Bianca Valenciani - RA: 168763
Felipe Soares Calderaro - RA: 168843
Isabella Maria Ribeiro - RA: 168870

Relatório Técnico: Implementação de um Compilador para a Linguagem C-Minus

São José dos Campos - Brasil

Dezembro de 2025

Bianca Valenciani - RA: 168763
Felipe Soares Calderaro - RA: 168843
Isabella Maria Ribeiro - RA: 168870

Relatório Técnico: Implementação de um Compilador para a Linguagem C-Minus

Relatório apresentado à Universidade Federal de São Paulo como parte de uma avaliação na disciplina de Compiladores.

Docente: Prof. Dr. Rodrigo Colnago Contreras

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Dezembro de 2025

Resumo

Este relatório apresenta o desenvolvimento de um compilador para a linguagem C- (C-Minus), um subconjunto didático da linguagem C, atendendo aos requisitos da disciplina de Compiladores. O projeto abrange as etapas de *front-end* da compilação: análise léxica, análise sintática e análise semântica. Foram utilizadas as ferramentas Flex e Bison para a geração dos analisadores, integradas através da linguagem C. As principais contribuições deste trabalho incluem a implementação de uma Tabela de Símbolos baseada em tabela hash com suporte a escopos aninhados, a construção da Árvore Sintática Abstrata (AST) no formato DOT, permitindo a visualização gráfica da estrutura do código fonte através da ferramenta Graphviz e a adoção de estratégias robustas de tratamento de erros, como o *Panic Mode Recovery*, bem como a geração do código de 3 Endereços. Os resultados demonstram a capacidade do compilador em validar a estrutura gramatical e as regras semânticas da linguagem, gerando saídas estruturadas para futuras etapas de geração de código.

Palavras-chaves: Compiladores, C-Minus, Flex, Bison, Análise Sintática, Tabela de Símbolos, Graphviz, AST, Código de 3 Endereços.

Sumário

1 Introdução

A construção de compiladores é uma das áreas fundamentais da Computação, unindo teoria de linguagens formais, estruturas de dados avançadas e arquitetura de computadores. O presente projeto tem como objetivo principal o desenvolvimento de um compilador funcional para a linguagem **C- (C-Minus)**, um subconjunto simplificado da linguagem C padrão, conforme especificado na obra de Kenneth C. Louden, referência clássica na área de construção de compiladores 1.

O escopo deste trabalho abrange as fases de *front-end* do processo de compilação, incluindo a análise léxica (scanner), a análise sintática (parser), a construção da Árvore Sintática Abstrata (AST) e a análise semântica com verificação de tipos e escopo. O objetivo final é transformar o código-fonte de entrada em uma representação estruturada e validada, gerando, assim, um código de 3 Endereços.

Para a implementação, foram utilizadas as ferramentas **Flex** (para geração do analisador léxico) e **Bison** (para o analisador sintático), integradas através da linguagem de programação **C**. A escolha dessas ferramentas deve-se à sua robustez, eficiência e ampla adoção na indústria e academia para o processamento de linguagens livres de contexto, além de indicação em sala pelo Professor.

Este relatório detalha as decisões de projeto, as estruturas de dados implementadas (como a Tabela de Símbolos baseada em *Hash Table*) e as estratégias de recuperação de erros adotadas para garantir a estabilidade do compilador.

2 Análise Léxica (Scanner)

A análise léxica é a primeira etapa do processo de compilação, responsável por ler o fluxo de caracteres do código-fonte e agrupá-los em unidades significativas chamadas *tokens*, conforme descrito por 1.

2.1 Implementação com Flex

O scanner foi implementado utilizando a ferramenta **Flex**. Foram definidas Expressões Regulares (Regex) para identificar os padrões da linguagem C-, tais como:

- **Palavras Reservadas:** `if`, `else`, `int`, `return`, `void`, `while`.
- **Identificadores:** Sequências alfanuméricas iniciadas por letras (id).
- **Números:** Sequências de dígitos inteiros (numero).
- **Símbolos Especiais:** Operadores aritméticos (`+`, `-`, `*`, `/`), relacionais (`<`, `<=`, `> >=`, `==`, `!=`) e Bloculares (`; , , , )`, `(, [, ], { , }`)

2.2 Tratamento de Comentários e Linhas

Um desafio específico foi o tratamento de comentários no estilo C (`/* ... */`), que podem abranger múltiplas linhas. Implementou-se uma máquina de estados manual dentro do Flex para consumir o conteúdo dos comentários sem gerar tokens, garantindo ao mesmo tempo a contagem correta das quebras de linha (`\n`) para fins de relatório de erros.

2.3 Detecção de Erros Léxicos

O scanner possui uma regra de "captura geral"(catch-all) que identifica qualquer caractere não pertencente ao alfabeto da linguagem (como `@` ou `$`), reportando imediatamente um erro léxico formatado com a linha da ocorrência, conforme requisito do projeto.

3 Análise Sintática (Parser)

A análise sintática verifica se a sequência de tokens gerada pelo scanner obedece à gramática livre de contexto da linguagem C-. Utilizamos o **Bison** para gerar um analisador *bottom-up* LALR(1).

3.1 Gramática e Resolução de Conflitos

A gramática foi descrita em notação BNF (*Backus-Naur Form*). Durante o desenvolvimento, foi necessário resolver o clássico conflito de *Shift/Reduce* conhecido como **"Dangling Else"** (Else Pendurado).

Para solucionar a ambiguidade de a qual `if` um `else` pertence, utilizamos as diretivas de precedência do Bison (`%nonassoc`), forçando o analisador a associar o `else` ao `if` mais próximo, comportamento padrão em linguagens como C e Java.

3.2 Recuperação de Erros (Panic Mode)

Para cumprir o requisito de robustez e clareza nas mensagens de erro, foram implementadas estratégias clássicas de recuperação de erros descritas na literatura de compiladores, em especial o modo de recuperação por pânico (*Panic Mode Recovery*) apresentado por 1:

1. A diretiva `%define parse.error verbose` (ou manipulação manual da string de erro) para exibir mensagens detalhadas como "*token inesperado 'X', esperado 'Y'*".
2. O modo de recuperação de pânico (*Panic Mode Recovery*) utilizando o token especial `error` do Bison. Isso permite que o compilador, ao encontrar um erro sintático, descarte tokens até encontrar um ponto de sincronização seguro (como um ponto e vírgula `;`), retomando a análise para identificar possíveis erros subsequentes.

4 Tabela de Símbolos e Análise Semântica

A Tabela de Símbolos é a estrutura central para a validação semântica, armazenando informações sobre variáveis e funções.

4.1 Estrutura de Dados

Implementou-se uma **Tabela Hash** com tratamento de colisões por encadeamento (listas ligadas). Cada entrada na tabela armazena:

- Nome do identificador;
- Escopo de declaração;
- Tipo de dado (`int` ou `void`);
- Categoria (`var`, `fun`, `vet`);
- Localização na memória (*offset*);
- Lista de linhas onde o identificador foi referenciado.

4.2 Gerenciamento de Escopo

O compilador gerencia múltiplos escopos (Global e Local de funções). A função de busca (`lookup`) verifica primeiramente o escopo local atual e, caso não encontre o símbolo, recorre ao escopo global. Isso permite o sombreamento de variáveis (variáveis locais com mesmo nome de globais), conforme permitido pela linguagem.

4.2.0.0.1 Escopos de Blocos Aninhados

Além do escopo global e do escopo associado a cada função, o compilador implementa escopos adicionais para blocos internos delimitados por chaves (`{` e `}`), como aqueles introduzidos por comandos condicionais (`if-else`) e laços de repetição (`while`). Cada novo bloco gera um escopo único identificado por um sufixo incremental (`_B1`, `_B2`, etc.), concatenado ao escopo imediatamente superior.

Essa estratégia permite representar com precisão o escopo léxico da linguagem C-, garantindo que variáveis declaradas em blocos internos sejam visíveis apenas dentro de seus respectivos contextos, sem interferir em declarações externas. O controle explícito de

blocos também facilita a detecção de erros semânticos relacionados ao uso indevido de identificadores fora de seu escopo válido.

4.2.0.0.2 Sombreamento de Identificadores

O modelo de escopo adotado permite o sombreamento de variáveis, ou seja, a declaração de identificadores com o mesmo nome em escopos distintos e aninhados. Nesses casos, a busca por um símbolo prioriza o escopo mais interno ativo no momento da análise. Caso o identificador não seja encontrado localmente, a busca prossegue recursivamente pelos escopos externos, até alcançar o escopo global.

Esse comportamento é compatível com a semântica da linguagem C e foi validado por meio de casos de teste específicos, garantindo que variáveis locais possam sobrescrever temporariamente variáveis externas sem causar conflitos ou ambiguidades.

4.3 Verificações Semânticas

O analisador semântico percorre a árvore ou atua durante o parsing para validar regras como:

- Uso de variáveis não declaradas.
- Redeclaração de variáveis no mesmo escopo.
- Verificação da existência e assinatura da função `main`.
- Incompatibilidade de tipos (ex: operações aritméticas com `void`).

5 Árvore Sintática Abstrata (AST)

A Árvore Sintática Abstrata (AST) representa a estrutura hierárquica simplificada do programa, eliminando detalhes sintáticos irrelevantes e destacando a organização semântica do código, conforme definido por 1. Foi definida uma estrutura genérica `TreeNode` no arquivo `globals.h`, capaz de representar declarações, comandos e expressões.

A árvore utiliza ponteiros para **Filhos** (`child`) para representar o aninhamento lógico (ex: um `if` tem como filhos a condição, o bloco *then* e o bloco *else*) e ponteiros para **Irmãos** (`sibling`) para representar sequências de comandos ou declarações.

Em vez de uma representação puramente textual, implementou-se no módulo `util.c` a geração de código na linguagem **DOT**.

A função percorre a árvore recursivamente e imprime nós e arestas formatados para a ferramenta **Graphviz**.

- **Nós:** Identificados unicamente pelo seu endereço de memória, garantindo que a estrutura do grafo seja fiel à estrutura de dados em C.
- **Arestas Sólidas:** Representam a relação Pai-Filho (ex: `If -> Condition`).
- **Arestas Tracejadas:** Representam a relação entre Irmãos (`sibling`), indicando a sequência de comandos.

Essa abordagem permite gerar diagramas visuais (PNG/PDF) da árvore, tornando a análise da estrutura sintática muito mais intuitiva.

6 Geração de Código Intermediário (Três Endereços)

A geração de código intermediário tem como objetivo transformar a representação estrutural do programa, obtida por meio da Árvore Sintática Abstrata (AST), em uma sequência linear de instruções de três endereços. Essa representação intermediária facilita etapas posteriores do processo de compilação, como otimizações e geração de código alvo, além de simplificar a análise do fluxo de controle.

6.0.0.0.1 Relação entre Escopo e Código Intermediário

Embora a geração do código intermediário utilize apenas os nomes dos identificadores, o controle de escopo é realizado previamente durante a análise semântica, por meio da Tabela de Símbolos. Dessa forma, cada identificador utilizado no código de três endereços já foi validado quanto à sua declaração, visibilidade e tipo, assegurando que o código intermediário represente fielmente a semântica do programa original.

A separação entre os escopos lógicos e a geração linear do código intermediário simplifica o processo de tradução, permitindo que variáveis com o mesmo nome em escopos distintos coexistam sem ambiguidades, uma vez que sua resolução ocorre antes da etapa de geração.

O processo de geração é realizado por meio da travessia da AST, de forma recursiva, respeitando a hierarquia das construções sintáticas da linguagem C-. Para cada expressão aritmética ou relacional, são criadas instruções que utilizam variáveis temporárias, normalmente denotadas por `t1`, `t2`, `t3`, entre outras. Essas instruções seguem o formato geral:

$$t_i = operando_1 \; op \; operando_2$$

Estruturas de controle, como comandos condicionais (`if-else`) e laços de repetição (`while`), são traduzidas por meio da criação de rótulos (*labels*) e instruções de desvio condicional e incondicional, como `ifFalse` e `goto`. Dessa forma, o fluxo lógico do programa é preservado na forma linear do código intermediário.

De acordo com os requisitos do projeto, o código intermediário gerado não passa por etapas de otimização. Seu objetivo principal é representar fielmente a semântica do programa-fonte, servindo como base para futuras extensões do compilador.

7 Resultados e Testes

Nesta seção, apresentamos os resultados obtidos com o compilador desenvolvido. Foram realizados testes de análise léxica, sintática, semântica e geração de código intermediário.

7.1 Teste de Erros Léxicos, Sintáticos e Semânticos

Com o objetivo de avaliar a robustez do compilador, foram realizados testes específicos para cada classe de erro tratada pelo *front-end*: erros léxicos, sintáticos e semânticos. Em todos os casos, o compilador foi capaz de identificar corretamente o problema, indicando a linha da ocorrência e a natureza do erro, além de permitir a continuidade da análise quando aplicável.

7.1.1 Erro Léxico

O erro léxico ocorre quando o analisador encontra um caractere que não pertence ao alfabeto da linguagem. A Figura 1 apresenta a saída do compilador ao processar um código-fonte contendo o caractere inválido `@`. O trecho de código responsável por esse erro é apresentado no Código 7.1.

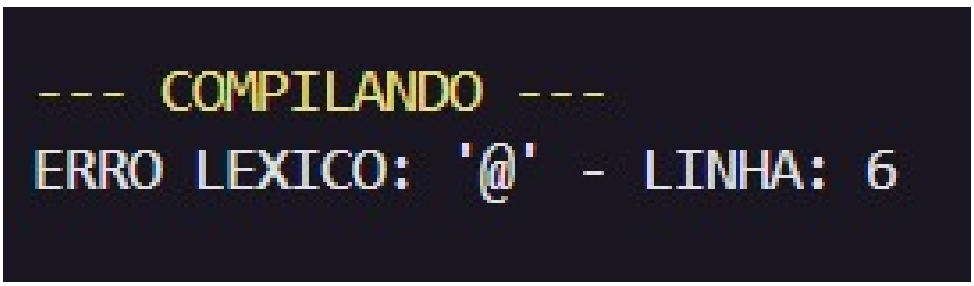

Figura 1 – Detecção de erro léxico pelo compilador.

```
1  /* Um programa de teste simples */
2  int main(void) {
3      int x;
4      int erro;
5
6      x = 10;
7      erro = @; /* Esse @ deve gerar erro lexico */
8
9      if (x > 0) {
10         return x;
```

```
11      }
12  }
```

Listing 7.1 – Código-fonte com erro léxico

7.1.2 Erro Sintático

Erros sintáticos ocorrem quando a sequência de tokens não respeita a gramática da linguagem. A Figura 2 ilustra a mensagem gerada pelo compilador ao identificar a ausência de um ponto e vírgula após a instrução `return`. O código responsável por esse erro é mostrado no Código 7.2.

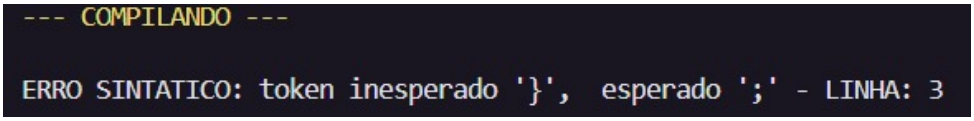

Figura 2 – Detecção de erro sintático pelo compilador.

```
1  /* Programa com erro sintatico - falta ponto e virgula */
2  int main(void) {
3      return 0
4  }
```

Listing 7.2 – Código-fonte com erro sintático

7.1.3 Erro Semântico

O erro semântico ocorre quando o código é sintaticamente válido, porém viola regras de significado da linguagem, como o uso de identificadores não declarados. A Figura 3 mostra a saída do compilador ao detectar o uso da variável `x` sem declaração prévia. O respectivo código-fonte é apresentado no Código 7.3.

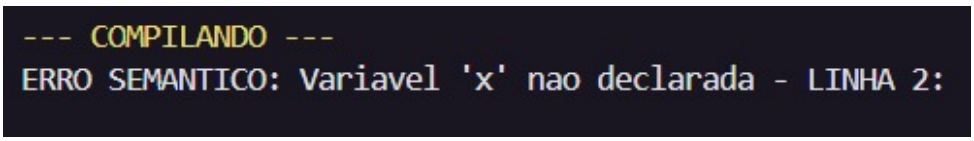

Figura 3 – Detecção de erro semântico pelo compilador.

```
1  /* Teste de erro semantico: variavel nao declarada */
2  int main(void) {
3      return x;
4  }
```

Listing 7.3 – Código-fonte com erro semântico

7.1.4 Teste de Escopo

O erro de escopo ocorre quando se tenta acessar um identificador fora do bloco ou função onde ele foi declarado, violando as regras de visibilidade da linguagem. A Figura 4 exibe a saída do compilador ao detectar a tentativa de acesso à variável local `temp` dentro da função `main`, sendo que ela pertence apenas ao escopo da função auxiliar. O código-fonte é apresentado no Código 7.4.

```
--- COMPILANDO ---
ERRO SEMANTICO: Variavel 'x' nao declarada - LINHA 2:
```

Figura 4 – Detecção de erro de escopo (semântico) pelo compilador.

```c
/* Exemplo de erro de escopo */
int soma(int a, int b) {
    int temp; /* temp eh local de soma */
    temp = a + b;
    return temp;
}

int main(void) {
    int resultado;
    /* Erro: temp n o eh visivel no escopo da main */
    resultado = temp + 1;
    return 0;
}
```

Listing 7.4 – Código-fonte com erro de escopo

7.2 Teste de Fatorial

Para validar a geração de código intermediário e o funcionamento geral do compilador em um algoritmo completo, utilizou-se o seguinte código fonte em C-Minus para o cálculo do fatorial.

7.2.1 Código Fonte (C-Minus)

```c
/* Teste com while e operadores */
int fatorial(int n) {
    int resultado;
    resultado = 1;

    while (n > 1) {
```

```
7           resultado = resultado * n;
8           n = n - 1;
9       }
10
11      return resultado;
12  }
13
14  int main(void) {
15      int x;
16      x = fatorial(5);
17
18      if (x >= 100) {
19          return 1;
20      } else {
21          return 0;
22      }
23  }
```

Listing 7.5 – Código fonte para cálculo do Fatorial em C-Minus

7.2.2 Tabela de Símbolos

Ao processar o código de teste `fatorial.cm`, um código válido, o compilador gera a Tabela de Símbolos e a representação textual da AST.

#	Nome	Escopo	TipoID	TipoDado	Loc	Linhas
0	resultado	fatorial	var	int	1	2
1	main	global	fun	int	0	13
2	n	fatorial	var	int	0	1
3	x	main	var	int	0	14
4	fatorial	global	fun	int	0	1

Figura 5 – Saída de compilação bem-sucedida com Tabela de Símbolos.

7.2.3 Árvore Sintática Abstrata (AST)

Abaixo apresentamos a representação gráfica da AST gerada para o código do fatorial, utilizando a ferramenta Graphviz.

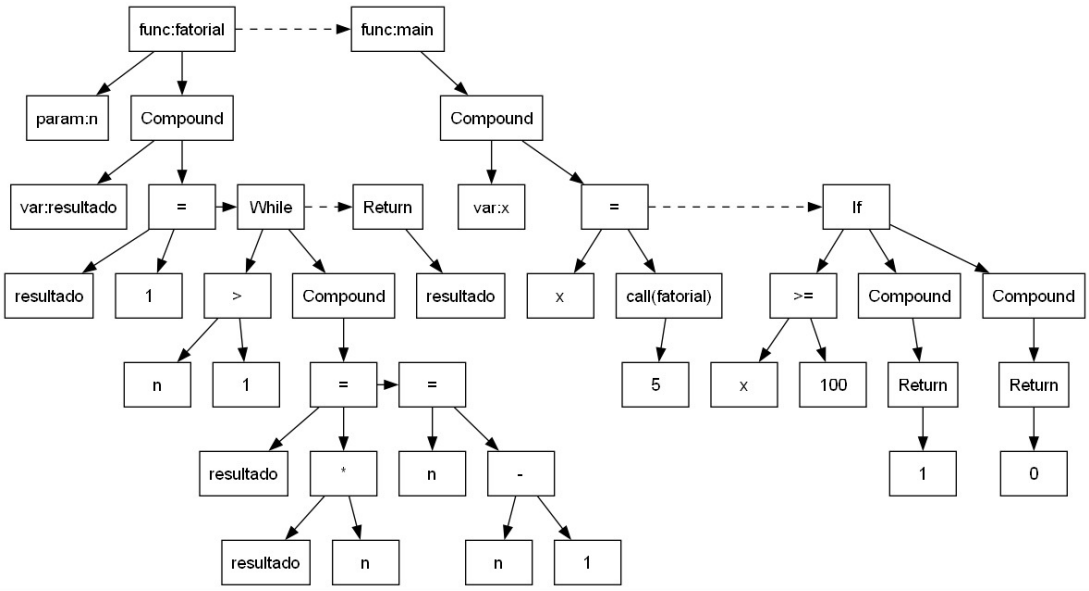

Figura 6 – Visualização da Árvore Sintática Abstrata (AST) para o programa Fatorial.

7.2.4 Código Intermediário Gerado (3 Endereços)

O compilador processou a AST e gerou o seguinte código de três endereços. Note o uso de rótulos (*labels*) e saltos (*goto*) para implementar as estruturas de controle `while` e `if`. O código intermediário apresentado a seguir foi gerado conforme o método descrito no Capítulo 6.

```
_entry fatorial
t1 = 1
resultado = t1
label L1
t2 = 1
t3 = n > t2
ifFalse t3 goto L2
t4 = resultado * n
resultado = t4
t5 = 1
t6 = n - t5
n = t6
goto L1
label L2
return resultado
_entry main
t7 = 5
param t7
t8 = call fatorial, 1
x = t8
t9 = 100
t10 = x >= t9
```

```
23 ifFalse t10 goto L3
24 t11 = 1
25 return t11
26 goto L4
27 label L3
28 t12 = 0
29 return t12
30 label L4
31 return
32 end main
```

Listing 7.6 – Código Intermediário gerado para o Fatorial

8 Conclusão

O desenvolvimento deste compilador permitiu a aplicação prática dos conceitos teóricos de linguagens formais e tradutores. A utilização das ferramentas Flex e Bison agilizou a construção do *front-end*, permitindo maior foco na implementação da lógica semântica e estruturação de dados.

Os principais desafios superados incluíram a integração correta entre o código C puro e as ferramentas geradoras, o gerenciamento de memória para a construção da AST e a implementação fiel das regras de escopo da linguagem C-. O resultado final é um compilador funcional que atende aos requisitos de análise léxica, sintática e semântica, servindo como base sólida para futuras etapas de geração e otimização de código.

Referências

1 LOUDEN, K. C. *Compiler Construction: Principles and Practice*. 1st edition. ed. Boston, MA, EUA: PWS Publishing Company, 1997. Citado 4 vezes nas páginas 5, 6, 7 e 10.