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.

2.3 ANALISADOR SEMÂNTICO E TABELA DE SÍMBOLOS

Após a análise sintática, o compilador inicia a fase de análise semântica, cuja função é verificar o programa fonte em busca de erros semânticos e coletar informações de tipo para as fases subsequentes. O componente central que apoia essa e outras fases é o gerenciador da tabela de símbolos.

2.3.1 GERENCIADOR DA TABELA DE SÍMBOLOS:

A tabela de símbolos é uma estrutura de dados utilizada para armazenar informações sobre os diversos identificadores encontrados no código-fonte. Para cada identificador, a tabela mantém atributos essenciais como seu lexema (nome), seu tipo de dado, seu escopo e, eventualmente, seu endereço relativo na memória em tempo de execução.

Uma das principais complexidades no gerenciamento da tabela de símbolos é o tratamento de **escopos**, que definem a visibilidade de um identificador dentro de diferentes blocos de código. A regra de aninhamento de blocos, presente em linguagens como C e C-, estabelece que um identificador se refere à declaração mais interna encontrada. Essa característica permite que a implementação da tabela de símbolos para escopos aninhados seja eficientemente modelada como uma **pilha**.

Ao entrar em um novo bloco, uma nova tabela para o escopo corrente é colocada no topo da pilha; ao sair do bloco, ela é removida. A busca por um identificador começa na tabela do topo (escopo atual) e prossegue para as tabelas abaixo (escopos envolventes) até que o identificador seja encontrado ou a base da pilha seja alcançada.

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.

Vale ressaltar que as gramáticas abaixo não possuem as regras semânticas, pois o foco é apenas mostrar como que a sintaxe das instruções é realizada. Para entender o desenvolvimento e funcionamento da análise semântica, basta

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 (lex.yy.c) e sintático (parser.tab.c) em linguagem C, é necessário compilá-los e vinculá-los, juntamente com os demais módulos de suporte (como a tabela de símbolos), para formar um único executável.

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

Neste comando, o compilador GCC é utilizado para compilar os arquivos gerados automaticamente pelo Flex e Bison, além do arquivo de implementação da tabela de símbolos. A opção `-o Compiler.o` define o nome do executável de saída. A flag `-lfl` é fundamental, pois realiza o vínculo com a biblioteca de tempo de execução do Flex (`libfl`), responsável por fornecer funções auxiliares utilizadas pelo analisador léxico, como `yywrap` e manipulações de entrada.

Ao final deste processo, obtém-se o executável `Compiler.o`, que representa o compilador completo da linguagem C-, integrando análise léxica, sintática e estruturas auxiliares para análise semântica.

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.

3.4 DESENVOLVIMENTO DA TABELA DE SÍMBOLOS

Para a implementação da análise semântica e posterior geração de código, foi desenvolvido um módulo de Tabela de Símbolos projetado para ser ao mesmo tempo robusto e flexível, atendendo a todos os requisitos da linguagem C-. A metodologia central foi a criação de uma estrutura de dados eficiente e uma interface de programação (API) clara para desacoplar a lógica de armazenamento da lógica de análise do parser.

3.4.1 ESTRUTURA DE DADOS E GERENCIAMENTO DE ESCOPO

A implementação da tabela de símbolos utiliza uma **tabela de hash com encadeamento separado** para garantir a eficiência na busca e inserção de

identificadores, minimizando colisões. Para o gerenciamento de escopos aninhados, fundamentais na linguagem C-, foi adotada a abordagem de **pilha de tabelas de símbolos**. Nesta arquitetura, cada escopo (`struct Scope`) contém um ponteiro para sua própria tabela de hash e um ponteiro para o escopo pai, formando uma lista ligada que opera como uma pilha.

A estrutura de dados principal, `struct Symbol`, foi desenhada para representar as diversas categorias de identificadores da linguagem. Para suportar a declaração de variáveis de tipo `struct` (ex: `struct Ponto p;`), a estrutura `var_info` foi projetada com o campo `char* struct_name` para armazenar o nome do tipo (`'Ponto'`). Adicionalmente, o campo `HashTable* members` foi incluído na estrutura.

```
C/C++
// trecho da struct Symbol em tabelaSimbolos.h
struct {
    DataType type;
    int relative_address;
    bool is_array;
    Dimension* dimensions;
    char* struct_name; // Guarda o nome do tipo struct, ex: "Ponto"
    HashTable* members; // Ponteiro para a tabela de membros da struct
} var_info;
```

Embora a gramática C- do projeto não inclua a sintaxe para acesso a membros (o uso do caractere `.` fora de um contexto numérico é um erro léxico), a inclusão do campo `members` na estrutura de dados estabelece uma base robusta que permitiria a implementação dessa funcionalidade em uma futura extensão da linguagem, servindo como um importante exercício de design.

3.4.2 INTERFACE PARA ANÁLISE SEMÂNTICA

Dada a variedade de informações necessárias para cada categoria de símbolo, foi projetada uma API com funções de inserção especializadas. Essa

abordagem torna a integração com o analisador sintático mais segura e intuitiva. A API pública oferece as seguintes funcionalidades:

- **Gerenciamento de Escopo:** Funções `open_scope()` e `close_scope()` para manipular a pilha de tabelas.
- **Busca de Símbolos:** A função `lookup_symbol(const char* name)` realiza a busca por um identificador a partir do escopo mais interno para o mais externo.
- **Inserção Especializada de Símbolos:**
    - **`insert_variable` e `insert_array`**: Funções para registrar variáveis e vetores. Suas assinaturas estão preparadas para lidar tanto com tipos primitivos quanto com tipos `struct`, recebendo o nome do tipo da `struct` como parâmetro para validação.

```
C/C++
Symbol* insert_variable(const char* name, DataType type, const char*
struct_name);
Symbol* insert_array(const char* name, DataType type, const char*
struct_name, Dimension* dims);
```

    - `insert_struct_def`: Uma função crucial para registrar a definição de um novo tipo `struct`, associando seu nome a uma tabela de hash contendo seus membros.

```
C/C++
Symbol* insert_struct_def(const char* name, HashTable* members);
```

- **Funções de Apoio Semântico:** Para centralizar a lógica de verificação e auxiliar o parser, o módulo também provê funções de apoio, como `symbols_compatible` e `verifica_argumentos`.

```C/C++
// Verifica se os tipos de dois símbolos são compatíveis para uma operação.
bool symbols_compatible(const Symbol* s1, const Symbol* s2);

// Valida se os argumentos de uma chamada de função correspondem à sua declaração.
int verifica_argumentos(Symbol *func, Node *args);
```

3.5 DESENVOLVIMENTO DA ANÁLISE SEMÂNTICA

A análise semântica é a etapa do compilador responsável por conferir a validade dos programas além da sintaxe, verificando a coerência dos tipos, declarações, usos de variáveis e funções, e regras específicas da linguagem que não podem ser garantidas apenas pela análise sintática.

3.5.1 FUNCIONAMENTO

O funcionamento básico pode ser dividido nas seguintes etapas:

3.5.1.1 PERCURSO DA ÁRVORE SINTÁTICA

A análise semântica faz um percurso (geralmente recursivo) da árvore sintática abstrata (AST), visitando cada nó para validar seu significado e coerência. Em cada nó que representa uma declaração (variável, função, struct), realiza inserção ou consulta na tabela de símbolos para registrar a existência do identificador. Em nós que representam expressões e comandos (atribuições, chamadas de função, operações aritméticas), consulta a tabela para buscar o tipo e propriedades dos identificadores envolvidos.

3.5.1.2 GERENCIAMENTO DE ESCOPOS

Quando a análise entra em um novo bloco (por exemplo, corpo de função, bloco de controle `if` ou `while`), abre um novo escopo, empilhando uma nova tabela hash. Todas as declarações feitas nesse escopo são inseridas na tabela atual, e buscas de símbolos respeitam a hierarquia — primeiro no escopo atual,

depois nos escopos ancestrais. Ao sair do bloco, o escopo é fechado, removendo a tabela do topo da pilha para garantir que os símbolos locais não fiquem acessíveis fora do seu contexto.

3.5.1.3 INSERÇÃO E CONSULTA NA TABELA DE SÍMBOLOS

- **Inserção:** Ao encontrar uma declaração, o analisador cria um símbolo (variável, função, struct, etc.) e tenta inseri-lo na tabela do escopo atual. Se já existir um símbolo com o mesmo nome nesse escopo, sinaliza erro de redeclaração.
- **Consulta:** Para uso de variáveis, funções e structs, a análise consulta a tabela a partir do escopo atual até o global, buscando o símbolo. Se não for encontrado, é erro de uso de símbolo não declarado.

3.5.1.4 VERIFICAÇÃO DE TIPOS

Em expressões, a análise consulta os tipos dos operandos e verifica se a operação é válida para esses tipos. Por exemplo, não é permitido somar um inteiro com uma struct diretamente.

- Em atribuições, verifica se o tipo do valor atribuído é compatível com o tipo da variável destino. Compatibilidades permitidas incluem, por exemplo, conversão implícita entre inteiro e float.
- Para arrays, verifica se as dimensões e o tipo do elemento são compatíveis no uso.
- Para structs, a análise verifica se o identificador pertence ao tipo struct declarado, e se o acesso aos seus membros é válido, consultando os membros definidos na tabela da struct.

3.5.1.5 VERIFICAÇÃO DE FUNÇÕES

Ao declarar uma função, registra seu nome, tipo de retorno, parâmetros e estrutura. Na chamada da função, verifica se a função está declarada. Confere se o número e tipo dos argumentos fornecidos na chamada correspondem à lista de parâmetros da função. Verifica se o valor retornado na função, em blocos `return`, corresponde ao tipo de retorno declarado.

3.5.1.6 TRATAMENTO DE ARRAYS E DIMENSÕES

- Para variáveis do tipo array, o analisador armazena uma lista encadeada de dimensões (tamanhos).
- Nas operações envolvendo arrays, verifica se os acessos aos índices estão dentro das dimensões definidas.
- Confirma compatibilidade de tipos em operações com arrays (por exemplo, passar um array como argumento para um parâmetro array compatível).

3.5.1.7 TRATAMENTO DE STRUCTS

- Ao declarar structs, cria uma tabela de símbolos própria para os membros da struct.
- Na declaração de variáveis do tipo struct, copia os membros da tabela da struct para o símbolo da variável para facilitar validações futuras.
- Em acessos a membros da struct, verifica a existência do membro e o tipo correspondente, acessando a tabela da struct.

3.5.1.8 GERAÇÃO DE MENSAGENS DE ERRO

- Ao detectar incompatibilidade, uso de símbolos não declarados ou redeclaração, a análise imprime mensagens claras indicando o erro e a localização aproximada.
- Mensagens específicas são geradas para erros comuns, como número incorreto de parâmetros em chamadas, tipos incompatíveis em expressões e usos inválidos de arrays ou structs.

3.5.1.8 GERAÇÃO DO CÓDIGO DE 3 ENDEREÇOS

- A geração de código intermediário em forma de **código de 3 endereços (three-address code – 3AC)** é uma etapa complementar à análise semântica, na qual são produzidas instruções simples e de baixo nível, que servem como ponte entre a representação sintática e a geração final de código.

- **FUNCIONAMENTO**
  - Durante a análise semântica, à medida que a árvore sintática é percorrida e verificada, são criados **nós intermediários (`Node`)** contendo informações semânticas e estruturais. Em cada nó, o campo `place` é preenchido com o nome de uma variável temporária ou identificador original, conforme a operação.
  - A geração de 3AC ocorre de forma integrada à construção desses nós e é implementada por meio da função `generate_code(Node *node)` que, de forma recursiva, gera código apenas para os nós válidos (não-`NULL`) e cujos campos semânticos foram corretamente analisados.
  - O resultado é escrito em um **arquivo separado** (`saida.3ac`) por meio do ponteiro global `code_output`.

- **PRINCIPAIS ELEMENTOS**
  - **Variáveis Temporárias**: Nomes como `t0`, `t1`, etc., são gerados com a função `new_temp()` e utilizados para armazenar resultados intermediários.
  - **Rótulos**: Rótulos como `L0`, `L1`, etc., são usados para estruturas de controle (`if`, `while`) e são criados com `new_label()`.
  - **Formato da instrução**: As instruções seguem o padrão:

```C/C++
t1 = t2 + t3
a = t4
param t5
t6 = call funcao, 2
if t7 goto L1
goto L2
L1:
```

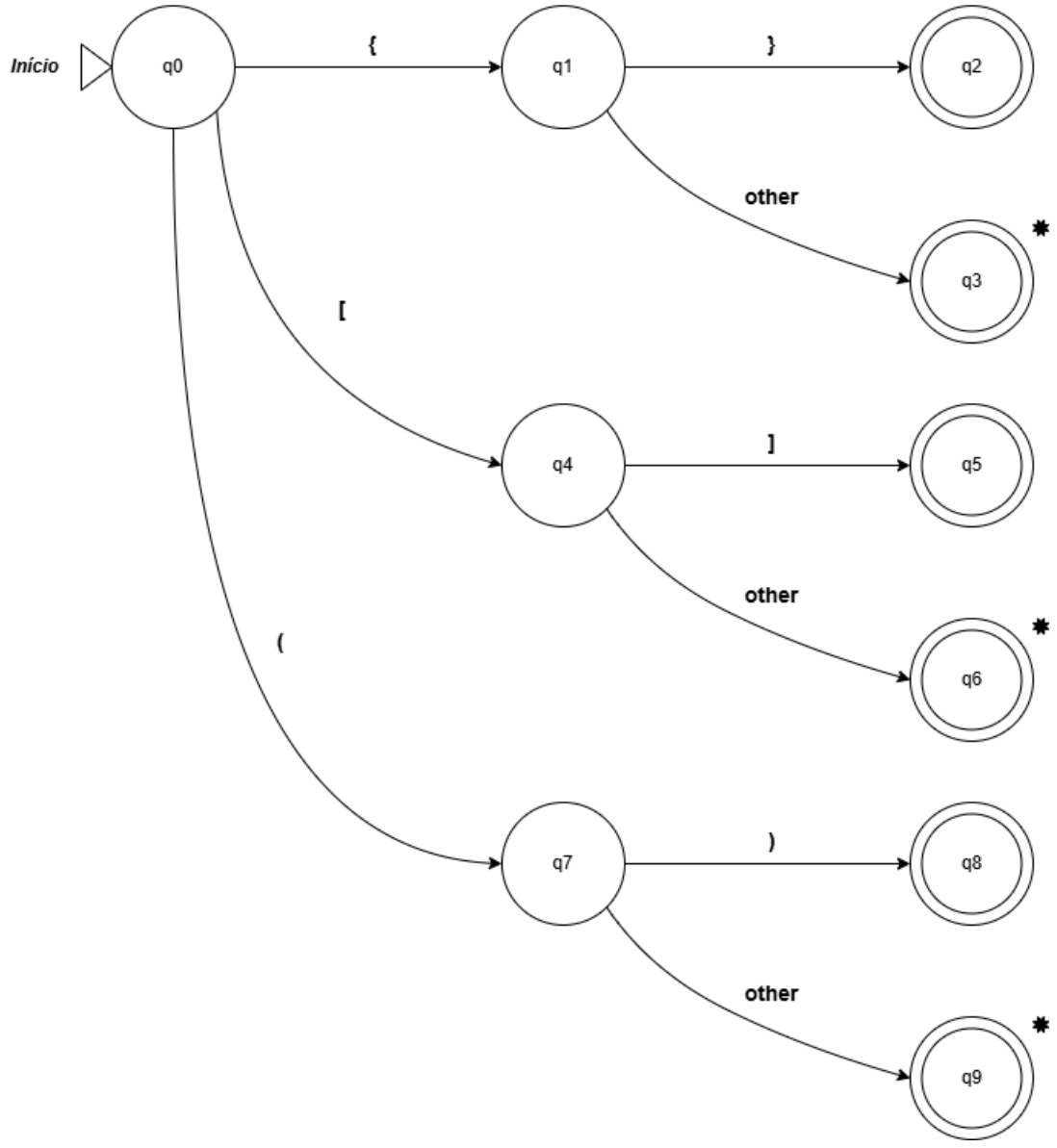
- **INTEGRAÇÃO COM A ANÁLISE**
  - A geração de código está embutida nas regras da gramática nas produções de expressões e comandos.
  - Em **expressões**, o código é gerado no momento da criação do `Node`, como em somas, subtrações, multiplicações, divisões e comparações. O resultado da expressão é armazenado em um temporário.
  - Em **atribuições**, a variável à esquerda recebe o resultado da expressão à direita.
  - Em **chamadas de função**, cada argumento é processado com `param x`, e a chamada resulta em um temporário que armazena o valor retornado.
  - Em **acessos a vetor**, são geradas instruções como `t3 = arr[t2]`.
    Em **estruturas de controle**, o código de rótulos e saltos é emitido diretamente nas regras como `if`, `else` e `while`, utilizando `fprintf(code_output, ...)`.

- **CONDIÇÕES E CONTROLE**
  - A geração do código só ocorre se o nó da árvore for válido e tiver passado pela verificação semântica. Isso evita a emissão de código para expressões mal formadas ou com símbolos inválidos. Em expressões inválidas, mensagens são impressas apenas no terminal de erro, mas o código de 3 endereços é gerado **apenas para trechos semanticamente válidos**.
  - Além disso, para evitar poluição do terminal, toda a saída do código intermediário é separada em `saida.3ac`, enquanto mensagens de erro continuam sendo emitidas em `stderr`.

4. DIAGRAMAS DE TRANSIÇÃO

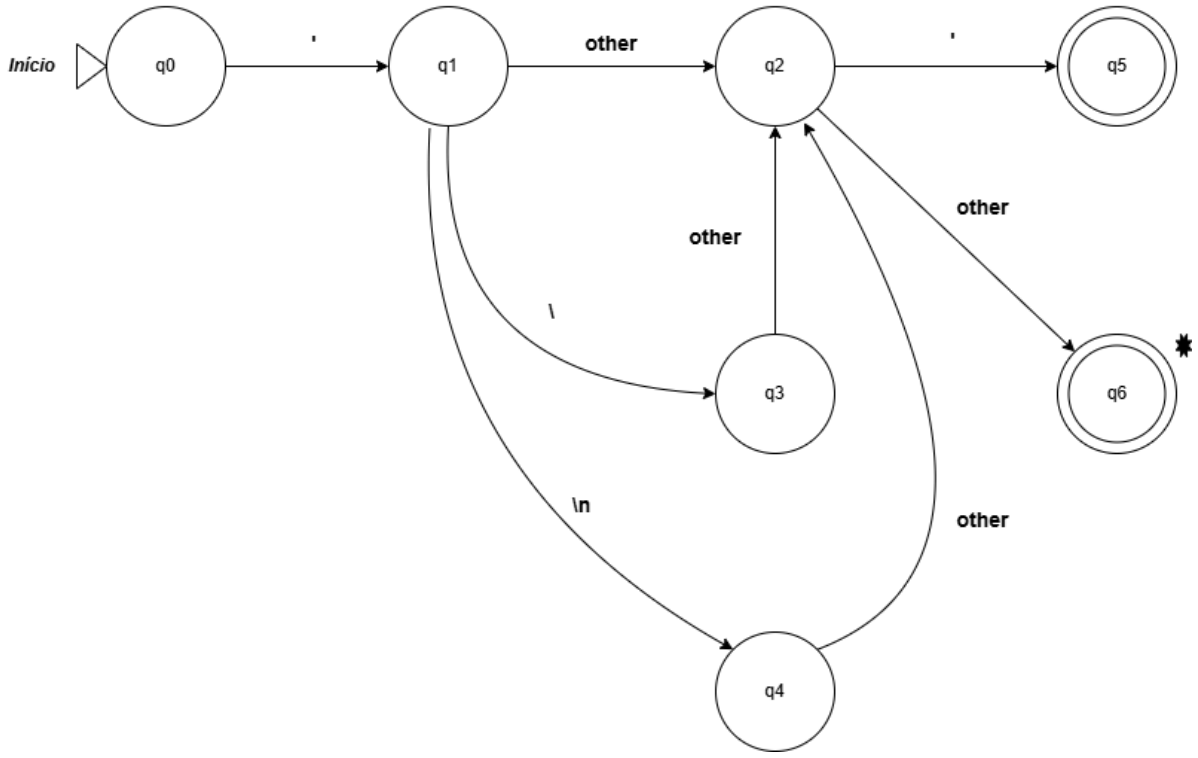
4.1 TRANSIÇÃO DE ABRE E FECHA

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

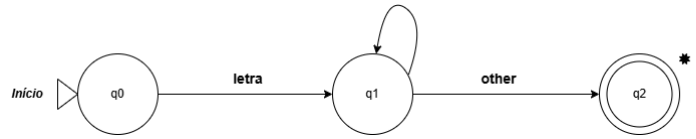


Diagrama de Transição de num

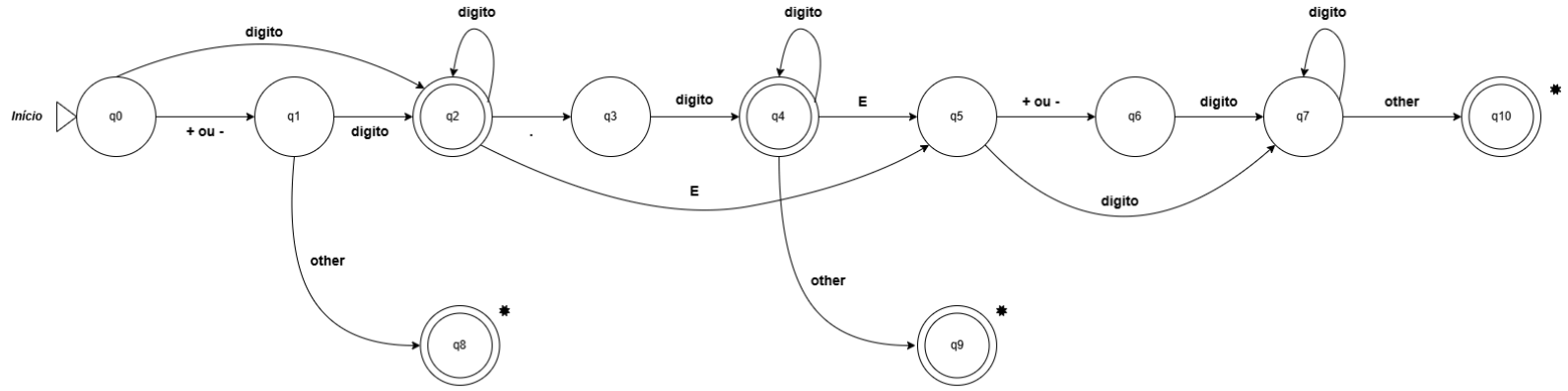
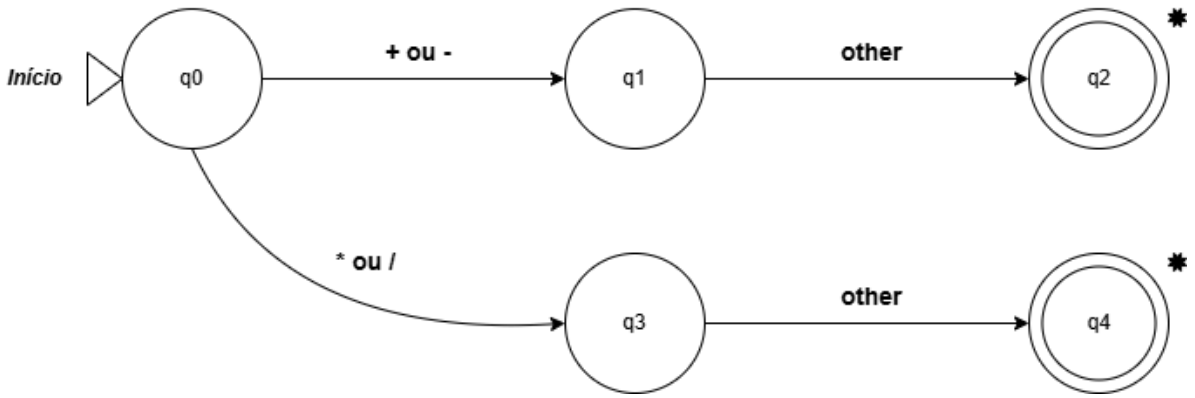


Diagrama de Transição de num_int

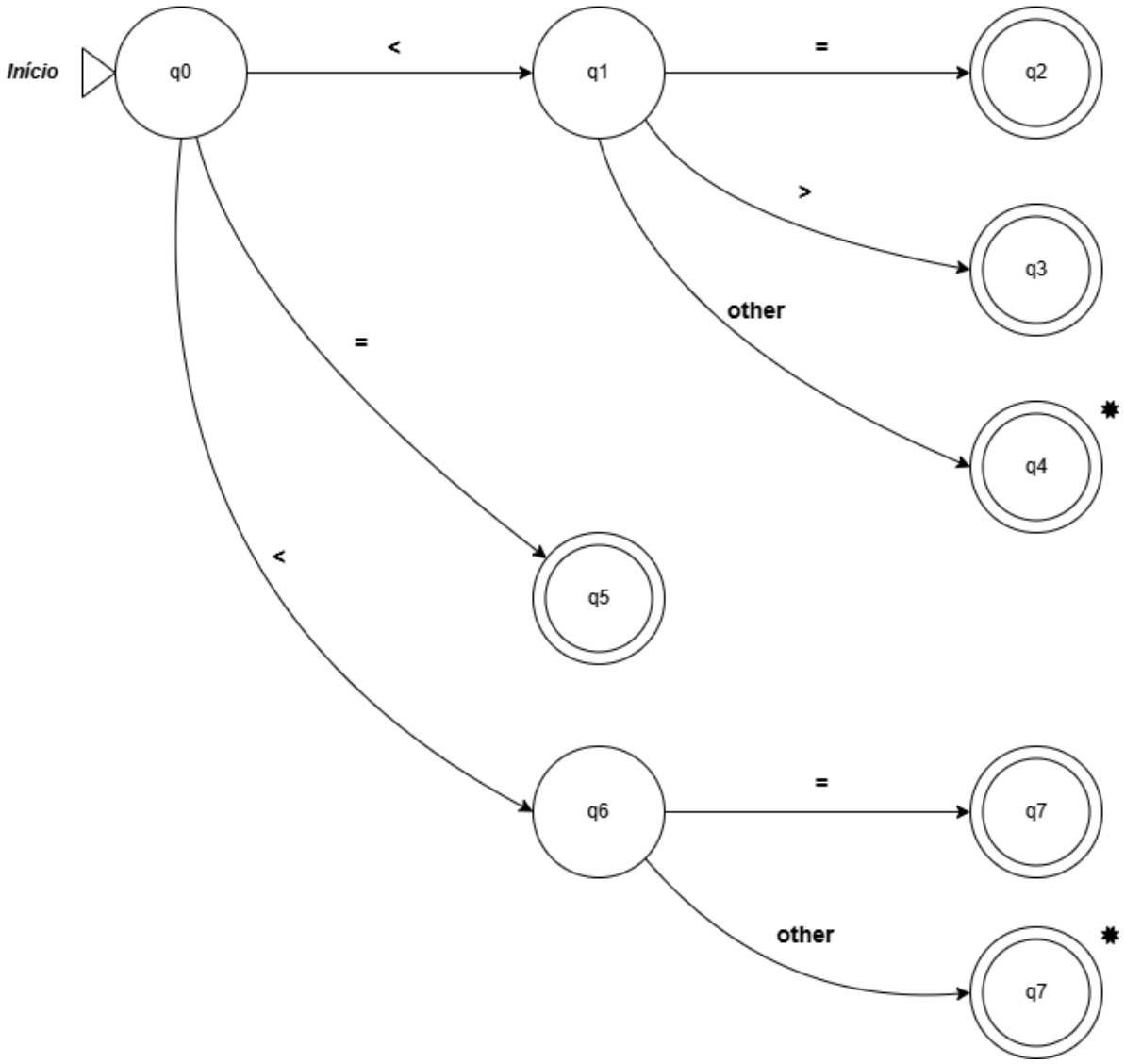
4.4 TRANSIÇÃO DE SOMA E MULTI

Diagrama de Transição de soma e multi



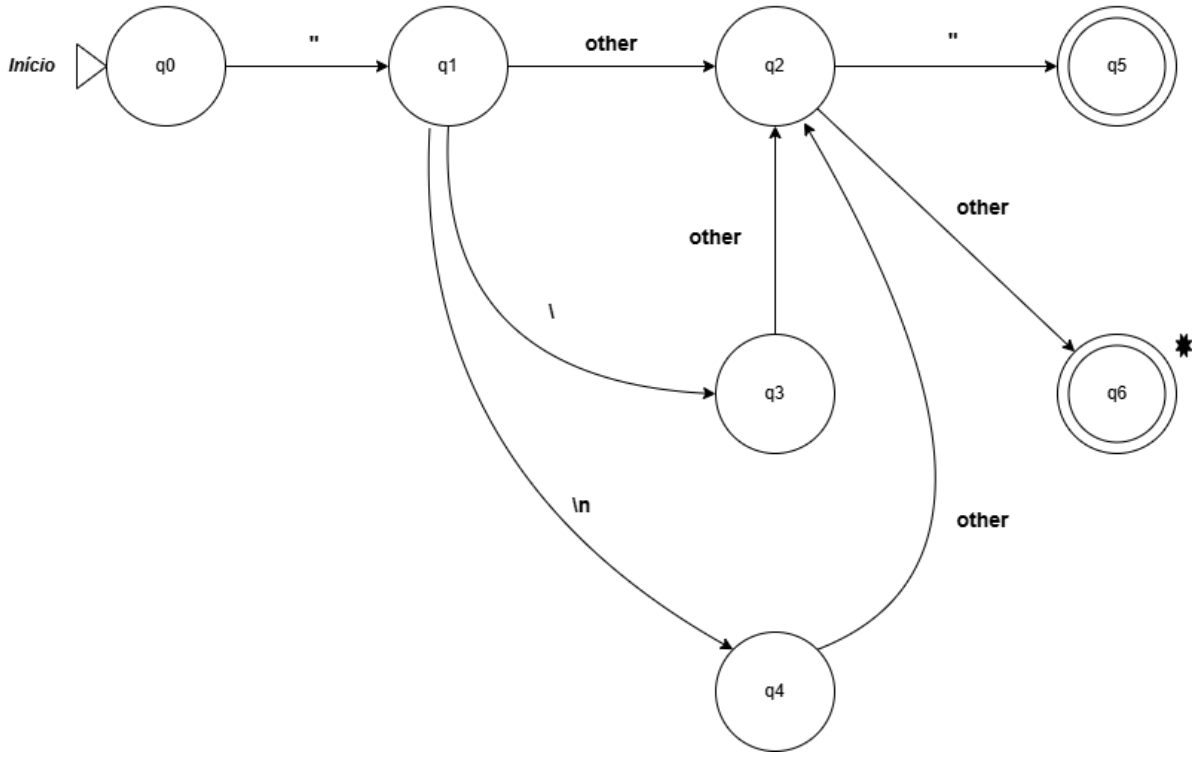
4.5 TRANSIÇÃO DE RELOP

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 CONCLUIDA COM SUCESSO!

Total de erros léxicos: 0

Total de erros sintáticos: 1

Total de erros semânticos: 0

Código de 3 Endereços Gravado em saida.3ac

Conteúdo de saida.3ac:

t0 = 0

return t0

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 CONCLUIDA COM SUCESSO!
Total de erros léxicos: 6
Total de erros sintáticos: 1
Total de erros semânticos: 0

Código de 3 Endereços Gravado em saida.3ac

Conteúdo de saida.3ac:
Nenhum código foi gerado

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

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

Código de 3 Endereços Gravado em saida.3ac
Conteúdo de saida.3ac:
t0 = x * x
return t0

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

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 CONCLUIDA COM SUCESSO!
Total de erros léxicos: 0
Total de erros sintáticos: 1
Total de erros semânticos: 0

Código de 3 Endereços Gravado em saida.3ac

Conteúdo de saida.3ac:
Nenhum código foi gerado

5.5 ENTRADA: int vet[10][];

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

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

Conteúdo de saida.3ac:
Nenhum código foi gerado

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 CONCLUIDA COM SUCESSO!

Total de erros léxicos: 0

Total de erros sintáticos: 1

Total de erros semânticos: 0

Código de 3 Endereços Gravado em saida.3ac

Conteúdo de saida.3ac:

return x

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

Erro na linha 1, coluna 3: syntax error

==== Resultado da Análise ====

ANALISE FALHOU DEVIDO A ERROS!

Total de erros léxicos: 0

Total de erros sintáticos: 0

Total de erros semânticos: 0

Código de 3 Endereços Gravado em saida.3ac

Nenhum código foi gerado

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

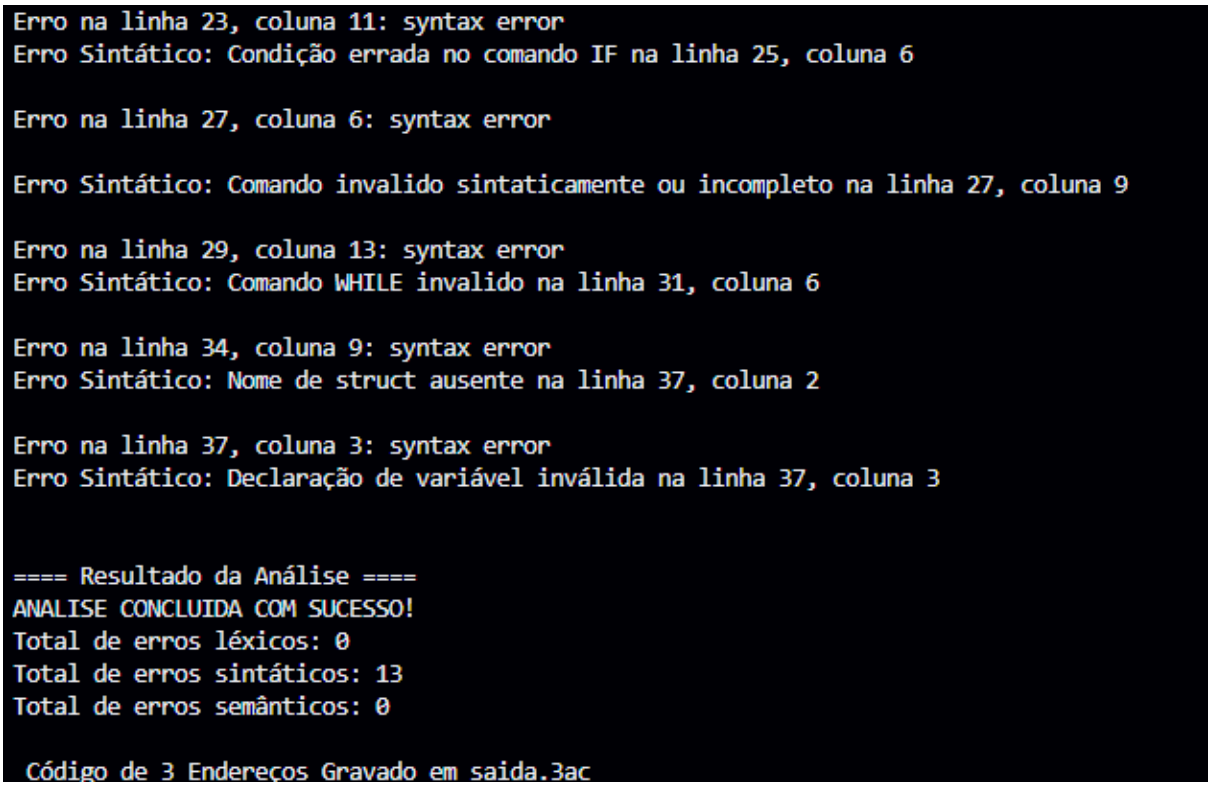
O arquivo de teste (**testeSintatico.c**) continha os seguintes erros:

```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:

```
estevao_augusto@Acer:~/Colocar_https/Trabalho-Compiladores/trabalho_pratico_lex$ ./ExecutarCompilador.sh
Erro na linha 1, coluna 6: syntax error
Erro Sintático: Declaração de variável inválida na linha 1, coluna 6

Erro na linha 3, coluna 19: syntax error
Erro Sintático: Dimensao do array invalida na linha 3, coluna 21

Erro na linha 5, coluna 16: syntax error
Erro Sintático: Lista de parâmetros malformada na declaração de função na linha 7, coluna 2

Erro na linha 9, coluna 6: syntax error
Erro Sintático: Função inexistente ou invalida apos o tipo de retorno na linha 11, coluna 2

Erro na linha 13, coluna 16: syntax error
Erro Sintático: Falta de abrir colchetes na linha 13, coluna 16

Erro na linha 15, coluna 15: syntax error
Erro Sintático: Inicialização de variável não suportada nesta linguagem na linha 15, coluna 16

Erro na linha 19, coluna 10: syntax error
Erro Sintático: Atribuição sem expressão à direita na linha 19, coluna 10

Erro na linha 21, coluna 12: syntax error
Erro Sintático: Argumentos invalidos no retorno da funcao na linha 21, coluna 15
```



Comentários importantes sobre os erros encontrados:

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

```
C/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/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`

5.9 Teste com 12 Erros Semânticos do Arquivo *testeSemantico.cm*

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

```
1   int a;
2   int b;
3   int c;
4   int arr[5];
5
6   struct exercito {
7       int qtdsoldados;
8       float forcamedia;
9   } exer;
10
11  int soma(int a, int b) {
12      return a + b;
13  }
```

```
14
15  int main(void) {
16
17      float d;
18      float multiplicacao;
19
20      qtdsoldados = 25 + 2 * 9E2;         /* 1 - Tipos incompativeis na multiplicação (2 é inteiro e 9E2 é float) */
21
22      forcamedia = 25.8;                  /* 2 - Acesso não suportado para membros de Struct */
23
24      arr = soma(b,1);                    /* 3 - 'arr' é um arranjo sem índice */
25
26      i = 0;                              /* 4 - Identificador não declarado */
27
28      d = 0;
29      while (d < 5) {                     /* 5 - Tipos incompativeis na comparação (d < 5) */
30          arr[d] = arr[d] + 1;            /* 6 - Indices de vetores devem ser inteiros */
31          d = d + 1;                      /* 7 - Tipos imcompativeis na soma (d é float e 1 é inteiro) */
32      }
33
34      b = arr[6];                         /* 8 - Indice foram dos limites da dimensao do array */
35
36      arr[4][5];                          /* 9 - Vetor com dimensões demais (arr[5]) */
37
38      d = subtracao(c,2);                 /* 10 - Funcao 'subtracao' não foi declarada */
39
40      c = multiplicacao(a,6.6);           /* 11 - multiplicacao não é um função */
41
42  }
```

```
43
44  int soma(int a, int c) {                /* 12 - Função de soma já foi declarada anteriormente */
45      return a+c;
46  }
```

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

```
Felipe Crisóstomo@DESKTOP-Q4AQB4S MINGW64 ~/Documents/Faculdade/Compiladores/Trabalho-Compiladores3.0/trabalho_pratico_lex (Codigo3Enderecos)
$ ./ExecutarCompilador.sh

Erro Sem├óntico: tipos incompat├¡veis na multiplica├º├úo (linha 20, coluna 32).

Erro Sem├óntico: Acesso a membros de struct n├úo suportado pela gram├ática (linha 22, coluna 23).

Erro Sem├óntico: Problema de Atribui├º├úo, uma das variaveis ├® um arranjo, mas nenhum indice foi colocado (linha 24, coluna 10).

Erro Sem├óntico: Identificador n├úo foi declarado (linha 26, coluna 8).

Erro Sem├óntico: Tipos incompat├¡veis na compara├º├úo (linha 29, coluna 18).

Erro Sem├óntico: ├ìndice de vetor deve ser inteiro (linha 30, coluna 17).

Erro Sem├óntico: ├ìndice de vetor deve ser inteiro (linha 30, coluna 26).

Erro Sem├óntico: tipos incompat├¡veis na soma (linha 31, coluna 19).

Erro Sem├óntico: ├ìndice [6] fora dos limites da dimens├úo 1 (tamanho = 5) do vetor 'arr' (linha 34, coluna 16).

Erro Sem├óntico: Vetor 'arr' possui dimens├Áes demais (fornecido 2, esperado 1) (linha 36, coluna 15).

Erro Sem├óntico: Fun├º├úo 'subtracao' n├úo declarada (linha 38, coluna 23).

Erro Sem├óntico: 'multiplicacao' n├úo ├® uma fun├º├úo (linha 40, coluna 29).

Erro Sem├óntico: Fun├º├úo 'soma' j├í declarada (linha 46, coluna 2).

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

 C├│digo de 3 Ender├ºos Gravado em saida.3ac
```

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

```C/C++
qtdsoldados = 25 + 2 * 9E2;
```

  - **Tipos incompatíveis na multiplicação**: o valor `2` é inteiro e `9E2` é float. A multiplicação mistura tipos sem coerção automática, e seu compilador exige tipos iguais.

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

```C/C++
forcamedia = 25.8;
```

  - **Acesso a membros de struct não suportado**: a variável `forcamedia` é membro da struct `exer`, mas foi usada diretamente como se fosse uma variável global. Seu compilador não suporta `exer.forcamedia`.

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

```C/C++
arr = soma(b,1);
```

  - **Arranjo usado sem índice**: `arr` é um vetor (`int arr[5]`), mas está sendo usado como variável comum na esquerda da atribuição. É necessário indicar um índice, como `arr[0]`.

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

```C/C++
i = 0;
```

  - **Identificador não declarado**: a variável `i` foi usada sem ser declarada. O analisador de símbolos não a reconheceu.

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

```C/C++
while (d < 5)
```

    - **Tipos incompatíveis na comparação**: `d` é `float`, `5` é `int`. Comparações lógicas no compilador exigem ambos do tipo `int`.

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

```C/C++
arr[d] = arr[d] + 1;
```

    - Há 2 erros equivalentes nessa mesma linha:
        - **Índice de vetor deve ser inteiro**: `d` é do tipo `float`, mas índices de vetores devem ser inteiros.
        - **Tipos incompatíveis na soma**: novamente, `arr[d]` é `int` e `1` é `int`, mas `d` como índice não é válido, e o erro propaga.

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

```C/C++
d = d + 1;
```

    - **Tipos incompatíveis na soma**: `d` é `float`, `1` é `int`. Sua linguagem não realiza promoção automática de tipos aritméticos.

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

```C/C++
b = arr[6];
```

    - **Índice fora dos limites do vetor**: `arr[6]` excede o limite superior do vetor `arr[5]`. O intervalo válido é `arr[0]` até `arr[4]`.

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

```C/C++
arr[4][5];
```

- **Dimensões demais**: `arr` é um vetor de uma dimensão (`int arr[5]`), mas está sendo acessado como se fosse bidimensional.

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

```C/C++
d = subtracao(c,2);
```

- **Função não declarada**: `subtracao` está sendo usada, mas nunca foi declarada nem definida anteriormente.

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

```C/C++
c = multiplicacao(a, 6.6);
```

- **Identificador não é função**: `multiplicacao` é uma variável `float`, mas está sendo usada como se fosse uma função — erro de uso indevido.

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

```C/C++
int soma(int a, int c) { return a+c; }
```

- **Função já declarada**: `soma` foi definida anteriormente na linha 13. Sua linguagem não permite redefinição da mesma função.

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.

6.7. PROJETO DA API PARA SÍMBOLOS HETEROGÊNEOS

Um desafio central no projeto da tabela de símbolos foi definir uma interface de inserção que fosse ao mesmo tempo robusta e clara, considerando que cada categoria de símbolo (variável, função, array, etc.) exige um conjunto distinto de atributos para ser completamente descrita. Uma única função de inserção genérica se mostraria excessivamente complexa, com muitos parâmetros, e propensa a erros de uso pelo parser.

A solução adotada foi a criação de uma API com funções de inserção especializadas, como `insert_variable`, `insert_array` e `insert_function`. Cada função possui uma assinatura voltada para a sua finalidade específica, exigindo apenas os parâmetros relevantes. Por exemplo, `insert_array` recebe uma lista de dimensões, algo que `insert_variable` não necessita.

```
C/C++
// Exemplo da API especializada
Symbol* insert_array(const char* name, DataType type, const char*
struct_name, Dimension* dims);
Symbol* insert_function(const char* name, DataType return_type, const char*
struct_name, Param* params);
```

Essa abordagem tornou a integração com o analisador sintático mais segura, legível e intuitiva, eliminando a ambiguidade e garantindo que cada símbolo seja inserido na tabela com todos os seus atributos corretos.

6.8. GERENCIAMENTO DE TIPOS ESTRUTURADOS E SEUS MEMBROS

O suporte ao tipo `struct` da linguagem C- apresentou um desafio de representação particular. A gramática permite a *definição* de um tipo `struct` (`struct Ponto { ... }`) e a *declaração* de variáveis desse tipo (`struct Ponto p;`), mas não contempla a sintaxe para *acesso* aos seus membros (ex: `p.x`). O desafio, portanto, foi criar um mecanismo na Tabela de Símbolos que validasse as declarações corretamente, sem implementar uma funcionalidade de

acesso que a própria linguagem não suporta. Era preciso garantir que, em `struct Ponto p;`, o tipo `Ponto` fosse, de fato, uma `struct` previamente definida.

A solução foi projetar uma Tabela de Símbolos que, embora preparada para um cenário mais completo, servisse estritamente aos requisitos da gramática.

1. **Distinção entre Definição e Declaração de struct:** O design adotado separa conceitualmente o registro da definição de um tipo `struct` da declaração de uma variável desse tipo. Uma função específica, `insert_struct_def`, foi criada para que o parser, ao reconhecer a criação de uma `struct`, possa registrá-la como um novo tipo no escopo atual.

```C/C++
// em tabelaSimbolos.c
// Função para registrar a definição de um tipo, como 'struct Ponto {...}'
Symbol* insert_struct_def(const char* name, HashTable* members) {
    Symbol* new_symbol = (Symbol*)malloc(sizeof(Symbol));
    new_symbol->name = strdup(name);
    new_symbol->kind = KIND_STRUCT_DEF;
    new_symbol->data.struct_info.members = members;
    // ... (lógica de inserção)
}
```

2. **Validação de Tipo na Declaração de Variáveis:** As funções de inserção de variáveis e vetores foram equipadas para validar a declaração de instâncias de `structs`. Elas recebem o nome do tipo `struct` como parâmetro e utilizam a função `lookup_symbol` para verificar se um símbolo com aquele nome e com a categoria `KIND_STRUCT_DEF` já existe na tabela.

```C/C++
// em tabelaSimbolos.c
Symbol* insert_variable(const char* name, DataType type, const char*
struct_name) {
    // ...
     Symbol* struct_sym = struct_name ? lookup_symbol(struct_name) :
NULL;
     if (type == TYPE_STRUCT && (!struct_sym || struct_sym->kind !=
KIND_STRUCT_DEF)) {
            return NULL; // Erro semântico: struct inválida ou não
declarada
    }
    // ... (continua a inserção)
}
```

Dessa forma, a Tabela de Símbolos cumpre seu papel de acordo com as regras da linguagem C- do projeto: ela valida que apenas tipos `struct` previamente definidos podem ser usados em declarações. Embora a infraestrutura para verificação de acesso a membros esteja presente (como o armazenamento da tabela de membros), sua utilização prática fica como um exercício de design, demonstrando como o compilador poderia ser estendido caso a gramática da linguagem fosse expandida para permitir tal operação, um cenário comum fora do contexto estritamente acadêmico deste trabalho.

6.9. COMPLEXIDADE DA MANIPULAÇÃO DA TABELA DE SÍMBOLOS E DAS FUNÇÕES AUXILIARES

A manipulação da tabela de símbolos durante a análise semântica revelou-se uma das partes mais complexas e delicadas do desenvolvimento do compilador. Cada instância da gramática exige o correto controle dos escopos para garantir que símbolos sejam inseridos e recuperados de forma consistente, respeitando regras como visibilidade, reuso e declaração única dentro de um mesmo escopo. Além disso, a diversidade de símbolos, que incluem variáveis simples, arrays, funções com seus parâmetros e definições complexas de structs, trouxe desafios adicionais, uma vez que cada tipo possui atributos e comportamentos específicos que precisam ser cuidadosamente gerenciados.

Para lidar com essa complexidade, foi fundamental a criação de um conjunto robusto de funções auxiliares, as quais são declaradas no arquivo tabelaSimbolos.h e implementadas na tabelaSimbolos.c. Essas funções encapsulam operações rotineiras, como a inserção de novos símbolos, busca e verificação de existência, clonagem de estruturas para replicar escopos ou membros de structs, além da liberação adequada da memória alocada. Com essa modularização, o código principal do analisador semântico ficou muito mais organizado, legível e fácil de manter, evitando repetição e potenciais erros que poderiam surgir da manipulação direta e dispersa das estruturas de dados.

Além da melhora na organização do código, essa abordagem promoveu também a portabilidade e a reutilização do sistema de tabela de símbolos. Ao concentrar funcionalidades específicas em funções dedicadas, foi possível criar uma estrutura genérica e flexível que pode ser adaptada a diferentes linguagens e requisitos sem grandes alterações. Isso torna o projeto escalável e facilita futuras manutenções ou extensões, como a adição de novos tipos de símbolos, regras semânticas mais complexas, ou até mesmo a integração com outras fases do compilador, como a geração de código intermediário e otimizações.

Por fim, o investimento no desenvolvimento dessa biblioteca interna de manipulação da tabela de símbolos não apenas tornou a implementação do analisador semântico mais eficiente e confiável, mas também serviu como uma base sólida para a construção das etapas seguintes do compilador, consolidando o projeto como um todo e preparando-o para possíveis evoluções futuras.

6.10. GERENCIAMENTO DE TIPOS E COMPATIBILIDADE EM EXPRESSÕES E ATRIBUIÇÕES

Um dos principais desafios enfrentados na implementação da análise semântica foi o gerenciamento adequado dos tipos de dados em diferentes contextos da linguagem, como expressões aritméticas, comparações, atribuições, chamadas de função e retorno de valores. Essa etapa é essencial para garantir que o código-fonte analisado seja coerente do ponto de vista semântico, e não apenas sintaticamente correto.

Embora a gramática definida no Bison conseguisse estruturar corretamente a linguagem, ela não era capaz, por si só, de validar se, por exemplo, uma operação entre um `int` e um `float` deveria ser permitida, se uma variável `char` podia receber um `int`, ou se uma `struct` era compatível com outra. Também foi necessário lidar com questões mais específicas, como o uso correto de arrays, verificação do número e tipo de argumentos em chamadas de função, além da consistência entre o tipo declarado de uma função e o tipo realmente retornado.

Para tratar essas verificações com clareza e reutilização, foram desenvolvidas diversas funções auxiliares no módulo da Tabela de Símbolos, como `types_compatible()` e `symbols_compatible()`. Essas funções encapsulam as regras de conversão e comparação entre tipos, tornando o código da análise semântica mais limpo, modular e reutilizável. Ao invés de duplicar lógicas de verificação ao longo do código, essas funções centralizam as regras e permitem uma manutenção mais simples e segura. Essa abordagem também melhora a portabilidade, permitindo aplicar essas verificações em diversos pontos da gramática, como em declarações, comandos de atribuição, argumentos de funções, retornos e expressões compostas.

Outro aspecto importante da análise semântica foi o uso da estrutura `Node`, que representa elementos semânticos da linguagem, como variáveis, literais, chamadas de função, operadores e seus operandos. Mesmo que a geração de código não tenha sido diretamente de nossa responsabilidade, compreender a estrutura e comportamento dos `Node` foi essencial para garantir que a análise semântica operasse corretamente. Cada `Node` carrega informações cruciais como o tipo (`type`), operador (`op`), ponteiro para símbolo (`symbol`) e flags adicionais como `is_array` ou `struct_name`. Esses campos permitiram aplicar as funções auxiliares corretamente, validando a coerência entre os nós de uma expressão e reportando erros semânticos com precisão.

Esse processo exigiu atenção ao encadeamento dos nós, já que muitas construções da linguagem são naturalmente representadas como listas ou árvores. Por exemplo, os argumentos de uma chamada de função estão ligados via ponteiro `next`, enquanto os operandos de expressões binárias são ligados via `left` e

`right`. Para garantir que cada argumento fosse verificado corretamente, funções como `verifica_argumentos()` iteram sobre essas listas usando tanto os `Node` quanto os `Param` da função alvo. Isso evidencia a necessidade de manter a estrutura `Node` bem populada e semanticamente correta logo no momento da construção da árvore, etapa anterior à verificação semântica.

No geral, o gerenciamento de tipos e compatibilidade envolveu uma integração cuidadosa entre a estrutura dos nós da árvore sintática e a lógica da tabela de símbolos, utilizando diversas funções auxiliares como suporte para validar cenários variados de forma modular, eficiente e clara. A complexidade dessa etapa reforça a importância da análise semântica como um elo entre a análise sintática e a posterior geração de código.

7. LIMITAÇÕES

Apesar de o compilador implementar com sucesso as funcionalidades centrais propostas para a linguagem C-, o processo de desenvolvimento e teste revelou uma série de limitações e pontos para otimizações futuras.

7.1 GERAÇÃO DE CÓDIGO INEFICIENTE PARA ESTRUTURAS CONDICIONAIS

Uma limitação notável reside na forma como o código de três endereços é gerado para comandos `if-else`. Atualmente, o compilador avalia as expressões de ambos os ramos, `if` e `else`, antes de realizar o salto condicional.

```C/C++
if (a < 10){
    d = d + 1.6;
} else{
    d = d + 2.0;
}
```

```C/C++
t7 = 1.600000
t8 = d + t7
d = t8 // Bloco IF é calculado incondicionalmente
t9 = 2.000000
t10 = d + t9
d = t10 // Bloco ELSE também é calculado incondicionalmente
if t6 goto L3 // Salto condicional ocorre apenas no final
goto L4
```

Esta abordagem é funcionalmente incorreta e altamente ineficiente, pois executa cálculos desnecessários. A estratégia correta, conforme apresentado na teoria, envolveria o uso de rótulos e saltos para garantir que apenas o bloco de código relevante seja executado com base na condição.

7.2 TRATAMENTO INCOMPLETO DE ARRANJOS MULTIDIMENSIONAIS

O gerador de código de três endereços atual não implementa corretamente o cálculo de endereço para arranjos com mais de uma dimensão. A lógica de acesso considera apenas o primeiro índice fornecido, ignorando os subsequentes.

```C/C++
matriz[13][29] = 4;
```

```C/C++
t20 = 13
t21 = matriz[t20] // Erro: Trata 'matriz[13]' como uma variável
t22 = 4
t21 = t22
```

O código gerado trata `matriz[13]` como um endereço final, ignorando completamente o segundo índice `[29]`. A implementação correta exigiria o cálculo do deslocamento "achatado" (flattened) em memória, usando uma fórmula como `offset = linha * LARGURA_DA_LINHA + coluna`, que ainda não foi implementada.

7.3 USO EXCESSIVO DE VARIÁVEIS TEMPORÁRIAS

A geração de código atual emprega um padrão não otimizado que resulta na criação de variáveis temporárias redundantes, especialmente para atribuições de valores constantes e cópias de variáveis.
Embora funcional, esta abordagem aumenta o número de instruções. Uma otimização local, como a propagação de cópia, otimizaria boa parte do código.

```C/C++
d = 0;
b = arr[5];
```

```
C/C++
t2 = 0
d = t2
t3 = 5
t4 = arr[t3]
b = t4
```

7.4 CONVERSÃO IMPLÍCITA DE TIPOS SEM ALERTAS

O analisador semântico permite certas conversões de tipo, como a atribuição de um float a uma variável int, mas o faz silenciosamente, sem alertar o programador sobre a potencial perda de precisão.

7.5 COBERTURA PARCIAL DE TIPOS DE DADOS

A análise semântica realizada no projeto identifica apenas erros que podem ser verificados estaticamente, ou seja, antes da execução do programa. Problemas que ocorrem apenas em tempo de execução, como divisão por zero, estouro de memória ou acessos inválidos, não são detectados nesta fase. Portanto, a análise semântica oferece uma garantia limitada e deve ser complementada por verificações em fases posteriores ou em tempo de execução.

8. CONCLUSÃO

A construção de um compilador completo, mesmo para uma linguagem de programação reduzida, envolve o domínio e a integração de múltiplas etapas da cadeia de tradução: análise léxica, análise sintática, análise semântica, gerenciamento de escopos, uso de tabelas de símbolos e geração de código intermediário. Este projeto demonstrou, de forma incremental e bem estruturada, a aplicação de todos esses conceitos fundamentais.

A análise léxica, desenvolvida com **Flex**, foi responsável por reconhecer os menores elementos significativos da linguagem – como identificadores, operadores, palavras-chave e literais – utilizando expressões regulares. A flexibilidade e a clareza das regras fornecidas pelo Flex permitiram a identificação precisa de tokens e a produção de informações relevantes, como número de linha e coluna, essenciais para a geração de mensagens de erro detalhadas e úteis.

A análise sintática, realizada com **Bison**, transformou a sequência de tokens em uma estrutura sintática válida, conforme regras formais definidas na gramática. Com o suporte de precedências, regras recursivas e produção de nós sintáticos, foi possível reconhecer e validar comandos de controle, expressões, declarações e definições de funções, organizando tudo em uma **árvore sintática abstrata (AST)**. Erros sintáticos foram tratados com recuperação e mensagens explícitas, permitindo que o compilador continuasse sua execução sem abortar na primeira falha.

A seguir, com a **análise semântica**, o compilador passou a verificar aspectos da linguagem que vão além da forma – como o uso correto de variáveis, tipos, vetores, structs, funções e parâmetros. Para isso, implementamos uma **tabela de símbolos com suporte a escopos aninhados**, que permite a definição e a busca de identificadores respeitando a hierarquia léxica do código. Foram tratados diversos aspectos semânticos: verificação de tipos em operações e atribuições, controle de uso correto de vetores e dimensões, validação de acessos a membros de `struct`, controle da assinatura e chamada de funções e detecção de variáveis não declaradas ou mal utilizadas.

A última etapa implementada foi a **geração de código intermediário na forma de código de 3 endereços (3AC)**. Esta fase traduziu os nós válidos da AST em instruções de baixo nível com variáveis temporárias e rótulos, permitindo a separação clara entre lógica e controle de fluxo. Foram geradas instruções como:

- Atribuições e operações binárias (`t1 = b + t2`);
- Acesso a vetores (`t3 = arr[t2]`);
- Chamadas de funções com parâmetros (`param t5`, `t6 = call soma, 2`);
- Comandos de controle com saltos e rótulos (`if t0 goto L1`, `goto L2`).

O código de 3 endereços foi enviado para um arquivo separado (`saida.3ac`), permitindo a separação entre o código intermediário e a exibição de mensagens de erro, o que contribui para uma organização clara da saída do compilador.

Ao longo do desenvolvimento, foram utilizadas estratégias de encapsulamento e modularização para facilitar a manutenção e a extensibilidade do compilador. A estrutura do projeto permite que futuras etapas – como otimizações ou geração de código final (que não foram pedidas no Trabalho Prático) – sejam acopladas sem a necessidade de retrabalhos significativos.

Portanto, o projeto alcançou seu objetivo de simular todas as fases iniciais de um compilador real, desde a leitura do código-fonte até a geração de uma representação intermediária robusta, abrindo caminho para transformações mais avançadas e execução real do código. A experiência prática com **Flex e Bison**, aliada à construção manual da **AST, da tabela de símbolos e do verificador semântico**, consolidou o entendimento de cada fase e mostrou a importância de sua interação dentro do compilador.

9. 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/