Universidade Federal de São Paulo
Instituto de Ciência e Tecnologia
Compiladores - Turma I

Compilador para a Linguagem C-

Integrantes:
168813 Enzo de Almeida Belfort Rizzi Di Chiara
168880 João Pedro da Silva Zampoli

Docente responsável:
Prof. Dr. Rodrigo Colnago Contreras

São José dos Campos
2025/2

Universidade Federal de São Paulo
Instituto de Ciência e Tecnologia
Compiladores - Turma I

Compilador para a Linguagem C-

Relatório do projeto de Compiladores (Turma I) da Universidade Federal de São Paulo para cumprimento dos requisitos de avaliação e aprovação na disciplina.

São José dos Campos
2025/2

Abstract

Este relatório documenta o desenvolvimento de um compilador para a linguagem C- como projeto final da disciplina de Compiladores, implementado utilizando as ferramentas Flex (analisador léxico) e Bison (analisador sintático). O projeto abrange a construção da Árvore Sintática Abstrata (AST), de uma tabela de símbolos e a geração de código intermediário no formato de três endereços.

Contents

1 Introdução

A linguagem C- é um subconjunto da linguagem C, definida no livro *Compiler Construction: Principles and Practice* de Kenneth C. Louden. Esta linguagem possui apenas dois tipos de dados (`int` e `void`), utiliza vetores, funções com parâmetros e estruturas de controle básicas (`if-else` e `while`).

O compilador desenvolvido realiza as seguintes etapas:

1. **Análise Léxica**: Reconhecimento de tokens (implementado com Flex)
2. **Análise Sintática**: Validação da estrutura gramatical e construção da AST (Bison)
3. **Análise Semântica**: Verificação de tipos e escopos
4. **Geração de Código Intermediário**: Produção de código de três endereços

2 Scanner (Léxico)

O analisador léxico foi implementado utilizando a ferramenta Flex, responsável por reconhecer os tokens da linguagem C- e passá-los ao parser. Com base no Louden, temos que a definição das convenções léxicas do C- são as seguintes:

A.1 CONVENÇÕES LÉXICAS DE C–

1. As palavras-chave da linguagem são as seguintes:

   ```
   else if int return void while
   ```

   Todas as palavras-chave são reservadas e devem ser escritas com caixa baixa.
2. Os símbolos especiais são os seguintes:

   ```
   + - * / < <= > >= == != = ; , () [] {} /* */
   ```

3. Há, ainda, os marcadores ***ID*** e ***NUM***, definidos pelas expressões regulares a seguir:

   ```
   ID = letra letra*
   NUM = dígito dígito*
   letra = a|..|z|A|..|Z
   dígito = 0|..|9
   ```

   Existe diferença entre caixa baixa e caixa alta.

Figure 1: Definições feitas por Louden em seu livro

2.1 Estrutura do Scanner

O scanner reconhece as seguintes categorias de tokens:

- **Palavras reservadas**: `else`, `if`, `int`, `return`, `void`, `while`
- **Operadores aritméticos**: `+`, `-`, `*`, `/`
- **Operadores relacionais**: `<`, `<=`, `>`, `>=`, `==`, `!=`
- **Símbolos especiais**: `;`, `,`, `(`, `)`, `[`, `]`, `{`, `}`

- **Identificadores**: sequências de letras
- **Números**: sequências de dígitos

2.2 Definições Regulares

```
digit       [0-9]
number      {digit}+
letter      [a-zA-Z]
identifier  {letter}+
newline     \n
whitespace  [ \t]+
```

Listing 1: Definições regulares em Flex

2.3 Regras de Reconhecimento

```
/* Palavras reservadas */
"else"          { return ELSE; }
"if"            { return IF; }
"int"           { return INT; }
"return"        { return RETURN; }
"void"          { return VOID; }
"while"         { return WHILE; }

/* Operadores */
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return TIMES; }
"/"             { return DIV; }
"<="            { return LESSEQUAL; }
">="            { return GREATEREQUAL; }
"=="            { return EQUAL; }
"!="            { return NOTEQUAL; }
"<"             { return LESSTHAN; }
">"             { return GREATERTHAN; }
"="             { return ASSIGN; }

/* Numeros e identificadores */
{number}        {
                  yylval.num = atoi(yytext);
                  return NUM;
                }
{identifier}    {
                  yylval.id = strdup(yytext);
                  return ID;
                }
```

Listing 2: Principais regras do scanner

2.4 Tratamento de Comentários

Os comentários em C- seguem o formato `/* ... */` e podem ocupar múltiplas linhas. O scanner implementa um autômato para consumir todo o conteúdo do comentário:

```
"/*"              {
                    char c;
                    int comment_line = linha;
                    do {
                      c = input();
                      if (c == EOF) {
                        fprintf(stderr,
                          "ERRO LEXICO: comentario nao fechado - LINHA: %d
    \n",
                          comment_line);
                        return ERROR;
                      }
                      if (c == '\n') linha++;
                    } while (c != '*' || (c = input()) != '/');
                  }
```

Listing 3: Tratamento de comentários multilinha

2.5 Tratamento de Erros Léxicos

Caracteres não reconhecidos geram erro léxico com indicação da linha:

```
.                 {
                    fprintf(stderr,
                      "ERRO LEXICO: '%s' - LINHA: %d\n", yytext, linha);
                    return ERROR;
                  }
```

Listing 4: Tratamento de caracteres inválidos

2.6 Controle de Linha

O scanner mantém uma variável global `linha` que é incrementada a cada caractere de nova linha encontrado, permitindo que mensagens de erro indiquem a localização precisa do problema:

```
int linha = 1;  /* Variavel global */

{newline}       { linha++; }
{whitespace}    { /* pular espaco em branco */ }
```

Listing 5: Controle de número de linha

3 Tabela de Símbolos

A tabela de símbolos é responsável por armazenar informações sobre identificadores (variáveis, funções, parâmetros e arrays) durante a compilação, permitindo verificações de escopo e tipo.

3.1 Estrutura de Dados

3.1.1 Tipos e Classes de Símbolos

```c
typedef enum {
    TIPO_INT,   // Tipo inteiro
    TIPO_VOID,  // Tipo void
    TIPO_ERRO   // Tipo invalido/erro
} Tipo;

typedef enum {
    SIMP_VAR,    // Variavel simples
    SIMP_ARRAY,  // Array
    SIMP_FUNC,   // Funcao
    SIMP_PARAM   // Parametro de funcao
} Classe;
```

Listing 6: Enumerações para tipos e classes de símbolos

3.1.2 Estrutura do Símbolo

```c
typedef struct Simbolo {
    char *nome;             // Nome do identificador
    Tipo tipo;              // Tipo (int ou void)
    Classe classe;          // Classe (var, array, func, param)
    int tamanho;            // Tamanho do array (0 se nao for array)
    int linha;              // Linha de declaracao
    struct Simbolo *prox;   // Proximo simbolo na lista
} Simbolo;
```

Listing 7: Estrutura de um símbolo na tabela

3.1.3 Estrutura do Escopo

A tabela utiliza uma pilha de escopos para gerenciar variáveis locais e globais:

```c
typedef struct Escopo {
    Simbolo *simbolos;      // Lista de simbolos do escopo
    struct Escopo *pai;     // Escopo pai (para busca hierarquica)
    struct Escopo *prox;    // Proximo escopo na lista historica
    int id;                 // Identificador unico do escopo
} Escopo;
```

Listing 8: Estrutura de um escopo

3.2 Operações da Tabela de Símbolos

3.2.1 Gerenciamento de Escopo

```c
void entra_escopo(void) {
    Escopo *novo = malloc(sizeof(Escopo));
    novo->simbolos = NULL;
    novo->pai = escopo_atual;
    novo->prox = NULL;
    novo->id = contador_escopos++;
    escopo_atual = novo;

    // Adiciona ao historico para impressao final
    if (todos_escopos == NULL) {
        todos_escopos = novo;
```

```
        ultimo_escopo = novo;
    } else {
        ultimo_escopo->prox = novo;
        ultimo_escopo = novo;
    }
}

void sai_escopo(void) {
    if (escopo_atual != NULL) {
        escopo_atual = escopo_atual->pai;
    }
}
```

Listing 9: Funções de entrada e saída de escopo

3.2.2 Inserção de Símbolos

```
void insere_simbolo(char *nome, Tipo tipo, Classe classe,
                    int tamanho, int linha_val) {
    // Verifica redeclaracao no escopo atual
    for (Simbolo *sim = escopo_atual->simbolos;
         sim != NULL; sim = sim->prox) {
        if (strcmp(sim->nome, nome) == 0) {
            fprintf(stderr,
                "ERRO SEMANTICO: identificador '%s' "
                "ja declarado neste escopo - LINHA: %d\n",
                nome, linha_val);
            exit(1);
        }
    }

    // Cria e insere novo simbolo
    Simbolo *novo = (Simbolo*)malloc(sizeof(Simbolo));
    novo->nome = strdup(nome);
    novo->tipo = tipo;
    novo->classe = classe;
    novo->linha = linha_val;
    novo->tamanho = tamanho;
    novo->prox = escopo_atual->simbolos;
    escopo_atual->simbolos = novo;
}
```

Listing 10: Função de inserção com verificação de redeclaração

3.2.3 Busca de Símbolos

A busca percorre a pilha de escopos do atual até o global, implementando as regras de escopo da linguagem C-:

```
Simbolo *busca_simbolo(char *nome) {
    // Percorre do escopo atual ate o global
    for (Escopo *esc = escopo_atual; esc != NULL; esc = esc->pai) {
        for (Simbolo *sim = esc->simbolos; sim != NULL; sim = sim->prox)
    {
            if (strcmp(sim->nome, nome) == 0) {
                return sim;
            }
```

```
        }
    }
    return NULL;  // Simbolo nao encontrado
}
```

Listing 11: Função de busca hierárquica de símbolos

3.3 Inicialização com Funções Pré-definidas

O escopo global é inicializado com as funções `input()` e `output()` conforme especificado na linguagem C-:

```
// No inicio da analise semantica
entra_escopo();  // Cria escopo global
insere_simbolo("input", TIPO_INT, SIMP_FUNC, 0, 0);
insere_simbolo("output", TIPO_VOID, SIMP_FUNC, 0, 0);
```

Listing 12: Inserção das funções pré-definidas

3.4 Impressão da Tabela

```
void imprime_tabela(void) {
    printf("\n=== TABELA DE SIMBOLOS (Completa) ===\n");
    printf("%-15s %-10s %-10s %-10s %-10s\n",
           "Nome", "Tipo", "Classe", "Tamanho", "Linha");

    Escopo *esc = todos_escopos;
    while (esc != NULL) {
        if (esc->simbolos != NULL) {
            printf("\n--- Escopo %d ---\n", esc->id);
            Simbolo *s = esc->simbolos;
            while (s != NULL) {
                printf("%-15s %-10s %-10s %-10d %-10d\n",
                       s->nome,
                       tipo_str(s->tipo),
                       classe_str(s->classe),
                       s->tamanho,
                       s->linha);
                s = s->prox;
            }
        }
        esc = esc->prox;
    }
}
```

Listing 13: Função de impressão formatada

3.5 Exemplo de Saída

Para um programa com função `soma` e `main`:

```
=== TABELA DE SIMBOLOS (Completa) ===
Nome            Tipo       Classe     Tamanho    Linha

--- Escopo 0 ---
main            void       func       0          8
```

```
soma            int         func        0           1
output          void        func        0           0
input           int         func        0           0

--- Escopo 1 ---
b               int         param       0           1
a               int         param       0           1

--- Escopo 2 ---
resultado       int         var         0           10
x               int         var         0           9
```

Listing 14: Exemplo de saída da tabela de símbolos

4 Parser (Sintático) AST

4.1 Estrutura da Gramática

A gramática da linguagem C- foi implementada seguindo as 29 regras definidas por Louden. A gramática foi levemente adaptada com tratamento especial para o problema do *dangling else*.

4.1.1 Principais Regras Gramaticais

```
/* 1. programa -> declaracao-lista */
programa
    : declaracao_lista
        { arvoreSintatica = $1; }
    ;

/* 6. fun-declaracao -> tipo-especificador ID ( params ) composto-decl
   */
fun_declaracao
    : tipo_especificador ID LEFTPAREN params RIGHTPAREN composto_decl
        {
            $$ = newNode(NODE_FUN_DECL);
            $$->filhos[0] = $1;
            $$->data.name = $2;
            $$->filhos[1] = $4;
            $$->filhos[2] = $6;
        }
    ;

/* 15. selecao-decl -> if ( expressao ) statement [else statement] */
selecao_decl
    : IF LEFTPAREN expressao RIGHTPAREN statement %prec LOWER_THAN_ELSE
        {
            $$ = newNode(NODE_IF);
            $$->filhos[0] = $3;
            $$->filhos[1] = $5;
            $$->filhos[2] = NULL;
        }
    | IF LEFTPAREN expressao RIGHTPAREN statement ELSE statement
        {
            $$ = newNode(NODE_IF);
```

```
            $$->filhos[0] = $3;
            $$->filhos[1] = $5;
            $$->filhos[2] = $7;
        }
    ;
```

Listing 15: Regras principais da gramática em Bison

4.2 Estrutura da AST

A Árvore Sintática Abstrata foi implementada utilizando uma estrutura de dados em C que permite representar todos os tipos de nós necessários para a linguagem C-.

4.2.1 Tipos de Nós

```
typedef enum {
    NODE_PROGRAM,        // Programa principal
    NODE_VAR_DECL,       // Declaracao de variavel
    NODE_ARRAY_DECL,     // Declaracao de array
    NODE_FUN_DECL,       // Declaracao de funcao
    NODE_PARAM,          // Parametro simples
    NODE_ARRAY_PARAM,    // Parametro array
    NODE_COMPOUND,       // Bloco composto { }
    NODE_IF,             // Estrutura if-else
    NODE_WHILE,          // Estrutura while
    NODE_RETURN,         // Comando return
    NODE_ASSIGN,         // Atribuicao
    NODE_OP,             // Operacao aritmetica (+, -, *, /)
    NODE_RELOP,          // Operacao relacional (<, <=, >, >=, ==, !=)
    NODE_VAR,            // Variavel
    NODE_ARRAY_ACCESS,   // Acesso a array
    NODE_CALL,           // Chamada de funcao
    NODE_NUM,            // Numero literal
    NODE_TYPE_INT,       // Tipo int
    NODE_TYPE_VOID       // Tipo void
} NodeType;
```

Listing 16: Enumeração dos tipos de nós da AST

4.2.2 Estrutura do Nó

```
typedef struct AST {
    NodeType type;           // Tipo do no
    union {
        int num;             // Para NODE_NUM
        char* name;          // Para identificadores
        char op;             // Para operadores (+, -, *, /)
        char* relop;         // Para operadores relacionais
    } data;
    struct AST* filhos[4];   // Ate 4 filhos por no
    struct AST* irmao;       // Proximo irmao (para listas)
    int lineno;              // Linha no codigo fonte
} AST;
```

Listing 17: Estrutura de um nó da AST

4.2.3 Justificativa da Estrutura

- **4 filhos**: Número máximo necessário para representar qualquer construção da gramática C-. Por exemplo, `NODE_FUN_DECL` usa: tipo (0), parâmetros (1), corpo (2).
- **Irmão**: Permite representar listas de elementos no mesmo nível hierárquico (declarações, parâmetros, argumentos) sem usar arrays dinâmicos.
- **Union para dados**: Economia de memória, já que cada nó usa apenas um tipo de dado.

4.3 Tratamento de Erros Sintáticos

O compilador implementa mensagens de erro descritivas seguindo o formato especificado:

```
void yyerror(const char* s) {
    errosSintaticos++;

    if (strstr(s, "unexpected") != NULL) {
        char* unexpected = strstr(s, "unexpected");
        char* expecting = strstr(s, "expecting");

        if (unexpected != NULL) {
            char tokenInesperado[64] = "";
            char tokenEsperado[256] = "";

            sscanf(unexpected, "unexpected %63[^,\n]", tokenInesperado);

            if (expecting != NULL) {
                strcpy(tokenEsperado, expecting + 10);
                char* nl = strchr(tokenEsperado, '\n');
                if (nl) *nl = '\0';
            }

            if (strlen(tokenEsperado) > 0) {
                fprintf(stderr,
                    "ERRO SINTATICO: token inesperado '%s', esperado '%s
    ' - LINHA: %d\n",
                    tokenInesperado, tokenEsperado, lineno);
            }
        }
    }
}
```

Listing 18: Função de tratamento de erros sintáticos

4.3.1 Exemplos de Mensagens de Erro

Erro no Código	Mensagem Gerada
Falta de ponto e vírgula	`token inesperado 'VOID', esperado 'SEMICOLON'`
Falta de parêntese	`token inesperado 'LEFTBRACE', esperado 'ID'`
Expressão inválida	`token inesperado 'PLUS', esperado 'NUM or ID'`
If sem parênteses	`token inesperado 'ID', esperado 'LEFTPAREN'`

Table 1: Exemplos de mensagens de erro sintático

4.4 Visualização da AST com GraphViz

O compilador gera automaticamente um arquivo `arvore.dot` que pode ser convertido em imagem usando o GraphViz.

```
void generateDotFile(AST* tree, const char* filename) {
    FILE* fp = fopen(filename, "w");

    fprintf(fp, "digraph AST {\n");
    fprintf(fp, "  graph [rankdir=TB, splines=ortho];\n");
    fprintf(fp, "  node [shape=box, fontname=\"Courier\"];\n");

    nodeCounter = 0;
    generateDotNodes(fp, tree);

    fprintf(fp, "}\n");
    fclose(fp);
}
```

Listing 19: Geração do arquivo DOT para visualização

Os nós são coloridos de acordo com seu tipo para facilitar a visualização:

- **Dourado**: Declarações de função
- **Verde claro**: Declarações de variáveis/arrays
- **Azul claro**: Estruturas de controle (if, while)
- **Lilás**: Operadores
- **Ciano**: Variáveis

5 Analisador Semântico

O analisador semântico percorre a AST para verificar a correção semântica do programa, garantindo que as regras da linguagem C- sejam respeitadas.

5.1 Verificações Implementadas

O analisador semântico realiza as seguintes verificações conforme especificado por Louden:

1. **Declaração de identificadores**: Todo identificador deve ser declarado antes de ser usado
2. **Unicidade de declarações**: Não pode haver redeclaração no mesmo escopo
3. **Função main**: Deve existir, ser do tipo `void main(void)` e ser a última declaração do programa
4. **Tipos de variáveis**: Variáveis e parâmetros não podem ser do tipo `void`
5. **Compatibilidade de tipos**: Operações e atribuições devem ter tipos compatíveis
6. **Retorno de funções**: Funções `int` devem ter `return` com valor; funções `void` não podem retornar valor
7. **Uso correto de arrays**: Acesso a array requer índice inteiro
8. **Chamadas de função**: Verificação de funções `input()` e `output()`

5.2 Estrutura do Analisador

5.2.1 Variáveis de Controle

```
static Tipo tipo_funcao_atual = TIPO_ERRO;  // Tipo da funcao sendo
    analisada
static int funcao_tem_return = 0;           // Flag de return encontrado
static int main_encontrada = 0;             // Flag de main declarada
static int dentro_funcao = 0;               // Flag de contexto
static int escopo_funcao_criado = 0;        // Evita escopo duplicado
```

Listing 20: Variáveis globais do analisador semântico

5.2.2 Função Principal

```
void semanticAnalysis(AST* root) {
    entra_escopo();  // Escopo global

    // Insere funcoes pre-definidas
    insere_simbolo("input", TIPO_INT, SIMP_FUNC, 0, 0);
    insere_simbolo("output", TIPO_VOID, SIMP_FUNC, 0, 0);

    main_encontrada = 0;
    checkNode(root);  // Percorre a AST

    // Verifica se main() foi declarada
    if (!main_encontrada) {
        fprintf(stderr,
            "ERRO SEMANTICO: funcao 'main' nao declarada\n");
        exit(1);
    }

    // Verifica se main e a ultima declaracao
    AST* ultimo = root;
    while (ultimo->irmao != NULL) {
        ultimo = ultimo->irmao;
```

```
    }
    if (ultimo->type != NODE_FUN_DECL ||
        strcmp(ultimo->data.name, "main") != 0) {
        fprintf(stderr,
            "ERRO SEMANTICO: 'main' deve ser a ultima declaracao\n");
        exit(1);
    }
}
```

Listing 21: Função principal da análise semântica

5.3 Verificação de Declarações

5.3.1 Declaração de Funções

```
case NODE_FUN_DECL: {
    Tipo tipoFunc = getAstType(node->filhos[0]);
    insere_simbolo(node->data.name, tipoFunc, SIMP_FUNC, 0, node->linha)
    ;

    // Verificacoes especiais para main
    if (strcmp(node->data.name, "main") == 0) {
        main_encontrada = 1;

        if (tipoFunc != TIPO_VOID) {
            semanticError("funcao 'main' deve retornar void",
                          node->linha);
        }
        if (!isVoidParam(node->filhos[1])) {
            semanticError("funcao 'main' nao deve ter parametros",
                          node->linha);
        }
    }

    tipo_funcao_atual = tipoFunc;
    funcao_tem_return = 0;
    dentro_funcao = 1;

    entra_escopo();
    escopo_funcao_criado = 1;

    checkNode(node->filhos[1]);  // parametros
    checkNode(node->filhos[2]);  // corpo

    sai_escopo();

    // Verifica return obrigatorio para funcao int
    if (tipo_funcao_atual == TIPO_INT && !funcao_tem_return) {
        semanticError("funcao do tipo 'int' deve ter return",
                      node->linha);
    }
    break;
}
```

Listing 22: Verificação de declaração de função

5.3.2 Declaração de Variáveis

```
case NODE_VAR_DECL:
    if (getAstType(node->filhos[0]) == TIPO_VOID) {
        semanticError("variavel nao pode ser do tipo void",
                      node->linha);
    }
    insere_simbolo(node->data.name, getAstType(node->filhos[0]),
                   SIMP_VAR, 0, node->linha);
    break;

case NODE_ARRAY_DECL:
    if (getAstType(node->filhos[0]) == TIPO_VOID) {
        semanticError("array nao pode ser do tipo void",
                      node->linha);
    }
    insere_simbolo(node->data.name, getAstType(node->filhos[0]),
                   SIMP_ARRAY, node->filhos[1]->data.num,
                   node->linha);
    break;
```

Listing 23: Verificação de declaração de variáveis

5.4 Verificação de Expressões

5.4.1 Uso de Variáveis

```
case NODE_VAR: {
    Simbolo* s = busca_simbolo(node->data.name);
    if (s == NULL) {
        semanticErrorId(node->data.name, node->linha);
    }
    if (s->classe == SIMP_FUNC) {
        semanticError("funcao usada como variavel", node->linha);
    }
    return s->tipo;
}

case NODE_ARRAY_ACCESS: {
    Simbolo* s = busca_simbolo(node->data.name);
    if (s == NULL) {
        semanticErrorId(node->data.name, node->linha);
    }
    if (s->classe != SIMP_ARRAY) {
        semanticError("variavel nao e um array", node->linha);
    }
    if (checkExpr(node->filhos[0]) != TIPO_INT) {
        semanticError("indice de array deve ser int", node->linha);
    }
    return s->tipo;
}
```

Listing 24: Verificação de uso de variáveis e arrays

5.4.2 Operações Aritméticas e Relacionais

```
case NODE_OP: {
    Tipo t1 = checkExpr(node->filhos[0]);
    Tipo t2 = checkExpr(node->filhos[1]);
    if (t1 != TIPO_INT || t2 != TIPO_INT) {
        semanticError("operacao aritmetica exige operandos int",
                      node->linha);
    }
    return TIPO_INT;
}

case NODE_RELOP: {
    Tipo t1 = checkExpr(node->filhos[0]);
    Tipo t2 = checkExpr(node->filhos[1]);
    if (t1 != TIPO_INT || t2 != TIPO_INT) {
        semanticError("operacao relacional exige operandos int",
                      node->linha);
    }
    return TIPO_INT;
}
```

Listing 25: Verificação de tipos em operações

5.5 Verificação de Chamadas de Função

```
case NODE_CALL: {
    Simbolo* s = busca_simbolo(node->data.name);
    if (s == NULL) {
        semanticErrorId(node->data.name, node->linha);
    }
    if (s->classe != SIMP_FUNC) {
        semanticError("identificador nao e uma funcao", node->linha);
    }

    // Verificacoes para output()
    if (strcmp(node->data.name, "output") == 0) {
        int numArgs = countParams(node->filhos[0]);
        if (numArgs != 1) {
            semanticError("output() requer exatamente 1 argumento",
                          node->linha);
        }
        if (node->filhos[0] != NULL) {
            if (checkExpr(node->filhos[0]) != TIPO_INT) {
                semanticError("argumento de output() deve ser int",
                              node->linha);
            }
        }
    }

    // Verificacoes para input()
    if (strcmp(node->data.name, "input") == 0) {
        if (countParams(node->filhos[0]) != 0) {
            semanticError("input() nao recebe argumentos",
                          node->linha);
        }
    }

    return s->tipo;
```

```
}
```

Listing 26: Verificação de chamadas de função

5.6 Verificação de Comandos de Controle

```
case NODE_RETURN:
    if (!dentro_funcao) {
        semanticError("return fora de funcao", node->linha);
    }

    if (node->filhos[0] != NULL) {
        Tipo t = checkExpr(node->filhos[0]);
        if (tipo_funcao_atual == TIPO_VOID) {
            semanticError("return com valor em funcao void",
                          node->linha);
        }
        if (t != tipo_funcao_atual && tipo_funcao_atual != TIPO_ERRO) {
            semanticError("tipo de retorno incompativel", node->linha);
        }
        funcao_tem_return = 1;
    } else {
        if (tipo_funcao_atual == TIPO_INT) {
            semanticError("funcao int deve retornar valor",
                          node->linha);
        }
    }
    break;
```

Listing 27: Verificação de return

5.7 Mensagens de Erro

```
static void semanticError(const char* msg, int linha) {
    fprintf(stderr, "ERRO SEMANTICO: %s - LINHA: %d\n", msg, linha);
    exit(1);
}

static void semanticErrorId(const char* id, int linha) {
    fprintf(stderr,
        "ERRO SEMANTICO: identificador '%s' nao declarado - LINHA: %d\n"
    ,
        id, linha);
    exit(1);
}
```

Listing 28: Funções de tratamento de erro semântico

5.8 Testes de Erros Semânticos

Arquivo	Erro Testado	Detectado
erro_main_param.c	main com parâmetros	Sim
erro_void_retorna_valor.c	função void com return valor	Sim
erro_int_sem_return.c	função int sem return	Sim
erro_return_fora.c	return fora de função	Sim
erro1.c	variável não declarada	Sim

Table 2: Testes de detecção de erros semânticos

6 Código Intermediário (3 endereços)

6.1 Estrutura das Quádruplas

O código intermediário é representado através de quádruplas, seguindo o formato clássico de três endereços.

```
typedef enum {
    OP_ADD,          // t = a + b
    OP_SUB,          // t = a - b
    OP_MUL,          // t = a * b
    OP_DIV,          // t = a / b
    OP_ASSIGN,       // t = a
    OP_ASSIGN_ARR,   // t[i] = a
    OP_ARR_ACCESS,   // t = a[i]
    OP_LT,           // t = a < b
    OP_LE,           // t = a <= b
    OP_GT,           // t = a > b
    OP_GE,           // t = a >= b
    OP_EQ,           // t = a == b
    OP_NE,           // t = a != b
    OP_LABEL,        // L:
    OP_GOTO,         // goto L
    OP_IF_FALSE,     // if_false a goto L
    OP_PARAM,        // param a
    OP_CALL,         // t = call f, n
    OP_RETURN,       // return a
    OP_FUNC_START,   // inicio de funcao
    OP_FUNC_END,     // fim de funcao
    OP_HALT          // fim do programa
} QuadOp;

typedef struct Quadruplas {
    QuadOp op;
    char* arg1;
    char* arg2;
    char* result;
    struct Quadruplas* next;
} Quadruplas;
```

Listing 29: Estrutura de uma quádrupla

6.2 Geração de Temporários e Labels

```c
int tempCount = 0;
int labelCount = 0;

char* newTemp(void) {
    char* temp = (char*) malloc(16);
    sprintf(temp, "t%d", tempCount++);
    return temp;
}

char* newLabel(void) {
    char* label = (char*) malloc(16);
    sprintf(label, "L%d", labelCount++);
    return label;
}

void emit(QuadOp op, char* arg1, char* arg2, char* result) {
    Quadruplas* quad = (Quadruplas*) malloc(sizeof(Quadruplas));
    quad->op = op;
    quad->arg1 = arg1 ? strdup(arg1) : NULL;
    quad->arg2 = arg2 ? strdup(arg2) : NULL;
    quad->result = result ? strdup(result) : NULL;
    quad->next = NULL;

    // Adiciona a lista de quadruplas
    if (quadList == NULL) {
        quadList = quad;
        quadTail = quad;
    } else {
        quadTail->next = quad;
        quadTail = quad;
    }
}
```

Listing 30: Funções auxiliares para geração de código

6.3 Geração de Código para Estruturas de Controle

6.3.1 Estrutura IF-ELSE

```c
case NODE_IF:
    t1 = genCode(node->filhos[0]);  // condicao
    label1 = newLabel();
    emit(OP_IF_FALSE, t1, NULL, label1);
    genCode(node->filhos[1]);  // bloco then
    if (node->filhos[2] != NULL) {
        // tem else
        label2 = newLabel();
        emit(OP_GOTO, NULL, NULL, label2);
        emit(OP_LABEL, NULL, NULL, label1);
        genCode(node->filhos[2]);  // bloco else
        emit(OP_LABEL, NULL, NULL, label2);
    } else {
        emit(OP_LABEL, NULL, NULL, label1);
    }
```

```
    break;
```

Listing 31: Geração de código para if-else

Exemplo de tradução:

```
// Codigo C-:                    // Codigo Intermediario:
if (n <= 1) {                     t1 = n <= 1
    return 1;                     if_false t1 goto L0
} else {                          return 1
    return n * fatorial(n-1);     goto L1
}                                 L0:
                                  t2 = n - 1
                                  param t2
                                  t3 = call fatorial, 1
                                  t4 = n * t3
                                  return t4
                                  L1:
```

Listing 32: Código C- e seu código intermediário correspondente

6.3.2 Estrutura WHILE

```
case NODE_WHILE:
    label1 = newLabel();  // inicio do loop
    label2 = newLabel();  // fim do loop
    emit(OP_LABEL, NULL, NULL, label1);
    t1 = genCode(node->filhos[0]);  // condicao
    emit(OP_IF_FALSE, t1, NULL, label2);
    genCode(node->filhos[1]);  // corpo
    emit(OP_GOTO, NULL, NULL, label1);
    emit(OP_LABEL, NULL, NULL, label2);
    break;
```

Listing 33: Geração de código para while

Exemplo de tradução:

```
// Codigo C-:                    // Codigo Intermediario:
while (i < 10) {                  L2:
    y[i] = i * 2;                 t7 = i < 10
    i = i + 1;                    if_false t7 goto L3
}                                 t8 = i * 2
                                  y[i] = t8
                                  t9 = i + 1
                                  i = t9
                                  goto L2
                                  L3:
```

Listing 34: While e seu código intermediário

6.4 Geração de Código para Chamadas de Função

```
case NODE_CALL:
    // Gera codigo para os argumentos
    genArgs(node->filhos[0]);
    // Conta argumentos
    sprintf(numStr, "%d", countArgs(node->filhos[0]));
```

```
    // Gera chamada
    temp = newTemp();
    emit(OP_CALL, node->data.name, numStr, temp);
    return temp;

static void genArgs(AST* argList) {
    if (argList == NULL) return;

    AST* arg = argList;
    while (arg != NULL) {
        char* argVal = genCode(arg);
        emit(OP_PARAM, argVal, NULL, NULL);
        arg = arg->irmao;
    }
}
```

Listing 35: Geração de código para chamadas de função

6.5 Exemplo Completo de Saída

Para o programa de teste:

```
int soma(int a, int b) {
    return a + b;
}

void main(void) {
    int x;
    int resultado;
    x = 5;
    resultado = soma(x, 10);
}
```

Listing 36: Programa C- de exemplo

O código intermediário gerado é:

```
  0: func_start soma
  1: t0 = a + b
  2: return t0
  3: func_end soma
  4: func_start main
  5: x = 5
  6: param x
  7: param 10
  8: t1 = call soma, 2
  9: resultado = t1
 10: func_end main
```

Listing 37: Código intermediário gerado

7 Testes Realizados

7.1 Testes de Aceitação

Arquivo	Descrição	Resultado
teste.c	Programa completo com funções e arrays	Passou
ok.c	Programa simples válido	Passou

Table 3: Testes de programas válidos

7.2 Testes de Rejeição (Erros)

Arquivo	Erro Testado	Detectado
erro_falta_pv.c	Falta ponto e vírgula	Sim
erro_falta_paren.c	Falta parêntese	Sim
erro_falta_chave.c	Falta chave de fechamento	Sim
erro_expr_invalida.c	Expressão malformada	Sim
erro_if_sem_paren.c	If sem parênteses	Sim
erro_call_sem_paren.c	Chamada sem fecha parêntese	Sim

Table 4: Testes de detecção de erros sintáticos

8 Participação dos Membros

Membro	Contribuições
Enzo de Almeida Belfort Rizzi Di Chiara	- Implementação do parser (Bison) - Estrutura da AST - Tratamento de erros sintáticos - Geração de código intermediário - Testes e documentação
João Pedro da Silva Zampoli	- Scanner (Flex) - Integração dos módulos - Tabela de Símbolos - Analisador Semântico - Testes e documentação

Table 5: Divisão de tarefas entre os membros do grupo

9 Conclusão

O desenvolvimento do compilador para a linguagem C- se mostrou completo e eficiente, gerando, de maneira clara e correta, como saída: uma tabela de símbolos, uma árvore sintática e o código intermediário baseado em 3 endereços. Além disso, a compilação de diversos casos de testes com erros léxicos, sintáticos e semânticos, provou a capacidade do compilador de indicar erros eficientemente.

Referências

1. LOUDEN, Kenneth C. *Compiler Construction: Principles and Practice*. PWS Publishing Company, 1997.

2. Documentação do Bison: `https://www.gnu.org/software/bison/manual/`

3. Documentação do Flex: `https://westes.github.io/flex/manual/`

4. Graphviz - Graph Visualization Software: `https://graphviz.org/documentation/`