma AST e uma tabela de símbolos.

### 3 TABELA DE SÍMBOLOS

A tabela de símbolo comportará todos os símbolos encontrados na gramática (variáveis neste caso).

A tabela de símbolos, tecnicamente falando, será uma tabela hash comportando os seguintes valores:

- O nome da variável;
- O seu valor;
- Caso ela seja uma função, o ponteiro para os nós da AST com procedimentos;
- Caso ela seja uma função, a lista de parâmetros;

Para encontrar o valor, a função lookup procura o nome do símbolo na tabela. Caso ele encontre é retornado o endereço da tabela onde está o símbolo, caso contrário ele alocará o símbolo na tabela. É importante salientar que, caso ocorra overflow na tabela, o programa é encerrado imediatamente.

```
struct symbol * lookup(char* sym)
{
    struct symbol *sp = &symtab[symhash(sym)%NHASH];
    int scount = NHASH;
    while(--scount >= 0) {
        if(sp->name && !strcmp(sp->name, sym)) { return sp; } //encontrou o simbolo

        if(!sp->name) { //a variavel n foi encontrada
            sp->name = strdup(sym);
            sp->value = 0;
            sp->func = NULL;
            sp->syms = NULL;
            return sp;
        }

        if(++sp >= symtab+NHASH) sp = symtab; //proximo indice
    }
    yyerror("symbol table overflow\n");
    abort(); //a tabela esta cheia. Abortar
}
```

**Figura 7 – A função lookup**

É utilizando a função lookup que valores serão resgatados e/ou atribuídos a determinada variável.

## 4 AST

A AST é uma árvore que constrói a gramática - assim saberemos para qual lado devemos percorrer .

Diversas structs são utilizadas para construir cada nó da árvore. A função eval é a chave que fará com que - a partir de um nó, podemos correr entre os seus filhos e realizar as operações necessárias. É nele que fazemos as expressões aritméticas, os operadores if/else e os laços.

Por exemplo: caso essa função encontre um nó da AST que realize uma soma, eval descerá recursivamente na árvore até encontrar os valores e, logo após, fará a soma e a retornará na pilha da recursão.

```
case '+': v = eval(a->l) + eval(a->r); break;
case '-': v = eval(a->l) - eval(a->r); break;
case '*': v = eval(a->l) * eval(a->r); break;
case '/': v = eval(a->l) / eval(a->r); break;

case '1': v = (eval(a->l) > eval(a->r)) ? 1 : 0; break;
case '2': v = (eval(a->l) < eval(a->r)) ? 1 : 0; 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;
```

**Figura 8 – Um trecho da função eval que realiza operadores aritméticos e comparações**

## 5 TESTES E RESULTADOS

Com o código pronto foram realizados alguns testes para analisar tanto a gramática quanto os resultados gerados pelo interpretador.

### 5.1 COMPILANDO O INTERPRETADOR

Para compilar os seguintes comandos devem ser digitados:

```
flex cmm.l
```

```
bison -d cmm.y
```

```
gcc -o $@ cmm.tab.c lex.yy.c cmm_func.c -ll
```

Para facilitar e agilizar o processo de testes, um makefile foi feito:

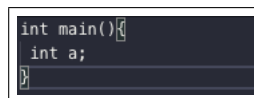
```
make cmm
```

Para rodar o tradutor devemos digitar:

```
.cmm < nome - do - arquivo.cmm
```

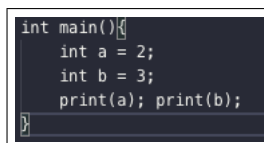
### 5.2 ARQUIVOS BEM-SUCEDIDOS

Segue abaixo exemplos de arquivos bem-sucedidos na interpretação, ou seja, que não obtiveram erros na sintaxe.



```
int main() {  
    int a;  
}
```

**Figura 9 – Primeiro teste**



```
int main() {  
    int a = 2;  
    int b = 3;  
    print(a); print(b);  
}
```

**Figura 10 – Segundo teste**

```
int main(){
    int a = 2;
    int b = 3;
    a = (1+2) - 3/4;
    b = a/2;
    print(a); print(b);
}
```

**Figura 11 – Terceiro teste**

```
int main(){
    int a = 2;
    int b = 1;
    if(a > b){
        if(a == 2){
            print(a);
        };
    }else{
        print(b);
    }
    ;
}
```

**Figura 12 – Quarto teste**

### 5.3 RESULTADOS

O resultado é escrito em um arquivo .txt intitulado de "resultado.txt"

Os resultados foram:

- Teste 1 = Não imprime nada;
- Teste 2 = Imprime 2 e 3;
- Teste 3 = Imprime 2.25 e 1.125;
- Teste 4 = Imprime 2;

### 5.4 CASOS EM QUE O INTERPRETADOR FOI MAL-SUCEDID

Algumas funcionalidades, apesar de estarem implementadas, não funcionam bem como:

- Funções;
- Laços (entra em loop);
- Os tipos de variável não são tratados

## 6 CONCLUSÃO

Trabalhar com o Flex e o Bison não é simples - uma vez que são bem sensíveis e algo fora da gramática não é aceito pelo programa. Muitas dificuldades foram encontradas para a realização (o próprio funcionamento do Bison foi complicado de entender), e nem todas as funções foram implementadas com sucesso - mostrando o quão complexo é trabalhar com um compilador

## REFERÊNCIAS

LEVINE, J. R. **flex bison**. [S.l.]: O'Reilly, 2009.

MOGENSEN, T. Ægidius. **Basics of Compiler Design**. [S.l.]: lulu.com., 2010. ISBN 978-87-993154-0-6.