UNIVERSIDADE FEDERAL DE LAVRAS

RELATÓRIO DE COMPILADORES

Professor: Ricardo Terra Nunes Bueno Villela

Alunos: Estevão Augusto da Fonseca Santos

Felipe Crisóstomo Silva Oliveira

Bernardo Coelho Pavani Marinho

1. INTRODUÇÃO

Este relatório apresenta o desenvolvimento e funcionamento de um compilador para a linguagem C-, um subconjunto simplificado da linguagem C, projetado com fins didáticos para a disciplina de Compiladores.

Nesta segunda etapa do trabalho, são detalhados dois componentes fundamentais do compilador: a análise léxica e a análise sintática.

A análise léxica, como etapa inicial, escaneia o código-fonte para identificar e categorizar seus elementos, transformando cadeias de caracteres em uma sequência de tokens significativos. Tal fase foi implementada utilizando a ferramenta Flex (Fast Lexical Analyzer Generator). Ademais, para representar visualmente o analisador, foi utilizado a ferramenta JFLAP para criar os diagramas de transição.

Em seguida, os tokens são enviados para a análise sintática, a qual verifica se as sequências se encontram na ordem sintática esperada da linguagem. Para a construção dessa parte, foi utilizada a ferramenta Yacc (Yet Another Compiler Compiler).

Serão exploradas as funcionalidades de ambas as ferramentas, as estratégias de implementação, os testes realizados e os resultados obtidos em cada fase da construção.

2. REFERENCIAL TEÓRICO

2.1 ANALISADOR LÉXICO

A análise léxica constitui a primeira fase do processo de compilação. Sua principal função é ler os caracteres do programa-fonte, agrupá-los em unidades lexicais chamadas lexemas, e produzir como saída uma sequência de tokens correspondentes a esses lexemas. Essa sequência é então enviada ao analisador sintático para que a análise gramatical seja realizada (AHO, A. V. et al, 2008).

Além de identificar lexemas, o analisador léxico desempenha outras funções importantes. Entre elas, destaca-se a remoção de comentários e espaços em branco (como espaços, quebras de linha e tabulações), que são utilizados para separar os tokens na entrada, mas não possuem significado sintático. Outra tarefa relevante é a associação das mensagens de erro geradas durante a compilação com a posição correta no código-fonte. Para isso, o analisador léxico pode monitorar o número de caracteres de quebra de linha processados, permitindo a associação precisa de mensagens de erro com o número da linha correspondente (AHO, A. V. et al, 2008).

Para automatizar o processo de análise léxica, utiliza-se a ferramenta Flex, que, com base em expressões regulares, permite ao programador definir padrões que representam os tokens de uma linguagem de programação. A partir dessas definições, o Flex gera automaticamente código em linguagem C, capaz de reconhecer tais padrões de forma eficiente. A especificação do Flex é dividida em três seções principais: (i) definições, onde são declaradas variáveis e expressões regulares reutilizáveis; (ii) regras, que associam padrões a ações específicas; e (iii) código do usuário, onde podem ser incluídas funções auxiliares e o código que será incorporado ao analisador (AHO, A. V. et al, 2008).

Essa automação facilita a construção de compiladores e interpretadores, reduzindo a complexidade e o tempo necessário para o desenvolvimento da fase léxica. Além disso, o Flex é amplamente documentado e suportado, o que o torna uma escolha popular tanto para fins educacionais quanto profissionais.

2.2 ANALISADOR SINTÁTICO

O analisador sintático tem como função principal receber os tokens produzidos pelo analisador léxico e verificar se a sequência corresponde à linguagem definida pela gramática. Além disso, ele deve ser capaz de emitir mensagens de erro claras em caso de falhas sintáticas e tentar recuperar-se para continuar a análise do programa. Em implementações práticas, a construção explícita da árvore de derivação pode ser dispensada, já que as ações de tradução e verificação podem ser realizadas durante a própria análise, permitindo que essa etapa seja integrada ao restante do front-end do compilador (AHO, A. V. et al, 2008).

Para essa etapa, foi utilizada a ferramenta Bison, que, assim como o Flex, gera código em linguagem C. O Bison é um gerador de analisadores sintáticos baseados principalmente na técnica LR(1) (Left-to-right, Rightmost derivation, com 1 token de lookahead), diferentemente da técnica LL(1). Ele lê uma gramática definida pelo programador e gera automaticamente um parser eficiente e capaz de detectar erros sintáticos durante a análise.

A gramática no Bison é definida por meio de regras de produção, que especificam como os símbolos não-terminais podem ser compostos por terminais (tokens) e outros não-terminais. Cada regra pode estar associada a uma ação em C, que é executada quando a regra é reconhecida pelo parser, permitindo a construção da árvore sintática, a realização de verificações semânticas ou outras operações necessárias.

O uso conjunto do Flex e do Bison facilita a implementação das fases léxica e sintática de compiladores, fornecendo ferramentas poderosas para o reconhecimento e a interpretação de linguagens formais.

3. DESCRIÇÃO DO TRABALHO

Neste projeto, desenvolvemos um analisador léxico e sintático para a linguagem de programação C- (subconjunto de C), que contém a seguinte descrição:

- Tipos de dados: inteiro, real, caractere, arranjo e registro.
- Funções: recursão, parâmetros passados por valor.
- Comandos: Atribuição, if/else, while, E/S simples (tratados como funções).
- Comentários: texto entre /* e */ (sem comentários aninhados).
- Palavras reservadas: int, float, struct, if, else, while, void, return (caixa baixa)
- Símbolo Inicial: <programa>

Tais dados são descritos na gramática abaixo, essa que utiliza a notação Forma Normal de Backus Estendida (FNBE):

1. <programa> ::= <declaração-lista>
2. <declaração-lista> ::= <declaração> { <declaração> }
3. <declaração> ::= <var-declaração> | <fun-declaração>
4. <var-declaração> ::= <tipo-especificador> <ident> **;** | <tipo-especificador> <ident> <abre-colchete> <num-int> <fecha-colchete> { <abre-colchete> <num-int> <fecha-colchete> } **;**
5. <tipo-especificador> ::= **int** | **float** | **char** | **void** | **struct** <ident> <abre-chave> <atributos-declaração> <fecha-chave>
6. <atributos-declaração> ::= <var-declaração> { <var-declaração> }
7. <fun-declaração> ::= <tipo-especificador> <ident> **(** <params> **)** <composto-decl>
8. <params> ::= <param-lista> | **void**
9. <param-lista> ::= <param> {**,** <param> }
10. <param> ::= <tipo-especificador> <ident> | <tipo-especificador> <ident> <abre-colchete> <fecha-colchete>
11. <composto-decl> ::= <abre-chave> <local-declarações> <comando-lista> <fecha-chave>
12. <local-declarações> ::= { <var-declaração> }
13. <comando-lista> ::= { <comando> }

14. <comando> ::= <expressão-decl> | <composto-decl> | <seleção-decl> | <iteração-decl> | <retorno-decl>

15. <expressão-decl> ::= <expressão> **;** | **;**

16. <seleção-decl> ::= **if (** <expressão> **)** <comando> | **if (** <expressão> **)** <comando> **else** <comando>

18. <iteração-decl> ::= **while (** <expressão> **)** <comando>

19. <retorno-decl> ::= **return ;** | **return** <expressão> **;**

20. <expressão> ::= <var> **=** <expressão> | <expressão-simples>

21. <var> ::= <ident> | <ident> <abre-colchete> <expressão> <fecha-colchete> { <abre-colchete> <expressão> <fecha-colchete> }

22. <expressão-simples> ::= <expressão-soma> <relacional> <expressão-soma> | <expressão-soma>

24. <relacional> :: **=** | **<=** | **<** | **>** | **>=** | **==** | **!=**

25. <expressão-soma> ::= <termo> {<soma> <termo>}

26. <soma> ::= **+** | **-**

27. <termo> ::= <fator> {<mult> <fator>}

28. <mult> ::= * | **/**

29. <fator> ::= **(** <expressão> **)** | <var> | <ativação> | <num> | <num-int>

30. <ativação> ::= <ident> **(** <args> **)**

31. <args> ::= [<arg-lista>]

32. <arg-lista> ::= <expressão> {, <expressão> }

33. <num> ::= [+ | -] <dígito> { <dígito> } [**.** <dígito> <dígito>] [**E** [**+** | **-**] <dígito> <dígito>]

34. <num-int> ::= <dígito> { <dígito> }

35. <digito> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

36. <ident> ::= <letra> { <letra> | <dígito> }

37. <letra> ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**

38. <abre-chave> ::= **{**

39. <fecha-chave> ::= **}**

40. <abre-colchete> ::= **[**

41. <fecha-colchete> ::= **]**

3.1 EXPRESSÕES REGULARES UTILIZADAS

A seguir, apresentam-se as expressões regulares implementadas no analisador léxico por meio da ferramenta Flex. Essas expressões têm como finalidade identificar os tokens utilizados pelo analisador sintático, além de permitir a detecção e contagem de erros léxicos durante a análise do código-fonte. Também foram empregadas a numeração de linhas e colunas do arquivo de entrada, o que facilita a localização de possíveis falhas.

O código-fonte completo do analisador léxico encontra-se abaixo:

```
C/C++
%{
/*-------------------------- Definitions --------------------------*/
#include <stdio.h>
#include<string.h>
#include "parser.tab.h"


int line_number = 1;
int column_number = 1;
int lexical_errors = 0;

#define TAB_SIZE 4
%}

%option noyywrap

/*-------------------------- Definições --------------------------*/

letra                           [a-z]
digito                          [0-9]
ident                           {letra}({letra}|{digito})*
num_int                         {digito}+
num
[+-]?[0-9]+(\.[0-9]+)?([E][+-]?[0-9]+)?

abre_chave                      \{
fecha_chave                     \}
abre_colchete                   \[
fecha_colchete                  \]
abre_parenteses                 \(
```

```
fecha_parenteses                \)

comment                         "/*"([^*]|\*+[^/])*\*+"/"
char_literal                    \'([^\\\n]|\\.)\'
other                           .


%%
%{
/*----------------------------- Regras
-----------------------------*/
%}

\n              { line_number++; column_number = 1; }
[ ]+            { column_number += yyleng; }
\t+           {
                  for (int i = 0; i < yyleng; i++) {
                      column_number += TAB_SIZE - ((column_number -
1) % TAB_SIZE);
                  }
              }

{comment}       { column_number += yyleng; }

"if"                { column_number += yyleng; return IF; }
"else"              { column_number += yyleng; return ELSE; }
"while"             { column_number += yyleng; return WHILE; }
"return"            { column_number += yyleng; return RETURN; }
"int"               { column_number += yyleng; return INT; }
"float"             { column_number += yyleng; return FLOAT; }
"char"              { column_number += yyleng; return CHAR; }
"struct"            { column_number += yyleng; return STRUCT; }
"void"              { column_number += yyleng; return VOID; }

{num_int}           { column_number += yyleng; return CONSTINT; }
{num}               { column_number += yyleng; return CONSTFLOAT; }
{ident}             { column_number += yyleng; return IDENTIFIER; }

"=="                { column_number += yyleng; return EQUAL_OP; }
"!="                { column_number += yyleng; return NOT_EQUAL_OP; }
"<="                { column_number += yyleng; return LESS_EQUAL_OP;
}
```

```
">="                { column_number += yyleng; return RIGHT_EQUAL_OP;
}
"<"                 { column_number += yyleng; return LEFT_OP; }
">"                 { column_number += yyleng; return RIGHT_OP; }
"="                 { column_number += yyleng; return ASSIGN_OP; }

{abre_chave}        { column_number += yyleng; return LEFT_BRACE; }
{fecha_chave}       { column_number += yyleng; return RIGHT_BRACE; }
{abre_colchete}     { column_number += yyleng; return LEFT_BRACKET; }
{fecha_colchete}    { column_number += yyleng; return RIGHT_BRACKET;
}
{abre_parenteses}   { column_number += yyleng; return LEFT_PAREN; }
{fecha_parenteses}  { column_number += yyleng; return RIGHT_PAREN; }
";"                 { column_number += yyleng; return SEMICOLON; }
","                 { column_number += yyleng; return COMMA; }

{char_literal}          { column_number += yyleng; return CONSTCHAR;
}
\"([^\\\"\n]|\\.)*\"    { column_number += yyleng; return
CONSTSTRING; }

"+"                 { column_number += yyleng; return PLUS; }
"-"                 { column_number += yyleng; return MINUS; }
"/"                 { column_number += yyleng; return DIVISION; }
"*"                 { column_number += yyleng; return MULTIPLY; }

{num_int}{ident}        {
                            fprintf(yyout, "Erro Léxico:
identificador invalido iniciado com numero na linha %d, coluna %d:
%s\n", line_number, column_number, yytext);
                            column_number += yyleng;
                            lexical_errors++;
                        }

[\u201C\u201D]          {
                            fprintf(yyout, "Erro Léxico: uso de aspas
invalidas (aspas curvas) na linha %d, coluna %d: %s\n", line_number,
column_number, yytext);
                            column_number += yyleng;
                            lexical_errors++;
                        }

\'([^\\\n]|\\.){2,}\'   {
```

```
                            fprintf(yyout, "Erro Léxico: constante
char com mais de um caractere na linha %d, coluna %d: %s\n",
line_number, column_number, yytext);
                            column_number += yyleng;
                            lexical_errors++;
                        }

\"([^\\\"\n]|\\.)*      {
                            fprintf(yyout, "Erro Léxico: string nao
fechada na linha %d, coluna %d: %s\n", line_number, column_number,
yytext);
                            column_number += yyleng;
                            lexical_errors++;
                        }

{other}                 {
                            fprintf(yyout, "Erro Léxico: simbolo
lexico invalido na linha %d, coluna %d: %s\n", line_number,
column_number, yytext);
                            column_number += yyleng;
                            lexical_errors++;
                        }

%%
```

3.2 GRAMÁTICAS FORMAIS UTILIZADAS

Abaixo, estão as gramáticas que criamos em Bison baseado na descrição do C- que formam sua sintaxe. Nela, é utilizada uma função que conta os erros sintáticos previstos e os informa com detalhes para o usuário junto de sua linha e coluna. Caso haja erros imprevistos, uma função própria do Bison é acionada automaticamente, que usa uma mensagem de erro padrão da ferramenta junto da linha e coluna que ocorreu.

C/C++

```
%{
    #include <stdio.h>
    #include <stdlib.h>

    extern FILE *yyin;
    extern int yylex();
    extern int yyparse();

    extern int line_number;
    extern int column_number;

    extern int lexical_errors;
    int syntax_errors = 0;


    void yyerror(const char *s);
    void erro_sintatico_previsto(const char *msg);
%}

/*------------------------ Tokens ------------------------*/
%token IF ELSE WHILE RETURN
%token INT FLOAT CHAR VOID STRUCT

%token PLUS MINUS DIVISION MULTIPLY
%token EQUAL_OP NOT_EQUAL_OP LESS_EQUAL_OP RIGHT_EQUAL_OP LEFT_OP
RIGHT_OP
%token ASSIGN_OP

%token LEFT_BRACE RIGHT_BRACE
%token LEFT_BRACKET RIGHT_BRACKET
%token LEFT_PAREN RIGHT_PAREN

%token SEMICOLON COMMA

%token CONSTINT CONSTFLOAT CONSTCHAR CONSTSTRING
%token IDENTIFIER

/*------------------------ Precedências ------------------------*/

%left PLUS
%left MULTIPLY
%right ASSIGN_OP
%left EQUAL_OP NOT_EQUAL_OP LESS_EQUAL_OP
```

```
%left  RIGHT_EQUAL_OP LEFT_OP RIGHT_OP
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%

/*------------------------ Regras Sintáticas
------------------------*/

/*----- 1° -----*/
programa
    : declaracao_lista
    ;

/*----- 2° -----*/
declaracao_lista
    : declaracao
    | declaracao_lista declaracao
    | declaracao_lista error SEMICOLON
    { erro_sintatico_previsto("Erro Sintático: Declaracao mal formada
e não reconhecida"); yyerrok; }
    ;

/*----- 3° -----*/
declaracao
    : func_declaracao
    | var_declaracao
    ;

/*----- 4° -----*/
var_declaracao
    : tipo_especificador IDENTIFIER SEMICOLON
    | tipo_especificador IDENTIFIER arrayDimensao SEMICOLON
    | tipo_especificador IDENTIFIER ASSIGN_OP error SEMICOLON
    { erro_sintatico_previsto("Erro Sintático: Inicialização de
variável não suportada nesta linguagem"); yyerrok; }
    | tipo_especificador IDENTIFIER error SEMICOLON
    { erro_sintatico_previsto("Erro Sintático: Declaração de variável
inválida"); yyerrok; }
    | tipo_especificador error SEMICOLON
    { erro_sintatico_previsto("Erro Sintático: Declaração de variável
inválida"); yyerrok; }
    ;
```

```
arrayDimensao
    : LEFT_BRACKET CONSTINT RIGHT_BRACKET arrayDimensao
    | LEFT_BRACKET CONSTINT RIGHT_BRACKET
    | LEFT_BRACKET error RIGHT_BRACKET
    { erro_sintatico_previsto("Erro Sintático: Dimensao do array
invalida"); yyerrok; }
    | LEFT_BRACKET error RIGHT_BRACKET arrayDimensao
    { erro_sintatico_previsto("Erro Sintático: Dimensao do array
invalida"); yyerrok; }
    ;

/*----- 5° -----*/
tipo_especificador
    : INT
    | FLOAT
    | CHAR
    | VOID
    | STRUCT IDENTIFIER LEFT_BRACE varDeclList RIGHT_BRACE
    | STRUCT error LEFT_BRACE varDeclList RIGHT_BRACE
    { erro_sintatico_previsto("Erro Sintático: Nome de struct
ausente"); yyerrok; }
    ;

/*----- 6°: sequência de declarações de variáveis -----*/
varDeclList
    : var_declaracao
    | var_declaracao varDeclList
    ;

/*----- 7° -----*/
func_declaracao
    : tipo_especificador IDENTIFIER LEFT_PAREN params RIGHT_PAREN
composto_decl
    | tipo_especificador error LEFT_PAREN params RIGHT_PAREN
composto_decl
    { erro_sintatico_previsto("Erro Sintático: Função inexistente ou
invalida apos o tipo de retorno"); yyerrok; }
    | tipo_especificador IDENTIFIER LEFT_PAREN error RIGHT_PAREN
composto_decl
    { erro_sintatico_previsto("Erro Sintático: Lista de parâmetros
malformada na declaração de função"); yyerrok; }
    ;
```

```
/*----- 8° -----*/
params
    : params_lista
    | VOID
    ;

/*----- 9° -----*/
params_lista
    : param
    | param COMMA params_lista
    ;

/*----- 10° -----*/
param
    : tipo_especificador IDENTIFIER
    | tipo_especificador IDENTIFIER LEFT_BRACKET RIGHT_BRACKET
    | tipo_especificador IDENTIFIER error RIGHT_BRACKET
    { erro_sintatico_previsto("Erro Sintático: Falta de abrir
colchetes"); yyerrok; }
    ;

/*----- 11° -----*/
composto_decl
    : LEFT_BRACE local_declaracoes comando_lista RIGHT_BRACE
    ;

/*----- 12° -----*/
local_declaracoes
    : local_declaracoes var_declaracao
    | /* vazio */
    ;

/*----- 13° -----*/
comando_lista
    : comando_lista comando
    | /* vazio */
    ;

/*----- 14° -----*/
comando
    : expressao_decl
    | composto_decl
```

```
    | selecao_decl
    | iteracao_decl
    | retorno_decl
    | error SEMICOLON
    { erro_sintatico_previsto("Erro Sintático: Comando invalido
sintaticamente ou incompleto"); yyerrok; }
    ;

/*----- 15° -----*/
expressao_decl
    : expressao SEMICOLON
    | SEMICOLON
    ;

selecao_decl
    : IF LEFT_PAREN expressao RIGHT_PAREN comando %prec
LOWER_THAN_ELSE
    | IF LEFT_PAREN expressao RIGHT_PAREN comando ELSE comando
    | IF LEFT_PAREN error RIGHT_PAREN comando
    { erro_sintatico_previsto("Erro Sintático: Condição errada no
comando IF"); yyerrok;}
    ;

/*----- 17° -----*/
iteracao_decl
    : WHILE LEFT_PAREN expressao RIGHT_PAREN comando
    | WHILE LEFT_PAREN error RIGHT_PAREN comando
    { erro_sintatico_previsto("Erro Sintático: Comando WHILE
invalido"); yyerrok; }
    ;

/*----- 18° -----*/
retorno_decl
    : RETURN SEMICOLON
    | RETURN expressao SEMICOLON
    | RETURN error SEMICOLON
    { erro_sintatico_previsto("Erro Sintático: Retorno invalido");
yyerrok;}
    ;

/*----- 19° -----*/
expressao
    : var ASSIGN_OP expressao
```

```
    | var ASSIGN_OP error
    { erro_sintatico_previsto("Erro Sintático: Atribuição sem
expressão à direita"); yyerrok; }
    | expressao_simples
    ;

/*----- 20° -----*/
expressao_simples
    : exp_soma relacional exp_soma
    | exp_soma
    ;

/*----- 21° -----*/
relacional
    : LEFT_OP
    | RIGHT_OP
    | LESS_EQUAL_OP
    | RIGHT_EQUAL_OP
    | EQUAL_OP
    | NOT_EQUAL_OP
    ;

/*----- 23°: Operação de Soma mesmo -----*/
exp_soma
    : termo
    | exp_soma PLUS termo
    | exp_soma MINUS termo
    ;

termo
    : fator
    | termo MULTIPLY fator
    | termo DIVISION fator
    | termo MULTIPLY fator error
    { erro_sintatico_previsto("Erro Sintático: Operação de '*' sem o
fator"); yyerrok; }
    | termo DIVISION fator error
    { erro_sintatico_previsto("Erro Sintático: Operação de '/' sem o
fator"); yyerrok; }
    ;

/*----- 24° -----*/
fator
```

```
    : LEFT_PAREN expressao RIGHT_PAREN
    | var
    | ativacao
    | CONSTFLOAT
    | CONSTINT
    | CONSTCHAR
    | CONSTSTRING
    | LEFT_PAREN error RIGHT_PAREN
    { erro_sintatico_previsto("Erro Sintático: Expressao Vazia");
yyerrok; }
    ;

/*----- 25° -----*/
ativacao
    : IDENTIFIER LEFT_PAREN args RIGHT_PAREN
    | IDENTIFIER LEFT_PAREN error RIGHT_PAREN
    { erro_sintatico_previsto("Erro Sintático: Argumentos invalidos
no retorno da funcao"); yyerrok; }
    ;

/*----- 26° -----*/
args
    : arg_lista
    |
    ;

/*----- 27° -----*/
arg_lista
    : expressao
    | arg_lista COMMA expressao
    | arg_lista COMMA COMMA error
    { erro_sintatico_previsto("Erro Sintático: Falta de parametro");
yyerrok; }
    | arg_lista COMMA error
    { erro_sintatico_previsto("Erro Sintático: Virgula excedente ao
final da lista de parametros"); yyerrok; }
    ;

/*----- 28° -----*/
var
    : IDENTIFIER
    | IDENTIFIER LEFT_BRACKET expressao RIGHT_BRACKET var_auxiliar
    ;
```

```
/*---- 29° ----*/

var_auxiliar
    : var_auxiliar LEFT_BRACKET expressao RIGHT_BRACKET
    |
    ;

%%


/*------------------------ Funções auxiliares
-------------------------*/

void yyerror(const char *s) {
    fprintf(stderr, "Erro na linha %d, coluna %d: %s\n", line_number,
column_number, s);
}

void erro_sintatico_previsto(const char *msg) {
    syntax_errors++;
    fprintf(stderr, "%s na linha %d, coluna %d\n\n", msg,
line_number, column_number);
}

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Provenha o arquivo de entrada para o compilador.\n");
        return -1;
    }

    FILE *compiled_arq = fopen(argv[1], "r");
    if (!compiled_arq) {
        printf("O arquivo não é válido.\n");
        return -2;
    }

    yyin = compiled_arq;
    int result = yyparse();

    printf("\n=== Resultado da Análise ===\n");
    if (result == 0) {
        printf("ANALISE SINTATICA CONCLUIDA COM SUCESSO!\n");
```

```
    } else {
        printf("ANALISE SINTATICA FALHOU DEVIDO A ERROS!\n");
    }

    printf("Total de erros léxicos: %d\n", lexical_errors);
    printf("Total de erros sintáticos: %d\n", syntax_errors);

    fclose(compiled_arq);
    return 0;
}
```

3.3 PROCESSO DE GERAÇÃO E EXECUÇÃO DO COMPILADOR

O desenvolvimento do analisador léxico e sintático da linguagem C- seguiu uma metodologia modular, empregando as ferramentas Flex e Bison em um fluxo de trabalho sequencial para a geração do executável do compilador. A seguir, detalha-se o processo de compilação e execução, utilizando os comandos de linha de comando como referência.

3.3.1 GERAÇÃO DO ANALISADOR SINTÁTICO COM BISON

A primeira etapa do processo consiste na definição e compilação da gramática da linguagem. Para tal, utilizou-se a ferramenta Bison, que traduz a especificação da gramática formal em código C.

```
None
bison -d parser.y
```

Este comando processa o arquivo parser.y, que contém as regras gramaticais da linguagem C-. A opção -d é crucial, pois instrui o Bison a gerar não apenas o código-fonte C do analisador sintático (parser.tab.c), mas também um arquivo de cabeçalho (parser.tab.h). Este arquivo de cabeçalho é fundamental para a interoperabilidade entre o analisador léxico (Flex) e o sintático, pois ele contém as definições simbólicas de todos os tokens que a gramática espera receber do analisador léxico.

3.3.2 GERAÇÃO DO ANALISADOR LÉXICO COM FLEX

Com as definições de tokens estabelecidas pelo Bison, a próxima etapa envolve a criação do analisador léxico, responsável por identificar os elementos básicos (tokens) no código-fonte da linguagem.

```
None
flex projeto.l
```

Este comando lê o arquivo projeto.l, que contém as expressões regulares para reconhecimento dos tokens e as ações associadas a cada um deles. O Flex, ao processar projeto.l, gera o arquivo lex.yy.c. Este arquivo C contém a implementação da função yylex(), que é a interface principal para o analisador léxico. A função yylex() é responsável por ler a entrada, corresponder padrões às expressões regulares e, ao identificar um token, retorna seu tipo e, se aplicável, seu valor textual para o analisador sintático. É importante notar que lex.yy.c inclui internamente parser.tab.h para que possa referenciar as definições dos tokens geradas pelo Bison.

3.3.3 COMPILAÇÃO E VINCULAÇÃO DOS MÓDULOS

Após a geração dos módulos léxico e sintático em C, ambos precisam ser compilados e vinculados para formar um único executável.

```
None
gcc -o Compiler.o lex.yy.c parser.tab.c -lfl
```

O compilador GCC é utilizado para compilar os arquivos lex.yy.c e parser.tab.c. A opção -o Compiler.o especifica o nome do arquivo executável de saída. A flag -lfl é essencial, pois vincula a biblioteca de tempo de execução do Flex (libfl), que contém funções auxiliares necessárias para o funcionamento do analisador léxico. Este passo resulta na criação do executável Compiler.o, que representa o compilador completo da linguagem C-.

3.3.4 EXECUÇÃO DO COMPILADOR

Uma vez que o executável do compilador foi gerado, ele pode ser invocado para processar arquivos de código-fonte escritos na linguagem C-.

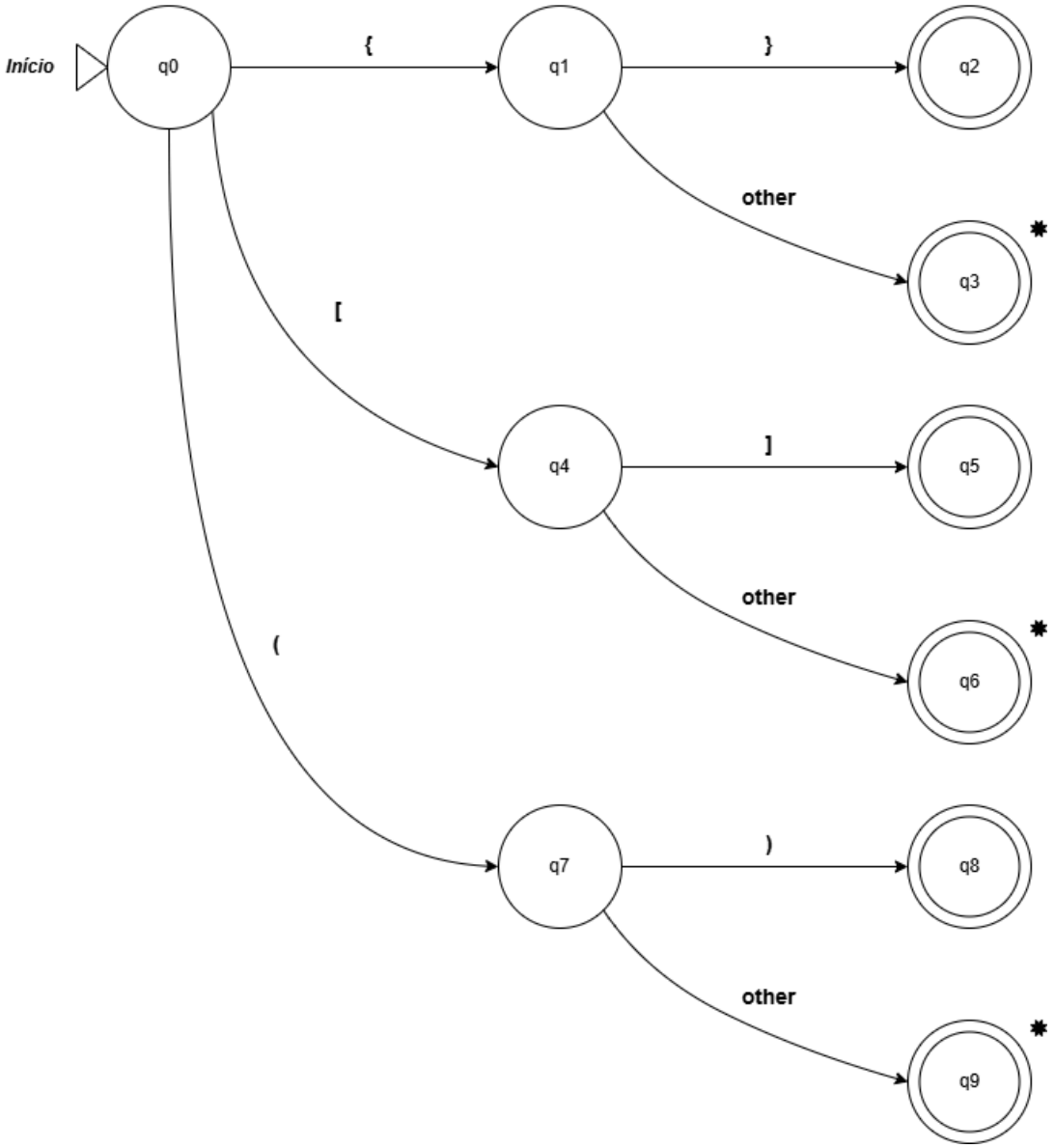
```
None
./Compiler.o <ARQUIVO-TEXTO-ENTRADA>
```

Este comando executa o compilador (Compiler.o) e passa o arquivo como entrada. O Compiler.o então utiliza o analisador léxico (yylex()) para ler o conteúdo do arquivo e extrair uma sequência de tokens. Essa sequência de tokens é então alimentada ao analisador sintático (yyparse()), que verifica a conformidade da estrutura do código com as regras gramaticais definidas. Durante este processo, quaisquer erros léxicos ou sintáticos detectados são reportados ao usuário com indicação da linha e coluna do problema.

4. DIAGRAMAS DE TRANSIÇÃO

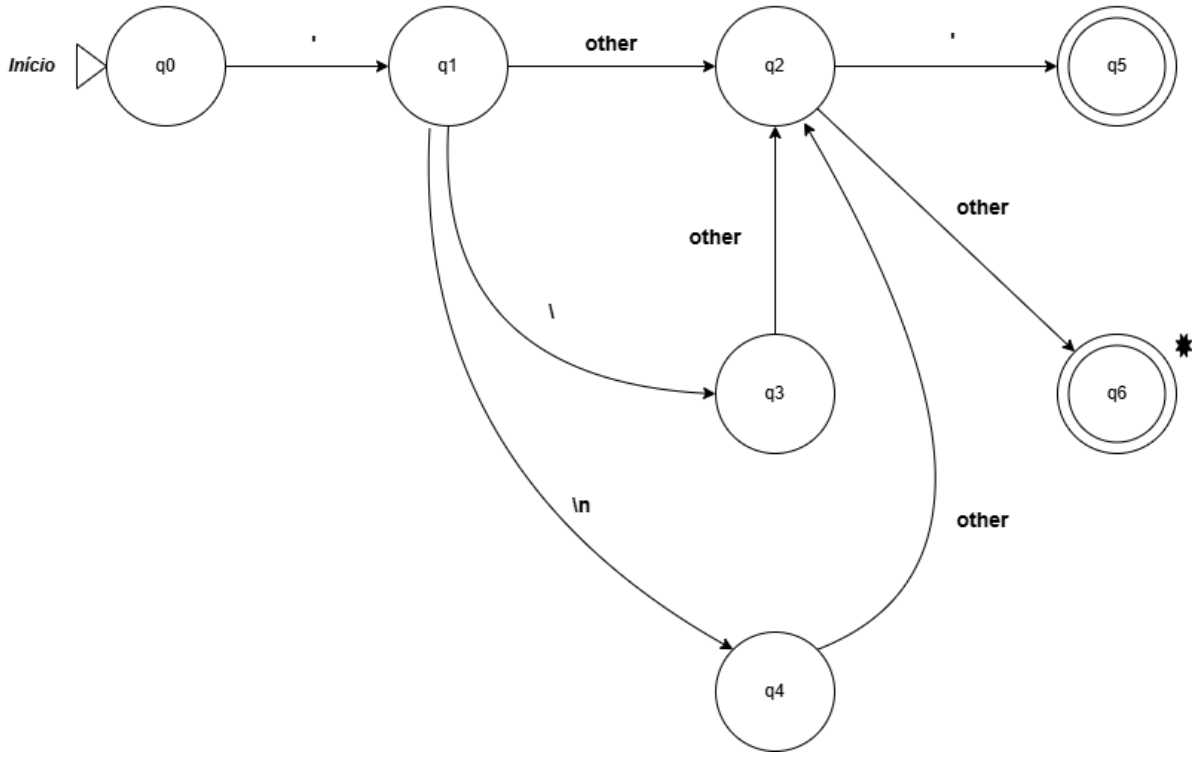
4.1 TRANSIÇÃO DE ABRE E FECHA

Diagrama de Transição de abre e fecha



4.2 TRANSIÇÃO DE CHAR_LITERAL

Diagrama de Transição de char_literal



4.3 TRANSIÇÃO DE LETRA, NUM E NUM_INT

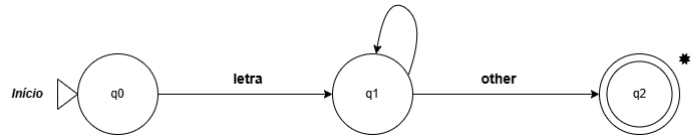


Diagrama de Transição de num

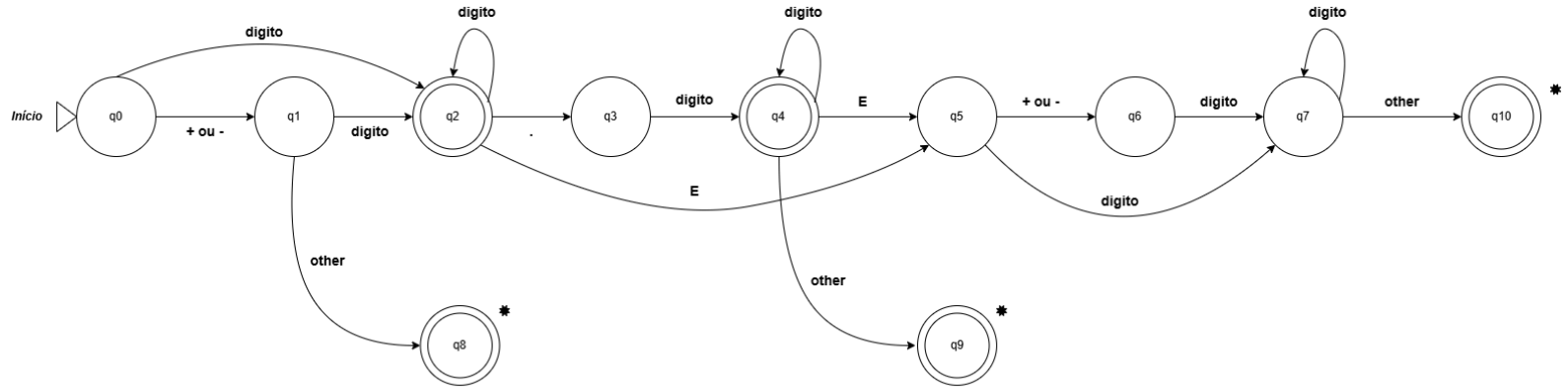
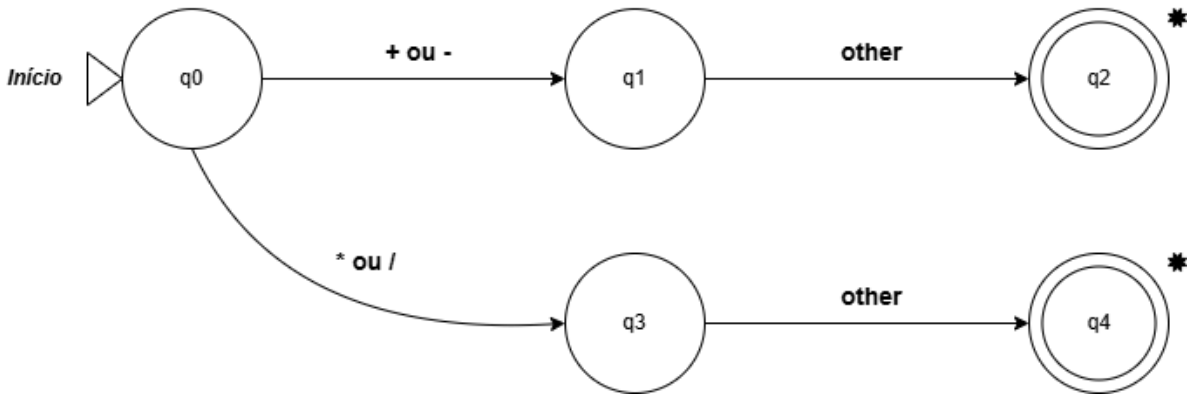


Diagrama de Transição de num_int

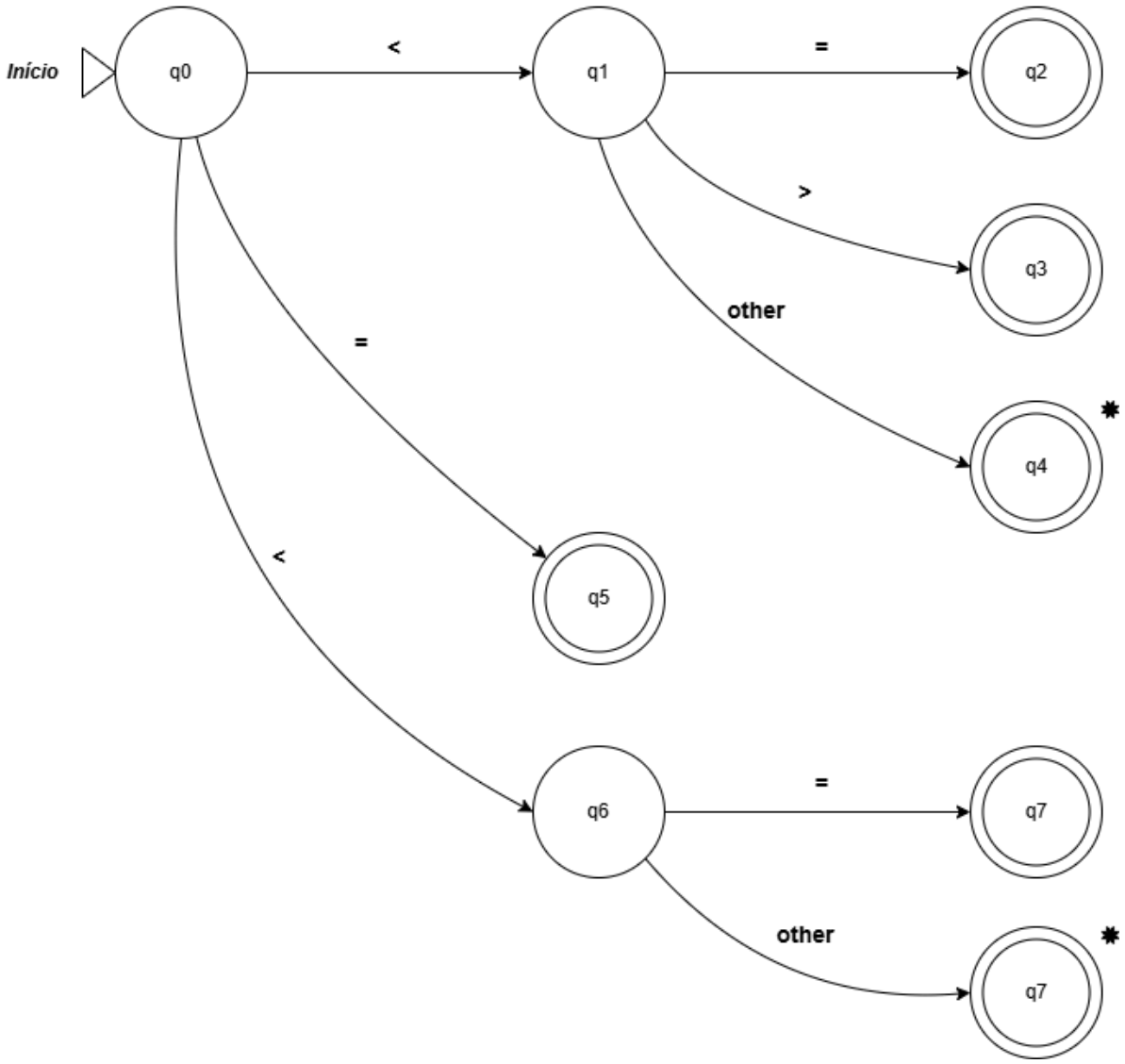
4.4 TRANSIÇÃO DE SOMA E MULTI

Diagrama de Transição de soma e multi



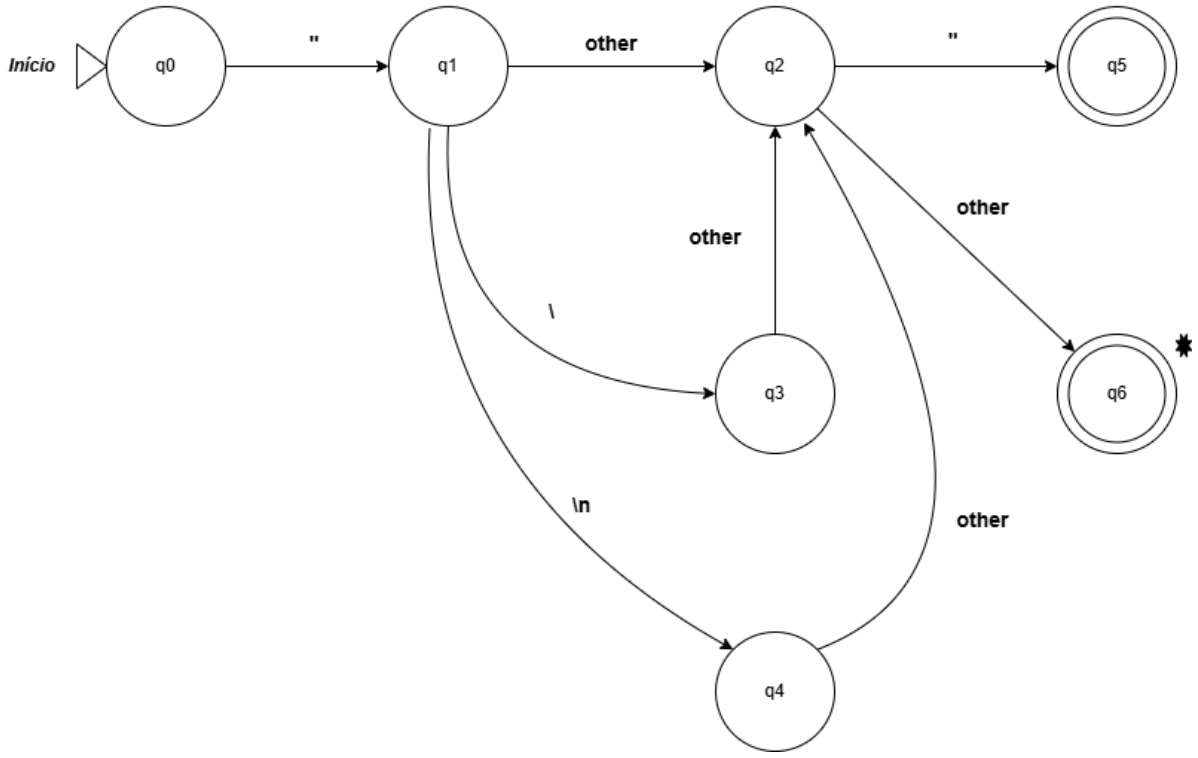
4.5 TRANSIÇÃO DE RELOP

Diagrama de Transição de relop



4.6 TRANSIÇÃO DE STRING_LITERAL

Diagrama de Transição de string_literal



5. TESTES EXECUTADOS E RESULTADOS OBTIDOS

5.1 ENTRADA: int main() { return 0; }

Resultado obtido:

Erro na linha 1, coluna 11: syntax error

Erro Sintático: Lista de parâmetros malformada na declaração de função na linha 1, coluna 25

=== Resultado da Análise ===

ANALISE SINTATICA CONCLUIDA COM SUCESSO!

Total de erros léxicos: 0

Total de erros sintáticos: 1

5.2 ENTRADA: char []txt = “texto”;

Erro na linha 1, coluna 7: syntax error

Erro Léxico: simbolo lexico invalido na linha 1, coluna 14:

Erro Léxico: simbolo lexico invalido na linha 1, coluna 15:

Erro Léxico: simbolo lexico invalido na linha 1, coluna 16:

Erro Léxico: simbolo lexico invalido na linha 1, coluna 22:

Erro Léxico: simbolo lexico invalido na linha 1, coluna 23:

Erro Léxico: simbolo lexico invalido na linha 1, coluna 24:

Erro Sintático: Declaração de variável inválida na linha 1, coluna 26

=== Resultado da Análise ===

ANALISE SINTATICA CONCLUIDA COM SUCESSO!

Total de erros léxicos: 6

Total de erros sintáticos: 1

5.3 ENTRADA: float quadrado(float x){ return x*x; }

Resultado obtido:

=== Resultado da Análise ===
ANALISE SINTATICA CONCLUIDA COM SUCESSO!
Total de erros léxicos: 0
Total de erros sintáticos:

5.4 ENTRADA: int res = 2*a - 9;

Resultado obtido:
Erro na linha 1, coluna 12: syntax error
Erro Sintático: Inicialização de variável não suportada nesta linguagem na linha 1, coluna 19

=== Resultado da Análise ===
ANALISE SINTATICA CONCLUIDA COM SUCESSO!
Total de erros léxicos: 0
Total de erros sintáticos: 1

5.5 ENTRADA: int vet[10][];

Resultado obtido:

Erro na linha 1, coluna 14: syntax error
Erro Sintático: Dimensao do array invalida na linha 1, coluna 15

=== Resultado da Análise ===
ANALISE SINTATICA CONCLUIDA COM SUCESSO!
Total de erros léxicos: 0
Total de erros sintáticos: 1

5.6 ENTRADA: int (int x) { return x; }

Resultado obtido:
Erro na linha 1, coluna 6: syntax error

Erro Sintático: Função inexistente ou invalida apos o tipo de retorno na linha 1, coluna 26

=== Resultado da Análise ===
ANALISE SINTATICA CONCLUIDA COM SUCESSO!
Total de erros léxicos: 0
Total de erros sintáticos: 1

5.7 ENTRADA: if x > 0 { x = x - 1; }

Resultado obtido:
Erro na linha 1, coluna 3: syntax error

=== Resultado da Análise ===
ANALISE SINTATICA FALHOU DEVIDO A ERROS!
Total de erros léxicos: 0
Total de erros sintáticos: 0

5.8 Teste com 12 Erros Sintáticos do Arquivo *teste.c*

O arquivo de teste (**teste.c**) continha os seguintes erros:

```c
int ;                   /* Declaração de variável inválida - C */

char tabel [1][int];    /* Declaracao Erronea de Matriz - C */

void function() {       /* Passagem Sem Parametro - C */
    int num;
}

int (int a, int b) {    /* Nome de função inválido (erro logo após tipo de retorno) - C */
    return a + b;
}

int soma(int x]) { }    /* Colchetes não abertos - C */

int number = 4;         /* Declara e Atribui ao mesmo tempo - C */
```

```c
void function(int soma) {       /* Atribuição errada - C */
    int a;
    a = ;

    soma( * 5);                 /* Retorno da função invalido - C */

    if ( ) {                /* Comando IF errado - C */
        int x;
    }

    = 5;                    /* Erro genérico de comando invalido sintaticamente - C */

    while (;soma < 30) {        /* Comando WHILE inválido - C */
        soma = soma + 1;
    }
}

struct {            /* Falta de IDENTIFIER no início da instrução - C */
    float tamanho;
    float peso;
};                      /* Novamente uma declaração de variável inválida - REPETIDO */
```

Ao executar o parser sobre esse código, foram obtidas as seguintes mensagens de erro:

```
Felipe Crisóstomo@DESKTOP-Q4AQB4S MINGW64 ~/Documents/Faculdade/Compiladores/Trabalho-Compiladores/trabalho_pratico_lex (bugfix)
$ ./ExecutarCompilador.sh
Erro na linha 1, coluna 6: syntax error
Erro Sint├ítico: Declara├º├úo de vari├ível inv├ílida na linha 1, coluna 6

Erro na linha 3, coluna 19: syntax error
Erro Sint├ítico: Dimensao do array invalida na linha 3, coluna 21

Erro na linha 5, coluna 16: syntax error
Erro Sint├ítico: Lista de par├ómetros malformada na declara├º├úo de fun├º├úo na linha 7, coluna 2

Erro na linha 9, coluna 6: syntax error
Erro Sint├ítico: Fun├º├úo inexistente ou invalida apos o tipo de retorno na linha 11, coluna 2

Erro na linha 13, coluna 16: syntax error
Erro Sint├ítico: Falta de abrir colchetes na linha 13, coluna 16

Erro na linha 15, coluna 15: syntax error
Erro Sint├ítico: Inicializa├º├úo de vari├ível n├úo suportada nesta linguagem na linha 15, coluna 16
```

```
Erro na linha 19, coluna 10: syntax error
Erro Sint├ítico: Atribui├º├úo sem express├úo ├á direita na linha 19, coluna 10

Erro na linha 21, coluna 12: syntax error
Erro Sint├ítico: Argumentos invalidos no retorno da funcao na linha 21, coluna 15

Erro na linha 23, coluna 11: syntax error
Erro Sint├ítico: Condi├º├úo errada no comando IF na linha 25, coluna 6

Erro na linha 27, coluna 6: syntax error
Erro Sint├ítico: Comando invalido sintaticamente ou incompleto na linha 27, coluna 9

Erro na linha 29, coluna 13: syntax error
Erro Sint├ítico: Comando WHILE invalido na linha 31, coluna 6

Erro na linha 34, coluna 9: syntax error
Erro Sint├ítico: Nome de struct ausente na linha 37, coluna 2

Erro na linha 37, coluna 3: syntax error
Erro Sint├ítico: Declara├º├úo de vari├ível inv├ílida na linha 37, coluna 3


=== Resultado da An├ílise ===
ANALISE SINTATICA CONCLUIDA COM SUCESSO!
Total de erros l├®xicos: 0
Total de erros sint├íticos: 13
```

Comentários importantes sobre os erros encontrados:

- ***Erro 1 (linha 1, coluna 19):***

```c
int ;          /* Declaração de variável inválida */
```

  - Faltou um identificador (nome da variável)
  - O parser esperava um `IDENTIFIER`, mas encontrou o ‘`;`’
  - **Regra afetada:** `var_declaracao` (caso: `tipo_especificador error SEMICOLON`)

- ***Erro 2 (linha 3, coluna 11):***

```c
char tabel [1][int];  /* Declaracao Erronea de Matriz */
```

- Dimensão inválida no array: o parser esperava uma constante (`CONSTINT`), mas encontrou um `INT`
- **Regra afetada:** `arrayDimensao` (caso: `LEFT_BRACKET error RIGHT_BRACKET`)

- ***Erro 3 (linha 5, coluna 16):***

```C/C++
void function() {      /* Passagem Sem Parametro */
     int num;
}
```

- Depois do '`(`' fica esperando ou **`VOID`** ou algum parâmetro da regra `params_lista`, mas o próximo token é '`)`', o que não é aceito pela regra de `params`

- ***Erro 4 (linha 9, coluna 6):***

```C/C++
int (int a, int b) {     /* Nome de função inválido */
    return a + b;
}
```

- Erro logo após o tipo `int`, faltando o nome da função
- **Regra afetada:** `func_declaracao` (caso: `tipo_especificador error LEFT_PAREN params RIGHT_PAREN`

- ***Erro 5 (linha 13, coluna 16):***

```C/C++
int soma(int x]) { }    /* Colchetes não abertos */
```

- Colchetes fechando sem abrir
- **Regra afetada:** `param` (caso: `tipo_especificador IDENTIFIER error RIGHT_BRACKET`)

- ***Erro 6 (linha 15, coluna 15):***

```C/C++
int number = 4;   /* Declara e Atribui ao mesmo tempo */
```

  - A própria linguagem C- **não suporta inicialização de variáveis** na declaração
  - **Regra afetada:** `var_declaracao` (caso: `tipo_especificador IDENTIFIER ASSIGN_OP error SEMICOLON`)

- ***Erro 7 (linha 19, coluna 10):***

```C/C++
a = ;              /* Atribuição errada - C */
```

  - Expressão incompleta após o operador `=`, espera-se um identificador à direita
  - **Regra afetada:** `comando` → cai no `error SEMICOLON`

- ***Erro 8 (linha 21, coluna 12):***

```C/C++
soma( * 5);      /* Retorno da função invalido */
```

  - Chamada de função com argumento inválido: `* 5` (fator malformado)
  - **Regra afetada:** dentro de `arg_lista`, pode dar erro de "argumentos inválidos"

- ***Erro 9 (linha 23, coluna 8):***

```
C/C++
if ( ) {                /* Comando IF errado */
     int x;
}
```

    - Faltou a expressão dentro do `if`
    - **Regra afetada:** `selecao_decl` (caso: `IF LEFT_PAREN error RIGHT_PAREN comando`)

- ***Erro 10 (linha 26, coluna 5):***

```
C/C++
= 5;           /* Erro genérico de comando invalido sintaticamente */
```

    - Comando inválido começando direto com `=` sem o identificador
    - **Regra afetada:** `comando` → novamente caindo no caso `error SEMICOLON`

- ***Erro 11 (linha 28, coluna 8):***

```
C/C++
while (;soma < 30) {    /* Comando WHILE inválido */
     soma = soma + 1;
}
```

    - Faltou a expressão dentro do parênteses do while antes do `;`
    - **Regra afetada:** `iteracao_decl` (caso: `WHILE LEFT_PAREN error RIGHT_PAREN comando`)

- ***Erro 12 (linha 33, coluna 8):***

```C/C++
struct {
     float tamanho;
     float peso;
};

/* Falta de IDENTIFIER no início da instrução */
/* Novamente uma declaração de variável inválida - REPETIDO */
```

    - Faltou o nome da struct depois da palavra `struct`
    - **Regra afetada:** `tipo_especificador` (caso: `STRUCT error LEFT_BRACE varDeclList RIGHT_BRACE`)
    - Há um 13° erro ao fim da struct onde ela exige um identificador para estar completamente aceita, mas esse caí no mesmo tipo de erro da primeira linha, mas específico para o `struct`

6. Problemas Encontrados e Suas Resoluções

Aqui estão todos os grandes, e alguns moderados, problemas que tivemos ao realizar o trabalho e como conseguimos resolvê-los.

6.1. PROBLEMA COM MÚLTIPLAS FUNÇÕES `MAIN`

Ao integrar o analisador léxico (gerado pelo Flex) com o analisador sintático (gerado pelo Bison), é necessário trabalhar com dois arquivos fonte distintos: um para o léxico (`projeto.l`) e outro para a sintaxe (`projeto.y`). No entanto, cada um desses arquivos, inicialmente, pode conter uma função `main` própria, o que causa um conflito durante a etapa de linkedição, gerando o erro de múltiplas definições de `main`.

Para resolver esse problema, foi necessário remover a função `main` presente no arquivo léxico. Assim, a execução principal do programa passa a ser controlada exclusivamente pelo `main` gerado (ou manualmente implementado) no código fonte relacionado ao Bison. Dessa forma, a integração entre o analisador léxico e o sintático ocorre de forma adequada, sem conflitos de definição de ponto de entrada.

6.2. PROBLEMA COM ERROS IMPRECISOS DE LINHA E COLUNA

Durante os testes iniciais do compilador, foi observado que as mensagens de erro sintático e léxico não indicavam corretamente as posições exatas dos erros dentro do código-fonte. As informações sobre linha e coluna estavam ausentes, incorretas ou completamente inconsistentes, dificultando a depuração por parte do usuário.

Esse problema ocorria porque, por padrão, o Flex e o Bison não mantêm controle automático da posição atual de leitura (linha e coluna) durante a análise.

Como resultado, sempre que ocorria um erro, a ferramenta não sabia informar a localização exata no arquivo de entrada.

Para resolver essa limitação, foram implementadas duas variáveis globais para rastreamento de posição:

```
8   int line_number = 1;
9   int column_number = 1;
```

Figura 1.1 — Variável de line_number e column_number

O `line_number` passou a ser incrementado toda vez que o analisador léxico encontrava um caractere de nova linha (`\n`), enquanto o `column_number` era incrementado a cada caractere lido e reiniciado a cada nova linha. Além disso, o Flex foi configurado para atualizar essas variáveis dentro da regra de reconhecimento de tokens.

Com isso, todas as funções de tratamento de erro (como `yyerror` ou `yyerrok` no Bison) passaram a incluir essas informações ao gerar mensagens de erro. Agora, sempre que ocorre um erro, o compilador exibe a linha e a coluna exatas onde o problema foi detectado, oferecendo ao usuário um feedback muito mais preciso e facilitando o processo de correção de código.

6.3. PROBLEMA COM INTERRUPÇÃO DA ANÁLISE SINTÁTICA

Durante a execução do parser, foram observadas falhas graves que faziam com que a análise sintática fosse abruptamente interrompida antes da leitura completa do arquivo de entrada. Essas falhas, chamadas aqui de "falhas catastróficas", impediam que o parser continuasse processando o restante do código, comprometendo a robustez do compilador.

Após uma revisão detalhada das regras gramaticais, foi identificado que essas falhas estavam relacionadas a um tratamento inadequado dos erros de sintaxe em algumas produções. Especificamente, algumas regras não estavam

capturando corretamente situações de erro, o que causava o término prematuro da análise.

Para mitigar esse problema, as regras foram completamente revisadas e aprimoradas para que a detecção e o tratamento de erros fossem mais precisos, evitando que erros pontuais fossem interpretados como falhas catastróficas que interrompesse toda a análise. Essa melhoria também envolveu ajustes em conflitos do tipo *shift/reduce*, que contribuem para a dificuldade de tratamento dos erros e que serão abordados posteriormente.

```
25  void func() {
26      while (i < 5) {     /* Chavers Abertas -  */
27          i = i + 1;
28  }
```

Figura 1.2 — Código exemplo testado

```
Erro na linha 25, coluna 12: syntax error
Erro na linha 28, coluna 2: syntax error

=== Resultado da An├ílise ===
ANALISE SINTATICA FALHOU DEVIDO A ERROS!
Total de erros l├®xicos: 0
Total de erros sint├íticos: 8
```

Figura 1.3 — Resposta do Analisador Sintático diante a uma falha catastrófica

6.4. PROBLEMA DE SHIFT/REDUCE — IF/ELSE ENCADEADO

Durante a implementação das regras gramaticais para comandos condicionais (`if-else`), surgiu um conflito do tipo *shift/reduce* relacionado ao encadeamento de estruturas `if` e `else`. Esse problema é comum em gramáticas LL e LALR, como as utilizadas pelo Bison, especialmente quando a linguagem permite comandos `if-else` aninhados e não obriga o uso de chaves ou blocos explícitos.

O conflito ocorria porque o analisador sintático não conseguia decidir, ao encontrar um `else`, se ele deveria fazer o *shift* (associar o `else` ao `if` mais próximo) ou realizar um *reduce* (encerrar o `if` anterior).

Para resolver essa ambiguidade, foi adotada a técnica tradicional de manipulação de precedência para esse tipo de caso. Foi declarada uma precedência fictícia chamada `%nonassoc LOWER_THAN_ELSE`, e o caso específico do `if` sem `else` foi tratado com a diretiva `%prec LOWER_THAN_ELSE`. Essa configuração força o Bison a sempre preferir o *shift*, associando o `else` ao `if` mais próximo, o que segue o comportamento padrão da maioria das linguagens de programação.

6.5. PROBLEMA COM TOKENS DECLARADOS E NÃO UTILIZADOS

Durante o desenvolvimento da gramática no Bison, foi identificado que alguns tokens, como `MULTIPLY` e `MINUS`, estavam devidamente declarados na seção de tokens, mas não estavam sendo utilizados em nenhuma regra gramatical. Essa situação gera um aviso ou erro durante a compilação do parser, pois o Bison alerta sempre que um token é declarado, mas não referenciado nas produções da gramática.

Para corrigir esse problema, foi necessário criar regras gramaticais específicas que incluíssem esses tokens de maneira funcional ou mesmo apenas de forma a consumi-los, garantindo sua utilização no processo de análise sintática. Essa abordagem eliminou os avisos relacionados e assegurou que todos os tokens declarados tivessem uma função definida dentro da especificação gramatical, mesmo que na *linguagem C-*, não haja regras para a utilização destes tokens.

6.6. PROBLEMA COM GENERALIZAÇÃO EXCESSIVA EM `declaracao_lista`

Durante o desenvolvimento da gramática, foi identificado que muitos dos exemplos de código de teste estavam sendo analisados de forma incorreta, pois acabavam caindo diretamente na regra mais genérica da gramática: a produção `declaracao_lista`. Por estar posicionada em um nível superior na hierarquia da árvore sintática, essa regra assumia prioridade na captura de erros, o que dificultava a identificação precisa de onde e por que o erro realmente ocorria.

A estrutura original da regra era a seguinte:

```
56  /*----- 2º -----*/
57  declaracao_lista
58      : declaracao
59      | declaracao_lista declaracao
60      | declaracao_lista error SEMICOLON
61      { erro_sintatico_previsto("Erro Sintático: Declaracao mal formada e não reconhecida"); yyerrok; }
62      ;
```

Figura 1.4 — Regra gramatical de declaracao_lista

O problema central é que, ao incorporar a captura de erros diretamente na `declaracao_lista`, qualquer erro de sintaxe que ocorresse dentro de uma declaração individual ou em outra parte da gramática era, muitas vezes, tratado de forma genérica como um erro de "declaração mal formada", mascarando a real origem do problema.

Como consequência, foi necessário revisar a estrutura de tratamento de erros, movendo os blocos de captura (`error`) para regras mais específicas e de nível inferior na gramática. Isso permitiu uma detecção mais localizada e informativa dos erros sintáticos, facilitando a correção e o diagnóstico por parte do usuário.

7. CONCLUSÃO

A implementação de um analisador léxico com Flex demonstrou ser eficiente e prática para identificar estruturas léxicas em uma linguagem de programação simples. A clareza das expressões regulares e a flexibilidade do Flex permitem adaptações rápidas para novas linguagens. A ferramenta provou ser útil tanto em contextos acadêmicos quanto como base para compiladores reais.

A integração com o Bison ampliou ainda mais o alcance do projeto, permitindo a construção de uma análise sintática robusta e bem estruturada. Com o Bison, é possível definir regras gramaticais formais, que verificam se a sequência de tokens gerada pelo Flex segue corretamente a sintaxe da linguagem. O analisador sintático pode validar comandos, estruturas de controle, declarações e expressões, além de detectar erros de forma precisa, apresentando mensagens claras com a linha e coluna do problema.

A combinação do Flex com o Bison permite não apenas reconhecer os elementos básicos da linguagem, mas também validar sua organização estrutural, sendo uma etapa fundamental na construção de compiladores e interpretadores. Essa abordagem modular facilita a manutenção e a escalabilidade do projeto, permitindo que, futuramente, sejam adicionadas etapas como análise semântica, geração de código intermediário e otimizações. Assim, a utilização conjunta dessas ferramentas se mostra extremamente eficiente tanto no meio acadêmico quanto como base para o desenvolvimento de compiladores completos e profissionais.

8. REFERÊNCIAS BIBLIOGRÁFICAS.

AHO, A. V. et al. Compiladores: princípios, técnicas e ferramentas. [S.l.]: Addison Wesley, 2008.

Levine, J. R., Mason, T., & Brown, D. (1992). *Lex & Yacc*. O'Reilly Media.

Projeto GNU. *Flex - The Fast Lexical Analyzer*. Disponível em: https://www.gnu.org/software/flex/

GNU Project. (n.d.). *GNU Bison*. Obtido em https://www.gnu.org/software/bison/manual/