



Universidade Estadual de Santa Cruz
Bacharelado em Ciência da Computação

RELATÓRIO CET058-CMP Projeto Final

Discente: Igor Lima Rocha
Docente: César Alberto Bravo Pariente

Ilhéus, 19 de Dezembro de 2022

IGOR LIMA ROCHA

RELATÓRIO CET058-CMP Projeto Final

Relatório da atividade proposta como
avaliação da matéria CET058 - Compiladores.

Ilhéus, 15 de Setembro de 2022

Sumário

RELATÓRIO CET058-CMP Projeto Final	2
1. INTRODUÇÃO	5
2. OBJETIVOS	5
3. LINGUAGEM ACEITA	5
3.1. Declaração de funções	5
3.2. Conjunto de comandos	6
3.3. Expressões	7
4. RESULTADOS	7
5. MÓDULOS	7
5.1. Analisador Sintático (Parser)	7
5.2. Analisador Léxico (Scanner)	8
5.3. ADA2ASA (Árvore de Análise para Árvore de Sintaxe Abstrata)	8
5.4. ASA2NP (Árvore de Sintaxe Abstrata para Notação Polonesa)	9
5.5. NP2GCI (Notação Polonesa para Geração de Código Intermediário)	10
5.6. GCI2GCO (Geração de Código Intermediário para Geração de Código Objeto)	10
6. METODOLOGIA	10
7. CRONOGRAMA	11
8. TESTES	11
8.1. Teste simples	12
a. Entrada	12
b. Saída	12
8.2. Teste com 1 chamada de função e 1 while	12
a. Entrada	12
b. Saída	13
8.3. Teste com 1 IF dentro de 1 while	13

a. Entrada	13
b. Saída	13
8.4. Teste com 1 while dentro de 1 IF dentro de 1 while	14
a. Entrada	14
b. Saída	14
8.5. Teste com outra função além da main	14
a. Entrada	14
b. Saída	15
9. APÊNDICES	15
A. Gramática	15
B. Analisador Sintático (Parser)	16
C. Analisador Léxico (Scanner)	27
10. REFERÊNCIA	35

1. INTRODUÇÃO

O objetivo deste relatório é apresentar o trabalho desenvolvido para a disciplina CET058 - Compiladores, ministrada pelo professor César Alberto Bravo Pariente, no período 2022.2, na Universidade Estadual de Santa Cruz (UESC).

O trabalho consiste na implementação de um compilador para a linguagem de programação C, utilizando os módulos desenvolvidos durante a disciplina, sendo estes: Analisador Sintático (Parser), responsável por analisar a entrada; Analisador Léxico (Scanner), responsável por identificar os tokens; ADA2ASA (Árvore de Análise para Árvore de Sintaxe Abstrata), responsável por gerar a Árvore de Sintaxe Abstrata a partir da Árvore de Análise; ASA2NP (Árvore de Sintaxe Abstrata para Notação Polonesa), responsável por gerar a Notação Polonesa a partir da Árvore de Sintaxe Abstrata; GCI (Gerador de Código Intermediário), responsável por gerar o código intermediário a partir da Notação Polonesa; e GCO (Gerador de Código Objeto), responsável por gerar o código objeto a partir do código intermediário.

2. OBJETIVOS

Implementar um compilador para a linguagem de programação C, utilizando os módulos desenvolvidos durante a disciplina. O compilador deve ser capaz de gerar código objeto p-code, seguindo as regras da gramática descrita nos apêndices.

3. LINGUAGEM ACEITA

A linguagem aceita pelo compilador é a linguagem C, levando em consideração as seguintes regras:

3.1. Declaração de funções

O compilador aceita a declaração de uma função Main ('m'), ou de uma função auxiliar ('g') juntamente com a função Main, ou duas funções auxiliares ('n' e 'g'), sendo que a função 'n' deve ser declarada antes da função 'g'. A estrutura de uma função é:

```
F() { A; r(E); }
```

, onde:

- F: nome da função ('m', 'h' ou 'g')
- A: conjunto de comandos
- E: expressão

3.2. Conjunto de comandos

A gramática de um conjunto de comandos é `A`, onde:

- $A \rightarrow CB$

Sendo que CB é um comando ou um conjunto de comandos aceito pelo compilador.

- C: comando aceito pelo compilador
- B: finalização de um comando ('.') ou conjunto de comandos

Os comandos aceitos pelo compilador podem ser vistos na listagem abaixo:

- Atribuição de função:

```
h = g()
```

- Comando de atribuição:

```
j = E
```

- Expressão:

```
(EXE)
```

- While:

```
w(E) { CD
```

- If:

```
f (E) { CD
```

- For:

```
o (E; E; E) { CD
```

'D' é a finalização de um bloco de código ('}') ou um conjunto de comandos aceito pelo compilador.

3.3. Expressões

A gramática de uma expressão é 'E', podendo ser um número inteiro (0...9), uma variável ('x' ou 'y') ou uma operação aritmética. A estrutura de uma operação aritmética é '(EXE)', sendo que 'X' é um operador aritmético ('+', '-', '*' ou '/').

4. RESULTADOS

O compilador foi parcialmente implementado e testado com sucesso, gerando código objeto p-code para a maioria dos programas de teste fornecidos pelo professor.

5. MÓDULOS

Os módulos desenvolvidos durante a disciplina foram utilizados para a implementação do compilador. A seguir, cada um dos módulos é descrito.

5.1. Analisador Sintático (Parser)

O Analisador Sintático (Parser) é responsável por analisar a entrada (texto), gerando a Árvore de Análise (AA) a partir da entrada. A estrutura da Árvore de Análise é uma árvore n-ária, onde cada nó representa um símbolo da gramática. A estrutura da Árvore de Análise é descrita na Figura 1.

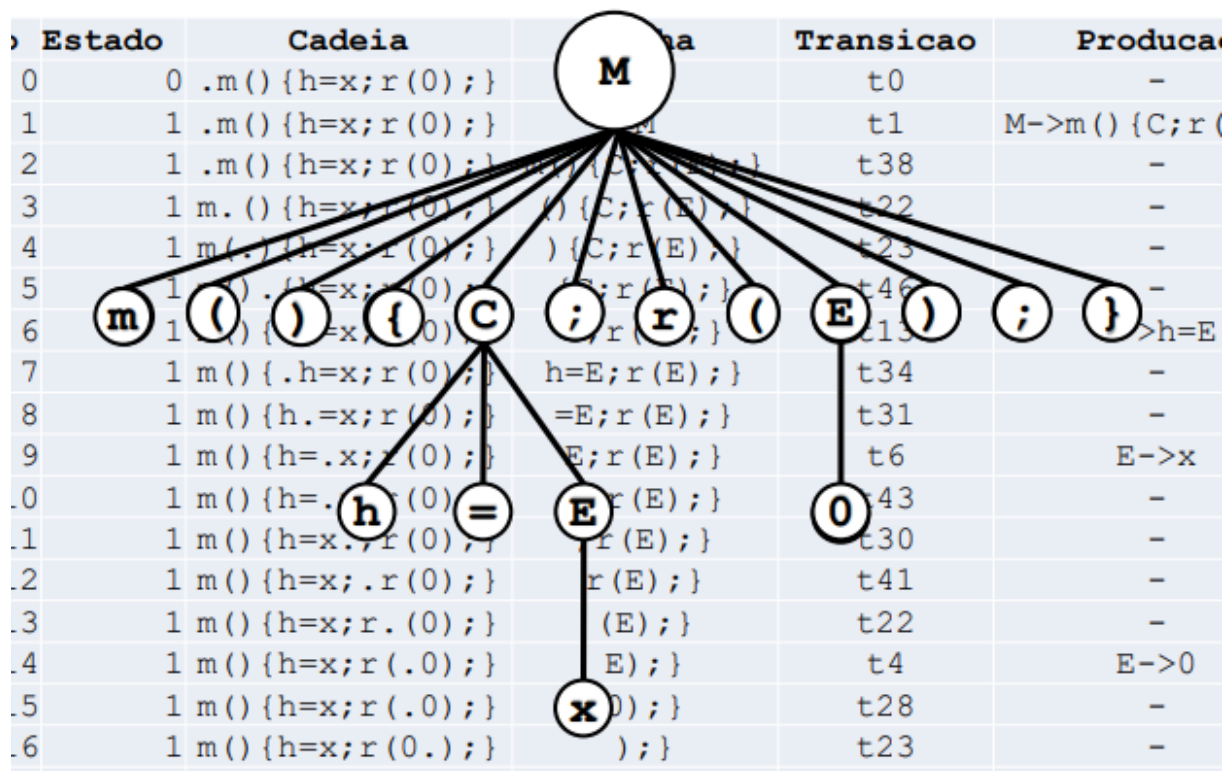


Figura 1: Estrutura da Árvore de Análise

A Árvore de Análise é gerada a partir da gramática descrita nos apêndices, que é utilizada para identificar os símbolos da entrada.

5.2. Analisador Léxico (Scanner)

O Analisador Léxico (Scanner) é responsável por identificar os tokens da entrada. Os tokens são identificados a partir de uma tabela de símbolos, que contém os tokens e seus respectivos valores, podendo ser palavras reservadas, constantes, identificadores ou qualquer outro símbolo da linguagem.

5.3. ADA2ASA (Árvore de Análise para Árvore de Sintaxe Abstrata)

O módulo ADA2ASA (Árvore de Análise para Árvore de Sintaxe Abstrata) é responsável por gerar a Árvore de Sintaxe Abstrata (ASA) a partir da Árvore de Análise (AA). A estrutura da Árvore de Sintaxe Abstrata é uma árvore binária, onde cada nó representa um símbolo da gramática. A estrutura da Árvore de Sintaxe Abstrata é descrita na Figura 2.

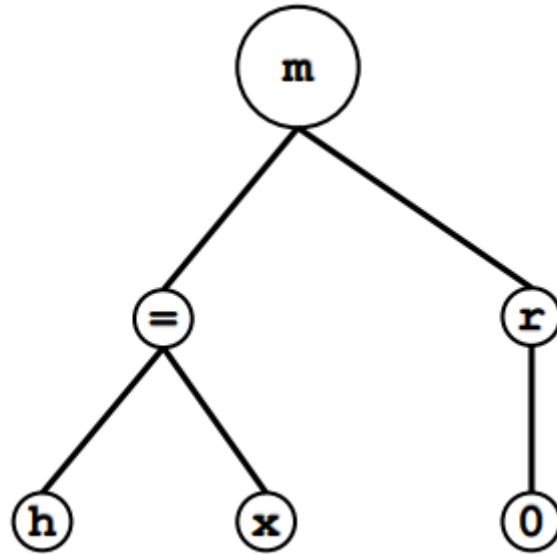


Figura 2: Estrutura da Árvore de Sintaxe Abstrata

A Árvore de Sintaxe Abstrata é gerada a partir da gramática descrita nos apêndices, que é utilizada para identificar os símbolos da entrada.

5.4. ASA2NP (Árvore de Sintaxe Abstrata para Notação Polonesa)

O módulo ASA2NP (Árvore de Sintaxe Abstrata para Notação Polonesa) é responsável por gerar a Notação Polonesa (NP) a partir da Árvore de Sintaxe Abstrata (ASA). A estrutura da Notação Polonesa é uma pilha, onde cada elemento da pilha representa um símbolo da gramática. A estrutura da Notação Polonesa é descrita na Figura 3.

Expressão polonesa:

$m=hxr0$

Expressão polonesa reversa:

$hx=0rm$

Figura 3: Estrutura da Notação Polonesa

A Notação Polonesa é gerada a partir da gramática descrita nos apêndices, que é utilizada para identificar os símbolos da entrada. A Notação Polonesa é utilizada para gerar o código objeto p-code.

5.5. NP2GCI (Notação Polonesa para Geração de Código Intermediário)

O módulo NP2GCI (Notação Polonesa para Geração de Código Intermediário) é responsável por gerar o código intermediário a partir da Notação Polonesa (NP). Fazer uso de uma representação intermediária entre a linguagem de alto nível e a linguagem de máquina é uma técnica utilizada para facilitar a implementação de compiladores, pois facilita a implementação do compilador para outras plataformas de hardware.

5.6. GCI2GCO (Geração de Código Intermediário para Geração de Código Objeto)

O módulo GCI2GCO (Geração de Código Intermediário para Geração de Código Objeto) é responsável por gerar o código objeto a partir do código intermediário. O código objeto é gerado em p-code, semanticamente equivalente ao código de entrada do usuário.

6. METODOLOGIA

A metodologia utilizada para o desenvolvimento do compilador foi a metodologia de desenvolvimento de software em cascata, onde cada módulo é desenvolvido e testado individualmente, antes de ser integrado ao módulo seguinte. A Figura 4 mostra os módulos desenvolvidos e a ordem de desenvolvimento.

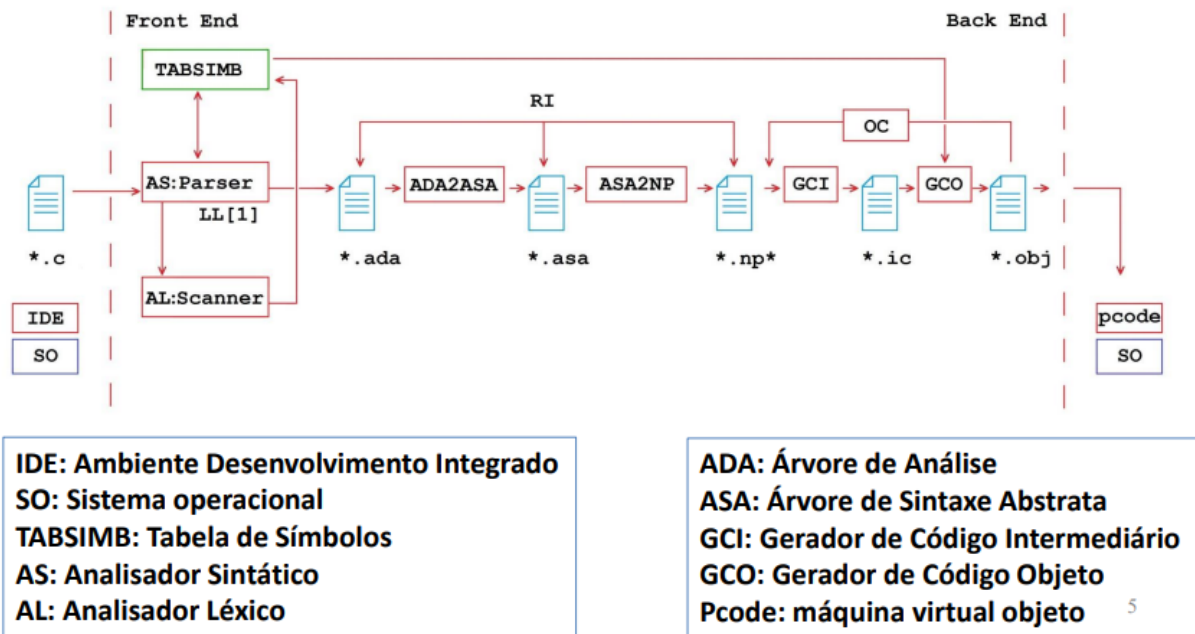


Figura 4: Metodologia de desenvolvimento de software em cascata

7. CRONOGRAMA

O cronograma de desenvolvimento do compilador é descrito na Tabela 1.

Módulo	Data de Término
Proj2B - Analisador Descendente Recursivo	17 de outubro
Proj2c - ADA2ASA	19 de outubro
Proj3b - ASA2NP	14 de novembro
Proj3c - NP2GCI	21 de novembro
Proj3d - GCI2GCO	01 de dezembro

8. TESTES

O objetivo dessa seção é descrever os testes realizados para verificar o funcionamento do compilador.

8.1. Teste simples

O teste simples consiste em compilar um programa simples, que possui uma inicialização de variável e um retorno.

a. Entrada

O arquivo de entrada para qualquer teste é o programa que será compilado, o qual neste caso é o programa abaixo.

```
m() {  
    j = (1+1);  
    r (0);  
}
```

b. Saída

O arquivo de saída varia de acordo com o módulo em questão. No módulo Parser, o arquivo de saída é uma representação da ADA em forma de vetor, com índices e valores correspondentes em cada posição. Já no módulo ADA2ASA, a saída é a representação da ASA. No módulo ASA2NP*, a saída é apenas a NP* em uma única linha. No módulo NP2GCI, o arquivo de saída é o código intermediário com operações que fazem referência à tabela de símbolos e à pilha. Por fim, no módulo GCI2GCO, o arquivo de saída é a tradução do código intermediário em código objeto, que é a representação do programa de entrada em p-code.

8.2. Teste com 1 chamada de função e 1 while

O teste consiste em compilar um programa que possui uma função chamada e um while.

a. Entrada

O arquivo de entrada para qualquer teste é o programa que será compilado, o qual neste caso é o programa abaixo.

```
m() {  
    h=g();  
    w(x) {  
        k=(1+(1*0))  
    };  
    r(y);  
}
```

b. Saída

Seguindo o mesmo padrão do teste anterior, o arquivo de saída varia de acordo com o módulo em questão. No módulo Parser, o arquivo de saída é uma representação da ADA em forma de vetor, com índices e valores correspondentes em cada posição. Já no módulo ADA2ASA, a saída é a representação da ASA. No módulo ASA2NP*, a saída é apenas a NP* em uma única linha. No módulo NP2GCI, o arquivo de saída é o código intermediário com operações que fazem referência à tabela de símbolos e à pilha. Por fim, no módulo GCI2GCO, o arquivo de saída é a tradução do código intermediário em código objeto, que é a representação do programa de entrada em p-code.

8.3. Teste com 1 IF dentro de 1 while

O teste consiste em compilar um programa que possui um while e um if dentro do while.

a. Entrada

O arquivo de entrada para qualquer teste é o programa que será compilado, o qual neste caso é o programa abaixo.

```
m() {  
    w(x) {  
        f(1) {  
            i=n();  
            x=(0+1);  
        };  
    };  
    r(y);  
}
```

b. Saída

Seguindo o mesmo padrão do teste anterior, o arquivo de saída varia de acordo com o módulo em questão. No módulo Parser, o arquivo de saída é uma representação da ADA em forma de vetor, com índices e valores correspondentes em cada posição. Já no módulo ADA2ASA, a saída é a representação da ASA. No módulo ASA2NP*, a saída é apenas a NP* em uma única linha. No módulo NP2GCI, o arquivo de saída é o código intermediário com operações que fazem referência à tabela de símbolos e à pilha. Por fim, no módulo GCI2GCO, o arquivo de saída é a tradução do código intermediário em código objeto, que é a representação do programa de entrada em p-code.

8.4. Teste com 1 while dentro de 1 IF dentro de 1 while

O teste consiste em compilar um programa que possui um while e um if dentro do while.

a. Entrada

O arquivo de entrada para qualquer teste é o programa que será compilado, o qual neste caso é o programa abaixo.

```
m() {  
  w(x) {  
    f(1) {  
      w((x+y)) {  
        k=g();  
      };  
    };  
  };  
  r(y);  
}
```

b. Saída

Seguindo o mesmo padrão do teste anterior, o arquivo de saída varia de acordo com o módulo em questão. No módulo Parser, o arquivo de saída é uma representação da ADA em forma de vetor, com índices e valores correspondentes em cada posição. Já no módulo ADA2ASA, a saída é a representação da ASA. No módulo ASA2NP*, a saída é apenas a NP* em uma única linha. No módulo NP2GCI, o arquivo de saída é o código intermediário com operações que fazem referência à tabela de símbolos e à pilha. Por fim, no módulo GCI2GCO, o arquivo de saída é a tradução do código intermediário em código objeto, que é a representação do programa de entrada em p-code.

8.5. Teste com outra função além da main

O teste consiste em compilar um programa que possui uma função além da main.

a. Entrada

O arquivo de entrada para qualquer teste é o programa que será compilado, o qual neste caso é o programa abaixo.

```

n() {
    w(1) {
        k=g();
    };
    r(1);
}
m() {
    h=n();
    i=g();
    r(0);
}

```

b. Saída

Seguindo o mesmo padrão do teste anterior, o arquivo de saída varia de acordo com o módulo em questão. No módulo Parser, o arquivo de saída é uma representação da ADA em forma de vetor, com índices e valores correspondentes em cada posição. Já no módulo ADA2ASA, a saída é a representação da ASA. No módulo ASA2NP*, a saída é apenas a NP* em uma única linha. No módulo NP2GCI, o arquivo de saída é o código intermediário com operações que fazem referência à tabela de símbolos e à pilha. Por fim, no módulo GCI2GCO, o arquivo de saída é a tradução do código intermediário em código objeto, que é a representação do programa de entrada em p-code.

9. APÊNDICES

A. Gramática

Abaixo está a gramática utilizada para o desenvolvimento do compilador.

- **p1:** $S \rightarrow M$
- **p2:** $S \rightarrow G \ M$
- **p3:** $S \rightarrow N \ G \ M$
- **p4:** $N \rightarrow n() \{ \ C; \ r(E); \ }$
- **p5:** $G \rightarrow g() \{ \ C; \ r(E); \ }$
- **p6:** $M \rightarrow m() \{ \ C; \ r(E); \ }$
- **p7:** $A \rightarrow CB$
- **p8:** $B \rightarrow .$
- **p9:** $B \rightarrow ;CB$
- **p10:** $E \rightarrow 0$
- **p11:** $E \rightarrow 1$

- p12: $E \rightarrow 2$
- p13: $E \rightarrow 3$
- p14: $E \rightarrow 4$
- p15: $E \rightarrow 5$
- p16: $E \rightarrow 6$
- p17: $E \rightarrow 7$
- p18: $E \rightarrow 8$
- p19: $E \rightarrow 9$
- p20: $E \rightarrow x$
- p21: $E \rightarrow y$
- p22: $E \rightarrow (EXE)$
- p23: $X \rightarrow +$
- p24: $X \rightarrow -$
- p25: $X \rightarrow *$
- p26: $X \rightarrow /$
- p27: $C \rightarrow h=g()$
- p28: $C \rightarrow i=n()$
- p29: $C \rightarrow j=E$
- p30: $C \rightarrow k=E$
- p31: $C \rightarrow z=E$
- p32: $C \rightarrow (EXE)$
- p33: $C \rightarrow w(E) \{ C; \}$
- p34: $C \rightarrow f(E) \{ C; \}$
- p35: $C \rightarrow o(E; E; E) \{ C; \}$
- p36: $D \rightarrow \}$
- p37: $D \rightarrow ;CD$

B. Analisador Sintático (Parser)

parser.h

```
#ifndef PARSER_H
#define PARSER_H
```



```

void program();

void expr();

void term();

void factor();

void atr();

void logic_expr();

void Y();

void stmt();

void stmts();

void var();

void if_stmt();

void for_stmt();


extern int nextToken;

extern char nextChar;

#endif

```

parser.c

```

#include <stdio.h>

#include <stdlib.h>

#include "parser.h"

#include "front.h"


static void error();


/**

```

```

* This is the example Recursive-Descent Parser in pp. 181 - 185 in the
* textbook
*
* Sebesta, R. W. (2012). Concepts of Programming Languages.
* Pearson, 10th edition.
*
*
* */

/* expr
* Parses strings in the language generated by the rule:
* <expr> -> <term> { (+ | -) <term> }
*/

void program() {
    printf("Enter <program>\n");
    if( nextToken == MAIN_KEYW) {
        lex();
        if( nextToken==LEFT_PAREN) {
            lex();
            if(nextToken==RIGHT_PAREN) {
                lex();
                if(nextToken==LEFT_BRACK) {
                    lex();
                    stmts();
                    if(nextToken ==RIGHT_BRACK)
                        lex();
                    else
                        error();
                }
            }
        }
    }
}

```

```

        error();

    }

    else

        error();

    }

    else

        error();

}

else

    error();

printf("Exit <program>\n");
}

void stmts(){

    printf("Enter <stmts>\n");

    stmt();

    if( nextToken == IF_KEYW || nextToken == FOR_KEYW || nextToken ==
IDENT){

        stmts();

    }

    printf("Exit <stmts>\n");

}

void stmt(){

    printf("Enter <stmt>\n");

    if( nextToken==IDENT){

        atr();

    }

}

```

```

else if( nextToken == IF_KEYW) {

    if_stmt();

}

else if( nextToken == FOR_KEYW) {

    for_stmt();

}

else

    error();

printf("Exit <stmt>\n");

}

void atr(){

    printf("Enter <atr>\n");

    if (nextToken == IDENT) {

        var();

        if(nextToken== EQ_OP) {

            lex();

            expr();

            if(nextToken==PONTO_VIRGULA)

                lex();

        }

        else

            error();

    }

    else

        error();

    printf("Exit <atr>\n");

}

```

```

void var(){

    printf("Enter <var>\n");

    if (nextToken==IDENT)

        lex();

    else

        error();

    printf("Exit <var>\n");

}


void expr()

{

    printf("Enter <expr>\n");

    term();


    while (nextToken == ADD_OP || nextToken == SUB_OP || nextToken ==
EQ_OP || nextToken == LESS_OP || nextToken==REL_OPS) {

        lex();

        term();

    }

    printf("Exit <expr>\n");

}

```

```

void term()
{
    printf("Enter <term>\n");

    factor();

    while (nextToken == MULT_OP || nextToken == DIV_OP) {
        lex();
        factor();
    }

    printf("Exit <term>\n");
} /* End of function term */

```

```

void factor()
{
    printf("Enter <factor>\n");

    if (nextToken == IDENT) {
        var();
    }

    else if (nextToken == INT_LIT) {
        lex();
    }

    else {

        if (nextToken == LEFT_PAREN) {
            lex();
            expr();

```

```

        if (nextToken == RIGHT_PAREN) {
            lex();
        } else {
            error();
        }
    }
}

else {
    error();
}

}

printf("Exit <factor>\n");
}

```

```

void for_stmt() {
    printf("Enter <for_stmt>\n");
    if (nextToken == FOR_KEYW) {
        lex();
        if (nextToken == LEFT_PAREN) {
            lex();
            atr();
            expr();
            if (nextToken == PONTO_VIRGULA) {
                lex();
                expr();
                if (nextToken == RIGHT_PAREN)
                    lex();
            } else

```

```

        error();

        if(nextToken==LEFT_BRACK){

            lex();

            stmts();

            if(nextToken==RIGHT_BRACK)

                lex();

            else

                error();

        }

        else

            error();

    }

    else

        error();

}

else

    error();

}

printf("Exit <for_stmt>\n");
}

void if_stmt(){

```



```

printf("Enter <if_stmt>\n");

if(nextToken==IF_KEYW){

    lex();

    if(nextToken== LEFT_PAREN){

        logic_expr();

        if(nextToken == RIGHT_PAREN)

            lex();

        if(nextToken == LEFT_BRACK ){

            lex();

            stmts();

            if(nextToken == RIGHT_BRACK)

                lex();

            else

                error();

        }

        else

            error();

    }

    else

        error();

}

printf("Exit <if_stmt>\n");

}

void logic_expr(){

    printf("Enter <logic_expr>\n");

    if(nextToken==IDENT){

        var();

```

```

    }

    else if ( nextToken == LEFT_PAREN){

        lex();

        logic_expr();

        Y();

        logic_expr();

        if( nextToken == RIGHT_PAREN)

            lex();

        else

            error();

    }

    else

        error();

    printf("Exit <logic_expr>\n");

}

void Y(){

    printf("Enter <Y>\n");

    if(nextToken==REL_OPS || nextToken==EQ_OP || nextToken==LESS_OP)

        lex();

    else

        error();

    printf("Exit <Y>\n");

}

static void error()

{

    printf("Error (more is desired, but not implemented).\n");

}

```

C. Analisador Léxico (Scanner)

front.h

```
#ifndef FRONT_H
#define FRONT_H

/* Character classes */

#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Token codes */

#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define EQ_OP 25
#define LESS_OP 26
#define MORE_OP 50
#define LEFT_PAREN 27
#define RIGHT_PAREN 28
#define LEFT_BRACK 29
#define RIGHT_BRACK 30
#define PONTO_VIRGULA 31
#define REL_OPS 32
#define MAIN_KEYW 40
#define IF_KEYW 41
```

```
#define FOR_KEYW 42

int lex();

#endif
```

front.c

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include "front.h"
#include "parser.h"

/* Global Variable */
int nextToken;
char nextChar;

/* Local Variables */
static int charClass;
static char lexeme [100];
static int lexLen;
static FILE *in_fp;
static char mainKW[]="main";
static char ifKW[]="if";
static char forKW[]="for";

/* Local Function declarations */
static void addChar();
static void getChar();
```

```

static void getNonBlank();

int main()
{
    /* Open the input data file and process its contents */
    if ((in_fp = fopen("front-in.txt", "r")) == NULL) {
        printf("ERROR - cannot open front.in \n");
    } else {
        getChar();
        do {
            lex();
            program();
        } while (nextToken != EOF);
    }

    return 0;
}

/*****
/* lookup - a function to lookup operators and parentheses and return
the
* token */
static int lookup(char ch) {
    switch (ch) {
        case '(':
            addChar();
            nextToken = LEFT_PAREN;
            break;
        case ')':
            addChar();
            nextToken = RIGHT_PAREN;

```

```

        break;

    case '+':
        addChar();
        nextToken = ADD_OP;
        break;

    case '-':
        addChar();
        nextToken = SUB_OP;
        break;

    case '*':
        addChar();
        nextToken = MULT_OP;
        break;

    case '/':
        addChar();
        nextToken = DIV_OP;
        break;

    case '{':
        addChar();
        nextToken = LEFT_BRACK;
        break;

    case '}':
        addChar();
        nextToken = RIGHT_BRACK;
        break;

    case ';':
        addChar();
        nextToken = PONTO_VIRGULA;
        break;

    case '=':

```

```

        addChar();

        nextToken = EQ_OP;

        break;

    case '&':

        addChar();

        nextToken = REL_OPS;

        break;

    case '!=':

        addChar();

        nextToken = REL_OPS;

        break;

    case '<':

        addChar();

        nextToken = LESS_OP;

        break;

    case '>':

        addChar();

        nextToken = REL_OPS;

        break;

    default:

        addChar();

        nextToken = EOF;

        break;

    }

    return nextToken;
}

/*****

/* addChar - a function to add nextChar to lexeme */

static void addChar() {

```

```

    if (lexLen <= 98) {
        lexeme[lexLen++] = nextChar;
        lexeme[lexLen] = 0;
    } else {
        printf("Error - lexeme is too long \n");
    }
}

/*****

/* getChar - a function to get the next character of input and
determine its
* character class */
static void getChar() {
    if ((nextChar = getc(in_fp)) != EOF) {
        if (isalpha(nextChar))
            charClass = LETTER;
        else if (isdigit(nextChar))
            charClass = DIGIT;
        else charClass = UNKNOWN;
    } else {
        charClass = EOF;
    }
}

/*****

/* getNonBlank - a function to call getChar until it returns a
non-whitespace
* character */
static void getNonBlank() {
    while (isspace(nextChar)) getChar();
}

```



```

/*****
/* lex - a simple lexical analyzer for arithmetic expressions */
int lex() {
    lexLen = 0;
    getNonBlank();

    switch (charClass) {
        /* Parse identifiers */
        case LETTER:
            addChar();
            getChar();
            while (charClass == LETTER || charClass == DIGIT) {
                addChar();
                getChar();
            }
            nextToken = IDENT;
            if(strcmp(mainKW,lexeme)==0){
                nextToken = MAIN_KEYW;
            }
            else if(strcmp(ifKW,lexeme)==0){
                nextToken = IF_KEYW;
            }
            else if(strcmp(forKW,lexeme)==0){
                nextToken = FOR_KEYW;
            }
            break;

        /* Parse integer literals */

```

```

        case DIGIT:

            addChar();

            getChar();

            while (charClass == DIGIT) {

                addChar();

                getChar();

            }

            nextToken = INT_LIT;

            break;

/* Parentheses and operators */

        case UNKNOWN:

            lookup(nextChar);

            getChar();

            break;

/* EOF */

        case EOF:

            nextToken = EOF;

            lexeme[0] = 'E';

            lexeme[1] = 'O';

            lexeme[2] = 'F';

            lexeme[3] = 0;

            break;

    } /* End of switch */

    printf("Next token is: %d, Next lexeme is %s\n", nextToken,
lexeme);

    return nextToken;

} /* End of function lex */

```

10. REFERÊNCIA

- a. Wikipedia contributors. (2022, August 8). **P-code machine**. In *Wikipedia, The Free Encyclopedia*. Retrieved 14:25, September 8, 2022, from https://en.wikipedia.org/w/index.php?title=P-code_machine&oldid=1103004966
- b. SETHI, Ravi; ULLMAN, Jeffrey D.; MONICA S. LAM. **Compiladores: princípios, técnicas e ferramentas**. Pearson Addison Wesley, 2008