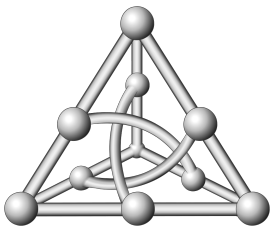
Micro C - um Compilador Acadêmico

Pedro Henrique Ferreira Carvalho

Monografia

Área de Estudo: Compiladores

Orientador: Prof. Brivaldo Alves da Silva Junior



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
Dezembro, 2025

Micro C - um Compilador Acadêmico

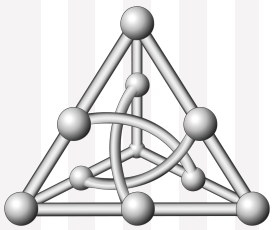
Pedro Henrique Ferreira Carvalho

Monografia

Área de Estudo: Compiladores

Orientador: Prof. Brivaldo Alves da Silva Junior

Apresentado em cumprimento parcial dos requisitos para o grau de Bacharel em Ciência da Computação



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
Dezembro, 2025

Agradecimentos

A conclusão deste trabalho marca o fim de um ciclo intenso de aprendizado e a realização do projeto mais desafiador da minha jornada acadêmica: a construção de um compilador. O processo foi árduo, mas não teria sido possível sem o apoio, a paciência e a orientação de muitas pessoas.

À minha família, expresso todo o meu amor e gratidão. O apoio, a compreensão e a presença de cada um foram importantes ao longo da minha trajetória, contribuindo para que eu tivesse a base e a tranquilidade necessárias para seguir avançando. Um agradecimento especial ao meu pai, Luiz Adauto Carvalho da Silva, cujo apoio incondicional sempre foi fundamental na minha vida.

Agradeço ao meu orientador, prof. Dr. Brivaldo Alves da Silva Jr. Sua orientação foi fundamental para a conclusão deste trabalho. Agradeço pela paciência nas incontáveis sessões de revisão e reuniões, que possibilitou apoiar o meu desenvolvimento como estudante.

Estendo meus agradecimentos a todos os professores que, ao longo de minha trajetória acadêmica, desde os primeiros anos da educação básica até minha passagem pela FACOM da Universidade Federal de Mato Grosso do Sul (UFMS), contribuíram de forma significativa para minha formação. Agradeço aos membros da banca, prof(a) Dra. Bianca de Almeida Dantas e prof. Dr. Diego Padilha Rubert, tanto pela disponibilidade em avaliar este trabalho, quanto por suas contribuições para minha formação enquanto professores.

Por fim, deixo um agradecimento especial ao meu gato, Krueger, que esteve ao meu lado em todos os momentos durante a realização deste trabalho.

Resumo

A construção de compiladores é uma área de grande interesse na ciência da computação pois ela é responsável pela tradução de linguagens de alto nível para código de máquina. No entanto, a complexidade dos compiladores modernos, como o GCC e o Clang/LLVM, ou mesmo da literatura associada ao assunto, tornam seu aprendizado e compreensão difíceis. Este trabalho busca minimizar essa dificuldade construindo um compilador para uma linguagem de programação minimalista (um subconjunto da linguagem de programação C) associando apenas o contexto teórico necessário para a compreensão da construção do compilador. Dessa forma, o objetivo do trabalho é construir um compilador que implemente as fases de compilação, traduzindo a linguagem definida, o *Micro C*, para código *Assembly* e, finalmente, um binário executável. Por se tratar de um compilador para aprendizado, todas as etapas de compilação são explícitas e podem ser executadas de forma independente seguindo o *pipeline* de compilação. Dessa forma, é possível compreender como a linguagem é convertida em seus artefatos intermediários a cada fase do processo. No compilador desenvolvido para a linguagem *Micro C*, foi utilizado um projeto modular simples dividido em *front-end* e *back-end*. O *front-end* está associado as fases de análise (léxica, sintática e semântica) para converter o código fonte em objetos prontos para conversão em linguagem de máquina. Por outro lado, o *back-end* realiza a conversão desses objetos em códigos simplificados e, por fim, código *Assembly*, pronto para serem convertidos em binários. Por fim, vários exemplos de teste foram disponibilizados para validar o compilador em diversos cenários como laços aninhados (*Bubble Sort*, Peneira de Eratóstenes) e recursão múltipla (Fibonacci, Máximo Divisor Comum). O compilador para a linguagem *Micro C* equivale, em linhas de código, menos do que $0,027\%$ do compilador GCC, tornando-o uma ferramenta valiosa para o aprendizado e ensino de compiladores. Além disso, o código fonte do compilador e todos os seus artefatos estão disponíveis publicamente.

Palavras Chave: compilador, **linguagem de programação**, **educação**

Sumário

Capítulo 1

Introdução

> *The introduction of many minds into many fields of learning along a broad spectrum keeps alive questions about the accessibility, if not the unity, of knowledge.*
>
> – Edward Levi

Apesar de sua importância, os compiladores modernos, como o GNU Compiler Collection (GCC) [15] e o *framework* LLVM (utilizado pelo Clang) [10, 11], são sistemas muito complexos. Eles possuem milhões de linhas de código e décadas de otimizações. Do ponto de vista de quem está aprendendo sobre compiladores, são sistemas difíceis de serem compreendidos, estudados e entendidos. Consequentemente, a maioria dos estudos acadêmicos desenvolvidos sobre compiladores fica restrita a algumas fases de compilação como análise sintática e semântica, e tópicos teóricos como autômatos finitos, gramáticas livres de contexto e gerenciamento de pilha de compilação.

A construção de um compilador com fins acadêmicos é um desafio em vários níveis. Primeiro, escrever um compilador para uma linguagem (das análises iniciais do código fonte até a geração de código de máquina) envolve inúmeros conceitos de computação, tanto teóricos quanto de implementação. Este capítulo introduz o trabalho, focando na construção de compiladores com o objetivo de criar um material que permita que outros acadêmicos possam compreender, de forma prática, como um compilador é construído, os desafios de implementação e os conceitos teóricos envolvidos.

O desenvolvimento da linguagem *Micro C*, como um subconjunto de uma linguagem conhecida, pode ajudar no aprendizado de programação e ainda permitir que o conhecimento adquirido com essa micro linguagem possa ser aproveitado quando do estudo da própria linguagem C. Além disso, pode servir como ferramenta para o estudo de compiladores. Pois, como o código do compilador é reduzido e cada uma das etapas de compilação é explícita, isso permite que seja compreendido como cada

uma das etapas envolvidas na conversão do código fonte em linguagem de máquina foi desenvolvida.

Por outro lado, o compilador do *Micro C* não está preocupado com performance ou otimização de código. Seus fins são didáticos para o aprendizado sobre construção de compiladores e conceitos necessários para o seu desenvolvimento. O objetivo é utilizar os conceitos teóricos da ciência da computação de livros clássicos [1, 4, 13] e utilizar apenas o necessário desses conceitos para implementar um compilador completo e com código acessível [14] publicamente sob a licença GPLv3.

1.1 Compiladores vs. Interpretadores

Programas de *software* são escritos em linguagens de alto nível, como C ou Python, pois são a forma como programadores conseguem expressar conjuntos de instruções para que um computador possa executá-las. Contudo, um computador não compreende a lógica e muito menos instruções textuais escritas em códigos de alto nível. O que a CPU de um computador consegue executar são instruções binárias de baixo nível, conhecidas como código de máquina.

Quem realiza a tradução do código que o programador desenvolveu em linguagem de alto nível em instruções que o hardware consegue executar é o compilador que é um tradutor entre as duas linguagens. Existem duas abordagens para esta tradução, a compilação e a interpretação.

Um *compilador* é um programa que traduz um código-fonte de alto nível (ex: *Micro C*) para um código-alvo de baixo nível (ex: *Assembly* x86-64), *antes* de sua execução. O resultado desse processo é um arquivo executável independente (ex: `programa.exe`) que o sistema operacional pode carregar e executar diretamente no processador.

A principal característica de um compilador é que a tradução ocorre apenas uma vez. O programa final gerado é o código de máquina nativo pronto para ser executado [1]. O compilador do *Micro C* é um exemplo dessa estratégia de tradução.

Um *interpretador*, por outro lado, não produz um arquivo executável. Em vez disso, ele intepreta as instruções em tempo de execução. O interpretador lê o código-fonte (ou um código intermediário chamado de *bytecode*) instrução por instrução e *executa* a ação correspondente.

A principal característica de um interpretador é que ele atua como um processador virtual, simulando a execução do programa. Linguagens como Python ou Ruby são exemplos de linguagens que dependem de um interpretador. Nystrom [13] mostra que esta abordagem é muito portável (o mesmo código-fonte pode ser executado em qualquer máquina que tenha o interpretador), mas incorre em penalidades de performance, pois a tradução ocorre a cada execução.

As diferenças fundamentais entre as duas abordagens impactam diretamente a

performance, a portabilidade e a forma como os erros são tratados. Enquanto o compilador traduz o programa uma única vez antes de sua execução, o interpretador executa instrução por instrução durante a execução do programa.

Dessa forma, com o compilador, a tradução do código fonte para o código de máquina é gerado apenas uma única vez para o sistema e arquitetura alvo, o que é diferente do interpretador que geralmente interpreta o código fonte todas as vezes ou, na melhor hipótese, gera um código intermediário (*bytecode*). Essa diferença no processamento e execução faz com que o compilador gere códigos executáveis pelo computador muito mais rápidos em relação aos códigos interpretados.

Além disso, a compilação consegue detectar vários erros antes da geração do código binário, pois ele realiza uma *verificação estática* antes de traduzir, enquanto o interpretador realiza *verificações dinâmicas* gerando erros e exceções em tempo de execução. Por fim, do ponto de vista da portabilidade, os códigos compilados ficam dependentes das arquiteturas e sistemas operacionais alvos da compilação enquanto os códigos interpretados são muito mais portáveis pois dependem apenas da existência do interpretador para o sistema desejado.

1.2 Estrutura do Trabalho

Este trabalho está organizado em seis capítulos, além da introdução. No Capítulo 2, são detalhadas a Análise Léxica, a teoria dos autômatos finitos e a implementação do `scanner.c` do compilador para converter texto em *tokens*. No Capítulo 3, é abordada a Análise Sintática, explicando a gramática da linguagem do *Micro C*, a implementação do *parser* recursivo descendente e a construção da Árvore Sintática Abstrata (ASA). Em seguida, no Capítulo 4, aborda-se a Análise Semântica, detalhando a implementação da Tabela de Símbolos, o gerenciamento de escopos, a checagem de tipos e o cálculo dos *offsets* de memória, garantindo que não existam inconsistências entre o que foi programado e a linguagem. Encerrando, dessa forma, a construção do *front-end* do compilador.

O próximo passo na construção do compilador é a construção do *back-end*, ou seja, a conversão das estruturas lógicas em linguagem de máquina. Dessa forma, no Capítulo 5 se inicia essa construção, explicando a arquitetura da Representação Intermediária (RI) e a tradução da ASA gerada nos capítulos anteriores para o Código de Três Endereços (CTE). Finalizando o *back-end*, no Capítulo 6 é feita a implementação e o detalhamento dos modelos de memória (*stack frame*), convenções de chamada (ABI) e a tradução final da RI para o código *Assembly* x86-64.

Vale destacar que, alinhado ao propósito didático do projeto, o compilador foi implementado para permitir a **execução passo-a-passo**. A cada fase concluída descrita nos parágrafos anteriores, a ferramenta é capaz de gerar um **artefato** correspondente àquela etapa (como a lista de *tokens*, a visualização da ASA, a tabela de símbolos, o código intermediário e o código em *Assembly*). Essa funcionalidade permite que o usuário inspecione e valide os resultados parciais de cada processo

de compilação do *Micro C* antes de prosseguir para a próxima fase do compilador. Além disso, a ferramenta também permite a **execução de todas as etapas** de uma única vez para gerar o executável final diretamente, imprimindo a saída do código, caso haja.

No Capítulo 7, são apresentados os resultados da implementação, discussão dos principais desafios de implementação e trabalhos futuros. Por fim, são apresentados quatro apêndices. O Apêndice A descreve a linguagem definida para o *Micro C*, o O Apêndice B descreve a Gramática Livre de Contexto aceita pelo *Micro C*, o Apêndice C apresenta alguns exemplos de códigos que podem ser implementados e compilados e, finalmente, o Apêndice D mostrando um exemplo de compilação completa passando por todas as etapas de compilação até a geração de um binário para o Linux.

Capítulo 2

Análise Léxica

> *It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way.*
>
> – *A Tale of Two Cities*

Neste capítulo, vamos abordar a análise léxica (a primeira fase do processo de compilação) do código e suas operações. A primeira fase da compilação de um programa tem como objetivo ler os caracteres do código fonte e identificar as sequências de caracteres que correspondem a formatos esperados para os *tokens* (símbolos ou elementos léxicos). Um *token* é uma categoria ou tipo de elemento no código que o compilador reconhece e associa a um significado específico seguindo um *padrão*.

Suponha, por exemplo, um programa escrito em uma linguagem de programação, com linhas de código como $x = 5$ ou $y = 10$. A análise léxica é a etapa inicial que lê essas linhas de código e identifica as sequências de caracteres que correspondem a operações aritméticas (como $=$), variáveis (como x e y) ou outros tipos de elementos.

A essa sequência é dado o nome de lexema, que é a sequência real de caracteres encontrada no código fonte que o compilador identifica e associa a um *token* específico. O *token* é uma unidade do código que o compilador reconhece e classifica. Cada *token* possui um nome que indica sua função ou tipo (por exemplo, uma palavra reservada, um número inteiro, etc.) e, em alguns casos, um valor adicional (como o valor de uma constante). Enquanto o *token* é a categoria abstrata (como **identificador** ou **operador de atribuição**), o lexema é o texto concreto que se encaixa nessa categoria (como `x` ou `=`), desde que corresponda a um padrão léxico definido. A análise léxica é o processo de ler os lexemas do código e *etiquetá-los* com seus respectivos tipos de *token*, preparando-os para as fases seguintes.

Por outro lado, o padrão é a descrição do formato que os lexemas de um *token* devem seguir. Em outras palavras, é o modelo que o compilador usa para identificar qual tipo de *token* está sendo processado. Por exemplo, o padrão para um *token* de **número inteiro** pode ser *uma ou mais ocorrências de dígitos de 0 a 9*. Esse padrão faz o compilador distinguir entre diferentes tipos de *tokens*, como palavras reservadas, operadores, identificadores e constantes, por exemplo.

Além de agrupar os lexemas em *tokens*, o analisador léxico frequentemente realiza o escandimento[1], que envolve a remoção de comentários e espaços em branco desnecessários do código fonte para facilitar as próximas etapas.

2.1 Análise Léxica: A Primeira Validação

Embora as fases posteriores do compilador sejam responsáveis pela maior parte da verificação de erros, o analisador léxico atua como a primeira linha de defesa. Ele não apenas agrupa caracteres, mas também valida se esses agrupamentos formam lexemas válidos. Se o *scanner* encontrar um caractere ou uma sequência de caracteres que não corresponde a nenhum padrão definido na linguagem, ele deve reportar um **erro léxico**.

Um erro léxico ocorre quando o *scanner* não consegue formar um *token* válido a partir da entrada. Por exemplo, na linguagem do *Micro C*, o caractere `@` não pertence a nenhum padrão válido. Se o *scanner* o encontrasse no código, ele geraria um erro imediato.

```
int x = 10 @ 20; // Erro Léxico!
```

Neste caso, o *scanner* reconheceria `int`, `x`, `=`, `10`, mas ao encontrar o `@`, ele não conseguiria classificá-lo. Neste caso, o compilador iria interromper sua execução e informar o programador com um erro como, por exemplo: `Erro Léxico na linha 1: Caractere inválido '@' encontrado`.

2.1.1 A Interação com as Fases Subsequentes

O analisador léxico não trabalha isoladamente; ele é a fonte de entrada para a próxima fase do compilador. Existem duas arquiteturas principais para essa interação: a *sob demanda* (em que o analisador sintático avalia um *token* de cada vez) ou a *em lote* (em que o analisador léxico gera todos os *tokens* primeiro).

Embora a abordagem *sob demanda* seja comum e eficiente em termos de uso de memória [1], no *Micro C*, optou-se pela abordagem *em lote* por ser mais simples de implementar e mais fácil de depurar.

[1]Em compiladores, o escandimento é uma leitura inicial do código que apenas identifica os elementos básicos, removendo comentários ou espaços em branco.

No *Micro C*, o processo de compilação ocorre em duas etapas. Na primeira etapa é realizada a análise **léxica completa**, em que a função `executar_analise_lexica()` é invocada e entra em um laço de repetição executando a função `proximo_token()` repetidamente, até que o *token* `END_OF_FILE` seja encontrado no código fonte. Cada *token* gerado é armazenado em uma grande lista na memória (`TokenList`). A segunda etapa é a **entrega para a próxima fase**, que ocorre somente após a análise léxica estar concluída e a `TokenList` completa. Essa lista será então enviada para o analisador sintático (*parser*).

O trecho de código a seguir ilustra a lógica de execução do *Micro C*. Primeiro, ele executa a Fase 1 por completo:

```
// arquivo: src/main.c (trecho da Fase 1)

/fase 1: análise léxica
    TokenList tokens;
    inicializar_token_list(&tokens, 100);
    if (executar_analise_lexica(nome_arquivo, &tokens) != 0) {
        liberar_token_list(&tokens);
        return -1; //encerra se houver erro léxico
    }
    if (strcmp(modo, "--scan") != 0) {
        salvar_tokens_em_arquivo(&tokens, false);
    }
    printf("Fase 1 (Lexica) concluida: %d tokens
    gerados.\n", tokens.tamanho);
```

Uma vez que a lista de *`tokens`* está completamente preenchida e validada, ela é enviada para a Fase 2:

```
// arquivo: src/main.c (trecho da Fase 2)

//fase 2: análise sintática
PilhaTabelasSimbolos* pilha_simbolos = criar_pilha_tabelas();
Parser parser;
// A lista "tokens" completa é entregue ao parser de uma só vez:
inicializar_parser(&parser, &tokens, pilha_simbolos);
ASTNode* arvore = parse(&parser);
// ...
```

O uso da estratégia *em lote* como decisão de projeto, permitiu uma depuração mais granular ao executar a análise léxica de forma independente (`make scan`). Como o objetivo é aprender e entender como compiladores são construídos, o *Micro C* permite que cada fase da compilação seja executada de forma incremental. Ou seja, o `make scan`, quando utilizado de forma independente, imprime a lista completa de *tokens* e permite verificar se o *vocabulário* pré-determinado foi reconhecido corretamente.

2.1.2 A Análise Léxica como uma Fase Independente

Separar a análise léxica das outras fases em módulos distintos é uma decisão de projeto na construção de um compilador e possui algumas vantagens como, por exemplo:

- **Simplicidade de Projeto:** a separação torna cada módulo mais simples. O *scanner*, por exemplo, fica dedicado apenas em reconhecer padrões de baixo nível (caracteres), enquanto as próximas etapas podem se concentrar em como esses *tokens* se combinam para formar estruturas maiores, sem se preocupar com os detalhes da leitura de caracteres, espaços em branco ou comentários.
- **Eficiência:** o *scanner* é a parte do compilador que mais interage com o sistema de arquivos. Técnicas de otimização de leitura, podem ser implementadas e aprimoradas dentro do módulo do *scanner* sem afetar o resto do compilador.
- **Portabilidade:** A análise léxica é uma das partes mais portáveis de um compilador. A lógica para reconhecer identificadores, números e palavras reservadas é muito semelhante entre diferentes linguagens de programação. Um *scanner* bem escrito pode ser mais facilmente adaptado para um novo projeto de compilador.

O *Micro C* pode ser facilmente reproduzido ou expandido dado o seu caráter modular com interfaces simples definidas em bibliotecas (`.h`) e implementações (arquivos `.c`) do projeto.

2.2 Do Código Fonte aos Lexemas e *Tokens*

A Análise Léxica é o primeiro passo do processo de compilação, responsável por ler o arquivo fonte caractere por caractere. Ela consiste em agrupar esses caracteres para transformar o código fonte em elementos léxicos (*tokens*), gerando uma estrutura que o compilador possa entender e utilizar. Essa transformação permite ao compilador correlacionar o que o programador escreveu com a sua real intenção, preparando o ambiente para as etapas subsequentes de análise.

A linguagem definida para o *Micro C* é um subconjunto minimalista da linguagem ANSI C. Embora sua sintaxe reduzida facilite a compreensão das etapas internas do compilador, permitindo observar com clareza como cada etapa funciona, isso não torna a linguagem menos poderosa. Ou seja, mesmo com uma linguagem reduzida, vários algoritmos puderam ser implementados como, por exemplo, Bubble Sort, Torre de Hanoi, Fibonnaci, etc (alguns exemplos estão no Apêndice C). A descrição completa da linguagem está definida no Apêndice A.

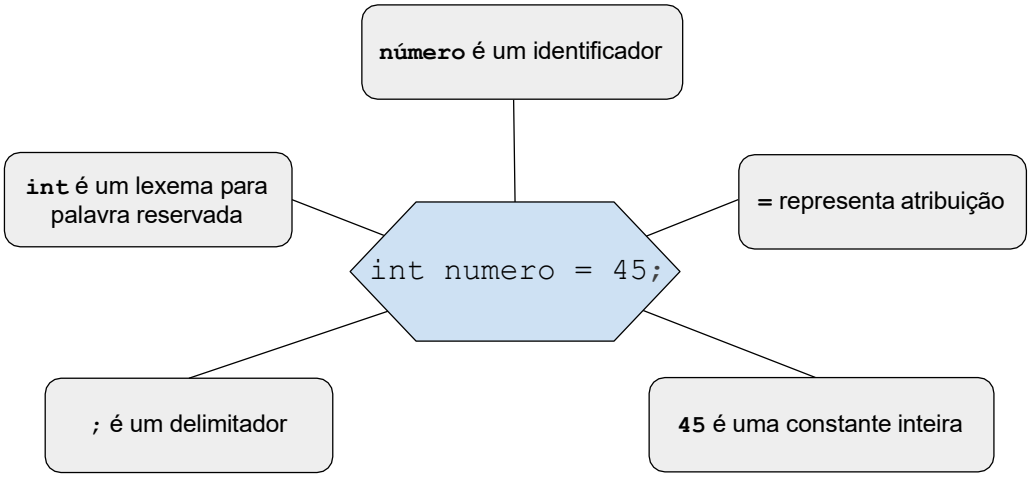


Figura 2.1: Uma expressão como `int numero = 45;` deve ser quebrada em vários *tokens* e cada um será interpretado de uma maneira diferente na análise léxica.

Como mostrado na Figura 2.1, cada lexema cria um *token*. Por exemplo, `int` é uma palavra reservada para números inteiros. `numero` é o nome de uma variável (um identificador). O símbolo `=` é um operador que atribui um valor à variável que vem antes dele, neste caso, `numero`. `45` é o valor inteiro que será guardado em `numero`. Por fim, `;` é um delimitador que marca o fim da instrução.

Quando o analisador léxico identifica um *token* do tipo `ID`, ele registra o lexema correspondente. Por exemplo, na Figura 2.1, o lexema `numero` é classificado como um `ID`. Embora o analisador léxico não associe esse nome a informações adicionais, como seu tipo de dado (`int`) ou seu futuro endereço de memória, essa correlação será feita posteriormente pelo analisador semântico. Dessa forma, o campo `lexema` do *token* serve como ponte entre a fase da análise léxica e as etapas seguintes do compilador.

Uma vez que o fluxo de *tokens* é gerado pela análise léxica, a primeira etapa da compilação está finalizada. O texto bruto foi transformado em uma sequência de unidades lógicas e estruturadas, permitindo que as fases seguintes do compilador possam analisar a gramática e o significado do programa.

2.3 Os *Tokens* no Compilador do *Micro C*

Na introdução e na primeira seção, foram definidos os conceitos teóricos de *token*, *lexema* e *padrão*. Em seguida, como a análise léxica atua transformando o código-fonte em unidades lógicas.

O próximo passo é entender a especificação formal desses conceitos, mostrando como eles foram traduzidos para estruturas de dados concretas no compilador. Esta seção detalha o *dicionário completo* do vocabulário do compilador do *Micro C* e como ele é implementado.

2.3.1 Vocabulário Completo: `tokens.h`

O primeiro passo na construção de um *scanner* é definir o seu vocabulário. No caso do *Micro C*, o arquivo `tokens.h` possui o *dicionário central* que é compartilhado por todas as fases do compilador. Este arquivo define duas estruturas de dados principais: a enumeração de todos os tipos de *token* e a estrutura que representa um *token* individual.

Primeiro, definimos todas as categorias de *tokens* possíveis usando uma enumeração `TokenType`. Esta é a lista completa de todos os *tokens* que o *scanner* do *Micro C* é capaz de reconhecer. Eles são agrupados por funcionalidade para facilitar a leitura e manutenção. Segue abaixo o trecho de código relativo aos tipos de *tokens*:

```
// arquivo: src/tokens/tokens.h (Parte 1: O Enum)
#ifndef TOKENS_H
#define TOKENS_H

typedef enum {
    // Tokens Fundamentais
    UNDEF,           // token indefinido (para erros)
    ID,              // identificador (ex: x, minhaVariavel)
    END_OF_FILE,     // token especial para o fim do arquivo

    // Constantes Literais
    INTEGERCONST,    // constante inteira (ex: 123)
    CHARCONST,       // constante de caractere (ex: 'a')
    STRINGCONST,     // string de caracteres (ex: "ola")

    // Operadores Aritméticos
    PLUS, MINUS, MUL, DIV, MOD,

    // Operadores Relacionais e Lógicos
    EQ, NEQ, LT, GT, LEQ, GEQ, AND, OR, NOT,

    // Símbolos de Atribuição e Pontuação
    ASSIGN, SEMICOLON, COMMA, LPAREN, RPAREN,
    LBRACE, RBRACE, LBRACKET, RBRACKET,

    // Palavras reservadas
    MAIN, IF, ELSE, FOR, RETURN, INT, CHAR, PRINT

} TokenType;
```

Cada valor dentro dessa enumeração representa um tipo diferente de *token* que o analisador léxico será capaz de identificar no código. Além da categoria, o compilador precisa armazenar as informações específicas daquele *token*, como o texto que ele representa (o lexema) e onde ele foi encontrado.

Esse comportamento é codificado na `struct Token`:

```
// arquivo: src/tokens/tokens.h (Parte 2: A Struct)
typedef struct {
    TokenType tipo;
    char lexema[100];
    int linha;
} Token;

#endif //TOKENS_H
```

Com as categorias definidas, a `struct Token` agrupa todas as informações que o *scanner* extrai do código para cada *token*. Esta estrutura é o *pacote de dados* que é utilizado entre as fases do compilador. Segue a seguir uma descrição de cada componente da estrutura:

- `TokenType tipo`: este é o campo mais importante para o **analisador sintático**. É a *etiqueta* que informa ao *parser* qual é a categoria do *token* (ex: `IF`, `LPAREN`, `ID`). O *parser* usará essa informação para verificar se a sequência de *tokens* obedece à gramática da linguagem.
- `char lexema[100]`: este é o atributo do *token*. Ele armazena o texto original do lexema. Como veremos na próxima subseção, este campo é vital para o **analisador semântico** (que o usará para inserir o nome da variável na Tabela de Símbolos) e para o **gerador de código** (que precisa saber o valor de uma constante, como `123`).
- `int linha`: Este campo armazena o número da linha em que o *token* foi encontrado. Sua única finalidade é permitir que **qualquer fase do compilador** (léxica, sintática ou semântica) possa reportar erros de forma clara e precisa para o programador.

2.3.2 Atributos: O Valor por trás do *Token*

Um *token* pode ter um *valor adicional* ou atributo. No *Micro C*, o campo `lexema` é o responsável por carregar esse atributo. Alguns *tokens*, como palavras reservadas (`IF`) ou pontuadores (`SEMICOLON`), são autossuficientes. O tipo de *token* `IF`, por exemplo, carrega todo o significado necessário.

Por outro lado, o lexema original ("if") é irrelevante para o resto do compilador, pois uma vez que o *scanner* o classifica como o tipo `IF`, o analisador sintático só precisa saber isso para validar as regras gramaticais (ex: `IF` seguido de `LPAREN`). Sendo assim, a string "if"por si só não carrega valor adicional. O que é diferente, por exemplo, para um `ID` ou um `INTEGERCONST`. No entanto, para outras categorias de *token*, o tipo sozinho não é relevante.

Considere o seguinte código:

```
int x = 10;
int y = 20;
```

Se o *scanner* gerasse *tokens* sem atributos (sem o campo `lexema`), o analisador sintático receberia a seguinte sequência de tipos de *token*:

`INT`, `ID`, `ASSIGN`, `INTEGERCONST`, `SEMICOLON`, `INT`, `ID`, `ASSIGN`, `INTEGERCONST`, `SEMICOLON`

Do ponto de vista da gramática (análise sintática), essa sequência está correta. O *parser* conseguiria validar que ambas são *declarações válidas*. No entanto, para a análise semântica e a geração de código, essa informação é desastrosa porque não seria possível determinar quais os nomes das variáveis que foram declaradas e nem qual o valor atribuído para qual variável.

Isso exemplifica o porquê do `lexema` ser essencial. O fluxo de *tokens* real que o *scanner* gera é, na verdade, uma sequência de `struct Token`, cada uma carregando seu lexema, ou seja:

- Token 1: `tipo`: INT, `lexema`: "int", ...
- Token 2: `tipo`: ID, `lexema`: "x", ...
- Token 3: `tipo`: ASSIGN, `lexema`: "=", ...
- Token 4: `tipo`: INTEGERCONST, `lexema`: "10", ...
- Token 5: `tipo`: SEMICOLON, `lexema`: ";", ...
- ... e assim por diante para a variável `y`.

Cada *token* cumpre um papel específico no processo de tradução. O primeiro indica ao compilador que será declarada uma variável inteira. O segundo fornece o nome dessa variável, `x`, que será registrada na tabela de símbolos. O terceiro sinaliza uma atribuição, enquanto o quarto entrega o valor literal `10`, que será associado à variável `x` e posteriormente carregado na memória. O quinto *token*, o delimitador `;`, encerra a instrução, indicando ao compilador o fim lógico dessa declaração. Esse mesmo processo se repete para a variável `y`. De forma geral, o campo `lexema` é o elo entre o texto escrito pelo programador e as informações que o compilador realmente entende e manipula.

2.3.3 Tratamento de Erros Léxicos

Como mencionado na Subseção 2.1, o *scanner* é responsável por identificar caracteres que não pertencem a nenhum padrão. O *Micro C* utiliza uma abordagem direta para tratar com situações em que um lexema não corresponde a nenhum *token* válido.

Dessa forma, um *token* `UNDEF` pode ser gerado em dois cenários principais pelo *scanner*:

1. **Padrões Malformados:** ocorre quando o *scanner* reconhece o início de um padrão válido (como " para uma string ou ' para um caractere), mas a sequência é terminada de forma inesperada. Como pode-se observar, uma string que não é fechada antes de uma quebra de linha ou um caractere que não é fechado corretamente são exemplos perfeitos. Nesses casos, o *scanner* identifica o erro e retorna `UNDEF`.
2. **Caracteres Inválidos:** ocorre quando o caractere lido não inicia *nenhum* padrão conhecido pela linguagem (como `@` ou `#`).

A seguir, temos um trecho de código do *'scanner.c'* para o reconhecimento de strings. Ele ilustra apenas o primeiro cenário. Se o laço `while` parar por causa de uma quebra de linha (`\n`) ou fim de arquivo (`EOF`) antes de encontrar a aspa de fechamento da string, ele entra no bloco `else` e gera um erro. Para referência, toda a estrutura léxica da linguagem *Micro C*, incluindo as definições e exemplos de todos os *tokens* e seus lexemas correspondentes, é apresentada no Apêndice A.

```
// arquivo: src/scanner/scanner.c (trecho do reconhecimento
de string)
if (c == '"') {
    char buffer[101];
    int i = 0;
    c = prox_char();

    while (c != '"' && c != '\n' && c != EOF && i < 100) {
        buffer[i++] = c;
        c = prox_char();
    }
    buffer[i] = '\0';

    if (c == '"') { // Sucesso
        token.tipo = STRINGCONST;
        strcpy(token.lexema, buffer);
    } else { // Erro: Padrão malformado
        token.tipo = UNDEF;
        strcpy(token.lexema, "String nao terminada");
    }
    return token;
}
```

O segundo cenário (caractere inválido) é tratado pelo caso `default` no final da função `proximo_token`.

Abaixo, um trecho que respresenta o *caso geral* para qualquer caractere que não iniciou nenhum dos padrões anteriores:

```
// arquivo: src/scanner/scanner.c (trecho do final do switch)
    //... (após todas as outras verificações)

    // Se nenhum padrão foi reconhecido, é um token indefinido
    default:
        token.lexema[0] = c;
        token.lexema[1] = '\0';
        token.tipo = UNDEF;
        break;
}
return token;
```

É importante notar que a lógica de parada não está no *scanner* em si, mas sim no orquestrador principal do compilador (o `main.c`). O `main` verifica cada *token* recebido do *scanner* e, se ele for do tipo `UNDEF`, interrompe imediatamente o processo de compilação.

```
// arquivo: src/main.c (trecho da função executar_analise_lexica)
do {
    token = proximo_token();
    adicionar_token(lista, token);
    if (token.tipo == UNDEF) {
        fprintf(stderr,
                "Erro Lexico: Token indefinido '%s' na linha %d\n",
                token.lexema, token.linha);
        return -1; // Interrompe a compilação
    }
} while (token.tipo != END_OF_FILE);
```

Essa estratégia de *parada imediata* evita que as fases seguintes (sintática e semântica) tentem processar uma entrada corrompida. Como o *Micro C* é um compilador didático, ele reporta o primeiro erro encontrado de forma clara e facilita a correção pelo usuário, sem a complexidade de algoritmos de *modo de pânico* que tentam continuar a análise mesmo após um erro.

2.4 Relação da Prática com a Teoria Formal

Nas seções anteriores foi definido o vocabulário da linguagem, expresso nas estruturas `tokens.h`, e o papel do *scanner* em identificar lexemas e gerar *tokens*. O próximo passo é fazer essas definições de modo mais formal.

A lógica por trás da implementação do *scanner* não é arbitrária. Ela representa uma aplicação de conceitos clássicos de teoria da computação. E, embora não seja o objetivo aprofundar nos conceitos teóricos relacionados a implementação do `scanner.c`, dois tópicos são importantes para compreender como ele é construído:

as **Expressões Regulares** e os **Autômatos Finitos**.

2.4.1 Expressões Regulares para Especificação dos *Tokens*

Na introdução do capítulo, definiu-se o conceito de **padrão** como *a regra que descreve os lexemas de um token*. A ferramenta mais comum, poderosa e concisa para descrever esses padrões é a **Expressão Regular** (Regex). Uma expressão regular é uma notação formal que define um conjunto de strings (uma *linguagem*).

Para compreender o uso das expressões regulares, são necessários alguns metacaracteres básicos que servem como base para descrever regras de construção.:

- `|` (alternância): significa "ou". Ex: `a|b` significa *o caractere 'a' ou o caractere 'b'*.
- `*` (fecho de Kleene): significa *zero ou mais ocorrências* do que veio antes. Ex: `a*` significa "" (string vazia), "a", "aa", "aaa", etc.
- `+` (fecho positivo): significa *uma ou mais ocorrências*. Ex: `a+` significa "a", "aa", "aaa", etc.
- `?` (opcional): significa *zero ou uma ocorrência*. Ex: `-?` significa *um sinal de menos opcional*.
- `[a-z]`: define uma *classe de caracteres*, ou seja, *qualquer caractere de 'a' a 'z'*.
- `[^abc]`: o `^` dentro de uma classe nega o conjunto, significando *qualquer caractere que* não *seja 'a', 'b' ou 'c'*.

Com essas regras de construção, é possível descrever um conjunto de todos os *tokens* de uma linguagem de programação. Isso forma o que é conhecido como uma **linguagem regular**. Abaixo, estão listados os padrões formais para os *tokens* mais importantes do *Micro C* e como eles se conectam ao código.

Identificadores (ID) O padrão para um identificador na linguagem é *uma letra ou underscore, seguido por zero ou mais letras, dígitos ou underscores.*

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Conexão com o Código:

Essa expressão regular é implementada diretamente no `scanner.c`. O primeiro `if` da função `proximo_token` verifica a primeira classe de caracteres (`isalpha(c) || c == '_'`) e o laço `do-while` subsequente implementa o `*` (zero ou mais) ao consumir todos os caracteres seguintes que se encaixam na segunda classe (`isalnum(c) || c == '_'`).

Constantes Inteiras (INTEGERCONST) O padrão para um inteiro na linguagem é *um ou mais dígitos*, opcionalmente precedido por um sinal de menos para representações de números negativos.

```
-?[0-9]+
```

Conexão com o Código:

A implementação foi realizada em duas partes: o bloco `if (isdigit(c))` trata a parte `[0-9]+`, e o `case '-':` (discutido em detalhes na próxima seção) usa *lookahead* para tratar a parte opcional `-?`.

Constantes String (STRINGCONST) O padrão para uma string é *um caractere de aspas duplas, seguido por zero ou mais caracteres de qualquer tipo, exceto outras aspas duplas ou uma quebra de linha, seguido por um caractere de aspas duplas.*

```
"[^"\n]*"
```

Conexão com o Código:

A classe `[^"\n]` (qualquer caractere que *não* seja aspa ou quebra de linha) é implementada pelo laço: `while (c != '"' && c != '\n' && c != EOF)`.

Essas são as principais expressões regulares para *tokens* no *Micro C*. As demais expressões regulares estão documentadas no código do compilador e seguem o mesmo padrão apresentado nos exemplos acima.

2.4.2 O *Scanner* como um Autômato Finito Determinístico

A construção das expressões regulares é o primeiro mecanismo. Contudo, para o reconhecimento do texto do código, são necessários mecanismos capazes de capturar a relação entre os caracteres de um *token*. Isso pode ser realizado por meio do uso de **Autômatos Finitos Determinísticos (AFD)**. Um AFD pode ser descrito, empiricamente, como um jogo de tabuleiro simples. Ele consiste em um conjunto de **estados** (círculos no diagrama, ou *casas* do tabuleiro). Um **estado inicial** (onde o jogo começa). Um conjunto de **estados de aceitação** (casas que, se a jogada encerrar nelas, o jogador *vence*, em outras palavras, um *token* é reconhecido). E um conjunto de **transições** (vetores direcionais entre os estados). Cada vetor é rotulado com um caractere ou classe de caracteres que precisam ser *processados* para que aquele caminho seja percorrido.

Para concretizar esse conceito teórico, a Figura 2.2 apresenta o autômato finito determinístico projetado para o reconhecimento de *identificadores* na linguagem *Micro C*. A regra léxica define que um *identificador* deve iniciar obrigatoriamente com uma letra ou sublinhado, podendo ser seguido por uma sequência de letras, dígitos ou sublinhados.

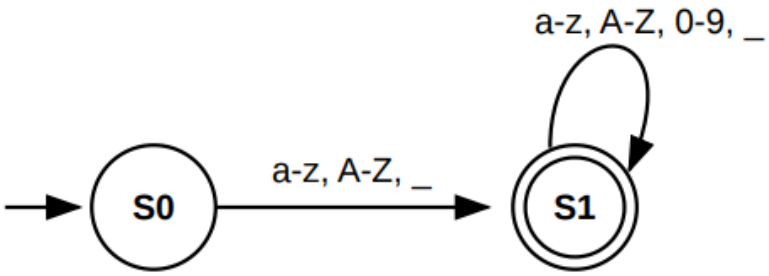


Figura 2.2: Autômato Finito Determinístico para reconhecimento de Identificadores.

Conforme ilustrado na Figura 2.2, o processo de reconhecimento inicia-se no estado S_0. Ao ler um caractere válido para o início de uma variável (letra ou _), o autômato transita para o estado S_1. Note que S_1 possui borda dupla, indicando ser um **estado de aceitação**, ou seja, a sequência processada até aquele momento já constitui um *token* válido.

A partir de S_1, existe uma transição de laço (*loop*) que permite ao autômato consumir sucessivos caracteres alfanuméricos, permanecendo no estado de aceitação. O reconhecimento do *token* é finalizado quando o *scanner* encontra um caractere que não satisfaz a condição do laço (como um espaço em branco ou um operador), momento em que o lexema acumulado é classificado como um `ID`.

A avaliação dos *tokens* por um AFD funciona de forma similar ao jogo descrito anteriormente. O AFD processa a entrada um caractere de cada vez. Ele começa no estado inicial, lê um caractere e percorre o vetor direcional correspondente até o novo estado até consumir todos os caracteres do lexema. Se, ao final, ele terminar em um estado de aceitação (estado final), o *token* é reconhecido. A palavra *determinístico*, neste contexto, significa que para qualquer estado e qualquer caractere, há no máximo **um** caminho possível. Não há ambiguidades.

A não ambiguidade é uma característica importante na construção de compiladores. Existem ferramentas automatizadas (ex: Lex[2], Flex[3], etc.) que transformam expressões regulares (padrões) em AFD's otimizados [9], além de gerar como saída o código em C que implementa o AFD. Não é o caso com relação a implementação do *Micro C*.

[2]Lex, abreviação de *Lexical Analyzer Generator*, é uma ferramenta que gera automaticamente o código de um analisador léxico em linguagem C a partir de um arquivo onde o programador define os padrões dos *tokens* usando expressões regulares.

[3]Flex, ou *Fast Lexical Analyzer*, é uma reimplementação mais rápida do Lex. Ele cumpre a mesma função: gerar um *scanner* automaticamente a partir de um arquivo de regras e padrões.

2.4.3 O *Scanner* como um AFD Manual

Embora o uso de ferramentas auxilie na conversão de expressões regulares em autômatos finitos determinísticos, o compilador do *Micro C* não os utiliza. O *scanner* implementado em `scanner.c` é uma implementação simples e direta de um AFD simulando um autômato finito. Isso permite que o compilador fique, embora mais limitado a linguagem definida, mais simples de entender como o processamento dos lexemas ocorre.

A função do código do *Micro C* que implementa o AFD é a `proximo_token()`. É nela que estão implementados os estados do autômato e suas transições. Iniciando pelo **estado inicial**, após invocar a função `ignora_espacos_e_comentarios()`, ele retorna com o primeiro caractere de um lexema. Cada **estado** é armazenado pela variável `c` (o caractere atual) e por qual bloco de código está sendo executado no momento (ex: `if (isalpha...)` ou `case '=':`). Caso o lexema seja consumido e a transição encerre em um **estado de aceitação**, a função decide qual tipo o *token* (ex: `token.tipo = INT;`) está relacionado e o retorna (`return token;`) para recomeçar e avaliar o próximo *token*. Por outro lado, quando termina em um **estado de erro**, que é o padrão, é retornado um *token* `UNDEF` (indefinido).

O exemplo a seguir demonstra como o AFD do *Micro C* reconhece os *tokens* `<` (LT) e `<=` (LEQ).

1. **Estado Inicial:** A função `proximo_token()` lê um caractere. Ele é `<`.
2. **Estado *Avaliando um Menor Que*:** O AFD agora está em um estado que consumiu um `<`. A única transição de saída deste estado depende do próximo caractere. O código executa o *lookahead* invocando a função `prox_char()`.
3. **Transições:**
   - Se o próximo caractere for `=`, o AFD transita para o **Estado de Aceitação** `LEQ`. O *scanner* retorna o *token* `LEQ`.
   - Se o próximo caractere for *qualquer outra coisa*, o AFD transita para o **Estado de Aceitação** `LT`. O *scanner* usa `ungetc()` para *retornar* um caractere (*retorna o ponteiro ao estado anterior*) e retorna o *token* `LT`.

Esse exemplo demonstra como o AFD é utilizado para consumir os lexemas apresentados e determinar o que são os elementos sendo avaliados. O uso do *lookahead* é necessário para determinar situações em que existem possíveis variações na ramificação de uma decisão, permitindo avaliar em qual estado, de fato, o autômato deve terminar sua execução.

2.5 O *Scanner* da Linguagem *Micro C*

As estruturas de dados para representação formal dos *tokens*, definidas no arquivo `tokens.h`, possibilitam iniciar a implementação do analisador léxico. Esta seção

apresenta a parte mais extensa e detalhada deste capítulo, dedicando-se a uma análise do arquivo `scanner.c`.

Embora ferramentas de automação, como Lex ou Flex, gerem analisadores léxicos a partir de um arquivo de especificações contendo expressões regulares, o *scanner* do *Micro C* foi feito manualmente. Essa metodologia, apesar de implicar um maior esforço de desenvolvimento, proporciona uma compreensão mais profunda da análise léxica.

2.5.1 Interface e Implementação do *Scanner*

A interface e implementação do *scanner* é simples e direta. Ele possui uma **interface pública** (`scanner.h`) que possui uma interface acessível por qualquer parte do compilador. Essa interface é simples e possui apenas duas ações possíveis: `int inicializar_scanner(const char* nome_arquivo)`, que requisita a abertura de um novo arquivo de código; e `Token proximo_token()`, que requisita o próximo *token* dentro do código fonte.

O *scanner*, no entanto, precisa *guardar o estado* de execução. Ou seja, ele precisa determinar o arquivo e linha que estava processando. Essa *memória interna* (estado), é armazenado por duas variáveis no `scanner.c`:

- `FILE *arquivo_fonte`: armazena qual o arquivo aberto no momento.
- `int linha_atual`: armazena a linha que estava sendo avaliada, informação utilizada para reportar erros de compilação apontando a linha aproximada do erro.

A execução do compilador é dependente da correta captura de todos os *tokens* de um código fonte. Dessa forma, ele vai requisitando *tokens* para o *scanner* até que todo o código seja processado. Se o arquivo for escaneado até o fim sem erro, signfica que todos os *tokens* foram corretamente processados e estão prontos para a próxima etapa da compilação.

2.5.2 A Função `proximo_token()`

A função `proximo_token()` é o procedimento chave do analisador léxico. A cada chamada é executado um ciclo completo de reconhecimento: consumir caracteres do arquivo fonte, ignorar o que for irrelevante (espaços ou comentários) e, por fim, identificar, construir e retornar o próximo *`Token`* válido. Essa função é, na prática, uma implementação de um **autômato finito determinístico**. Ou seja, ela é uma máquina de estados que executa três fases:

1. Chamada da função (estado inicial);
2. Invocação da função `ignora_espacos_e_comentarios()` para avançar até o primeiro caractere de um potencial lexema.

3. E, uma cadeia `if-else if-switch` para classificação dos caracteres e *mudança* para o estado de reconhecimento correto ou um erro.

Além disso, a ordem das verificações determina se o *scanner* irá produzir os resultados de forma correta. Dessa forma, ele precisa avaliar os *tokens* recebidos começando pela verificação do `EOF` primeiro, para garantir que o programa pare corretamente. Depois, verifica se é uma sequência de letras e `_` (`isalpha()`). Isso permite que o *scanner* entre no *modo de reconhecimento de identificador*, que é capaz de processar letras e números (ex: `var1`). Na sequência, se não for uma letra, verifica se é um dígito (`isdigit()`), o que faria o *scanner* entrar no modo de *reconhecimento de números*. O passo seguinte é validar se estão sendo utilizados delimitadores literais como `'` e `"`. Por fim, validar todos os caracteres importantes da linguagem como os operadores (`+`, `=`, `<`) e delimitadores (`;`, `( )` ou `{ }`). O caso padrão, quando nenhum estado final válido é alcançado é encerrar o processamento em um *estado de erro*, gerando um *token* `UNDEF` de erro.

Essa ordem garante que o *scanner* tente reconhecer o *padrão mais longo* possível. Por exemplo, `int` é reconhecido como um `ID` primeiro, e só depois classificado como `INT`, em vez de ser lido erroneamente como três *tokens* `ID` separados (`i`, `n`, `t`).

2.5.3 Desafios Práticos e Soluções no Código

Essa subseção detalha a estrutura do código `scanner.c` analisando e mostrando como algumas soluções foram implementadas dependendo do desafio prático encontrado durante a compilação pelo analisador léxico.

O *Ruído* do Código: Ignorando Espaços e Comentários

A primeira tarefa do *scanner* a cada chamada de `proximo_token()` é remover o *ruído*. Isso é feito pela função `ignora_espacos_e_comentarios`. Ela é construída utilizando um laço `while(1)` (loop infinito) que só termina quando um caractere válido é encontrado e retornado.

O primeiro passo dentro do laço é consumir todos os espaços em branco e quebras de linha:

```
// arquivo: src/scanner/scanner.c (trecho 1)
static char ignora_espacos_e_comentarios() {
    char c = prox_char();
    while (1) {
        //ignora espaços em branco
        while (isspace(c)) {
            if (c == '\n') linha_atual++;
            c = prox_char();
        }
//...
```

A função `isspace()` (da biblioteca `ctype.h`) é usada para consumir rapidamente todos os caracteres de espaço (espaço, tab, etc.). Um fato importante é que é necessário verificar se o caractere sendo avaliado é um `'\n'`, pois é necessário incrementar a variável global `linha_atual` de controle que auxilia quando é necessário determinar em qual linha de código o erro ocorreu.

Após o término do laço, `c` não é um espaço em branco. Em seguida, a próxima verificação é se ele é o início de um comentário (`/`). O `if` a seguir usa e estratégia de *lookahead* (olhar o próximo caractere):

```
//... (continuação de ignora_espacos_e_comentarios)
        //verifica comentários
        if (c == '/') {
            char next_c = prox_char();
//...
```

Ao verificar que o caractere é uma barra (`/`), o *scanner* precisa decidir se se trata de um comentário ou uma operação de divisão. Essa decisão depende do próximo caractere. O que acontece é que é avaliado o próximo caractere (que é armazenado em `next_c`). Com esse caractere extra, o *scanner* consegue decidir se a sequência é um comentário ou não.

O trecho de código a seguir, demonstra essa ideia:

```
//... (continuação)
            if (next_c == '/') { //comentário de linha única
                do {
                    c = prox_char();
                } while (c != '\n' && c != EOF);
            } else if (next_c == '*') { //comentário de
            múltiplas linhas
                char prev_c = '\0';
                do {
                    prev_c = c;
                    c = prox_char();
                    if (c == '\n') linha_atual++;
                } while (!(prev_c == '*' && c == '/') && c
                != EOF);
                c = prox_char(); //consome o char após o
                '*/'
//...
```

Se `next_c` for `/`, ele entra no modo de comentário de linha e processa tudo até encontrar um `'\n'` ou `EOF`. Por outro lado, se `next_c` for `*`, ele entra no modo de comentário de bloco. Este modo é mais complexo e precisa de **estado**: ele usa a variável `prev_c` para procurar a sequência de término `'*/'`, consumindo tudo (incluindo quebras de linha) até encontrá-la.

Finalmente, se não for um comentário, ou se o caractere original não for `/`, o *scanner* trata os casos restantes:

```
//... (continuação)
            } else {
                //não é um comentário, devolve o caractere
                e retorna a barra
                ungetc(next_c, arquivo_fonte);
                return c;
            }
        } else {
            //não é espaço nem comentário, retorna o
            caractere encontrado
            return c;
        }
    }
}
```

O primeiro `else` é crucial. Se `next_c` não era `/` nem `*`, o *scanner* descobriu que era o operador de divisão. Ele usa `ungetc()` para *devolver* o caractere avaliado e retorna o `/` para a função `proximo_token()` processar. Por outro lado, o segundo `else` interrompe o laço `while(1)`. Ele retorna o caractere `c` (que pode ser uma letra, dígito, `=`, etc.) para a função `proximo_token()`, que agora pode classificar o lexema.

Identificadores vs. Palavras Reservadas: Uma Decisão em Duas Etapas

Um dos desafios centrais do *scanner* é diferenciar identificadores (ex: `contador`) de palavras reservadas (ex: `if`). A abordagem utilizada no *Micro C* segue a estratégia clássica. Ou seja, primeiro, o *scanner* reconhece o padrão mais genérico, o de `ID`, e depois, após ter o lexema completo, ele o compara com uma lista de palavras reservadas conhecidas.

Primeiro, o *consumidor* de identificadores:

```
// arquivo: src/scanner/scanner.c (trecho)
if (isalpha(c) || c == '_') {
    char buffer[101];
    int i = 0;
    do {
        buffer[i++] = c;
        c = prox_char();
    } while ((isalnum(c) || c == '_') && i < 100);

    ungetc(c, arquivo_fonte);
    buffer[i] = '\0';
//...
```

A primeira verificação é garantir que o caractere atual pode iniciar um identificador válido (O `if (isalpha(c) || c == '_')`). Em seguida, o laço `do-while` constrói o lexema no *`buffer`* enquanto os caracteres seguintes forem alfanuméricos ou `_`. E, por último, a chamada `ungetc(c, arquivo_fonte);` retorna um caractere

para continuar o processamento de forma correta. O laço só para quando é lido um caractere que *não* pertence ao identificador (ex: um espaço). Esse caractere precisa ser devolvido ao fluxo para ser processado na próxima chamada.

Com o lexema completo no *buffer*, a próxima fase é a de *classificação*:

```
//... (continuação: O Classificador)
    //--- Início da Etapa 2: Classificação ---
    if (strcmp(buffer, "main") == 0) token.tipo = MAIN;
    else if (strcmp(buffer, "if") == 0) token.tipo = IF;
    else if (strcmp(buffer, "else") == 0) token.tipo = ELSE;
    else if (strcmp(buffer, "for") == 0) token.tipo = FOR;
    else if (strcmp(buffer, "return") == 0) token.tipo =
    RETURN;
    else if (strcmp(buffer, "int") == 0) token.tipo = INT;
    else if (strcmp(buffer, "char") == 0) token.tipo = CHAR;
    else if (strcmp(buffer, "print") == 0) token.tipo =
    PRINT;
    else token.tipo = ID; // Se não for palavra reservada, é um ID

    strcpy(token.lexema, buffer);
    return token;
}
```

O bloco `if-else if` compara o lexema no *buffer* com todas as palavras reservadas da linguagem. Se nenhuma delas corresponder, ele classifica o *token* como um `ID` genérico.

Ambiguidade e a Técnica de *Lookahead*: *Observando* o Futuro

O *lookahead* é utilizado para tratar operadores que compartilham prefixos, como `=` e `==`, por exemplo. O trecho de código a seguir demonstra como um trecho de código de como o *lookahead* é implementado:

```
// arquivo: src/scanner/scanner.c (trecho do switch)
    case '=':
        next_c = prox_char(); // 1. "Observa" o próximo caractere
        if (next_c == '=') {  // 2. Compara
            token.tipo = EQ; strcpy(token.lexema, "==");
        } else {
            ungetc(next_c, arquivo_fonte); // 3. Devolve se
            não for
            token.tipo = ASSIGN; strcpy(token.lexema, "=");
        }
        break;
// ... (casos similares para '!', '<', '>') ...
```

Este exemplo ilustra como funciona o padrão de avaliação. O *scanner* lê `=`, mas *observa antecipadamente* (`prox_char()`) o caractere seguinte. Se for outro `=`, ele consome ambos e gera o *token* `EQ` (`==`). Caso contrário, ele devolve o caractere

observado antecipadamente (`ungetc()`) e gera o *token* `ASSIGN` (`=`).

O caso mais complexo é a desambiguação entre o operador de subtração (`MINUS`) e um número negativo (`INTEGERCONST`). Ao encontrar o caractere `-`, é necessário avaliar também o próximo caractere (`next_c`). Isso divide o fluxo em duas rotas possíveis.

Rota 1: Reconhecimento de Número Negativo Se o caractere seguinte for um dígito, o *scanner* entra no modo de reconhecimento de número. Esta é uma mudança de estado, ou seja, o caractere `-` não é mais um operador, mas sim o prefixo de um literal numérico. O processo de construção do *token* se inicia armazenando o `-` em um *buffer* temporário. Em seguida, o *scanner* entra em um laço para consumir o primeiro dígito (que foi *observado*) e todos os dígitos subsequentes, anexando cada um ao *buffer*. O laço é interrompido apenas quando o primeiro caractere não-dígito é encontrado. Finalmente, o *token* é classificado como `INTEGERCONST` e o lexema completo (ex: `-123`) é salvo.

O bloco de código a seguir detalha esta implementação:

```
// arquivo: src/scanner/scanner.c (Rota 1: Número Negativo)
case '-':
    next_c = prox_char(); // Espia o caractere após o '-'
    if (isdigit(next_c)) { // Se for um dígito...
        // ...então é um número negativo.
        char buffer[12];
        buffer[0] = c; // c == '-'
        int i = 1;
        // (Laço para consumir o restante dos dígitos)
        do {
            buffer[i++] = next_c;
            next_c = prox_char();
        } while (isdigit(next_c) && i < 11);

        ungetc(next_c, arquivo_fonte); // Devolve o não-dígito
        buffer[i] = '\0';
        token.tipo = INTEGERCONST;
        strcpy(token.lexema, buffer);
    } else {
        /* ... (trata operador MINUS, veja Rota 2) ... */
    }
    break;
```

Rota 2: Reconhecimento do Operador de Subtração Contudo, se o caractere seguinte *não* for um dígito (como um espaço, uma letra ou outro operador), o *scanner* identifica que se trata do operador `MINUS`. Ou seja, ele devolve o caractere observado (`next_c`) de volta ao fluxo de entrada. O trecho de código a seguir detalha esta implementação:

```
// arquivo: src/scanner/scanner.c (Rota 2: Operador MINUS)
case '-':
    next_c = prox_char(); // Espia o caractere após o '-'
    if (isdigit(next_c)) { // Se for um dígito...
        /* ... (trata número negativo, veja Rota 1) ... */
    } else { // Se não for um dígito...
        // ...então é o operador MINUS.
        ungetc(next_c, arquivo_fonte); // Devolve o não-dígito
        token.tipo = MINUS;
        strcpy(token.lexema, "-");
    }
    break;
```

Esta abordagem demonstra uma técnica central do analisador léxico: o uso de *lookahead* (olhar à frente) com `prox_char()` para tomar uma decisão e, caso o caractere *observado* não pertença ao *token* atual (como na Rota 2), devolvê-lo ao fluxo de entrada com `ungetc()` para que seja processado na próxima iteração.

Reconhecimento de Literais: Strings e Caracteres

Finalmente, o *scanner* precisa reconhecer literais definidos por seus delimitadores (`'` e `"`). Este é um tipo de reconhecimento que torna claro como são tratados os erros léxicos pelo compilador *Micro C*. O trecho a seguir mostra como o *scanner* trata essa situação:

```
// arquivo: src/scanner/scanner.c (trecho)
if (c == '"') {
    char buffer[101];
    int i = 0;
    c = prox_char(); // Pula a aspa de abertura

    // Consome caracteres até encontrar o fim, uma nova
    linha ou EOF
    while (c != '"' && c != '\n' && c != EOF && i < 100) {
        buffer[i++] = c;
        c = prox_char();
    }
    buffer[i] = '\0';
//...
```

Ao avaliar o `"`, o *scanner* entra no *modo string* e começa a salvar os caracteres no *`buffer`*. O laço `while` será executado até encontrar a aspa de fechamento (`"`) ou até encontrar um erro como, por exemplo, uma quebra de linha (`\n`) ou o fim do arquivo (`EOF`), pois strings na linguagem não podem ter múltiplas linhas. Continuando o reconhecimento da string:

```
//... (continuação do reconhecimento de string)
    if (c == '"') { // Se parou por causa da aspa...
        token.tipo = STRINGCONST;
        strcpy(token.lexema, buffer);
    } else { // Se parou por '\n' ou EOF...
        token.tipo = UNDEF; // É um erro!
        strcpy(token.lexema, "String nao terminada");
    }
    return token;
}
```

Após o laço, o `if (c == '"')` verifica *por que* o laço parou. Se foi por causa da aspa, o *token* é um `STRINGCONST` válido. Se foi por qualquer outro motivo, é um erro, e é retornado o *token* `UNDEF` com uma mensagem de erro. O processamento de constantes de caracteres `CHARCONST` é processado de forma similar.

2.6 Sumário

Neste capítulo, iniciamos o processo de compilação com a primeira fase: a **análise léxica**. Foram definidos os três conceitos centrais desta etapa: o ***token*** (categoria abstrata), o **lexema** (texto do código fonte) e o **padrão** (a regra que descreve o lexema).

Foi possível observar que o analisador léxico, ou *scanner*, atua como a *primeira linha de processamento* do compilador, sendo responsável por validar o texto de entrada e reportar erros léxicos, como caracteres inválidos (`@`) ou strings malformadas.

Além disso, foram discutidas decisões de projeto que deixam claro porque algumas decisões de implementação foram tomadas ao desenvolver o *scanner*. Por exemplo, optando pelo uso da abordagem *em lote* ao invés de outras mais comuns que são utilizadas em compiladores clássicos.

Em seguida, discutiu-se a implementação do vocabulário (arquivo `tokens.h`) da linguagem e como os tipos de tokens são processados com o uso de estruturas de dados como a `struct Token`, que agrupa o tipo, o lexema e o número da linha associada ao lexema. A estrutura dessa forma, permite que erros comuns sejam informados ao programador apontando em qual linha de código o erro ocorreu.

Finalizando, foram abordados conceitos importantes que são utilizados na implementação do `scanner.c` como o uso de expressões regulares e autômatos finitos determinísticos. O *Micro C* segue os mesmos princípios e não utiliza mecanismos automatizados de geração de AFD's como Lex ou Flex. Isso tornou o *scanner* mais simples, mas ao mesmo tempo, dependente da linguagem definida.

Capítulo 3

Análise Sintática

Trees sprout up just about everywhere in computer science.

– Donald E. Knuth

Neste capítulo será discutida a construção do analisador sintático do compilador da linguagem *Micro C*. Após a conversão do código fonte em uma sequência de *tokens* pelo analisador léxico, o processo de compilação realiza a análise sintática. Esta fase tem como objetivo verificar se a estrutura do código e as regras gramaticais da linguagem estão em conformidade. Ou seja, o analisador sintático valida se a sequência de *tokens* forma construções sintáticas válidas, tais como expressões, comandos e declarações, em concordância com uma gramática formal previamente definida.

Nesta fase da compilação é que a estrutura do programa começa a ganhar forma. Além de simplesmente validar a gramática, a responsabilidade principal desta fase é *construir* a estrutura de dados hierárquica que será usada por todas as fases subsequentes. Essa estrutura, conhecida como Árvore Sintática Abstrata (ASA), é o resultado da análise sintática e serve como a representação concreta da lógica do programa, pronta para a análise de significado (semântica).

O analisador sintático recebe como entrada uma cadeia de *tokens* fornecida pelo analisador léxico e verifica se essa cadeia pertence à linguagem definida pela gramática do *Micro C*. Caso encontre erros de sintaxe, o analisador deve ser capaz de reportá-los de forma clara e, sempre que possível, seguir adiante com a análise, permitindo que outros erros também sejam detectados.

Existem três abordagens principais para a construção de analisadores sintáticos: a abordagem *universal*, a *descendente* e a *ascendente* [1, 3, 13]. A abordagem universal é capaz de analisar qualquer gramática, mas seus algoritmos, como o clássico **algoritmo de Earley [7]**, são pouco eficientes. Eles precisam explorar múltiplas árvores de derivação possíveis, um processo computacionalmente custoso que atinge a complexidade de $O(n^3)$ em relação ao tamanho do programa e, por isso, raramente

são usados em compiladores reais. As estratégias realmente utilizadas em compiladores são as análises descendente e ascendente. A análise descendente constrói a árvore de derivação do topo (raiz) até as folhas, enquanto a análise ascendente faz o caminho inverso, começando das folhas em direção à raiz. Em ambos os casos, os *tokens* são processados da esquerda para a direita. O *Micro C* utiliza a estratégia descendente, por simplicidade.

Nas próximas seções serão discutidas como essas técnicas funcionam, usando exemplos práticos da linguagem do *Micro C*. Também sera abordado como o analisador sintático trata erros e como se conecta com as demais fases do compilador.

3.1 Expressões e Gramáticas

Grande parte da complexidade da análise sintática em linguagens de programação está relacionada ao processamento de expressões, principalmente por causa das regras de associação e precedência entre operadores. Enquanto construções iniciadas por palavras reservadas como `if`, `while` ou `int` são relativamente fáceis de identificar e tratar, expressões exigem mais cuidado, já que podem gerar ambiguidade dependendo da ordem em que os elementos são analisados.

Para tratar essa situação, são utilizadas gramáticas que incorporam essas regras. Uma forma tradicional de representar expressões é por meio de uma estrutura em árvore onde os operadores de menor precedência aparecem no topo da árvore, e os de maior precedência mais próximo das folhas.

3.1.1 Hierarquia de Precedência em Expressões

Para que a gramática possa tratar corretamente a precedência e associatividade de operadores, ela é dividida em três níveis hierárquicos, utilizando os símbolos não terminais `E` (para Expressão), `T` (para Termo) e `F` (para Fator) para expressar essa hierarquia. Essa estrutura é um padrão clássico no projeto de compiladores, que garante a ordem correta de avaliação das operações matemáticas, por exemplo. A estrutura lógica é construída do nível de maior para o de menor precedência. A seguir uma descrição de cada um dos níveis hierárquicos definidos:

- **Fator (F):** representa a unidade de maior precedência em uma expressão. Um fator é algo que pode ser avaliado imediatamente, como um identificador (`id`), um número (`num`), ou uma *expressão* inteira entre parênteses `(E)`. O uso de parênteses altera a precedência natural, forçando a avaliação do que está dentro antes de qualquer outra operação. Isso gera a regra para Fator[1]: $F \rightarrow (E) \mid \text{id} \mid \text{num}$.

[1] A leitura desta notação, *adotada ao longo de todo este trabalho*, deve ser realizada da seguinte forma: a seta ($\rightarrow$) deve ser interpretada como "pode ser formado por" e a barra vertical ($\mid$) como um "ou". Sendo assim, a regra define, especificamente, que um **Fator** pode ser formado por uma expressão entre parênteses, **ou** por um identificador, **ou** por um número.

- **Termo (T):** representa uma sequência de fatores conectados pelos operadores de alta precedência: multiplicação (`*`) e divisão (`/`). Um termo pode ser um único fator, ou uma multiplicação/divisão de um termo com outro fator. Isso gera a regra para Termo: $T \rightarrow T * F \mid T/F \mid F$.
- **Expressão (E):** representa a forma mais geral, composta por uma sequência de termos conectados pelos operadores de baixa precedência: soma (`+`) e subtração (`-`). Uma expressão pode ser um único termo, ou uma soma/subtração de uma expressão com outro termo. Isso gera a regra para Expressão: $E \rightarrow E + T \mid E - T \mid T$.

Dessa forma, é criada uma hierarquia bem definida, ou seja, uma **Expressão** é feita de **Termos**, e um **Termo** é feito de **Fatores**. Como as operações de maior precedência (`*`, `/`) estão "aninhadas"na regra de `T`, que está um nível abaixo de `E`, a gramática determina que as multiplicações e divisões sejam reconhecidas e agrupadas antes das somas e subtrações, garantindo a ordem correta da avaliação matemática.

3.1.2 Gramática como Especificação Formal da Linguagem

A gramática é a ferramenta formal que permite ao compilador entender qual a sequência de símbolos que forma um programa válido. Ela funciona como uma descrição precisa da linguagem, definindo as regras que determinam a organização correta dos elementos no código. Assim como a análise léxica identifica os *tokens*, a gramática define como esses *tokens* se combinam para formar construções sintaticamente corretas, como expressões, comandos e blocos.

Em outras palavras, a gramática é um conjunto de instruções que explica ao compilador o que é esperado na estrutura do programa. Qualquer código que não obedeça a essas regras será considerado inválido durante a análise sintática, garantindo que erros estruturais sejam detectados o quanto antes no processo de compilação.

Dessa forma, a gramática atua como um contrato entre o programador e o compilador, assegurando que o código esteja organizado conforme as convenções da linguagem. Por isso, o estudo e a definição da gramática são essenciais para que o compilador possa interpretar corretamente o código fonte e avançar para as etapas seguintes da compilação.

Para ilustrar o papel da gramática, considere uma linguagem extremamente simples de expressões aritméticas que permite somar números e agrupar operações entre parênteses. Essa linguagem pode ser descrita pela seguinte gramática:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow (E) \mid \textbf{num}$$

Nessa gramática, o símbolo inicial é E, que representa uma expressão. A regra diz que uma expressão pode ser a soma de outra expressão com um termo ($E + T$)

ou apenas um termo (T). O termo, por sua vez, pode ser uma expressão entre parênteses, permitindo agrupamento, ou um número terminal, representado aqui por **num**.

Assim, uma sequência válida nessa linguagem é `3 + (5 + 2)`, pois a gramática especifica que números e somas entre expressões agrupadas são aceitos. Por outro lado, uma sequência como `3 + + 5` não é válida, pois não existe regra na gramática que permita dois operadores `+` consecutivos.

A gramática completa que representa essa hierarquia é a seguinte:

$$
\begin{aligned}
E &\to E + T \mid E - T \mid T \\
T &\to T * F \mid T/F \mid F \\
F &\to (E) \mid \text{id} \mid \text{num}
\end{aligned}
$$

Embora a gramática acima pareça expressar de forma natural a hierarquia das operações aritméticas, ela apresenta um problema para a **análise sintática descendente**: a recursão à esquerda. É importante destacar que, enquanto analisadores ascendentes lidam bem com regras recursivas à esquerda, na abordagem descendente essa estrutura inviabiliza o algoritmo. O exemplo a seguir deixará mais claro o porquê dessa situação ocorrer. Suponha a seguinte expressão, que será avaliada passo a passo:

```
a + b * c
```

Ao tentar derivar essa expressão a partir do símbolo inicial E, a estratégia natural seria expandir E de acordo com a produção:

$$E \to E + T$$

No entanto, essa escolha faz com que o símbolo E apareça novamente no lado direito da produção, logo na primeira posição. Assim, o analisador tentará expandir E novamente por $E + T$, e em seguida mais uma vez, e assim sucessivamente, sem processar nenhum símbolo de entrada. Isso gera um ciclo infinito de derivações, como ilustrado abaixo:

$$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow E + T + T + T \Rightarrow \ldots$$

Para a correta interpretação desta demonstração, deve-se notar a mudança na simbologia: diferente da seta simples ($\to$), que define uma regra estática, a seta dupla ($\Rightarrow$) representa um passo de **derivação**. Ou seja, ela indica uma ação de transformação. A sequência acima deve ser lida como: o símbolo E *deriva* em $E+T$, que por sua vez *deriva* em $E+T+T$, e assim sucessivamente, demonstrando a expansão infinita.

Em outras palavras, o analisador continua expandindo E indefinidamente, sem avançar na análise dos *tokens* do programa.

Esse comportamento caracteriza a **recursão à esquerda direta**, um padrão em que um não terminal faz referência a si mesmo como o primeiro símbolo de seu lado direito. A gramática apresentada contém esse tipo de recursão nas produções de E e T:

$$E \to E + T \qquad \text{e} \qquad T \to T * F$$

Esse problema impede o funcionamento correto de analisadores descendentes recursivos, que baseiam suas decisões em chamadas de funções recursivas. Portanto, antes de construir o analisador sintático propriamente dito, será necessário reformular essa gramática para eliminar a recursão à esquerda.

Para resolver o problema da recursão, transformamos a gramática eliminando a recursão à esquerda e reestruturando as produções.

O resultado é uma gramática equivalente, mas adequada para análise descendente:

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid -TE' \mid \epsilon \\ T &\to FT' \\ T' &\to *FT' \mid /FT' \mid \epsilon \\ F &\to (E) \mid \text{id} \mid \text{num} \end{aligned}$$

Ao analisar novamente a expressão, é possível notar que o problema foi resolvido:

```
a + b * c
```

Na nova gramática, a derivação começa pelo símbolo E e segue pela produção:

$$E \to TE'$$

Primeiro, T consome o termo mais à esquerda, que é a:

$$T \Rightarrow FT' \Rightarrow \text{id}\, T'$$

Em seguida, E' processa os operadores à direita. O próximo *token* é $+$, então é aplicada a derivação:

$$E' \Rightarrow +TE'$$

O T seguinte processa $b * c$ de forma correta, utilizando $T \rightarrow FT'$ e $T' \rightarrow *FT'$. Ao final, a gramática permite a derivação $E' \Rightarrow \epsilon$, encerrando o processamento.

Dessa forma, cada operador e operando é processado na ordem correta, respeitando a precedência de multiplicação sobre adição, e o analisador descendente consegue processar todos os *tokens* sem entrar em um ciclo infinito.

Contextualização da Implementação: Estruturas e Fluxo

Para compreender a implementação prática da análise sintática no *Micro C* é necessário, primeiramente, entender como o analisador é estruturado e quais dados ele manipula. O compilador utiliza uma abordagem **Descendente Recursiva** (*Recursive Descent*), onde a estrutura das funções em C espelha diretamente a hierarquia da gramática.

Em alto nível, o fluxo de execução começa na regra mais geral (o Programa) e desce, por meio de chamadas de funções, até atingir os elementos atômicos (Fatores e Tokens), conforme ilustrado na Figura 3.1.

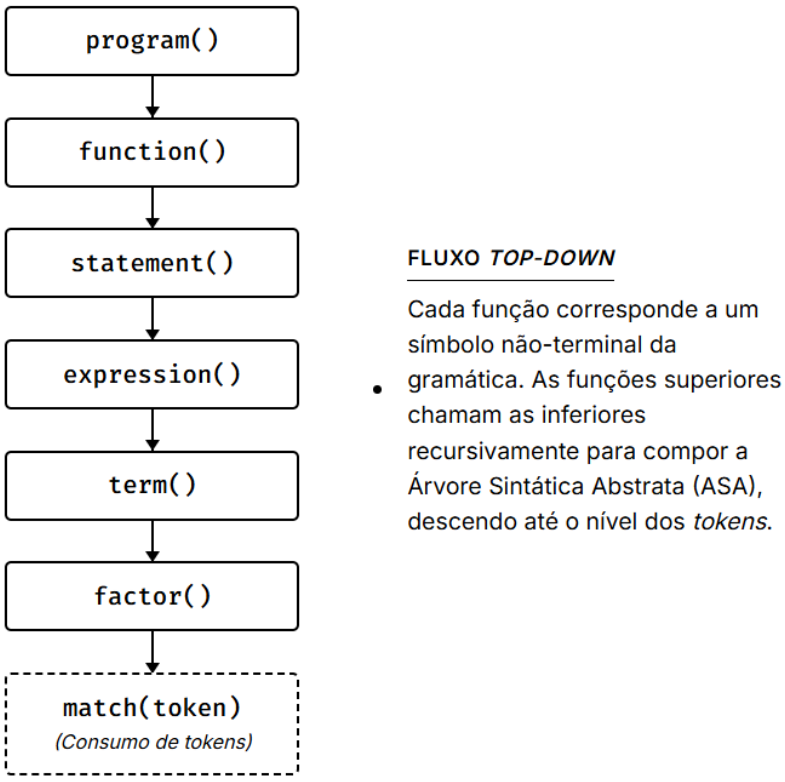


Figura 3.1: Fluxo de chamadas do Analisador Descendente Recursivo.

É importante notar a distinção entre o fluxo de controle e a construção da estrutura de dados. O analisador segue uma estratégia ***Top-Down*** (descendente), navegando das regras gramaticais mais gerais (Programa) para as mais específicas

(Fatores). No entanto, a Árvore Sintática Abstrata (ASA) é efetivamente montada de maneira ***Bottom-Up*** (ascendente): os nós folhas (números e variáveis) são criados primeiro nas funções mais profundas da recursão e, à medida que as funções retornam, esses nós são capturados e agrupados por nós operadores (pais) nas funções superiores.

Para sustentar esse processo, o analisador manipula duas estruturas de dados fundamentais:

1. A Árvore Sintática (ASTNode)

Diferente de implementações simplificadas que utilizam vetores fixos, o *Micro C* adota uma estrutura dinâmica baseada em listas encadeadas. Conforme definido em `ast.h`, cada nó possui um ponteiro para seu primeiro filho (`filho`) e um ponteiro para o seu próximo irmão (`proximo_irmao`). Essa flexibilidade permite que um nó pai tenha quantidade variável de filhos.

2. O Estado do Analisador (Parser)

O controle do fluxo de *tokens* é encapsulado na estrutura *`Parser`*, que mantém a referência para a lista completa de *tokens* carregada do arquivo e um cursor para o elemento atual.

As definições reais dessas estruturas, utilizadas no código, são apresentadas abaixo:

```
// --- Definições baseadas em ast.h e parser.h ---

//estrutura de um Nó da Árvore (ASA)
typedef struct ASTNode {
    NodeType node_type;           //tipo do nó (ex: NODE_BINARY_OP)
    int linha;                    //linha de origem no código

    struct ASTNode *filho;        //ponteiro para o primeiro filho
    struct ASTNode *proximo_irmao;//ponteiro para o próximo irmão (lista)

    union {
        long int_value;           //para constantes inteiras
        char* string_value;       //para identificadores e strings
        TokenType op_type;        //para operadores (+, -, *, /)
    } data;
} ASTNode;

//estrutura de controle do Parser
typedef struct {
    TokenList *lista;       //lista contendo todos os tokens lidos
    int current;            //índice numérico do token atual
    Token *current_token;   //ponteiro direto para o token atual
    // ... outros campos (tabela de símbolos)
} Parser;
```

Com o contexto das estruturas estabelecido, a implementação da função `arithmetic_expression` pode ser analisada.

Para a implementação no *Micro C*, realiza-se uma otimização em relação à gramática teórica. É essencial distinguir que, ao eliminar a recursão à esquerda (que causa *loops* infinitos), a gramática teórica resultante passa a utilizar **recursão à direita** no símbolo auxiliar E'. Embora a recursão à direita seja válida para analisadores descendentes, pois consome *tokens* antes da chamada recursiva, convertê-la diretamente em código geraria uma função extra apenas para processar o *resto* da expressão.

Por isso, na prática, não se cria uma função recursiva separada para E'. A lógica de repetição definida por E', que continuaria chamando a si mesma ao final da regra, é substituída de forma equivalente e mais eficiente por um laço iterativo (`while`). O laço continua rodando enquanto houver operadores $(+, -)$, desempenhando exatamente o mesmo papel lógico da recursão à direita, mas sem o custo de múltiplas chamadas de função e uso excessivo da pilha. O código abaixo demonstra como a árvore é construída de baixo para cima (*bottom-up*) dentro do laço, garantindo a precedência correta:

```
ASTNode* arithmetic_expression(Parser* parser) {
    //1. processa o termo da esquerda (maior precedência,
    ex: multiplicações)
    ASTNode* node = term(parser);

    //2. loop para processar somas (+) e subtrações (-) sequencialmente
    while (parser->current_token &&
          (parser->current_token->tipo == PLUS ||
           parser->current_token->tipo == MINUS)) {

        TokenType op = parser->current_token->tipo;
        int line = parser->current_token->linha;
        match(parser, op); //consome o operador e avança

        //cria o nó da operação binária
        ASTNode* op_node = criar_no(NODE_BINARY_OP, line);
        op_node->data.op_type = op;

        //a árvore cresce para cima:
        //o nó acumulado node vira o filho da esquerda do novo operador
        adicionar_filho(op_node, node);

        //o próximo termo vira o filho da direita
        adicionar_filho(op_node, term(parser));

        //atualiza a raiz da sub-árvore para o novo operador
        node = op_node;
    }
    return node;
}
```

Essa implementação garante que uma expressão como `A + B + C` seja estruturada corretamente. O laço processa primeiro `A + B`, criando um nó. Na próxima iteração, esse nó torna-se filho da esquerda da soma com `C`. Isso respeita a associatividade à esquerda da linguagem C e evita o estouro de pilha por recursão infinita.

3.2 Tratamento de Erros na Análise Sintática

Durante a análise sintática, é inevitável que erros de sintaxe apareçam no código fonte. O papel do analisador sintático não é apenas detectar esses erros, mas também tentar se recuperar deles para continuar a análise e identificar o máximo possível de problemas em uma única execução. Para isso, diversas estratégias de recuperação de erro podem ser utilizadas, cada uma com suas vantagens e limitações.

Modo de pânico. Essa é uma das estratégias mais comuns. Quando um erro é detectado, o analisador descarta símbolos da entrada até encontrar um *token* que pertença a um conjunto de sincronização (geralmente delimitadores como ponto e vírgula ou chaves). Isso permite que o analisador pule instruções malformadas e continue com o restante do código.

```
int a, 5abcd, sum, $2;
```

Neste exemplo, o *parser* reconhece o `int a` como uma declaração válida. Contudo, quando encontra *tokens* inválidos como 5abcd ou $2, ele os ignora, sem interromper a análise. A leitura continua até que seja encontrado o `;` que indica o fim da declaração, permitindo que o restante do código seja processado corretamente.

Essa abordagem é simples e evita que o compilador entre em ciclos infinitos. No entanto, pode mascarar erros posteriores, levando a interpretações incorretas na análise semântica ou na geração de código.

Recuperação de nível de frase. Essa técnica tenta realizar correções locais. Quando um erro é detectado, o analisador tenta fazer ajustes simples na entrada, como inserir um ponto e vírgula ausente ou corrigir uma vírgula no lugar errado.

> **Exemplo:** `int a, b` ⇒ `int a, b;`
> (O compilador insere o ponto e vírgula faltando.)

É uma abordagem eficiente e adotada por diversos compiladores, mas exige cuidado para evitar correções em cascata que causem ciclos ou novos erros.

Produções de erro. Nesse método, a própria gramática é enriquecida com produções específicas para reconhecer construções comuns incorretas. Quando uma dessas produções é acionada, uma mensagem de erro personalizada pode ser gerada. Isso exige conhecimento prévio sobre os tipos de erro que os programadores geralmente cometem.

> **Exemplo:** Se a gramática original não reconhece uma entrada como `abcd`, pode ser introduza uma produção auxiliar para aceitar esse padrão e indicar o erro:
>
> ```
> E → S B
> S → A
> A → aA | bA | a | b
> B → cd
> ```

Apesar de poderosa, essa estratégia é de difícil manutenção, pois qualquer mudança na gramática principal exige ajustes nas produções de erro.

Correção global. É uma abordagem teórica que tenta transformar a cadeia de entrada incorreta na mais próxima possível de uma entrada válida, realizando o menor número de alterações (inserções, remoções ou substituições de *tokens*). Embora seja um conceito interessante, não é aplicável na prática devido ao alto custo computacional.

Verificação de Tipos e Coerção. Embora o gerenciamento de tipos seja de responsabilidade da análise semântica, o analisador sintático pode interagir com a tabela de símbolos para validar operações básicas. É importante distinguir erro de conversão. Quando ocorre uma incompatibilidade de tipos que a linguagem suporta (como atribuir um valor real a um inteiro), não se trata de um erro, mas sim de uma **coerção** (conversão implícita). Nesses casos, o compilador utiliza as informações da tabela de símbolos para ajustar o valor automaticamente, garantindo a continuidade da compilação sem emitir falhas.

> **Exemplo de Coerção:** `int x = 5.2;` ⇒ `int x = (int)5.2;`
> *(O compilador trunca o valor para 5)*

A tabela de símbolos torna-se essencial para a recuperação de erros apenas quando essa conversão não é possível (por exemplo, tentar atribuir uma *string* a uma variável numérica), momento em que o compilador deve, efetivamente, reportar a incompatibilidade e tentar sincronizar a análise.

No *Micro C*, o analisador sintático não implementa estratégias de recuperação de erro avançadas, como modo de pânico, correção de nível de frase, produções de erro ou correção global. O propósito do compilador é didático: ele interrompe a análise ao encontrar o primeiro erro sintático, garantindo que a compreensão da estrutura da gramática e da árvore sintática abstrata seja clara e simples.

3.3 Leitura dos *Tokens*

Após a etapa de análise léxica, o compilador gera uma sequência de *tokens* que representa os elementos significativos do código fonte. O analisador sintático, por sua

vez, trabalha diretamente com essa sequência de *tokens* para verificar se a estrutura do programa está correta de acordo com a gramática da linguagem.

Neste estágio, a entrada para o analisador sintático não é mais o código fonte original, mas uma lista de *tokens* previamente gerados pelo analisador léxico. Cada *token* contém informações essenciais, como o tipo (representando a categoria do lexema), o lexema (que e a sequência de caracteres que corresponde ao *token*), e a linha do código onde o *token* foi encontrado. Com esses dados, o analisador sintático consegue construir a árvore de derivação e verificar se a construção sintática do programa está de acordo com as regras definidas pela gramática da linguagem.

O objetivo desta seção é descrever o processo de leitura e armazenamento desses *tokens* para que possam ser usados pelo analisador sintático. O arquivo de entrada utilizado para esse processo, o `tokens.txt`, contém os *tokens* gerados pela fase de análise léxica, e é a partir deste arquivo que o analisador sintático vai realizar sua análise. Um exemplo real da estrutura interna desse arquivo, gerado a partir do código fonte `soma.mcc`, pode ser consultado no Apêndice D.2 *Análise Léxica*.

A seguir, será discutida a implementação que permite a leitura desse arquivo e o armazenamento dos *tokens* em uma estrutura apropriada para posterior uso no processo de análise sintática.

3.3.1 Estrutura para Armazenamento de *Tokens*

Antes de prosseguir, é importante entender como os dados são organizados. Para isso, é utilizada a estrutura `TokenList` que mantém um vetor dinâmico de *tokens* e garante que a memória seja gerenciada de forma eficiente e todos os *tokens* gerados pela análise léxica sejam acessíveis, pelo analisador sintático.

A estrutura `TokenList` é definida como segue:

```
typedef struct {
  Token *tokens;
  int tamanho;
  int capacidade;
} TokenList;
```

A estrutura `TokenList` é formada por três membros: `tokens`, que é um ponteiro para o vetor de *tokens*; `tamanho`, que é o número atual de *tokens* armazenados; e, `capacidade`, que é a quantidade máxima de *tokens* que a estrutura pode armazenar antes de precisar ser redimensionada.

3.3.2 Leitura do Arquivo `tokens.txt`

A leitura do arquivo `tokens.txt` é realizada pela função `carregar_tokens` no *Micro C*. Essa função abre o arquivo e processa cada linha usando a função `sscanf` que

faz a leitura formatada de uma linha extraindo tipo, lexema e número de cada linha para cada *token*.

O trecho abaixo ilustra a leitura dos tokens do arquivo `tokens.txt`:

```
int carregar_tokens(const char *nome_arquivo, TokenList *lista){
    FILE *arquivo = fopen(nome_arquivo, "r");
    if (arquivo == NULL) {
        return -1; }

    char linha_str[256];
    while (fgets(linha_str, sizeof(linha_str), arquivo)) {
    int tipo, linha_num;
    char lexema[100];

    //extrai os dados da linha formatada
    if (sscanf(linha_str, "Token: tipo = %d, lexema = '%[^']',
    linha = %d", &tipo, lexema, &linha_num) != 3) {
        continue; }

    //verifica se é necessário redimensionar a lista
    if (lista->tamanho >= lista->capacidade) {
        lista->capacidade *= 2;
        Token *nova_lista = realloc(lista->tokens,
                                    lista->capacidade * sizeof(Token));
        if (nova_lista == NULL) {
            fclose(arquivo);
            return -1; //falha de memória
        }
        lista->tokens = nova_lista;
    }

    //armazena o token na lista
    lista->tokens[lista->tamanho].tipo = (TokenType)tipo;
    strncpy(lista->tokens[lista->tamanho].lexema, lexema, 100);
    lista->tokens[lista->tamanho].linha = linha_num;
    lista->tamanho++;
    }
    fclose(arquivo);
    return 0;
}
```

A função lê o arquivo linha a linha utilizando um *buffer* temporário (`linha_str`). O formato esperado para cada linha segue o padrão definido na etapa léxica:

```
Token: tipo = <tipo do token>, lexema = <texto>, linha = <número da linha>
```

Esta função abre o arquivo `tokens.txt` e lê cada linha, que contém as informações do *token*. Caso a leitura da linha seja bem sucedida, as informações do *token* são extraídas e armazenadas na lista de *tokens*. Se a lista estiver cheia, ela é redimensionada dinamicamente para comportar novos *tokens*.

Notem que, ao lidar com a lista de *tokens*, é importante garantir que a memória

seja gerida de forma eficiente, redimensionando o vetor conforme necessário para evitar desperdício ou falta de espaço. O uso da função `realloc` garante que a lista de *tokens* possa crescer dinamicamente durante o processamento do arquivo.

3.3.3 Conectando as Fases Léxica e Sintática

Neste estágio do compilador, a leitura dos *tokens* serve como uma *ponte* entre a fase de análise léxica e a análise sintática. A fase de análise léxica, que ocorre primeiro, tem como objetivo identificar os *tokens* no código fonte. Esses *tokens* são, então, carregados no analisador sintático para que a estrutura do programa seja verificada e processada.

É importante notar que, nesta etapa de leitura dos *tokens*, ainda não há validação sintática. Ou seja, o objetivo não é verificar se a sequência de *tokens* segue as regras da gramática da linguagem de programação. Em vez disso, estamos simplesmente carregando e organizando os *tokens* identificados pela análise léxica para que possam ser usados na próxima fase de análise sintática. Esse processo garante que o compilador tenha acesso à sequência de *tokens* que será analisada mais detalhadamente na fase subsequente.

Dessa forma, a etapa de leitura dos *tokens* fornece uma entrada organizada e estruturada ao analisador sintático, permitindo que ele realize suas operações com eficiência, sem precisar se preocupar com a identificação e categorização de *tokens*, uma vez que isso já foi feito pela análise léxica. Ou seja, os *tokens* armazenados no arquivo representam a única interface entre a análise léxica e a análise sintática, portanto qualquer erro nessa transição pode comprometer toda a compilação.

Para facilitar o acesso sequencial e ordenado a essas informações, utilizamos uma estrutura em C chamada `TokenList`, que é um vetor dinâmico de estruturas `Token`. Cada elemento dessa lista contém três informações: o tipo do *token* (valor numérico), seu *lexema* (texto associado) e a linha do código fonte onde ele foi encontrado.

Essa organização em forma de lista permite que o analisador sintático percorra os *tokens* de maneira eficiente, respeitando a ordem em que foram identificados. Além disso, ao manter o número da linha, facilita-se a geração de mensagens de erro mais informativas durante a análise sintática, contribuindo para uma melhor depuração do código fonte pelo programador.

Observem que a separação entre a análise léxica e a sintática permite que cada fase seja desenvolvida, testada e mantida de forma independente. A estrutura de dados escolhida para armazenar os *tokens* serve como uma ponte entre essas fases, garantindo que os dados sejam compartilhados de forma clara e organizada.

O trecho abaixo mostra um exemplo do arquivo `tokens.txt`:

```
// Conteúdo exemplo de um arquivo tokens.txt
Token: tipo = 31, lexema = 'int', linha = 1
Token: tipo = 1, lexema = 'x', linha = 1
Token: tipo = 19, lexema = '=', linha = 1
Token: tipo = 2, lexema = '10', linha = 1
Token: tipo = 20, lexema = ';', linha = 1
```

Esse trecho simula a saída do analisador léxico para uma linha simples de declaração e atribuição. Cada linha representa um *token* com seu tipo, conteúdo textual e a linha correspondente do código-fonte. Ao serem carregados em uma estrutura `TokenList`, esses dados são organizados sequencialmente, mantendo a integridade da ordem original do programa, o que é essencial para a análise sintática.

Compreender como os *tokens* são lidos e organizados pelo *scanner* nos permite preparar o *parser* para a análise sintática. Na seção a seguir, será introduzida a *Gramática da Linguagem*, que define formalmente as regras que o *parser* seguirá para validar a estrutura do programa. Serão descritos os símbolos terminais e não terminais, bem como as convenções adotadas para representar expressões, funções e instruções do *Micro C*, garantindo que a análise sintática esteja totalmente alinhada com a implementação prática do compilador.

3.4 Gramática da Linguagem

Gramáticas são manuais de regras que definem a estrutura de uma linguagem de programação. No *Micro C*, elas descrevem como funções, declarações, expressões e instruções devem ser organizadas, sendo essenciais para a construção do compilador e para a geração da Árvore Sintática Abstrata (ASA).

Uma *Gramática Livre de Contexto* (GLC) é um tipo especial de gramática que expressa estruturas hierárquicas. No *Micro C*, ela permite representar funções, blocos, expressões aninhadas, laços e comandos condicionais, refletindo diretamente a implementação do *parser* recursivo descendente.

Além disso, cada tipo de construção sintática possui correspondência direta com funções do *parser*. Por exemplo, as funções `function()`, `var_declaration()`, `expression()`, `arithmetic_expression()`, `term()` e `factor()` representam não terminais que estruturam a linguagem e determinam a hierarquia de operações e regras de escopo. Esse mapeamento garante que a gramática esteja alinhada com a implementação prática da linguagem, considerando que os *tokens* são fornecidos pelo *scanner*.

3.4.1 Notações de Gramática

Para evitar ambiguidades e facilitar a leitura, foram adotadas convenções claras para representar os símbolos e produções da gramática do *Micro C*:

1. **Símbolos terminais:** representam *tokens* reconhecidos pelo scanner, como:
   - Palavras reservadas: `int`, `char`, `if`, `else`, `for`, `return`, `print`.
   - Operadores e símbolos de pontuação: `+`, `-`, `*`, `/`, `%`, `=`, `==`, `<`, `>`, `<=`, `>=`, `!=`, `;`, `,`, `(`, `)`, `{`, `}`, `[`, `]`.
   - Literais e identificadores: **id**, **num**, **char_const**, **string_const**.
2. **Símbolos não terminais:** representam construções sintáticas do *Micro C*:
   - `program` → representa o ponto de entrada do compilador, contendo uma sequência de funções.
   - `function` → definição completa de função, incluindo tipo de retorno, nome, parâmetros e corpo.
   - `type` → especificador de tipo (`int` ou `char`).
   - `var_declaration` → declaração de variáveis locais ou parâmetros.
   - `expression`, `arithmetic_expression`, `term`, `factor` → hierarquia de expressões, determinando precedência e associatividade de operadores.
   - `arg_list` e `param_list` → listas de argumentos em chamadas de função e parâmetros de funções, respectivamente.
   - `statement` → representa instruções individuais, como atribuições, laços (`for`), condicionais (`if-else`) e comandos de saída (`print`).
3. **Produções:** uma regra genérica da gramática é escrita como:

   $$A \to \alpha$$

   onde A é um não terminal e α é uma sequência de símbolos terminais e/ou não terminais. Alternativas podem ser representadas com o operador `|`:

   $$A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

4. **Símbolo inicial:** o não terminal da primeira produção é o símbolo inicial da gramática, que no *Micro C* é `program`.

Pelas convenções adotadas, `E`, `T` e `F` são não terminais. O símbolo `E` é o símbolo inicial. Por outro lado, o **id**, os operadores aritméticos e os parênteses são todos *terminais*.

Com as notações e convenções estabelecidas, torna-se possível definir formalmente a estrutura sintática da linguagem. O Apêndice B possui a Especificação da Gramática Livre de Contexto (GLC) completa do *Micro C*. Esta especificação fundamenta a implementação do analisador sintático, detalhando como os *tokens* devem ser combinados para formar estruturas válidas, abrangendo desde a declaração da função principal (`main`) até as regras de precedência em expressões aritméticas e lógicas.

3.4.2 Relação entre Gramática e Analisador Sintático

Após a análise léxica, que transforma o código fonte em uma sequência de *tokens*, o compilador precisa verificar se essa sequência está organizada de forma sintaticamente correta. É nesse momento que o analisador sintático, ou *parser*, entra em ação.

O analisador sintático utiliza a gramática da linguagem como um guia para reconhecer se a sequência de *tokens* pode ser derivada a partir do símbolo inicial da gramática. Em outras palavras, o *parser* verifica se os *tokens* seguem as regras formais definidas pela gramática, confirmando que o programa obedece à estrutura sintática esperada. Caso a sequência não se encaixe nas regras, o analisador sintático sinaliza um erro indicando que o código está mal estruturado.

Além de validar a sequência de *tokens*, o analisador sintático constrói uma representação intermediária chamada *árvore sintática* ou *árvore de derivação*. Essa árvore mostra a estrutura hierárquica das construções do programa, evidenciando como as regras da gramática foram aplicadas para gerar a sequência de *tokens*. Cada nó da árvore representa um símbolo da gramática, e as ramificações indicam como as produções foram usadas para decompor a entrada.

A gramática, portanto, influencia diretamente a forma da árvore sintática. Dependendo das regras e da organização das produções, a árvore pode ter diferentes formatos, o que afeta etapas posteriores do compilador, como análise semântica e geração de código. Assim, é fundamental que a gramática seja projetada de forma cuidadosa para garantir que o analisador sintático funcione corretamente e produza estruturas que facilitem as fases seguintes da compilação.

Considere a expressão aritmética simples:

$$3 + (5 + 2)$$

Supondo a gramática para expressões aritméticas vista anteriormente:

$$\begin{aligned} E &\to E + T \mid E - T \mid T \\ T &\to T * F \mid T/F \mid F \\ F &\to (E) \mid \textbf{id} \mid \textbf{num} \end{aligned}$$

Neste exemplo, **num** representa um número, que é um terminal da gramática. A análise sintática verifica se a sequência de *tokens* correspondente à expressão acima pode ser derivada a partir do símbolo inicial E.

Na Figura 3.2 (árvore sintática para a expressão `3 + (5 + 2)`), cada nó representa um símbolo não terminal ou terminal da gramática, mostrando como a expressão foi decomposta seguindo as regras sintáticas. O analisador sintático constrói essa estrutura para garantir que a entrada está correta e facilitar as etapas seguintes, como a geração de código ou a análise semântica.

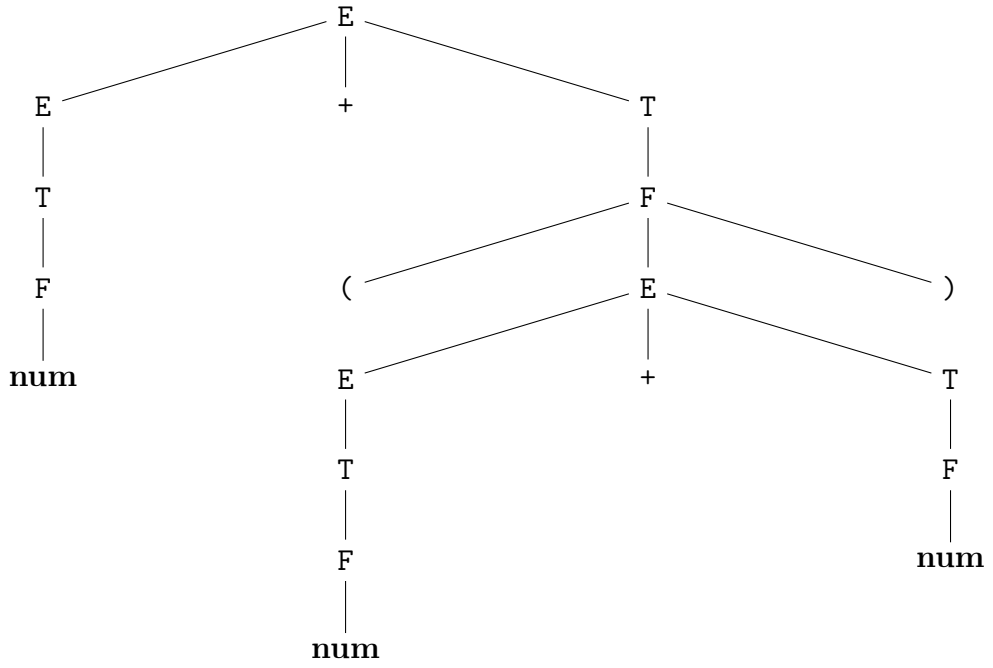


Figura 3.2: Árvore Sintática para a expressão `3 + (5 + 2)`.

Assim, é possível notar que a gramática orienta e guia o analisador sintático e a construção da árvore sintática. Isso estabelece uma ligação direta entre as regras formais da linguagem e a representação estrutural do programa.

3.5 Análise Sintática Descendente

A análise sintática descendente é uma técnica que constrói a árvore de derivação do código-fonte de forma *top-down*, ou seja, da raiz até as folhas. Nesse tipo de abordagem, a estrutura da árvore é criada em ordem pré-fixada (ou pré-ordem [5]), seguindo o mesmo raciocínio de uma busca em profundidade. O objetivo é derivar a cadeia de entrada de acordo com as regras da gramática, partindo do símbolo inicial e aplicando sucessivamente produções até gerar uma sequência de símbolos terminais que coincida com a sequência de *tokens* gerada pela análise léxica.

Essa estratégia corresponde a uma derivação mais à esquerda, pois a cada etapa o analisador foca na substituição da subárvore mais à esquerda que ainda contém um não terminal. A escolha da produção correta a ser aplicada em cada passo é o principal desafio desse método, especialmente em gramáticas que oferecem múltiplas alternativas para um mesmo símbolo não terminal.

3.5.1 Descida Recursiva

Um dos métodos mais utilizados na implementação da análise descendente é a **descida recursiva**. Ele é baseado em um conjunto de procedimentos recursivos, um

para cada não terminal da gramática. A execução tem início no procedimento correspondente ao símbolo inicial e avança por chamadas recursivas até que todos os *tokens* sejam reconhecidos corretamente.

O funcionamento básico pode ser esquematizado da seguinte forma: para cada não terminal da gramática, existe uma função associada que tenta reconhecer a produção correta. Essa função pode invocar outras funções (recursivamente) para reconhecer os símbolos não terminais do lado direito da produção. Finalmente, para cada símbolo terminal, verifica-se se ele corresponde com o símbolo atual da entrada.

No entanto, esse método, em sua forma geral, pode exigir *retrocesso*, isto é, pode ser necessário tentar diversas alternativas para um mesmo não terminal até encontrar aquela que corresponde corretamente à entrada. Essa tentativa e erro é pouco eficiente e raramente usada em compiladores reais [1, 3, 9], especialmente porque muitas construções em linguagens de programação não exigem essa complexidade.

Exemplo: Considere a seguinte gramática:

$$S \to xBz$$
$$B \to y \mid w$$

Suponha a cadeia de entrada `x w z` que será analisada utilizando a análise sintática descendente. Inicialmente, a descida começa do símbolo inicial S e aplica-se a produção $S \to xBz$. Em seguida, tenta-se expandir o não terminal B para que ele reconheça o símbolo intermediário da entrada. Como o próximo símbolo é `w`, a produção $B \to w$ é escolhida. A derivação é bem-sucedida, pois todos os símbolos da entrada são consumidos corretamente, resultando na árvore de derivação da Figura 3.3.

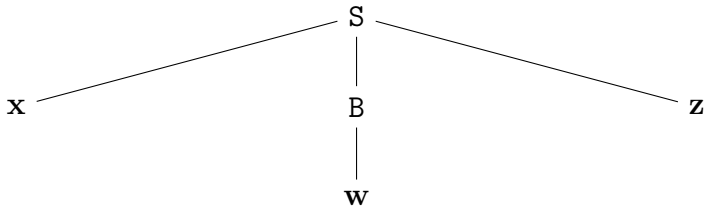


Figura 3.3: Árvore de Derivação da gramática.

Neste ponto, é experimentado a primeira alternativa para B, que é $B \to y$. Como o símbolo `y` não corresponde ao próximo símbolo da entrada, que é `w`, ocorre um erro. O analisador então realiza o retrocesso: retorna ao ponto em que B foi invocado e tenta a próxima alternativa. Utilizando $B \to w$, ocorre uma validação bem-sucedida com a entrada, permitindo o avanço para o símbolo final `z`, completando a derivação com sucesso.

3.5.2 Problemas com Recursão à Esquerda

Um dos desafios clássicos ao implementar analisadores descendentes recursivos é a presença de recursão à esquerda [1]. Quando uma função recursiva chama a si mesma de forma direta sem computar nenhum símbolo da entrada, o *parser* pode entrar em um ciclo infinito, impedindo que a análise avance.

Com o objetivo de reforçar a compreensão mas com enfoque na mecânica da descida recursiva, considere o seguinte analisador descendente recursivo para comandos de repetição simples a seguir:

- Função `repetir_comando()` tenta processar qualquer comando que comece com a palavra reservada `repeat`.

Se implementada de forma recursiva e direta, sem consumir a palavra `repeat` antes da chamada recursiva, a função poderia se auto invocar repetidamente sem avançar na entrada. Por exemplo, a sequência de chamadas poderia ocorrer da seguinte forma:

$$\texttt{repetir\_comando()} \Rightarrow \texttt{repetir\_comando()} \Rightarrow \texttt{repetir\_comando()} \Rightarrow \ldots$$

O *parser* nunca chega a processar a palavra `repeat`, resultando em um laço infinito, mesmo que a entrada contenha comandos válidos.

Para evitar isso, cada chamada recursiva precisa processar, pelo menos, um *token* da entrada antes de se invocar recursivamente. Alternativamente, é possível reestruturar a função para tratar a repetição de maneira iterativa ou dividir a produção em partes que garantam avanço da análise. Assim, o *parser* consegue processar corretamente os comandos e avançar pela entrada sem problemas adicionais.

O exemplo apresentado reforça que a recursão à esquerda precisa ser cuidadosamente tratada em analisadores descendentes recursivos. Ao compreender e aplicar esta técnica, é possível garantir que o *parser* funcione, reconhecendo sequências de *tokens* corretamente e evitando ciclos infinitos.

3.5.3 Implementação Sem Recursão

Além da versão recursiva, também é possível implementar um analisador descendente com uma **pilha explícita**, simulando o comportamento das chamadas recursivas. Essa técnica é particularmente útil em situações em que o uso de recursão na linguagem de implementação não é desejado.

A pilha mantém os símbolos que ainda precisam ser analisados. A cada passo, o analisador consulta o topo da pilha e o símbolo corrente da entrada. Se o topo da pilha for um terminal que casa com a entrada, ambos são processados. Se for um não terminal, usa-se a tabela preditiva para decidir qual produção aplicar, substituindo o não terminal por seus componentes. Por fim, se não houver regra aplicável, ocorre um erro de sintaxe.

3.6 Conjuntos FIRST e FOLLOW

Na construção de analisadores sintáticos descendentes especialmente os preditivos e também na análise sintática ascendente, os conjuntos **FIRST** e **FOLLOW** desempenham um papel fundamental. Essas duas funções, associadas aos símbolos de uma gramática livre de contexto, ajudam a decidir qual produção deve ser utilizada em cada etapa da análise.

Durante a análise descendente, por exemplo, o conjunto FIRST permite prever qual produção aplicar com base no próximo símbolo da entrada. Já o conjunto FOLLOW é especialmente útil em momentos de recuperação de erro e no tratamento de produções que podem derivar a cadeia vazia (ε).

O conjunto **FIRST** de uma cadeia α, denotado por $\texttt{FIRST}(\alpha)$, é o conjunto de símbolos terminais que podem iniciar alguma cadeia derivada a partir de α. Em outras palavras, é o conjunto de possíveis primeiros símbolos da entrada quando aplicadas as produções da gramática a partir de α. Se $\alpha \Rightarrow^* \varepsilon$, então $\varepsilon \in \texttt{FIRST}(\alpha)$.

Por outro lado, o conjunto **FOLLOW** de um não terminal A, denotado por $\texttt{FOLLOW}(A)$, é o conjunto de símbolos terminais que podem aparecer imediatamente à direita de A em alguma derivação da gramática. Ou seja, se houver uma derivação do tipo $S \Rightarrow^* \alpha A a \beta$, então $a \in \texttt{FOLLOW}(A)$. Além disso, o marcador de fim de entrada, representado por \$, também pertence a $\texttt{FOLLOW}(\text{S})$, onde S é o símbolo inicial da gramática.

Para computar o conjunto `FIRST` para todos os símbolos da gramática, são aplicadas as regras a seguir repetidamente, até que o conjunto se torne estável, ou seja, não ocorram mais mudanças:

1. Se X é um terminal, então $\mathsf{FIRST}(X) = \{X\}$.
2. Se X é um não terminal e há uma produção $X \to Y_1 Y_2 \ldots Y_k$, então:
   - Inclua todos os símbolos não-ε de $\mathsf{FIRST}(Y_1)$ em $\mathsf{FIRST}(X)$;
   - Se $\varepsilon \in \mathsf{FIRST}(Y_1)$, então também inclua os símbolos não-ε de $\mathsf{FIRST}(Y_2)$, e assim por diante;
   - Se $\varepsilon \in \mathsf{FIRST}(Y_i)$ para todo $i = 1, \ldots, k$, então $\varepsilon \in \mathsf{FIRST}(X)$.
3. Se $X \to \varepsilon$ é uma produção, então $\varepsilon \in \mathsf{FIRST}(X)$.

Por outro lado, para calcular o conjunto `FOLLOW` para todos os não terminais, são utilizadas as seguintes regras:

1. Inclua \$ em $\texttt{FOLLOW}(\text{S})$, onde S é o símbolo inicial da gramática.
2. Se houver uma produção $A \to \alpha B \beta$, então todos os símbolos de $\texttt{FIRST}(\beta)$ exceto ε devem estar em $\texttt{FOLLOW}(\text{B})$.

3. Se $\varepsilon \in$ `FIRST`(β) ou $\beta = \varepsilon$, então tudo que estiver em `FOLLOW`(A) deve estar também em `FOLLOW`(B).

E os procedimentos seguem o mesmo padrão do conjunto `FIRST`, até que o conjunto estabilize.

3.6.1 Exemplo de Cálculo dos Conjuntos

Nesta seção é apresentado um exemplo prático de cálculo dos conjuntos `FIRST` e `FOLLOW`, utilizando uma gramática para aritmética simples ajustada para não conter recursões à esquerda. O objetivo é compreender como esses conjuntos são construídos e como eles auxiliam o *parser* a tomar decisões durante a análise sintática.

Considere a seguinte gramática sem recursão à esquerda:

$$
\begin{aligned}
E &\to T\,E' \\
E' &\to +\,T\,E' \mid \varepsilon \\
T &\to F\,T' \\
T' &\to *\,F\,T' \mid \varepsilon \\
F &\to (\,E\,) \mid \text{id}
\end{aligned}
$$

As reapresentações a seguir demonstram a execução dos passos utilizados para calcular e construir os conjuntos `FIRST` e `FOLLOW`:

`FIRST`(F) = { (, id }

`FIRST`(T) = `FIRST`(F) = { (, id }

`FIRST`(E) = `FIRST`(T) = { (, id }

`FIRST`(E') = { +, ε }

`FIRST`(T') = { *, ε }

`FOLLOW`(E) = {), $ }

`FOLLOW`(E') = `FOLLOW`(E) = {), $ }

`FOLLOW`(T) = { +,), $ }

`FOLLOW`(T') = `FOLLOW`(T) = { +,), $ }

`FOLLOW`(F) = { *, +,), $ }

Este exemplo mostra, de forma concreta, como os conjuntos `FIRST` e `FOLLOW` são determinados para cada não terminal. Ao calcular esses conjuntos, é possível garantir que o *parser* do *Micro C* consiga prever corretamente quais produções devem ser aplicadas em cada situação, facilitando a construção da árvore sintática e evitando erros durante a compilação.

3.6.2 Gramáticas LL(1)

Os conjuntos `FIRST` e `FOLLOW` são as ferramentas formais que permitem responder a uma pergunta importante sobre a gramática. Ou seja, é necessário determinar se uma gramática é simples o suficiente para ser analisada por um **parser preditivo**. E esses conjuntos auxiliam nesse processo.

Um *parser* preditivo, como o nome sugere, tenta *prever* qual regra gramatical aplicar olhando apenas um *token* à frente na entrada. Esta é a abordagem mais rápida e eficiente. A alternativa seria um *parser* com **retrocesso** (ou *backtracking*): um método onde o *parser adivinha* uma regra. Se essa regra falhar (ex: 50 *tokens* depois), ele deve voltar atrás (retroceder) e tentar outra regra. O *backtracking* é extremamente lento e complexo [1, 13] de implementar e por isso não foi utilizado no compilador do *Micro C*.

Para garantir que o *parser* possa ser preditivo e não precise de *backtracking*, a gramática deve atender a uma propriedade especial. Essa propriedade é conhecida como LL (*Left-to-right*, *Leftmost*), onde o primeiro "L"indica que a entrada é lida da esquerda para a direita, e o segundo "L"indica que a análise constrói uma derivação mais à esquerda. Usando uma **gramática LL(1)**, é possível construir um analisador sintático preditivo que reconhece a cadeia de entrada (da esquerda para a direita), produzindo uma derivação mais à esquerda, e que decide qual produção usar examinando apenas um símbolo à frente da entrada, o "1"da notação `LL(1)` indica exatamente essa quantidade de *lookahead*.

Para que uma gramática G seja LL(1), ela deve satisfazer as seguintes condições, para toda par de produções $A \to \alpha$ e $A \to \beta$:

1. $\texttt{FIRST}(\alpha) \cap \texttt{FIRST}(\beta) = \emptyset$
2. No máximo uma entre α ou β pode derivar ε
3. Se $\varepsilon \in \texttt{FIRST}(\beta)$, então $\texttt{FIRST}(\ \alpha) \cap \texttt{FOLLOW}(\text{A}) = \emptyset$

Essas condições garantem que a produção correta pode ser escolhida apenas com base no próximo símbolo da entrada.

3.6.3 Gramáticas que não são LL(1)

A definição de LL(1) é estrita, e nem toda gramática pode ser transformada para satisfazer suas condições. Uma gramática falha em ser LL(1) se for intrinsecamente ambígua ou se requerer mais de um símbolo de antecipação (*lookahead*) para decidir qual produção aplicar. Quando uma gramática viola as condições (por exemplo, se $\texttt{FIRST}(\alpha) \cap \texttt{FIRST}(\beta) \neq \emptyset$), a tabela de análise preditiva resultante conteria múltiplas produções, indicando que o *parser* não pode tomar uma decisão determinística.

Exemplo: Considere a gramática abaixo, que simula a ambiguidade do uso de `else`:

$$S \to \texttt{if}\ \ E\ \ \texttt{then}\ \ S\,S' \mid a$$
$$S' \to \texttt{else}\ \ S \mid \varepsilon$$
$$E \to b$$

Essa gramática não é LL(1), pois a entrada da tabela para S' com símbolo `else` contém duas produções possíveis, o que causa ambiguidade. A resolução mais comum para esse tipo de problema é adotar a convenção de associar o `else` ao `then` mais próximo, que corresponde a escolher a produção $S' \to \texttt{else}\ \ S$.

3.6.4 Análise Preditiva

Uma variação eficiente da descida recursiva é o **analisador preditivo**, que evita o uso de retrocesso. Nessa abordagem, a escolha da produção correta para um não terminal é feita de forma determinística, com base no próximo símbolo da entrada (ou nos próximos k símbolos, no caso geral de uma gramática $LL(k)$. Na prática, costuma-se usar $k = 1$, dando origem às gramáticas $LL(1)$.

A partir desses conjuntos, é possível construir uma **tabela preditiva** que relaciona não terminais e símbolos de entrada às produções apropriadas. Com essa tabela, o analisador preditivo pode decidir, com base no próximo símbolo, qual produção utilizar sem necessidade de retrocesso.

Exemplo: Suponha que temos a seguinte gramática transformada:

$$E \to TE'$$
$$E' \to +TE' \mid \epsilon$$
$$T \to \text{id}$$

Com base nos conjuntos FIRST e FOLLOW, a tabela preditiva pode ser construída para guiar o analisador na escolha correta das produções para E' com base no símbolo seguinte na entrada. Se o próximo símbolo for `+`, é escolhida a derivação $E' \to +TE'$. Se for um símbolo do conjunto FOLLOW de E', é aplicada a derivação $E' \to \epsilon$.

3.6.5 Recuperação de Erros

Assim como discutido anteriormente, analisadores descendentes também podem adotar estratégias de recuperação de erros. Mesmo que a análise falhe em um determinado ponto, é desejável que o compilador continue examinando o restante do código, reportando múltiplos erros em uma única execução. Para isso, é comum utilizar o modo de **pânico**, onde símbolos da entrada são descartados até que se encontre um símbolo seguro que pertença ao conjunto FOLLOW de algum não terminal relevante.

Essa abordagem simples permite que o compilador avance a análise e retome a derivação a partir de pontos bem definidos, como o final de uma instrução ou um delimitador importante.

3.7 A Árvore Sintática Abstrata (ASA)

Enquanto a árvore de derivação é uma representação literal de como a gramática gera uma sequência de *tokens*, na prática, os compiladores utilizam uma estrutura mais otimizada e informativa: a *Árvore Sintática Abstrata* (ASA), ou *AST* (do inglês, *Abstract Syntax Tree*). A ASA é o principal resultado da análise sintática e serve como a estrutura de dados central para todas as fases subsequentes.

Diferente da árvore de derivação, a ASA condensa a estrutura do programa, removendo nós intermediários e *tokens* que não carregam significado essencial, como parênteses para agrupamento ou pontos e vírgulas como terminadores de instrução. Ela captura a estrutura hierárquica e lógica do código de uma forma muito mais direta. Por exemplo, uma expressão como `a + b * c` seria representada em uma árvore que reflete diretamente a precedência dos operadores, com a multiplicação com um nó folha mais profundo que o da adição.

3.7.1 Estrutura de um Nó da ASA no *Micro C*

Para construir a ASA, primeiro foi definida a estrutura de um nó individual no arquivo `ast.h`. Cada nó na árvore representa um conceito gramatical, como uma declaração, uma expressão ou um laço, e precisa de uma *etiqueta* para que o compilador saiba o que ele representa. Esta etiqueta é a informação central que as fases subsequentes (como a análise semântica e a geração de código) usarão para decidir qual ação tomar.

Para implementar este sistema de etiquetas, foi utilizado um enumerador (`enum`) chamado `NodeType`. Este `enum` é, essencialmente, o *vocabulário* da ASA. Ele define formalmente todo tipo de construção que o analisador sintático é capaz de reconhecer e armazenar. Cada membro deste `enum` (como `NODE_PROGRAM` ou `NODE_BINARY_OP`) atua como um tipo de nó, permitindo que a estrutura `ASTNode` (que será visto a seguir) saiba como interpretar os dados que armazena em sua `union`, definida dentro da estrutura.

O trecho de código a seguir, extraído do `ast.h`, detalha alguns tipos de nós que o compilador *Micro C* utiliza:

```
//arquivo: src/ast/ast.h (trecho)
typedef enum {
    NODE_UNDEFINED,
    NODE_PROGRAM,           // Nó raiz da AST
    NODE_FUNCTION_DEF,      // Definição de uma função
    NODE_VAR_DECL,          // Declaração de uma variável
    NODE_ASSIGN,            // Operação de atribuição
    NODE_IF,                // Estrutura condicional 'if-else'
    NODE_FOR,               // Laço 'for'
    NODE_RETURN,            // Comando 'return'
    NODE_CALL,              // Chamada de função
    NODE_BINARY_OP,         // Operação binária (ex: +, <, ==)
    NODE_ID,                // Um identificador
    NODE_INTEGER_CONST,     // Uma constante inteira
    //... outros tipos de nós
} NodeType;
```

Com os tipos de nós definidos, a estrutura principal, `ASTNode`, é criada. Ela contém o tipo do nó, a linha do código para facilitar mensagens de erro, ponteiros para conectar os nós e formar a árvore e uma estrutura do tipo `union` para armazenar dados específicos de forma eficiente.

A forma como os nós são conectados é um detalhe de implementação importante. Em vez de cada nó ter um número fixo de filhos (o que seria inflexível), foi utilizada a representação *primeiro filho, próximo irmão*. Ou seja, o `filho` tem um ponteiro que aponta para o **primeiro** filho do nó atual e o `proximo_irmao` um ponteiro que aponta para o próximo filho do **mesmo nó pai**. Os filhos de um nó formam uma lista encadeada por meio deste ponteiro.

Essa técnica é poderosa, pois permite que um nó tenha um número variável de filhos. Um nó de atribuição (`NODE_ASSIGN`) sempre terá dois filhos (o lado esquerdo e o direito), mas um nó de bloco (`NODE_BLOCK`) pode ter dezenas de filhos, um para cada instrução.

O trecho de código abaixo mostra a estrutura do `ASTNode`:

```
//arquivo: src/ast/ast.h (trecho)
typedef struct ASTNode {
    NodeType node_type;
    int linha;
    struct ASTNode *filho;
    struct ASTNode *proximo_irmao;

    union {
        long int_value;
        char char_value;
        char* string_value;
        TokenType op_type;
    } data;
} ASTNode;
```

3.7.2 Construindo e Manipulando a Árvore

Uma vez que o objetivo era que o código do *parser* fosse simples para facilitar a compreensão de sua implementação, a alocação de memória e conexão entre os nós foi encapsulada em funções auxiliares (em `ast.c`). A função `criar_no()`, por exemplo, foi usada para alocação e inicialização de um novo `ASTNode`:

```
//arquivo: src/ast/ast.c (trecho)
ASTNode* criar_no(NodeType type, int linha) {
    ASTNode* no = (ASTNode*) malloc(sizeof(ASTNode));
    if (no == NULL) { /*...*/ exit(EXIT_FAILURE); }

    no->node_type = type;
    no->linha = linha;
    no->filho = NULL;
    no->proximo_irmao = NULL;
    memset(&no->data, 0, sizeof(no->data));

    return no;
}
```

A função mais importante para a construção da árvore é a `adicionar_filho()`. Ela implementa a lógica da representação *primeiro filho, próximo irmão*, garantindo que novos filhos sejam adicionados ao final da lista de irmãos, preservando a ordem das instruções do código.

O trecho de código que implementa a função `adicionar_filho` está abaixo:

```
//arquivo: src/ast/ast.c (trecho)
void adicionar_filho(ASTNode* pai, ASTNode* filho) {
    if (pai == NULL || filho == NULL) return;

    if (pai->filho == NULL) {
        //Se for o primeiro filho
        pai->filho = filho;
    } else {
        //Se já existem filhos, percorre a lista de irmãos até o final
        ASTNode* irmao_atual = pai->filho;
        while (irmao_atual->proximo_irmao != NULL) {
            irmao_atual = irmao_atual->proximo_irmao;
        }
        //Adiciona o novo filho no final da lista
        irmao_atual->proximo_irmao = filho;
    }
}
```

Com essas estruturas e funções, o analisador sintático pode construir uma representação fiel do programa. Por exemplo, para a instrução `v[i] = 47;`, o *parser* executaria uma sequência de chamadas como:

1. `criar_no(NODE_ASSIGN, ...)` para criar o nó de atribuição.
2. `criar_no(NODE_ARRAY_ACCESS, ...)` para o lado esquerdo, `v[i]`.
3. `criar_no(NODE_INTEGER_CONST, ...)` para o lado direito, `47`.
4. `adicionar_filho(no_assign, no_array_access)`.
5. `adicionar_filho(no_assign, no_integer_const)`.

O resultado é uma sub-árvore que captura a operação e a deixa pronta para ser validada pelo analisador semântico.

3.8 Analisador Sintático

O analisador sintático, ou *parser*, é a etapa responsável por verificar se a sequência de *tokens* produzida pelo analisador léxico está estruturada conforme as regras da gramática da linguagem. Em outras palavras, ele garante que o código fonte siga a sintaxe correta da linguagem.

No compilador do *Micro C*, foi implementado um analisador sintático do tipo descendente recursivo, onde cada regra gramatical é representada por uma função em `C` que processa os *tokens* na ordem correta. Essa abordagem permite um controle detalhado do processo de análise e facilita a detecção de erros sintáticos e sua localização no código.

3.8.1 Processamento de *tokens* esperados

Uma operação fundamental do *analisador sintático* é a verificação e o processamento do *token* atual para garantir que ele corresponda ao símbolo esperado conforme as regras da gramática. Para isso, foi utilizada a função `match`, que recebe dois argumentos: um ponteiro para o `parser` e o tipo do *token* que se espera encontrar na entrada.

Se o tipo do *token* atual coincidir com o esperado, o analisador avança para o próximo *token*, processando-o com a função `advance`. Caso contrário, a função `error` é acionada para reportar uma mensagem de erro sintático e encerrar o processo de compilação.

Esse procedimento garante que o código-fonte está sintaticamente correto e aderente à estrutura da linguagem, facilitando a detecção precoce de desvios e inconsistências durante a análise.

A implementação da função `match` é dividida em duas partes. A primeira parte, mostrada abaixo, trata se o *token* atual é o esperado, ele é processado:

```
void match(Parser *parser, TokenType esperado) {
    if (parser->current_token && parser->current_token->tipo == esperado) {
        advance(parser);
    } else {
```

A segunda parte da função é o bloco `else`, que trata o erro sintático quando o *token* é inesperado. A função constrói uma mensagem de erro clara e interrompe a compilação:

```
//... (continuação da função match)
    } else {
        char msg[100];
        snprintf(msg, sizeof(msg), "Token inesperado. Esperado: %s",
                 token_name(esperado));
        error(parser, msg);
    }
}
```

A função `match` encapsula uma operação de verificação simples, porém crítica. Ela compara o tipo do *token* atual armazenado em `parser->current_token` com o tipo especificado como argumento. Havendo correspondência, o analisador chama a função `advance`, que realiza a leitura do próximo *token* da lista encadeada de entrada.

Quando há discrepância entre o tipo atual e o esperado, a função monta uma mensagem de erro que informa o que era esperado, utilizando a função `token_name` para converter o tipo do *token* em uma representação textual amigável. Essa mensagem é repassada à função `error`, que imprime o erro com o número da linha e interrompe a compilação.

Esse mecanismo é aplicado sistematicamente em todas as funções de análise, servindo como um verificador local que reforça a precisão e previsibilidade do `parser`. Como resultado, erros de sintaxe são detectados no ponto exato de ocorrência, com mensagens explicativas que auxiliam na correção rápida do código-fonte.

Além disso, a centralização da verificação na função `match` é uma prática de projeto importante. Ela promove a clareza, evita duplicação de código nas funções de análise e assegura uma resposta uniforme para erros sintáticos, o que torna o analisador mais robusto e de fácil manutenção.

3.8.2 Avanço para o próximo *token*

A cada vez que um *token* esperado é encontrado e corretamente processado, o *analisador sintático* precisa avançar para o próximo elemento da sequência léxica. Essa transição é realizada pela função `advance`, que atualiza o ponteiro `current_token` do `parser` para apontar para o próximo nó da lista de *tokens* gerada pelo analisador léxico.

A implementação é simples, mas desempenha um papel essencial na análise sintática, pois garante que o `parser` esteja sempre posicionado corretamente na sequência de entrada. A seguir o código da função `advance`:

```
void advance(Parser *parser) {
    if (parser->current_token != NULL) {
        parser->current_token = parser->current_token->proximo;
    }
}
```

O funcionamento da função `advance` é simples: ele apenas atualiza o campo `current_token` para apontar para o próximo elemento da lista de *tokens*. Isso supõe que os *tokens* foram previamente organizados em uma lista encadeada simples, na qual cada *token* aponta para o próximo por meio do campo `proximo`.

Esse mecanismo sequencial reflete o fluxo natural da leitura de um programa fonte, permitindo ao analisador percorrer *token* por *token* enquanto valida a estrutura sintática com base nas regras da gramática.

É importante destacar que, ao projetar a estrutura de *tokens*, deve-se garantir que o último elemento da lista aponte para `NULL`, marcando assim o final da entrada. Isso permite à função `advance` lidar corretamente com o fim do fluxo de entrada, evitando acessos inválidos à memória.

Por outro lado, a função `advance` implementa o avanço linear da entrada e atua em conjunto com a função `match` para controlar o fluxo de reconhecimento de *tokens*. Juntas, essas duas funções formam a base do controle sintático sobre a leitura da entrada.

3.9 Sumário

Neste capítulo, avançamos para a segunda fase da compilação: a **análise sintática**. Foi explicado como o *parser* recebe uma sequência linear de *tokens* da análise léxica e valida se essa sequência obedece à estrutura gramatical da linguagem do *Micro C*.

Inicialmente foi discutida a teoria sobre **Gramáticas Livres de Contexto (GLC)**, que são a especificação formal para linguagens de programação. Aém disso, foi detalhado como a hierarquia clássica de expressões (Expressão, Termo, Fator) é usada para garantir a precedência de operadores. Em seguida, foram identificados os problemas que a **recursão à esquerda** (ex: $E \rightarrow E + T$) pode causar e o porquê isso impede o uso de *parsers* descendentes. Em seguida, apresentou-se como o compilador do *Micro C* implementa o seu **parser descendente recursivo**.

O próximo passo foi discutir a importância da **Árvore Sintática Abstrata (ASA)** como o principal produto desta fase de compilação, em contraste com a árvore de derivação teórica. E foi comparada a implementação da ASA do *Micro C* com a representação *primeiro filho, próximo irmão* (`filho` e `proximo_irmao`), que permite que nós como `NODE_BLOCK` tenham um número variável de filhos.

Por fim, foram revisadas as funções auxiliares que encapsulam a lógica de construção da árvore, mantendo o código do parser limpo e concentrado na gramática. Dessa forma, com a lista linear de *tokens* e a estrutura de árvore hierárquica (a ASA), o código está pronto para a próxima etapa que será explorada no próximo capítulo: a análise de significado (semântica).

Capítulo 4

Análise Semântica

> *Get the habit of analysis - analysis will in time enable synthesis to become your habit of mind.*
>
> – Frank Lloyd Wright

No Capítulo 3, exploramos a análise sintática e como o compilador constrói a Árvore Sintática Abstrata (ASA) a partir das regras da gramática. Essa estrutura é essencial, mas não é suficiente, pois a análise sintática apenas valida a *forma* do código, não o seu *significado*. Um programa pode ser gramaticalmente perfeito e, ainda assim, ser semanticamente inválido, por exemplo, ao tentar somar um número inteiro a um *vetor*.

Neste capítulo, será abordada a seguinte fase: a **análise semântica**. Esta é a etapa responsável por interpretar a ASA e verificar se ela faz sentido dentro do contexto da linguagem *Micro C*. Nesta etapa de compilação serão explorados como o compilador utiliza uma Tabela de Símbolos para rastrear variáveis, checar tipos, validar escopos e garantir que as construções feitas pelo programador sejam lógicas e coerentes.

4.1 Além da Gramática: Sentido e Contexto

A melhor analogia para a diferença entre a análise sintática e a semântica vem da linguística. Por exemplo, a frase: *ideias verdes incolores dormem furiosamente*. É possível observar que do ponto de vista sintático, a frase está correta pois segue a estrutura convencional de `sujeito + verbo + advérbio`. No entanto, a frase não tem significado coerente e, por isso, está semanticamente incorreta.

Da mesma forma, um programa pode seguir todas as regras gramaticais de uma linguagem e ainda assim ser semanticamente inválida.

Considere o seguinte trecho de código em *Micro C*:

```
int a[10];
int b[10];
int c;

c = a * b; // Erro Semântico!
```

Para o analisador sintático, a linha `c = a * b;` é perfeitamente válida. Ele a enxerga como a estrutura `ID = ID * ID;`, que corresponde a uma regra gramatical de atribuição e expressão. O *parser* não sabe (e nem precisa saber) o que `a`, `b` e `c` realmente são.

A análise semântica, por outro lado, adiciona o **contexto**. Ela consulta a Tabela de Símbolos e descobre que `a` e `b` não são números, mas sim vetores. A regra semântica da linguagem determina que o operador de multiplicação (`*`) não é definido para operandos do tipo vetor. Portanto, é o analisador semântico que identifica e reporta este erro.

4.1.1 Verificação Estática vs. Dinâmica

A análise semântica é a principal responsável pela **verificação estática**. Verificações estáticas são aquelas que o compilador pode realizar **antes** da execução do programa, apenas analisando o código fonte. A checagem de tipos, a verificação de declaração de variáveis e a validação de parâmetros de funções são todos exemplos de verificações estáticas. O objetivo é capturar o maior número possível de erros.

No entanto, nem todos os erros podem ser detectados estaticamente. Existem erros que só podem ser identificados durante a execução, pois dependem dos valores que as variáveis assumem. Estes são erros de **verificação dinâmica**.

Exemplos clássicos incluem:

- **Acesso a um vetor fora dos limites:** em uma expressão como `v[i]`, se `i` for uma variável, o compilador não tem como saber em tempo de compilação se o valor de `i` será válido.
- **Divisão por zero:** em `x / y`, o compilador não pode garantir que o valor de `y` nunca será zero durante a execução.
- **Desreferência de ponteiro nulo:** o compilador não pode prever se um ponteiro terá o valor `NULL` em um determinado ponto da execução.

Linguagens como C e, por consequência, do *Micro C*, priorizam a performance e, portanto, realizam um número limitado de verificações dinâmicas. A responsabilidade de evitar esses erros de execução é de responsabilidade do programador. Linguagens como Java ou Python, por outro lado, inserem automaticamente verificações dinâmicas em seu ambiente de execução (como a *Java Virtual Machine* ou o

interpretador Python), o que as torna mais seguras, mas com um custo de performance. Esse custo se deve aos ciclos extras de CPU gastos para realizar checagens de segurança, por exemplo, verificar se um índice está dentro dos limites de um vetor em cada execução, algo que linguagens como C não fazem.

4.1.2 Atributos, Regras Semânticas e a Tabela de Símbolos

Para realizar suas tarefas, o analisador semântico trabalha com o conceito de **atributos**, que são informações associadas aos nós da ASA. O atributo mais importante é o **tipo** de uma expressão. O trabalho do analisador é percorrer a árvore e calcular os atributos para cada nó.

Esse processo é guiado por **regras semânticas**. Cada produção da gramática (representada por um tipo de nó na ASA) tem uma ou mais regras semânticas associadas. Por exemplo, para um nó de operação binária que representa a regra `expr -> expr1 + expr2`, a regra semântica seria:

> *"O tipo de* `expr` *será* `inteiro` *se os tipos de* `expr1` *e* `expr2` *forem ambos* `inteiro`*. Caso contrário, reporte um erro de tipo."*

Para aplicar essa regra, o analisador precisa saber os tipos de `expr1` e `expr2`. Se eles forem identificadores (variáveis), o analisador consulta a Tabela de Símbolos para obter seus tipos declarados. Se forem outras expressões, ele calcula seus tipos recursivamente.

Dessa forma, a ASA fornece a estrutura, a Tabela de Símbolos fornece o contexto inicial, e as regras semânticas guiam o processo de validação e de enriquecimento da Tabela de Símbolos com os atributos calculados. Este processo não se limita à verificação de tipos, mas inclui o registro de informações vitais para o *back-end*, como o cálculo do desvio de memória (*offset*) na pilha de execução para cada variável e parâmetro.

4.2 Objetivos da Análise Semântica

Enquanto a análise sintática se preocupa com a **forma** do código: *esta frase está gramaticalmente correta?*, a análise semântica se preocupa com o **significado**: *esta frase, embora gramaticalmente correta, faz algum sentido?* ou seja, ela atua como uma camada de verificação profunda, utilizando a ASA gerada pelo *parser* como sua principal estrutura de entrada.

Para realizar suas tarefas, o analisador semântico percorre a árvore e, com o auxílio da Tabela de Símbolos, valida a coerência do programa, respondendo a perguntas como *esta variável já foi declarada?* ou *é possível somar um inteiro com um caractere?*

A análise semântica tem como principais objetivos: verificar se todos os identificadores usados foram devidamente declarados; garantir que operadores e operandos

sejam compatíveis em termos de tipo; detectar usos incorretos de variáveis, funções e estruturas; controlar escopos e visibilidade de símbolos em blocos de código; validar chamadas de funções quanto ao número e tipos dos argumentos; e, finalmente, gerar anotações semânticas na árvore de sintaxe para uso na tradução posterior.

Por meio dessas tarefas, a análise semântica atua como uma ponte entre a *forma* (sintaxe) e o *comportamento* (execução), consolidando a informação necessária para transformar código fonte em passos executáveis pelo computador com segurança e precisão.

Considere o exemplo abaixo com um erro semântico:

```
int x;
x = 'P'; // Erro: atribuição de caractere a uma variável inteira
```

Esse trecho é sintaticamente válido: a atribuição `x = 'P';` está de acordo com a gramática da linguagem. No entanto, semanticamente ele está incorreto, pois em C não é permitido atribuir um valor booleano diretamente a uma variável do tipo `int` sem conversão explícita. Esse tipo de erro não pode ser capturado pela análise sintática apenas pela análise semântica.

Como foi possível observar, a análise semântica atua sobre a árvore gerada pela análise sintática, enriquecendo-a com informações contextuais: tipos, escopos, declarações, coerência de uso e outros atributos. Além disso, ela também prepara a árvore para a próxima fase do compilador: a geração de código intermediário que será o tema do Capítulo 5.

4.3 O Papel da Tabela de Símbolos

Um dos elementos centrais da análise semântica é a **Tabela de Símbolos**. Essa estrutura de dados armazena informações relevantes sobre cada identificador do programa: tipo, categoria (variável, função, constante), escopo, entre outros.

Sempre que uma nova declaração é encontrada, um símbolo correspondente é inserido na tabela. Quando um identificador é utilizado, o compilador consulta essa tabela para verificar se ele foi declarado corretamente, se está acessível no escopo atual e se está sendo usado de maneira coerente com seu tipo.

A Tabela de Símbolos é fundamental para a verificação semântica, pois permite rastrear e validar o uso de nomes ao longo do programa, garantindo consistência e evitando ambiguidades.

4.3.1 Exemplos de Erros Semânticos

A seguir, serão apresentados exemplos de erros semânticos comuns, que não seriam detectados por um analisador sintático.

Uso de variável não declarada

```
y = 10; // Erro: 'y' não foi declarada
```

Incompatibilidade de tipos

```
int x;
x = "abc"; // Erro: tentativa de atribuir string a inteiro
```

Chamada de função com número incorreto de argumentos

```
int soma(int a, int b);
soma(10); // Erro: número de argumentos incompatível
```

Acesso a identificadores fora de escopo

```
int funcao() {
  int z = 5;
}
z = 6; // Erro: 'z' não está mais em escopo
```

É possível observar que esses erros envolvem significado e contexto e não apenas forma. Por isso, a análise semântica é a única fase capaz de detectá-los adequadamente, completando o processo de validação do programa.

4.4 Estrutura da Tabela de Símbolos

Como discutido na Seção 4.3, a Tabela de Símbolos é uma estrutura essencial no processo de compilação, especialmente durante a análise semântica. Ela armazena informações sobre os identificadores presentes no código fonte, como variáveis e funções, incluindo seus tipos e propriedades. Por meio dessa estrutura, o compilador assegura a coerência semântica do programa, verificando, por exemplo, se uma variável foi declarada antes de seu uso, se há redeclarações indevidas no mesmo escopo ou se os tipos envolvidos em uma operação são compatíveis.

Além disso, a Tabela de Símbolos precisa tratar corretamente com os escopos de variável, permitindo que blocos de código, como funções ou estruturas de controle, mantenham declarações independentes. Essa característica é essencial para respeitar as regras de visibilidade definidas pela linguagem, onde variáveis locais têm precedência sobre variáveis globais.

O objetivo desta seção é apresentar a implementação da Tabela de Símbolos no compilador do *Micro C*, detalhando sua organização, as operações realizadas sobre ela e sua integração com o analisador sintático. A solução adotada utiliza uma pilha

de tabelas de símbolos para gerenciar escopos aninhados, garantindo uma análise semântica robusta e eficiente.

4.4.1 Organização em Pilha de Escopos

Para gerenciar escopos, a Tabela de Símbolos é organizada como uma pilha de tabelas, representada pela estrutura `PilhaTabelasSimbolos`. Cada tabela na pilha corresponde a um escopo específico, com o escopo mais interno localizado no topo. Ao entrar em um novo bloco (como o corpo de uma função), uma nova tabela é empilhada. Ao sair do bloco, a tabela é desempilhada, descartando os identificadores locais e preservando a hierarquia de escopos.

A estrutura `PilhaTabelasSimbolos` é definida como segue:

```
/*representa a pilha de tabelas de símbolos para controle de escopo. */
typedef struct {
    TabelaSimbolos *tabelas[MAX_SIMBOLOS];
    int topo;
} PilhaTabelasSimbolos;
```

Os campos da estrutura são representados por: `tabelas`, que é um vetor de ponteiros para as estruturas `TabelaSimbolos`, responsável por armazenar os escopos aninhados até o limite de `MAX_SIMBOLOS`; e `topo`, um número inteiro que indica o índice da tabela atualmente ativa (o escopo corrente no topo da pilha).

O uso de uma pilha para gerenciar escopos é uma técnica consagrada em compiladores [1, 4, 13], pois reflete a natureza hierárquica da análise sintática e semântica, garantindo que os identificadores sejam acessados conforme as regras de visibilidade da linguagem.

4.4.2 Estrutura dos Símbolos e Tabelas

Cada Tabela de Símbolos, representada pela estrutura `TabelaSimbolos`, armazena um conjunto de identificadores, cada um descrito pela estrutura `Simbolo`. Essas estruturas são definidas como segue:

```
/** @brief armazena informações sobre um único símbolo. */
typedef struct {
    char nome[100];
    TokenType tipo;
    int is_function;
    TokenType param_tipos[MAX_PARAMETROS];
    int num_parametros;
    int is_array;
    int array_size;
    int memory_offset;
    int is_parameter;
} Simbolo;
```

```
/* representa uma tabela de símbolos para um único escopo. */
typedef struct {
    Simbolo simbolos[MAX_SIMBOLOS];
    int tamanho;
} TabelaSimbolos;
```

Os campos da estrutura `Simbolo` armazenam os dados essenciais para a análise semântica e geração de código: `nome` (o lexema do identificador); `tipo` (categoria de dado, ex: `INT`); flags booleanas como `is_function`, `is_array` e `is_parameter` para categorizar o identificador; dados específicos como `array_size` e assinatura de funções (`param_tipos`, `num_parametros`); e o `memory_offset`, que determina o endereço relativo da variável na pilha de execução.

Já a estrutura `TabelaSimbolos` gerencia o escopo individual através dos campos: `simbolos`, que é um vetor estático responsável por armazenar as definições do escopo; e `tamanho`, um contador inteiro que indica a quantidade de símbolos atualmente registrados na tabela.

Essa organização permite que cada escopo mantenha um conjunto independente de identificadores, com suporte a crescimento dinâmico para acomodar programas de tamanho variável.

4.4.3 Inserção e Busca de Símbolos

As operações principais na Tabela de Símbolos são a inserção e a busca de identificadores, implementadas pelas funções:

`adicionar_simbolo`

`buscar_simbolo_no_escopo_atual`

`buscar_simbolo_em_todos_escopos`

A função `adicionar_simbolo` é o principal método para popular uma tabela. Ela insere um novo identificador no escopo atual (o topo da pilha), gerenciando a alocação de memória do vetor de símbolos dinamicamente, como demonstra sua implementação:

```
Simbolo* adicionar_simbolo(PilhaTabelasSimbolos *pilha,
const char *nome, TokenType tipo, int is_function, int is_array,
```

```
/*...continuação...*/
int array_size, int is_parameter) {
    if (pilha->topo < 0) {
        printf("Erro: Nenhuma tabela de símbolos disponível\n");
        return NULL;
    }
    TabelaSimbolos *tabela = pilha->tabelas[pilha->topo];
    if (tabela->tamanho >= MAX_SIMBOLOS) {
        printf("Erro: Tabela de símbolos cheia\n");
        return NULL;
    }
    Simbolo *s = &tabela->simbolos[tabela->tamanho];
    strncpy(s->nome, nome, 100);
    s->tipo = tipo;
    s->is_function = is_function;
    s->num_parametros = 0;
    //salva as novas informações do array
    s->is_array = is_array;
    s->array_size = array_size;

    s->is_parameter = is_parameter;
    s->memory_offset = 0; //será calculado pelo semantic.c

    tabela->tamanho++;

    return s;
}
```

Para evitar a redeclaração de um símbolo no mesmo escopo, o analisador semântico utiliza a função `buscar_simbolo_no_escopo_atual`. Conforme detalhado no código abaixo, esta função realiza uma busca linear simples apenas na tabela que está no topo da pilha:

```
Simbolo* buscar_simbolo_no_escopo_atual(PilhaTabelasSimbolos
*pilha, const char *nome) {
    if (pilha->topo < 0) {
        return NULL;
    }
    TabelaSimbolos *tabela = pilha->tabelas[pilha->topo];
    for (int i = 0; i < tabela->tamanho; i++) {
        if (strcmp(tabela->simbolos[i].nome, nome) == 0) {
            return &tabela->simbolos[i];
        }
    }
    return NULL;
}
```

De forma complementar, para validar o uso de um identificador (garantindo que ele foi declarado), o analisador utiliza a função `buscar_simbolo_em_todos_escopos`. Esta implementação demonstra a lógica de busca em escopos aninhados, iterando do topo da pilha (`pilha->topo`) até a base (o escopo global):

```
Simbolo* buscar_simbolo_em_todos_escopos(PilhaTabelasSimbolos
*pilha, const char *nome) {
    //itera da tabela do topo (escopo local) para a base (escopo global)
    for (int i = pilha->topo; i >= 0; i--) {
        TabelaSimbolos *tabela = pilha->tabelas[i];
        for (int j = 0; j < tabela->tamanho; j++) {
            if (strcmp(tabela->simbolos[j].nome, nome) == 0) {
                return &tabela->simbolos[j];
            }
        }
    }
    return NULL;
}
```

4.4.4 Gerenciamento de Escopos

O gerenciamento de escopos é realizado pelas funções:

`empilhar_tabela`

`desempilhar_tabela`

Essas funções implementam a semântica de pilha para o controle de escopos. A função `empilhar_tabela` é invocada sempre que o analisador entra em um novo bloco (como o corpo de uma função), alocando dinamicamente uma nova estrutura `TabelaSimbolos` na memória, desde que o limite máximo da pilha fixa não tenha sido atingido.

Abaixo a implementação da função:

```
void empilhar_tabela(PilhaTabelasSimbolos *pilha) {
    if (pilha->topo + 1 >= MAX_SIMBOLOS) {
        printf("Erro: Pilha de tabelas cheia\n");
        exit(EXIT_FAILURE);
    }
    TabelaSimbolos *nova_tabela = malloc(sizeof(TabelaSimbolos));
    if (nova_tabela == NULL) {
        printf("Erro: Falha ao alocar memória para nova tabela\n");
        exit(EXIT_FAILURE);
    }
    nova_tabela->tamanho = 0;
    pilha->tabelas[++pilha->topo] = nova_tabela;
}
```

De forma simétrica, a função `desempilhar_tabela` é chamada ao sair do bloco. Nesta implementação didática, a função apenas decrementa o índice do topo, fechando logicamente o escopo e impedindo o acesso a variáveis locais fora de seu contexto. A memória alocada é preservada temporariamente para permitir a geração de relatórios de depuração ao final da compilação. Abaixo o código relativo à função:

```
void desempilhar_tabela(PilhaTabelasSimbolos *pilha) {
    if (pilha->topo < 0) {
        printf("Erro: Pilha de tabelas vazia\n");
        exit(EXIT_FAILURE);
    }

    pilha->topo--;
}
```

Vale ressaltar que, embora a memória não seja liberada no momento do desempilhamento, o ciclo de vida dos dados é gerido corretamente: a liberação física de todos os recursos é delegada à função `destruir_pilha_tabelas`, executada ao término do processo de análise.

4.4.5 Processamento da Árvore Sintática e Validação

Diferente de compiladores de passagem única, a implementação do *Micro C* realiza a análise semântica de forma sequencial à sintática. O módulo semântico percorre a Árvore Sintática Abstrata (ASA) gerada anteriormente, utilizando a Tabela de Símbolos para validar regras de contexto e escopo. O percurso da árvore ocorre em profundidade (*Tree Walking*), interagindo com a tabela em três momentos-chave:

1. **Declaração de identificadores**: ao visitar nós de declaração (ex: `NODE_VAR_DECL`), o analisador verifica se o identificador já existe no escopo corrente antes de invocar a função `adicionar_simbolo`.
2. **Uso de identificadores**: em nós de expressão ou atribuição, utiliza-se `buscar_simbolo_em_todos_escopos` para garantir que a variável foi declarada e para recuperar seu tipo para checagem.
3. **Gerenciamento de escopos**: ao descer em nós de bloco ou função, o analisador invoca `empilhar_tabela`, e ao retornar da visita aos filhos, executa `desempilhar_tabela`, garantindo o isolamento das variáveis locais.

O trecho abaixo, extraído da função `analisar_no` do módulo semântico, ilustra como essa validação é aplicada na prática ao encontrar uma declaração de variável na ASA:

```
case NODE_VAR_DECL: {
    ASTNode* tipo_no = no->filho;
    ASTNode* id_no = tipo_no->proximo_irmao;
    char* nome_var = id_no->data.string_value;

    //1. verifica duplicidade no escopo atual
    if (buscar_simbolo_no_escopo_atual(pilha, nome_var)) {
        char msg[200];
        snprintf(msg, sizeof(msg), "Variavel ‘%s’ ja foi declarada.",
                 nome_var);
        erro_semantico(msg, id_no->linha);
    }

    //2. adiciona à Tabela de Símbolos
    adicionar_simbolo(pilha, nome_var, tipo_var, 0, is_array, ...);

    //3. linkagem: Salva o ponteiro do símbolo no nó da árvore
    id_no->symbol = buscar_simbolo_no_escopo_atual(pilha, nome_var);
    break;
}
```

Como foi possível observar, essa abordagem permite que o compilador trate escopos complexos e recursão, validando os tipos antes da geração de código.

Vale ressaltar que a implementação da Tabela de Símbolos utilizou vetores com busca linear. Em cenários de produção, essa abordagem é considerada ineficiente devido à complexidade $O(n)$ nas operações de busca [5], sendo preferível o uso de *Tabelas Hash* $O(1)$ [1].

No entanto, para o escopo do *Micro C*, a escolha pelos vetores foi intencional e pedagógica. Essa abstração simplificada reduz a sobrecarga de código (como o tratamento de colisões), tornando a implementação mais legível e facilitando a compreensão dos conceitos fundamentais de escopo e tipagem.

4.4.6 Importância da Tabela na Análise Semântica

Se a Árvore Sintática Abstrata (ASA) é o esqueleto do programa, a Tabela de Símbolos é o seu cérebro e memória. A ASA, por si só, não tem contexto; um nó de identificador `x` em uma expressão (ex: `y = x + 5;`) é apenas um nó, sem conexão com a sua declaração, `int x;`, que pode ter ocorrido muitas linhas antes, no início do bloco. Para que o compilador possa determinar que esses dois usos `x` se referem à mesma entidade, que é uma variável inteira, ele precisa de uma estrutura que, neste caso, é a Tabela de Símbolos.

Essa tabela funciona como um banco de dados para todos os identificadores do

programa. Durante o percurso da ASA, o analisador semântico interage constantemente com a Tabela de Símbolos para realizar suas verificações, permitindo validações como: **a declaração prévia de variáveis antes de seu uso**, por exemplo, em `x = 10;` se `x` não foi declarado; **compatibilidade de tipos em atribuições**, por exemplo, em `int x = 'a';`; e, finalmente, **a validação de funções**, como a existência de `main` e a ausência de redeclarações.

Além disso, a Tabela de Símbolos não apenas suporta a análise semântica, mas também é a base para etapas posteriores de compilação, como a geração de código e otimização, onde informações sobre tipos e escopos são importantes.

Por esses motivos, essa tabela é um componente indispensável do compilador, facilitando a análise semântica por meio do armazenamento e gerenciamento de identificadores. Sua implementação, baseada em uma pilha de tabelas dinâmicas, suporta escopos aninhados de forma eficiente, respeitando as regras de visibilidade da linguagem.

4.5 Sumário

Neste capítulo, exploramos a terceira fase do processo de compilação, a análise semântica, que é responsável por verificar o *significado* e a coerência do código-fonte. Dessa forma, embora um programa possa estar sintaticamente correto, ele ainda pode conter erros de lógica, como incompatibilidade de tipos ou o uso de variáveis não declaradas.

Para realizar essa validação, foi introduzida uma estrutura de dados chamada de Tabela de Símbolos. Durante a avaliação de funções importantes e do uso da tabela no processo de compilação, foram explorados detalhes da implementação de uma *pilha de tabelas*, que é a estratégia fundamental para gerenciar os escopos aninhados da linguagem. Em seguida, analisados os códigos das principais funções que manipulam essa pilha e como elas colaboram no processo de resolução de erros semânticos no código do *Micro C*.

Por fim, foram avaliados os cenários de como o analisador semântico interage com essa Tabela de Símbolos ao percorrer a ASA para validar o código e suas regras de contexto. Com a conclusão desta fase, o compilador não apenas tem uma representação estrutural do programa que é a Árvore Sintática Abstrata, mas também uma compreensão completa do contexto e do significado de cada identificador, deixando o caminho preparado para as fases de *back-end* de compilação (geração de *intercode* e código em *Assembly*).

Capítulo 5

Geração de Código Intermediário

> *The greatest revolution of our generation is the discovery that human beings, by changing the inner attitudes of their minds, can change the outer aspects of their lives.*
>
> – William James

Os capítulos anteriores abordaram a parte do compilador conhecida como *front-end*, ou seja, a compilação ocorria em torno da análise léxica, sintática e semântica do código fonte. Na análise léxica (Capítulo 2), foram validados os lexemas e processamento correto de tokens, na análise sintática (Capítulo 3) foi construida a Árvore Sintática Abstrata (ASA) e, finalmente, na análise semântica (Capítulo 4) validou-se essa árvore, verificando tipos, escopos e preenchendo a Tabela de Símbolos. Contudo, somente a validação da ASA, embora essencial, não era suficiente porque ela ainda representa o programa em um formato hierárquico e de alto nível, muito distante do que um processador pode executar. A análise semântica apenas confirmou que o programa *faz sentido*, mas não definiu *como* ele deve ser executado.

Neste capítulo se iniciará a implementação do *back-end* do compilador, abordando, inicialmente, a **geração de código intermediário**. Esta fase em si inicia no processo de *síntese*, traduzindo a ASA hierárquica e, previamente verificada, em uma nova representação: uma sequência linear de instruções simples, independente da máquina, conhecida como Código Intermediário. Esta fase do processamento realiza a conexão entre a análise (o *o quê*) e a geração de código final (o *como*).

5.1 A Transição da Análise para a Síntese

Com a conclusão da análise semântica, se encerra a construção do *front-end* do compilador. Dessa forma, o compilador agora sabe o que cada variável significa, qual o seu tipo, onde ela está armazeanda e se as operações fazem sentido.

A partir deste ponto, a função do compilador muda da análise para a **síntese**, marcando o início da construção do *back-end*. O *back-end* não questiona mais o significado do programa; ele aceita a representação validada pelo front-end como correta. Sua nova missão é *construir* um novo artefato funcional a partir dessa representação. Essa construção é o programa executável, ou uma representação que pode ser facilmente convertida em um. A primeira etapa desse processo de síntese é a **geração de código intermediário**.

5.1.1 O que é um Código Intermediário?

Um Código ou Representação Intermediária (RI) (*Intermediate Representation* - IR) é uma linguagem "no meio do caminho". Ela é mais baixo nível e mais explícita que a linguagem fonte (como a descrita para o compilador do *Micro C*), mas é mais alto nível e mais abstrata que a linguagem de máquina (ex.: *Assembly* x86-64).

Ou seja, o RI é um *idioma universal* para compiladores. Em vez de escrever um tradutor direto de Português para Japonês e outro de Português para Alemão, é mais eficiente traduzir o Português para uma língua intermediária, como o Esperanto, e depois criar tradutores menores de Esperanto para Japonês e de Esperanto para Alemão. Tornando o exemplo mais concreto:

- A ASA é a linguagem de alto nível, complexa e hierárquica.
- O RI é a linguagem intermediária, simples, linear e independente de máquina.
- O *Assembly* é a linguagem de baixo nível, específica para um processador.

Uma boa RI deve possuir algumas características para torná-lo mais eficiente. A primeira é a **Independência de Máquina:** A RI não deve conter nenhuma informação específica de um processador, como o nome de um registrador (ex: `%eax`). Isso garante que a primeira metade do *back-end* (gerador de RI) seja portável. E a segunda é a **Linearidade e Explicitude:** Ela representa o programa como uma longa sequência de instruções muito simples, onde todo o fluxo de controle (como os saltos de um `if`) e a ordem de avaliação de expressões se tornam explícitos. Existem muitos tipos de RI. Na implementação do compilador do *Micro C*, optou-se pela construção clássica utilizando **Códigos de Três Endereços**(`CTE`).

5.1.2 *Front-End* vs. *Back-End*: Onde Traçar a Linha?

A fronteira precisa entre o *front-end* e o *back-end* de um compilador é um tema controverso. Em geral, entende-se que o *front-end* é responsável pela análise do programa-fonte e pela geração de uma representação intermediária (RI), enquanto o *back-end* converte essa RI em código de máquina. No modelo clássico [1], a geração de código intermediário é tratada como a etapa final do *front-end*, uma vez que a RI ainda é uma forma abstrata e independente de máquina. Por outro lado, abordagens mais modernas [4] propõem uma divisão em três grandes fases: o *front-end*, responsável pela análise; o *middle-end*, dedicado à geração e otimização da RI;

e o *back-end*, encarregado da geração de código para a arquitetura alvo.

No entanto, no compilador do *Micro C*, foi adotada uma divisão conceitual ligeiramente diferente, com foco na **intenção** da fase. No ***Front-End* (Fases de Análise)**, a missão é exclusivamente *entender e validar* o código fonte. Todas as operações são feitas sobre representações diretas do código original (*tokens*, ASA, tabela de símbolos). Essa fase termina quando o compilador pode afirmar com 100% de certeza: "Eu entendo este programa e ele está semanticamente correto". Por outro lado, para o ***Back-End* (Fases de Síntese)**, o objetivo é *construir e traduzir*. Essa fase começa no momento em que o compilador, pela primeira vez, gera uma **nova representação** do programa que não é mais um reflexo da estrutura do código fonte, mas sim um passo em direção à lógica de execução de uma máquina. Em nosso caso, essa primeira representação é o RI.

Essa divisão faz sentido para o compilador do *Micro C*, porquê: primeiro, ela cria uma separação clara. Ou seja, o *front-end* termina quando a análise e a verificação do código terminam, que é onde o trabalho de validação termina. E, segundo, o *back-end* é um processo de duas etapas que gera o RI e depois transforma-o em *Assembly*. Essas fases tem como objetivo a *síntese* do programa final. Considerar a geração do RI como o passo inicial do *back-end* agrupa todas as tarefas de tradução de forma coesa, pois ambas as fases (geração de RI e geração de *Assembly*) são, em essência, etapas de tradução que transformam uma representação em outra, de nível mais baixo. Por simplicidade, a construção do compilador do *Micro C* considera o *front-end* como um *motor de compressão* e o *back-end* (a começar pela geração do RI) como o *motor de construção*.

5.1.3 O Papel do *Back-End*: Do "O Quê" para o "Como"

O *front-end* nos entregou uma descrição precisa do *o quê*. Ele nos disse: *Esta parte do programa é um laço* `for` *que itera 10 vezes; dentro dele, há uma atribuição de um valor inteiro a uma variável chamada* `x`. A ASA e a Tabela de Símbolos são esse "o quê"detalhado.

O trabalho do *back-end* é descobrir o "como". Ele precisa responder a perguntas fundamentalmente diferentes, que estão muito mais próximas do hardware:

- *Como* eu implemento um laço? – usando rótulos e instruções de desvio condicional e incondicional.
- *Como* eu acesso a variável `x`? – acessando sua posição na pilha de execução, que está no endereço relativo `[rbp-4]`.
- *Como* eu realizo uma soma? – carregando os valores para registradores da CPU e usando a instrução de máquina `ADD`.

O *back-end*, portanto, é a ponte que conecta a lógica abstrata da linguagem de programação com a realidade concreta do processador. A geração de código intermediário é o primeiro e mais importante passo nessa tradução, pois cria uma

representação que já começa a pensar em termos de “como", mas de forma ainda abstrata e independente de uma máquina específica.

5.1.4 A ASA e a Tabela de Símbolos

Para realizar a tradução do *o quê* para o *como*, o gerador de código intermediário precisa de um mapa detalhado. Isso é fornecido pelo *front-end* por duas estruturas de dados. A primeira é a **A Árvore Sintática Abstrata (ASA)** que dita a ordem e a estrutura das operações. O gerador de código irá *caminhar* por esta árvore, nó-a-nó, e para cada nó visitado, ele irá gerar uma ou mais instruções correspondentes. A ASA define o fluxo de controle e a estrutura das expressões. A segunda estrutura é a **A Tabela de Símbolos** que fornece o conjunto de informações contextuais. A ASA por si só é incompleta. Quando o gerador encontra um nó `NO_ID` com o nome `x`, ele não sabe nada sobre `x`. É nesse momento que ele consulta a Tabela de Símbolos para obter os detalhes sobre a variável: *`x` é uma variável do tipo `int` e seu endereço na pilha é `-8`*. Sem essa informação, seria impossível gerar o código correto para acessar a variável.

A interação entre essas duas estruturas é constante. O gerador de código segue o mapa da ASA e, a cada passo, consulta as informações da Tabela de Símbolos para obter os detalhes necessários para executar aquele passo.

5.2 O Papel da Representação Intermediária

No desenvolvimento de um compilador uma questão natural é o porquê não traduzir uma Árvore Sintática Abstrata diretamente para o código final em *assembly*. A resposta está nos princípios de design de software que priorizam a modularidade, a abstração e a separação de objetivos e resultados em cada uma das fases do compilador.

Gerar código *Assembly* diretamente a partir da ASA é uma tarefa complexa. O compilador precisaria tratar simultaneamente com a lógica de alto nível da linguagem (como a semântica de um laço `for` ou de um `if-else` aninhado) e com os detalhes de baixo nível da máquina (gerenciamento de registradores, convenções de chamada de função, conjunto de instruções específico do processador). Ou seja, essa abordagem construiria um compilador monolítico que seria, ao mesmo tempo, difícil de escrever, depurar, manter e, principalmente, muito complexo para portar para outras arquiteturas de computador.

O compilador do *Micro C* foi implementado utilizando técnicas modernas de compiladores, utilizando a estratégia de divisão e conquista, quebrando tarefas complexas em etapas mais simples. Uma delas, por exemplo, é o uso da geração intermediária. Essa segmentação das etapas de compilação traz vantagens robustas que serão verificadas nas seções seguintes.

5.2.1 Abstração e a Simplificação do Problema

A principal vantagem de usar uma representação intermediária é a aplicação da estratégia divisão e conquista. Em vez de um problema único e massivo de tradução, passamos a ter dois problemas menores e bem definidos.

O primeiro problema é a **Tradução da Linguagem Fonte para a Representação Intermediária**. Nesta etapa, realizada pelo módulo `intercode.c`, a tradução de alto nível é direcionada, exclusivamente, para traduzir a lógica de alto nível da ASA para uma representação linear simples. O compilador não precisa saber sobre registradores ou instruções de máquina. Sua única tarefa é *desenrolar* as estruturas complexas em versões mais simples. Por exemplo, ele transforma um nó `NODE_IF` da árvore na seguinte sequência lógica:

1. Calcule a condição.
2. Se a condição for falsa, pule para o rótulo do `else`.
3. Execute o bloco `then`.
4. Pule para o rótulo do fim do `if`.
5. Marque o início do rótulo do `else`.

E a etapa final, a **Tradução da Representação Intermediária para a Linguagem Alvo (*Assembly*)**. Nessa etapa, o compilador (`assembly.c`) tem uma tarefa mais direta. Ele recebe a lista de instruções simples da RI (como *some dois valores*, *salte para o rótulo L1* ou *atribua um valor a uma variável*) e sua única responsabilidade é **mapear** cada uma dessas instruções abstratas para uma ou mais instruções de *Assembly* específicas da máquina-alvo (como `addl`, `jmp` ou `movl`). O compilador não precisa mais se preocupar em decompor a lógica de alto nível de um `if` ou `for`; essa complexidade já foi *compactada* em instruções de desvio e rótulos simples pela fase anterior.

Essa separação permite que cada módulo tenha uma única responsabilidade, tornando o código do compilador mais limpo, mais fácil de testar e muito mais simples de depurar.

5.2.2 Portabilidade entre Arquiteturas

Diferentes processadores *processam* diferentes instruções de máquina (dialetos) que são mapeadas em mnemônicos (o que ficou conhecido como *assembly*). O conjunto de instruções de um processador Intel x86-64 por exemplo, é completamente diferente do de um processador ARM. A tarefa de gerar código de máquina (binário) é, portanto, inerentemente dependente da arquitetura alvo. Um compilador moderno deve ser capaz de gerar código para múltiplas arquiteturas, um desafio conhecido como *portabilidade*.

Se o compilador do *Micro C* gerasse código de máquina diretamente da ASA

para uma única arquitetura, seria necessária muitas mudanças para ser possível gerar código de máquina para outras arquiteturas. A representação intermediária resolve este problema atuando com uma *linguagem universal* que é independente da arquitetura da máquina.

A Figura 5.1 mostra como o compilador do *Micro C* utiliza uma representação modular para desacoplar completamente o *front-end* do *back-end*. O *front-end* e o *Gerador ASA para RI* formam um componente coeso e reutilizável. A complexidade fica dividida entre a primeira parte do *back-end* relacionada a tradução da lógica da linguagem para a RI, que é uma tarefa independente de máquina. E a segunda parte, que consiste em múltiplos *Tradutores* menores e simples. A tarefa de um *Tradutor RI para x86* é mais fácil do que a de um *Back-End Completo para x86*, pois ele parte de uma sequência de instruções linear, explícita e simplificada.

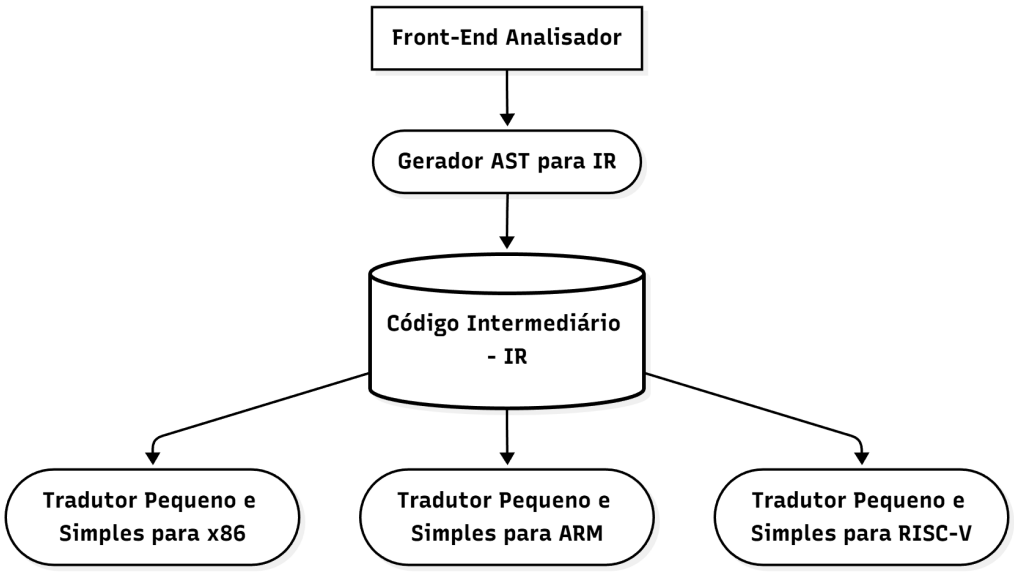


Figura 5.1: A abordagem modular, com Código Intermediário.

A principal vantagem na estratégia adotada na implementação do compilador do *Micro C* é a portabilidade. Ou seja, para dar suporte a outras arquiteturas como o RISC-V, por exemplo, bastaria adicionar um novo *Tradutor RI para RISC-V*. Todo o trabalho do *front-end* e da geração da RI seria reaproveitado. Isso torna o esforço de portar o compilador menor e incentiva um design limpo e modular.

Este modelo, que separa o *front-end* (que produz a RI) do *back-end* (que consome a RI), tornou-se o padrão de fato na indústria de compiladores modernos, uma abordagem que foi solidificada por *frameworks* como o LLVM [11]. Lattner *et al.* [11], por exemplo, demonstrou como uma Representação Intermediária (RI) comum e reutilizável permite que múltiplos *front-ends* (para linguagens diferentes) e múltiplos *back-ends* (para arquiteturas diferentes) sejam desenvolvidos de forma independente.

5.3 O Código de Três Endereços (CTE)

O Código de Três Endereços (CTE) do inglês (*Three Address Code* - TAC) é uma representação intermediária usada por compiladores para facilitar a otimização e a tradução do código-fonte para o código de máquina.

O princípio fundamental do CTE é a simplicidade, ou seja, cada instrução deve conter no máximo uma operação. Isso significa que expressões complexas e aninhadas do código fonte são decompostas em uma sequência de instruções simples e explícitas. No CTE, cada instrução contém no máximo três endereços (operandos), geralmente no formato:

```
x = y op z
```

Ou seja, `x`, `y`, e `z` são os *endereços* (operandos). Eles podem representar variáveis do programa, constantes ou variáveis temporárias criadas pelo próprio compilador. Essa estrutura linear e explícita se assemelha a uma espécie de *Assembly universal*, tornando a tradução final para o *Assembly* de máquina mais sistemática e direta.

5.3.1 A Estrutura do CTE no compilador do *Micro C*

A implementação CTE, como representação intermediária do compilador, requer uma definição formal em linguagem C de sua estrutura. Essa especificação abrange três componentes essenciais: os opcodes (que definem o conjunto de operações atômicas), os operandos (que representam os dados manipulados) e as instruções (que são a combinação estruturada de opcodes e operandos, representando a unidade de código executável). A estrutura léxica e a sintática da RI do compilador do *Micro C* são especificados no arquivo de cabeçalho `ir.h`

Os Opcodes: Os *Verbos* da RI

Para representar todas as operações possíveis da nossa RI, criamos uma enumeração chamada `IROpcode`. Ela contém uma entrada para cada tipo de ação que o código intermediário pode executar. Os opcodes foram agrupados por categoria para facilitar a compreensão. As categorias definidas foram: operações aritméticas, de comparação, de movimentação de dados, de controle de fluxo e de chamada de função.

O trecho de código a seguir, extraído diretamente do arquivo de cabeçalho `ir.h`, apresenta a definição da enumeração `IROpcode`. Esta estrutura é o núcleo estrutural da RI, definindo cada *verbo* (operação) que o *back-end* é capaz de processar. Como descrito anteriormente, os opcodes são agrupados por sua função (aritmética, controle de fluxo, etc.) para facilitar a implementação e a manutenção do tradutor de *Assembly*:

```
// arquivo: src/ir/ir.h (trecho)
typedef enum {
    //opcodes aritméticos, lógicos e de comparação
    IR_ADD, IR_SUB, IR_MUL, IR_DIV, IR_MOD,
    IR_EQ, IR_NEQ, IR_LT, IR_LEQ, IR_GT, IR_GEQ,
    IR_AND, IR_OR, IR_NOT, IR_NEG,

    //opcodes de movimentação de dados e acesso a memória
    IR_ASSIGN,      // Atribuição (x := y)
    IR_STORE,       // Armazenar em array (v[i] := x)
    IR_LOAD,        // Carregar de array (x := v[i])

    //opcodes de controle de fluxo e funções
    IR_LABEL, IR_GOTO, IR_IF_FALSE,
    IR_PARAM, IR_CALL, IR_RETURN,
} IROpcode;
```

Os Operandos: Os *Substantivos* da RI

Os endereços em no CTE podem ser de diferentes naturezas (uma variável, um número, um rótulo, etc.). Para representar todas essas possibilidades, foram criadas estruturas de `struct IROperand`. Elas utilizam um `enum OperandType` para identificar o tipo de dado que carrega e uma `union` para armazenar o valor de forma eficiente, economizando memória.

```
// arquivo: src/ir/ir.h (trecho)
typedef enum {
    OPERAND_SYMBOL,     // Uma variável ou parâmetro
    (ponteiro para Tabela de Símbolos)
    OPERAND_TEMP,       // Um temporário gerado pelo
    compilador (t0, t1...)
    OPERAND_CONST,      // Uma constante inteira
    OPERAND_LABEL,      // Um rótulo de código (L0, L1..., main)
    OPERAND_STRING_LBL  // Um rótulo para uma string literal
} OperandType;

typedef struct {
    OperandType type;
    union {
        Simbolo* symbol;
        int temp_id;
        int const_val;
        char* label_name;
    } data;
} IROperand;
```

As Instruções: As *Frases* da RI

Finalmente, a `struct IR_Instruction` unifica os opcodes e os operandos descritos anteriormente. Ela representa uma única linha do nosso código intermediário. Isso

ocorre pois esta estrutura contém o opcode (a operação a ser executada) e ponteiros para até três operandos (resultado, argumento 1 e argumento 2). O ponteiro `next` é o que permite conectar as instruções em uma lista encadeada, formando a sequência linear que representa o programa.

```
// arquivo: src/ir/ir.h (trecho)
typedef struct IR_Instruction {
    IROpcode opcode;
    IROperand* result;
    IROperand* arg1;
    IROperand* arg2;
    struct IR_Instruction* next;
} IR_Instruction;
```

5.3.2 Formato das Instruções: Tradução para CTE

Com as estruturas de dados da RI definidas, é possível ver, na prática, como as construções da linguagem do *Micro C* são convertidas para o Código de Três Endereços (CTE) pelo módulo `intercode.c`. Esta subseção servirá como um guia de referência, mostrando exemplos práticos e explicando o processo de conversão para cada tipo de construção.

Atribuições e Expressões Aritméticas

A tradução de expressões é onde o princípio do CTE de no máximo uma operação por instrução se torna mais evidente. O exemplo a seguir mostra de forma simples como funciona a atribuição de uma constante:

- **Código Fonte:**

  ```
  int x = 10;
  ```

- **RI Gerada:**

  ```
  x := 10
  ```

Neste caso, a tradução é direta. A instrução `IR_ASSIGN` é usada para mover um valor de origem (o operando constante `10`) para um destino (o operando de símbolo `x`). Como há apenas uma operação implícita (a atribuição), uma única instrução na RI é suficiente para representar toda a linha de código.

O próximo exemplo mostra uma operação aritimética. Neste exemplo é possível observar o benefício da decomposição do problema para tornar as operações mais simples e atômicas.

- **Código Fonte:**

```
int z;
z = x + y;
```

- **RI Gerada:**

```
t0 := x + y
z := t0
```

A linha `z = x + y;` no exemplo possui *duas* operações: uma soma (`+`) e uma atribuição (`=`). Para respeitar a regra de que cada instrução deve ser atômica, o gerador de código divide o processo em dois passos lógicos.

Essa separação reflete como o processador opera fisicamente: ele não consegue realizar cálculos diretamente nas variáveis armazenadas na memória (RAM). Primeiro, é necessário carregar os valores e realizar a soma em um local de acesso rápido (representado aqui pela variável temporária `t0`, que simula um registrador da CPU). Somente após o cálculo, o resultado é movido de `t0` para o endereço de memória definitivo da variável `z`.

1. **Operação:** primeiro, ele resolve a operação `x + y`. Ou seja, ele gera a instrução `t0 := x + y`, que realiza a soma dos valores representados pelos operandos `x` e `y`, e armazena o resultado em uma nova variável temporária, `t0`.
2. **Atribuição:** com o resultado da expressão inicial guardado em `t0`, o gerador pode realizar a atribuição. Neste caso, ele gera a uma nova instrução `z := t0`, que move o valor do temporário para a variável de destino `z`.

Essa decomposição garante que cada linha da RI seja simples. É possível observar que esse pseudo-código é praticamente como funcionam as instruções em códigos de máquina utilizando instruções de operações entre registradores ou saltos para posições de memórias (condicionais ou não).

Acesso a Vetores (*Arrays*)

O acesso a vetores na RI são tratados com duas instruções especializadas: `LOAD` para leitura de um valor da memória e `STORE` para escrita de um valor na memória. Segue um exemplo de como uma posição de um vetor é carregada e atribuída para uma variável:

- **Código Fonte (Leitura):**

```
x = v[4];
```

- **RI Gerada:**

```
t0 := LOAD v[4]
x := t0
```

De forma similar, o acesso a uma posição de um vetor é decomposto para manter a simplicidade das operações. Ou seja, a **leitura** (***LOAD***) gera uma instrução especializada `IR_LOAD`. Isso significa que o endereço de memória correspondente ao índice `4` do vetor `v` é calculado, permitindo que o dado seja acessado diretamente, sem a necessidade de percorrer os elementos anteriores. Em seguida, o valor armazenado nesta posição é carregado para a variável temporária `t0`. No próximo passo, a **atribuição** ocorre com o valor que agora está disponível em `t0`, por meio de uma instrução simples (`x := t0`), finalizando a operação e movendo o valor para a variável de destino `x`.

A operação de escrita em um vetor (***STORE***) segue a mesma lógica de acesso direto, sendo encapsulada pela instrução `IR_STORE`. Essa instrução determina que o valor do operando `x` seja armazenado na posição de memória correspondente ao índice `5` do vetor `v`. Visto que a instrução opera diretamente na memória, não são necessários passos adicionais.

- **Código Fonte (Escrita):**

```
v[5] = x;
```

- **RI Gerada:**

```
STORE v[5] := x
```

Controle de Fluxo

Estruturas hierárquicas como o `if-else` são *compactadas* em uma sequência linear de testes e desvios (saltos) usando rótulos e instruções condicionais, que é como um processador de fato executa essa cadeia de operações. O exemplo a seguir mostra uma operação que compara se uma variável é menor do que a outra e atribui um valor a uma terceira variável, bem como a RI gerada no processo:

- **Código Fonte:**

```
if (a < b) {
    x = 1;
}
```

- **RI Gerada:**

```
    t0 := a < b
    if_false t0 goto L0
    x := 1
L0:
```

A conversão da estrutura condicional `if` para o formato CTE segue alguns passos. O primeiro é avaliar a **condição**, ou seja a verificar o resutlado da expressão condicional `a < b`. O resultado de expressões condicionais são boleanos (que em C

é `1` para verdadeiro e `0` para falso) é armazenado na variável temporária `t0`. Em seguida, é realizado um **desvio condicional**, ou seja, a instrução `if_false t0 goto L0` gerada controla o fluxo de execução do código. Se o valor em `t0` for falso (ou seja, 0), salte (goto) para o rótulo `L0`. Se for verdadeiro, a execução continua na próxima linha. Se o desvio condicional não for executado (condição verdadeira), o corpo do `if` é executado (`x := 1`). Por outro lado, sempre que uma condição estrutura condicional for inserida, é criado um **rótulo de saída**. Este rótulo `L0:` serve para permitir que o fluxo seja desviado (condição falsa) para fora do laço condicional do `if`.

5.3.3 Variáveis Temporárias e Rótulos

Como observado, o gerador de código precisou adicionar entidades extras que não existiam no código original (variáveis temporárias e rótulos) para respeitar a estrutura definida pela CTE. Para uma melhor compreensão as subseções a seguir descrevem estes dois tipos de entidades mais profundamente.

Variáveis Temporárias (`t0`, `t1`, ...)

Elas são essenciais para manter a simplicidade do Código de Três Endereços. Sempre que uma expressão complexa é avaliada, o resultado intermediário de cada operação é armazenado em uma nova variável temporária. A abstração neste caso é similar a ideia de existir um número infinito de registradores em uma máquina abstrata.

A tarefa de mapear essas variáveis temporárias para o número limitado de registradores reais do processador (ex: `%eax`, `%ebx`) fica a cargo da fase final de geração do código em *Assembly*. Na implementação do compilador do *Micro C* (`intercode.c`), é utilizado uma simplificação com o uso de um contador (`temp_counter`) para geração de nomes únicos.

Rótulos (`L0`, `L1`, ..., `main`)

Os rótulos são marcadores de posição no código. Eles servem como alvos para as instruções de desvio (`GOTO`, `IF_FALSE`) e como pontos de entrada para funções. Eles são a ferramenta que nos permite *agrupar* a estrutura de controle aninhada da ASA (como `if`s e `for`s) em um fluxo de código linear, que é como um processador executa uma sequência de códigos. O gerador de código do compilador do *Micro C* também usa um contador (`label_counter`) para garantir que cada rótulo gerado para as estruturas de controle seja único.

5.4 Gerador de Código: ASA para RI

Com as estruturas da Representação Intermediária (RI) definidas, está na hora de definir o núcleo do *back-end*: a construção do tradutor. Este componente, implementado no módulo `intercode`, tem a tarefa de usar a Árvore Sintática Abstrata

(ASA) que possui as informações hierárquicas de de significado e convertê-la em uma lista linear de instruções simples do Código de Três Endereços (CTE).

Para realizar essa tradução de uma estrutura em árvore (a ASA) para uma estrutura linear (a lista de RI), a abordagem mais comum é percorrer a árvore nó por nó, gerando o código correspondente para cada um. Esta foi a estratégia utilizada na implementação.

5.4.1 *Tree Walker* e a Interação com a Tabela de Símbolos

Para traduzir a ASA, é utilizado um padrão de projeto clássico conhecido como o ***Tree Walker*** (ou "caminhante da árvore"). Como o nome sugere, ele consiste em uma função recursiva que *visita* cada nó da ASA e, para cada nó, ele executa uma ação de tradução específica.

Contudo, o *tree walker* não realiza suas operações de forma independente. Ele usa as duas grandes estruturas de dados produzidas pelo *front-end* (ASA e tabela de símbolos). Ou seja, a **ASA** fornece a **representação estrutural** das operações, ditando a ordem e a hierarquia do programa. Por outro lado, a **Tabela de Símbolos** atua como o **repositório de informações semânticas**. Para cada identificador (ID) encontrado na ASA, o gerador de código consulta a Tabela de Símbolos para obter seu contexto, seu tipo, seu escopo e, o mais importante para o *back-end*, seu `memory_offset`. Dessa forma, a cada passo, o gerador de código segue a estrutura da ASA e consulta o repositório da Tabela de Símbolos para obter os detalhes.

Do ponto de vista da implementação no compilador do *Micro C*, a lógica envolvida na organização do *tree walker* é dividida em duas funções: `gerar_ir_expr()` para expressões que calculam um valor, e `gerar_ir_no()` para instruções que realizam uma ação.

Essa separação de responsabilidades é uma decisão de projeto. Expressões (como `a + b`) são recursivas, podem ser aninhadas e devem sempre *retornar um valor* (ou o local onde o valor foi armazenado). Em contraste, instruções (ou *statements*, como `if` ou `for`) controlam o fluxo do programa e não retornam valores. As subseções seguintes detalharão a implementação de cada uma dessas funções, começando pela `gerar_ir_expr`.

5.4.2 Traduzindo Expressões: A Função `gerar_ir_expr`

Esta função é especialista em traduzir qualquer parte do código que produza um valor. Sua principal característica é que ela sempre **retorna** um `IROperand`, que representa o local onde o resultado do cálculo foi armazenado.

O Ponto de Partida: Variáveis e Constantes

Os casos mais simples de tradução de expressão ocorrem nos nós folha da árvore, como constantes ou identificadores.

O processamento para o nó `NODE_ID` (que representa o uso de uma variável) é detalhado no trecho de código a seguir, extraído do `intercode.c`:

- **Código Fonte:** `x` (o uso de uma variável)

```
// arquivo: src/intercode/intercode.c (trecho)
case NODE_ID: {
    // 1. Consulta a Tabela de Símbolos para encontrar 'x'
    Simbolo* s = buscar_simbolo_em_todos_escopos(pilha,
                                    no->data.string_value);

    // 2. Cria um operando que aponta para as informações de 'x'
    IROperand* var = criar_operando(OPERAND_SYMBOL);
    var->data.symbol = s;

    // 3. Retorna este operando
    return var;
}
```

O código mostra que ao encontrar um `NODE_ID`, o gerador executa invoca o método `buscar_simbolo_em_todos_escopos` que retorna um ponteiro para um `Simbolo` que contém o tipo, o nome, o *offset* de memória, etc., é armazenado no `IROperand`. Nenhuma instrução de RI é gerada neste momento; a função apenas retorna a referência para a variável, para que quem a chamou possa usá-la.

5.4.3 Traduzindo Instruções: função `gerar_ir_no`

Esta função é a contraparte da `gerar_ir_expr`; ela é responsável por traduzir *instruções* (*statements*), que são os nós da ASA que representam ações e não retornam valores. Isso inclui o controle de fluxo (como `NODE_IF` e `NODE_FOR`) e as operações de efeito colateral, como as atribuições. O exemplo mais fundamental de instrução é a atribuição (`NODE_ASSIGN`), detalhada a seguir, que demonstra a colaboração entre `gerar_ir_no` e `gerar_ir_expr`.

Atribuição (`NODE_ASSIGN`)

- **Código Fonte:** `y = x + 10;`

```
// arquivo: src/intercode/intercode.c (trecho)
case NODE_ASSIGN: {
    // 1. Identifica o destino da atribuição ('y')
    Simbolo* s = buscar_simbolo_em_todos_escopos(pilha, ...);
    IROperand* dest = criar_operando(OPERAND_SYMBOL);
    dest->data.symbol = s;

    // 2. Pede para gerar_ir_expr avaliar o lado direito
    //    ('x + 10')
    //    Isso gera "t0 := x + 10" e retorna o operando 't0'
    IROperand* src = gerar_ir_expr(no->filho
    ->proximo_irmao, pilha);

    // 3. Emite a instrução final de atribuição
    emitir(IR_ASSIGN, dest, src, NULL); // Gera: y := t0
    break;
}
```

A tradução mostra a colaboração entre as duas funções. A `gerar_ir_no` processa a atribuição, mas delega a avaliação da expressão complexa para a `gerar_ir_expr`. Após a `gerar_ir_expr` retornar o operando temporário (`t0`) com o resultado da soma, a `gerar_ir_no` finaliza o processo, emitindo a instrução que move o valor do temporário para a variável final.

Operações Binárias

Diferente dos nós folha (como `NODE_ID` ou `NODE_INTEGER_CONST`), que apenas retornam um operando existente, os nós de operação binária são o primeiro exemplo onde o tradutor *gera* ativamente novas instruções de código intermediário.

- **Código Fonte:** `x + 10`

```
// arquivo: src/intercode/intercode.c (trecho)
case NODE_BINARY_OP: {
    // A. Resolve recursivamente o operando da esquerda ('x')
    IROperand* arg1 = gerar_ir_expr(no->filho, pilha);

    // B. Resolve recursivamente o operando da direita ('10')
    IROperand* arg2 = gerar_ir_expr(no->filho
    ->proximo_irmao, pilha);

    // C. Cria um novo "local" temporário para o resultado
    IROperand* temp = criar_operando_temporario(); // ex: t0

    // D. Emite a instrução de três endereços
    emitir(IR_ADD, temp, arg1, arg2); // Gera: t0 := x + 10

    // E. Retorna o local do resultado
    return temp;
}
```

O processo segue uma lógica de pós-ordem: primeiro resolve os filhos, depois executa sua própria ação. As chamadas recursivas para `gerar_ir_expr` retornam os operandos para `x` e `10`. Dessa forma, a função cria um novo temporário (`t0`), emite a instrução de soma e retorna o operando `t0`, com o contexto necessário para que a função superior tenha informações de que o resultado esperado (de `x + 10`) está armazenado na variável `t0`.

Controle de Fluxo (`NODE_IF`)

A tradução de estruturas hierárquicas como o `if-else` é *compactada* em um fluxo linear usando rótulos e desvios.

- **Código Fonte:** `if (cond) { A } else { B }`

O processo de tradução segue o seguinte passo-a-passo:

1. **Criação de Rótulos:** O gerador *cria* dois novos rótulos para marcar pontos no código: um para o início do bloco `else` (que podemos chamar conceitualmente de **L_ELSE**) e outro para o fim da instrução `if` (o **L_FIM_IF**). No nosso código, a função `criar_operando_label_novo()` faz isso, criando nomes reais como `L0` e `L1`.
2. **Tradução da Condição:** o gerador invoca `gerar_ir_expr` para a condição. O resultado é um temporário, `t_cond`.
3. **Desvio Condicional:** a instrução `if_false t_cond goto L_ELSE` é emitida. Ela significa: "se a condição em `t_cond` for falsa, pule para o início do bloco `else`".
4. **Tradução do Bloco 'Then':** o corpo do `if` (bloco `A`) é traduzido em seguida. Após sua última instrução, um desvio incondicional `goto L_FIM_IF` é emitido para *pular por cima* do bloco `else`.
5. **Tradução do Bloco 'Else':** o rótulo `L_ELSE:` é emitido, seguido pela tradução do bloco `B`.
6. **Rótulo Final:** por fim, o rótulo `L_FIM_IF:` é emitido, marcando o ponto para onde a execução continua após o `if` ou o `else`.

O código que implementa essa lógica mostra como a estrutura complexa do `if` é traduzida.

A primeira parte é responsável por avaliar a condição e emitir o desvio condicional para o bloco `else`:

```
// arquivo: src/intercode/intercode.c (trecho: parte 1)
case NODE_IF: {
    IROperand* cond_result = gerar_ir_expr(no->filho,
    pilha);

    // Nossos nomes conceituais L_ELSE e L_FIM_IF viram
    // nomes reais aqui:
    IROperand* label_else = criar_operando_label_novo();
    // ex: L0
    IROperand* label_fim_if = criar_operando_label_novo();
    // ex: L1

    emitir(IR_IF_FALSE,
    criar_operando_label_nome(label_else->data.label_name),
           cond_result, NULL);
//...
```

A segunda parte do bloco de código gera a tradução para os blocos `then` e `else`, inserindo os rótulos e o desvio `goto` incondicional para garantir o fluxo correto:

```
//... (continuação do case NODE_IF)
    gerar_ir_no(no->filho->proximo_irmao, pilha);
    // bloco 'then'
    emitir(IR_GOTO, criar_operando_label_nome(label_fim_if
    ->data.label_name),
           NULL, NULL);

    emitir(IR_LABEL, label_else, NULL, NULL);
    if (no->filho->proximo_irmao->proximo_irmao) { // se
    existe 'else'
        gerar_ir_no(no->filho->proximo_irmao
        ->proximo_irmao, pilha);
    }

    emitir(IR_LABEL, label_fim_if, NULL, NULL);
    break;
}
```

Este trecho de código é a tradução direta da receita descrita anteriormente. Ele demonstra como o gerador de código, de forma sistemática, transforma a estrutura de árvore do `if` em uma sequência linear de testes e saltos. Essa mesma técnica de usar rótulos (*labels*) e desvios (*goto's*) é a base para a tradução de todas as outras estruturas de controle de fluxo, como os laços `for`. Ao final, a lógica complexa e aninhada do programador é convertida em um formato simples e explícito que se assemelha muito mais à forma como um processador de fato executa o código.

5.5 Sumário

Este capítulo expande a compreensão sobre os processos relacionados ao *back-end* do compilador. Inicialmente são discutidas as etapas relacionadas a síntese como, por exemplo, a *geração de código intermediário*. Em seguida, são apresentadas as justificativas do porquê utilizar uma representação intermediária garante maior modularidade e portabilidade do compilador levando em conta diferentes arquiteturas de computadores.

O próximo passo foi explorar e determinar o uso do **Código de Três Endereços (CTE)**, principalmente pela simplicidade na conversão por sua simplicidade e formato linear, que se assemelha a um *Assembly* abstrato. Detalhamos a implementação dessa RI em C, analisando o arquivo `ir.h` e o papel de suas três estruturas centrais: `IROpcode` (os *verbos*), `IROperand` (os *substantivos*) e `IR_Instruction` (a *frase* que une tudo em uma lista encadeada).

Com a estrutura da RI definida, o foco principal do capítulo foi a construção do tradutor, o módulo `intercode.c`. Explicamos a nossa estratégia de implementação, o padrão ***Tree Walker*** (caminhante da árvore), que utiliza a ASA como um *mapa* e a Tabela de Símbolos como um *dicionário*. Detalhamos como as funções `gerar_ir_no` (para instruções) e `gerar_ir_expr` (para expressões) trabalham em conjunto, percorrendo a ASA e consultando a Tabela de Símbolos para traduzir cada nó.

Finalmente, demonstramos com exemplos de código como as construções do compilador do *Micro C*, desde simples atribuições e expressões aritméticas até estruturas complexas como acesso a vetores e o controle de fluxo do `if-else`, são sistematicamente *compactadas* e convertidas para a sequência linear de instruções do CTE. Ao final deste capítulo, temos um programa que traduz com sucesso a ASA validada pelo *front-end* para uma Representação Intermediária correta e pronta para a próxima etapa: a geração de código *Assembly*.

Capítulo 6

Geração de Assembly

"Vision without execution is hallucination."

– Thomas Edison

No capítulo anterior, foi concluída a primeira fase do *back-end* do compilador ao traduzir a Árvore Sintática Abstrata (ASA) para uma Representação Intermediária (RI). Essa RI, por definição, é abstrata e independente de plataforma; ela nos diz *o que* o programa deve fazer, mas não *como* uma máquina específica deve executá-lo.

Neste capítulo, será abordada a etapa final da compilação (*execução*): a **geração de código *Assembly***. Esta é a *última milha* da tradução, onde a lógica abstrata da RI é convertida em um conjunto de instruções concretas, projetadas para uma arquitetura de um processador, no caso, o x86-64.

Esta é a fase onde a *síntese*, iniciada no Capítulo 5, se completa. O objetivo é pegar a lista de instruções do Código de Três Endereços (CTE), que é linear e explícita, e produzir um arquivo de texto (`.s`) que contém um código *Assembly* x86-64 equivalente. Este arquivo, por sua vez, pode ser entregue a um **montador** (ou *assembler*), que é o programa responsável por traduzir o código *Assembly* textual em código de máquina binário (um arquivo objeto `.o`). Em seguida, esse arquivo objeto passa por um *linker* (ou *ligador*), que o combina com bibliotecas do sistema (como a biblioteca C, para funções como `printf`) e cria o programa executável final.

No compilador do Micro C, é utilizada a ferramenta `gcc` para realizar essas duas últimas etapas e gerar um executável que o sistema operacional pode carregar e executar diretamente no hardware. O **GCC** [15] (GNU Compiler Collection) é um conjunto de compiladores e ferramentas de desenvolvimento de software de código aberto. Embora seja mais conhecido como um compilador C, ele é usado pelo Micro C para, especificamente, atuar como o **montador** e ***linker*** da saída do *Asssembly* gerado pelo compilador do *Micro C*.

6.1 A Plataforma Alvo: x86-64 e a Sintaxe AT&T

A geração de código final é, por definição, dependente de plataforma. A Representação Intermediária (RI) que foi construída no Capítulo 5 é abstrata e universal, mas o processador de um computador não. Ele entende apenas um conjunto específico de instruções binárias. A primeira decisão de projeto do *back-end* é, portanto, definir qual será a arquitetura de máquina para a qual a linguagem intermediária será traduzida.

6.1.1 A Arquitetura x86-64

A arquitetura escolhida para saída do compilador do *Micro C* foi a **x86-64** (também conhecida como AMD64). Esta é a arquitetura de 64 bits que sucedeu a popular arquitetura de 32 bits (IA-32, ou x86). A escolha se justifica por sua onipresença em computadores pessoais, desktops e servidores modernos, englobando a vasta maioria dos processadores da Intel (Core i3/i5/i7/i9) e da AMD (Ryzen).

O termo *64 bits* se refere ao tamanho dos registradores de propósito geral (como `%rax`) e ao tamanho dos endereços de memória, permitindo ao processador acessar uma quantidade de memória muito maior do que seu predecessor de 32 bits.

Além disso, a x86-64 é uma arquitetura **CISC** (*Complex Instruction Set Computer*), o que significa que ela possui um conjunto de instruções rico e complexo, onde uma única instrução de *Assembly* pode realizar múltiplas micro-operações (como carregar um valor da memória, somá-lo a um registrador e salvar o resultado de volta na memória, em uma única instrução). Como a linguagem do *Micro C* é limitada, será utilizado apenas um pequeno subconjunto dessas instruções.

A escolha da arquitetura também foi pragmática, pois ela é o alvo padrão do `gcc` no ambiente utilizado no desenvolvimento do compilador do *Micro C*. Isso garantiu que o código *Assembly* gerado pelo compilador seria compatível também com as ferramentas e bibliotecas de desenvolvimento do próprio sistema operacional.

6.1.2 Sintaxe AT&T vs. Sintaxe Intel

Escolher a arquitetura x86-64 não foi a única decisão. O *Assembly* para esta arquitetura possui duas sintaxes principais diferentes, a **Sintaxe Intel** e a **Sintaxe AT&T**. Embora ambas produzam o mesmo código de máquina, o formato textual da sintaxe é muito diferente.

Enquanto a **Sintaxe Intel** é utilizada, principalmente, pela Microsoft, de acordo com a documentação oficial da Intel [8] (com montadores como MASM [6] ou NASM [16]), ambientes derivados do Unix, como Linux e macOS, utilizam a **Sintaxe AT&T** e, por isso, muitos projetos como GCC [15] e Clang [10] e suas bibliotecas, seguem este modelo. A Tabela 6.1 apresenta, de forma resumida, diferenças entre as duas sintaxes.

Tabela 6.1: Comparação de Sintaxe: Intel vs. AT&T (Ex: NASM vs. GCC)

Item	**Sintaxe Intel**	**Sintaxe AT&T**
Ordem	`instrução destino, fonte`	`instrução fonte, destino`
Registrador	`eax`, `rbp`	`%eax`, `%rbp`
Constante	`10`	`$10`
Sufixo (Tamanho)	`mov`, `add` (inferido)	`movl`, `addq` (explícito)
Memória	`[rbp-4]`	`-4(%rbp)`

Como o compilador do *Micro C* utiliza o `gcc` como montador e *linker*, a sintaxe utilizada como saída da etapa de geração de código intermediário segue o padrão da **Sintaxe AT&T**. As subseções seguintes detalham as implicações práticas dessa escolha.

Ordem dos Operandos

A diferença que causa maior confusão no *Assembly* gerado pelas sintaxes está relacionada a ordem dos operandos utilizados nas instruções. Enquanto na sintaxe da Intel as operações seguem a ordem `destino` ⇒ `origem`, na sintaxe da AT&T, elas seguem a ordem `origem` ⇒ `destino`. Por exemplo:

- **Intel:** `mov eax, 10`
  significa *mova 10* ***para dentro*** *do registrador* ***eax***.
- **AT&T:** `movl $10, %eax`
  significa *mova $10* ***para dentro*** *de* ***%eax***.

A lógica de ordem das instruções da AT&T, embora menos intuitiva, é consistente em todas as suas instruções.

Prefixos de Registradores e Imediatos

A sintaxe AT&T é mais explícita e menos ambígua sobre o que é um operando, por exemplo:

- **Registradores** são sempre prefixados com `%`. O que a Intel chama de `rbp`, a AT&T chama de `%rbp`. O que é `eax` vira `%eax`.
- **Valores Imediatos** (constantes) são sempre prefixados com `$`. O número `10` é escrito como `$10`.

Isso resolve ambiguidades. Enquanto na sintaxe Intel: `mov eax, 10`, que significa *mova o valor 10 para* ***eax***, pode causar confusão com outra instrução como, por exemplo, `mov eax, [10]`, que significa *mova o valor do endereço de memória 10 para* ***eax***. Isso não ocorre com a sintaxe da AT&T pois, `movl $10, %eax` (o valor 10), é sintaticamente distinto de `movl 10, %eax` (endereço de memória 10).

Sufixos de Mnemônico

Onde a sintaxe Intel muitas vezes infere o tamanho da operação (8, 16, 32 ou 64 bits) a partir dos operandos, a sintaxe AT&T exige que o tamanho seja explícito no nome da instrução (o mnemônico):

- `movl` (move long, 32 bits): este é o mnemônico que será usado para todas as operações com os **dados** do *Micro C*. Embora um `char` tenha 1 byte, para simplificar o alinhamento da pilha, o compilador do *Micro C* trata tanto `int`s quanto `char`s como valores de 32 bits (4 bytes) na memória.
- `movq` (move quad, 64 bits): este mnemônico é reservado para manipular os **ponteiros da própria arquitetura**. Como o alvo é um sistema de 64 bits, os endereços de memória (e, portanto, os registradores que os armazenam, como `%rsp` e `%rbp`) têm 64 bits. Foram utilizadas as instruções `movq` (e `pushq`/`popq`) exclusivamente para gerenciar o *stack frame*, como em `movq %rsp, %rbp`.
- **Strings (Literais):** strings não são movidas com uma única instrução `mov`. Em vez disso, elas são armazenadas na seção de dados `.rodata`. Quando é necessário utiliza-las (como em `print("ola")`), é usada a instrução `leaq` (Load Effective Address) para carregar o seu **endereço** (um ponteiro de 64 bits) em um registrador.

O gerador de código do *Micro C* usará `movl` para todas as operações de dados e `movq` (ou `pushq`/`popq`) para manipulação da pilha.

Endereçamento de Memória

Esta é a diferença sintática final e mais importante do compilador do *Micro C*. O acesso à memória (especialmente à pilha) é escrito de forma diferente. A sintaxe da Intel e da AT&T seriam representadas da seguinte maneira:

- **Intel:** `[rbp - 4]`
- **AT&T:** `-4(%rbp)`

Embora ambas signifiquem a mesma coisa: *pegue o endereço no registrador `%rbp` e subtraia 4 bytes dele*. A sintaxe AT&T usa o formato `offset(%base)`. Nas próximas seções, esta sintaxe será explorada com maior profundidade, pois ela pode ser estendida para o cálculo complexo de arrays, como `offset(%base, %indice, escala)`, que implementa `base + (indice * escala)`.

Com essas decisões de projeto, a arquitetura **x86-64** como alvo e a **Sintaxe AT&T** de saída definida, a engenharia do *back-end* está completa. Todas as instruções que o compilador do *Micro C* gerar, a partir deste ponto, deverão obedecer estritamente a essas convenções.

Definir as convenções e sintaxe, no entanto, são apenas metade do problema. Antes de serem realizadas as traduções das instruções da RI, é necessário definir

um plano de gerenciamento de memória. A RI opera com um número infinito de *temporários* e *variáveis*, que são recursos limitados para o *Assembly*. A próxima seção detalhará o modelo de memória que será utilizada para resolver este problema, o *stack frame* (registro de ativação), que permitirá mapear os símbolos abstratos da RI para endereços de memória na pilha de execução.

6.2 O Modelo de Memória e Execução

Antes de traduzir a primeira instrução da RI, é fundamental entender o ambiente de execução de um programa em *Assembly*. Diferente do C, não existem *variáveis* automáticas; o que existe é apenas memória e esta deve ser gerenciada manualmente por meio da pilha de execução (*stack*). A tradução da RI para o *Assembly* é, em essência, um processo de mapear os conceitos abstratos de *variáveis*, *parâmetros* e *temporários* para endereços de memória e registradores do processador.

6.2.1 Registradores: A Memória Rápida da CPU

Um programa em *Assembly* não opera diretamente em variáveis na memória. A CPU utiliza um pequeno conjunto de áreas de armazenamento de altíssima velocidade, chamados **registradores**, para realizar operações. Para traduzir a RI, é necessário definir um subconjunto desses registradores e atribuir a eles papéis específicos.

Para isso, o *Micro C* utilizará os seguintes registradores (em sintaxe AT&T):

- `%rbp` (*Base Pointer*): é o registrador mais importante. Ele atua como o *ponteiro base* do *stack frame*. Durante a execução de uma função, seu valor é estável, servindo como uma âncora para acessar todas as variáveis locais e parâmetros.
- `%rsp` (*Stack Pointer*): é o *ponteiro da pilha*. Ele aponta para o topo atual da pilha e seu valor muda dinamicamente toda vez que algo é *empurrado* (`push`) ou *retirado* (`pop`) da pilha.
- `%rax` (*Accumulator*): é o registrador *acumulador*. É usado como o principal registrador de *rascunho* para cálculos aritméticos. Mais importante, pela convenção de chamada, é o registrador padrão para armazenar o **valor de retorno** de uma função. (Será utilizada sua porção de 32 bits, `%eax`, para os inteiros do *Micro C*).
- `%rbx`, `%rcx`, `%rdx`: são registradores de propósito geral que serão usados como *rascunhos* secundários, principalmente para conter o segundo operando em operações binárias (como `arg2` em `t0 := arg1 + arg2`).
- `%rdi`, `%rsi`: são registradores para chamadas de funções externas. A Convenção de Chamada System V ABI [12] (que o `gcc` utiliza) determina que o primeiro argumento de uma função é passado em `%rdi` e o segundo em `%rsi`.

Isso será utilizado extensivamente na função `print` exclusiva do *Micro C* (sua implementação é diferente do `printf` padrão da linguagem C).

6.2.2 O Registro de Ativação (Stack Frame)

Quando uma função é chamada, ela não opera em um vácuo; ela aloca uma área de trabalho temporária na pilha de execução. Essa área é conhecida como **Registro de Ativação** ou **Stack Frame**. Este *frame* contém todo o contexto necessário para a função executar: seus parâmetros, suas variáveis locais e espaço para valores temporários.

No x86-64, o acesso às variáveis locais e parâmetros é feito de forma *relativa* ao ponteiro base `%rbp`. A Figura 6.1 ilustra a anatomia de um *stack frame* típico, que serve de base para o cálculo de endereços na pilha.

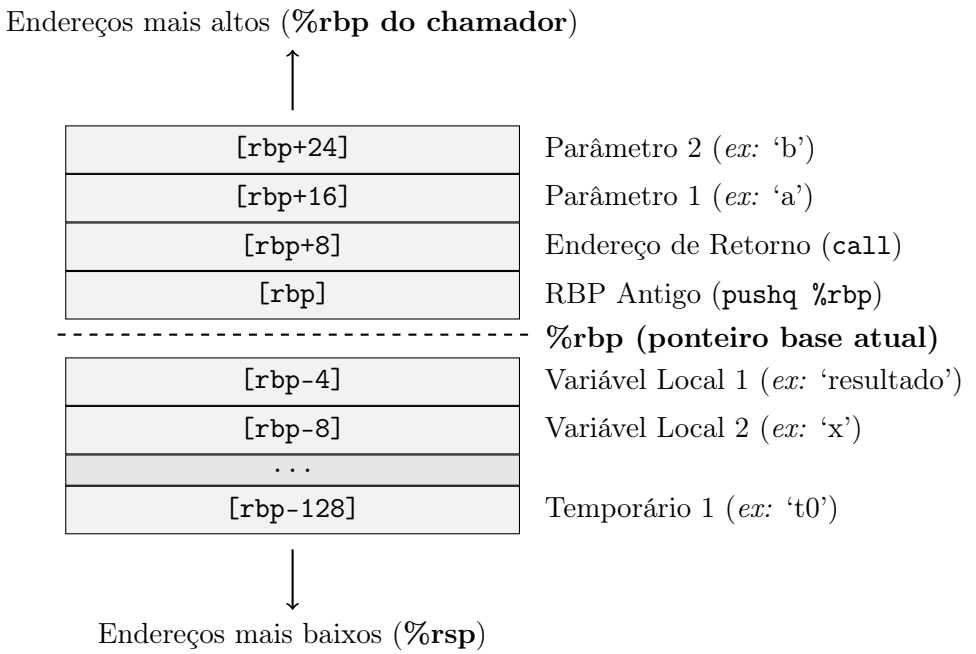


Figura 6.1: Estrutura do Registro de Ativação (Stack Frame) no x86-64.

Este diagrama é a conexão direta com o analisador semântico do *Micro C*. A função `calcular_offsets` (`semantic.c`) foi projetada especificamente para preencher a Tabela de Símbolos com os valores exatos deste mapa:

- **Offsets Positivos (Parâmetros):** endereços *acima* do `%rbp` pertencem ao escopo da função que os chamou. Por convenção, o primeiro parâmetro (ex: `a`) é alocado em `16(%rbp)`, o segundo (ex: `b`) em `24(%rbp)` e assim por diante. A *Passagem 1* do `calcular_offsets` calcula, exatamente, esses valores.
- **Offsets Negativos (Locais):** endereços *abaixo* do `%rbp` são o espaço de trabalho da função *atual*. É aqui que as variáveis locais (ex: `x`, `y`, `z`) são ar-

mazenadas. A *Passagem 2* da função `calcular_offsets` calcula esses valores (ex: `-4(%rbp)`, `-8(%rbp)`).

- **Temporários:** os temporários da *IR* (ex: `t0`, `t1`) também precisam de espaço. Para evitar conflitos com as variáveis locais, a função do gerador de *Assembly* do *Micro C* (`assembly.c`) os aloca em uma área negativa separada, começando em `-128(%rbp)` e alocando os temporários subsequentes em endereços de memória cada vez menores.Dessa forma, enquanto `t0` fica associado ao endereço `-128(%rbp)`, `t1` passa a ocupar `-132(%rbp)`, `t2 -136(%rbp)`, e a sequência continua seguindo esse padrão.

6.2.3 Prólogo e Epílogo: Gerenciamento do Frame

O (*stack frame*) descrito anteriormente não é construído automaticamente. Sua criação é um processo explícito, que exige a construção no início da execução de cada função e a sua subsequente destruição no fim. Essas duas sequências de instruções são denominadas, respectivamente, prólogo e epílogo da função. No gerador de *Assembly* do *Micro C* foram criadas duas funções auxiliáres para este fim: `asm_gen_prologue` e `asm_gen_epilogue`.

Prólogo da Função O prólogo é gerado toda vez que encontrado um `IR_LABEL` que é uma função (ex: `main:` ou `soma:`).

```
// Código gerado pela função asm_gen_prologue()
    pushq %rbp
    movq %rsp, %rbp
    subq $256, %rsp
```

A lógica, que é executada em ordem, é a seguinte:

1. `pushq %rbp`: salva o ponteiro base da função *anterior* (o chamador) na pilha. Isso é crucial para que seja possível restaurá-lo depois e retornar corretamente.
2. `movq %rsp, %rbp`: define o novo *ponto zero* do frame. O ponteiro base (‘%rbp‘) agora aponta para o topo da pilha (que contém o `%rbp` antigo). Todos os acessos (‘-4(%rbp)‘, ‘+16(%rbp)‘) serão relativos a este novo ponto.
3. `subq $256, %rsp`: *abre espaço* na pilha para as variáveis locais e temporários. Ao subtrair do ponteiro da pilha (`%rsp`), é possível movê-los para baixo, alocando 256 bytes de memória. No *Micro C* este é um tamanho fixo, mas compiladores otimizados calculariam o tamanho exato necessário.

Epílogo da Função O epílogo é gerado toda vez que uma instrução `IR_RETURN` é encontrada.

```
// Código gerado pela função asm_gen_epilogue()
    movq %rbp, %rsp
    popq %rbp
    ret
```

A lógica é inversa ao prólogo, destruindo o frame:

1. `movq %rbp, %rsp`: desaloca todo o espaço local (os 256 bytes) de uma só vez, movendo o ponteiro da pilha de volta para o ponteiro base.
2. `popq %rbp`: restaura o ponteiro base da função *anterior*, que foi salva no início. A pilha agora está exatamente como estava antes da função ser chamada.
3. `ret`: recupera o *Endereço de Retorno* (que agora está no topo da pilha) e redireciona a execução de volta para ele, saindo da função.

6.2.4 Desafios de Implementação

Embora o diagrama do *stack frame* pareça simples, implementar a lógica para o compilador foi um dos maiores desafios do projeto. O código *Assembly* gerado inicialmente estava logicamente incorreto, com todas as variáveis e parâmetros sendo mapeados para o mesmo endereço (`0(%rbp)`), o que causava a sobrescrita de dados.

O problema não estava no gerador de *Assembly*, mas na análise semântica, que falhava em popular a Tabela de Símbolos com os *offsets* corretos. A depuração revelou dois problemas (*bugs*).

Bug 1: Diferenciação de Parâmetros e Variáveis Locais. O primeiro problema era que a `struct Simbolo` não tinha como diferenciar um parâmetro (que precisa de um *offset* positivo) de uma variável local (que precisa de um *offset* negativo). A solução foi modificar o a definição da tabela de símbolos (em `symbol_table.h`) para adicionar uma flag:

```
// arquivo: src/symbol_table/symbol_table.h (trecho)
typedef struct {
    // ... (campos anteriores como nome, tipo, is_array)
    int memory_offset;
    int is_parameter; // <-- FLAG ADICIONADA
} Simbolo;
```

Em seguida, a função `adicionar_simbolo` (em `symbol_table.c`) foi atualizada para receber e armazenar essa flag:

```
// arquivo: src/symbol_table/symbol_table.c (trecho)
int adicionar_simbolo(..., int is_parameter) {
    // ... (código de alocação)
    s->is_array = is_array;
    s->array_size = array_size;
    s->is_parameter = is_parameter; // <-- Salva a flag
    s->memory_offset = 0;           // Inicializa o offset
    tabela->tamanho++;
    return 0;
}
```

Bug 2: Lógica de Percurso e Momento do Cálculo. O segundo bug, era que o analisador semântico (em `semantic.c`) estava invocando `calcular_offsets` no momento errado. A lógica de percurso da árvore (o *tree walker*) estava calculando os *offsets* de um escopo antes que todos os símbolos tivessem sido adicionados a ele, resultando em tabelas vazias e *offsets* zerados.

A solução foi reestruturar a função `analisar_no` para garantir uma ordem de execução correta (Pós-Ordem):

1. **Pré-Ordem (Entrada no Escopo):** ao encontrar um `NODE_FUNCTION_DEF` ou `NODE_BLOCK`, o analisador *apenas* empilha uma nova tabela (`empilhar_tabela (pilha)`) e marca que um `novo_escopo` foi criado.
2. **Descida Recursiva:** o analisador visita todos os nós filhos. É durante esta etapa que os nós `NODE_VAR_DECL` são processados e invocam `adicionar_simbolo`, populando a tabela do topo da pilha (que é o escopo atual).
3. **Pós-Ordem (Saída do Escopo):** somente após todos os filhos terem sido visitados (e a tabela de símbolos estar completa), a função verifica a flag `novo_escopo` e, então, invoca `calcular_offsets(pilha->tabelas[pilha->topo])`.

Finalmente, a própria função `calcular_offsets` (em `semantic.c`) foi reescrita para usar a flag `is_parameter`, implementando a lógica de duas passagens que o modelo de memória exige.

Dessa forma, a primeira passagem itera pela tabela de símbolos do escopo e calcula os *offsets* positivos para todos os símbolos marcados como parâmetros. Neste trecho (*Passagem 1*), o `param_offset` é inicializado em `16`, que corresponde ao primeiro endereço disponível acima do ponteiro base (`%rbp`), conforme o diagrama de *stack frame*. Cada parâmetro encontrado incrementa o *offset* em 8 bytes. A seguir, o trecho de código correspondente:

```
// arquivo: src/semantic/semantic.c (trecho: passagem 1)
static void calcular_offsets(TabelaSimbolos* tabela) {
    int param_offset = 16;
    int local_offset = 0;

    //passagem 1: calcular offsets de parâmetros
    for (int i = 0; i < tabela->tamanho; i++) {
        Simbolo* s = &tabela->simbolos[i];
        if (s->is_function) continue;

        if (s->is_parameter) {
            s->memory_offset = param_offset;
            param_offset += 8;
        }
    }
//...
```

Em seguida, a segunda passagem da mesma função itera novamente pela tabela para calcular os *offsets* negativos para todas as variáveis locais. Nesta passagem (*Passagem 2*), o `local_offset` começa em `0` e é decrementado pelo tamanho de cada símbolo (4 bytes para `int` ou o tamanho total para vetores). Isso aloca espaço na pilha *para baixo* a partir do `%rbp`, como `-4`, `-8`, etc. Trecho do código relativo a segunda passagem da função:

```
//... (continuação de calcular_offsets)
    //passagem 2: calcular offsets de variáveis locais
    for (int i = 0; i < tabela->tamanho; i++) {
        Simbolo* s = &tabela->simbolos[i];
        if (s->is_function) continue;

        if (!s->is_parameter) {
            int tamanho_simbolo = get_symbol_size(s);
            local_offset -= tamanho_simbolo;
            s->memory_offset = local_offset;
        }
    }
}
```

Com essas correções, a Tabela de Símbolos foi transformada em um *mapa de memória* preciso, permitindo ao gerador de *Assembly* (Seção 6.3) consultar os valores em `memory_offset` de qualquer símbolo e traduzi-lo para o endereço correto na pilha, como `16(%rbp)` ou `-4(%rbp)`.

Com o modelo de registradores, a estrutura do *stack frame* e o gerenciamento desse frame (por meio do prólogo e epílogo) agora é o compilador está pronto para gerar o código *Assembly*. A preparação está encerrada e agora existe um *mapa de memória* preciso da Tabela de Símbolos. Esse mapa armazena o endereço exato de cada símbolo (como `16(%rbp)` ou `-4(%rbp)`) e de cada variável temporária (como `-128(%rbp)`).

A próxima seção irá detalhar a tradução na prática. Será analisado como o gerador de *Assembly* do *Micro C* (`assembly.c`) percorre a lista de RI e traduz cada `IROpcode` (como `IR_ASSIGN`, `IR_ADD` e `IR_LOAD`) na sequência de instruções *Assembly x86-64* correspondente.

6.3 Tradução da RI para *Assembly*

Com o modelo de memória e a estrutura do *stack frame* definidos, podemos agora analisar a implementação do módulo que gerador do *Assembly* (`assembly.c`). O processo de tradução é um percurso linear pela lista de RI, onde a função principal `gerar_assembly` utiliza um `switch` para traduzir cada `IROpcode` em sua sequência de *Assembly x86-64 AT&T* correspondente.

6.3.1 O Tradutor de Operandos

A primeira peça-chave da implementação é a função auxiliar `get_operand_asm`. Esta função atua como um *dicionário* que traduz a estrutura de dados abstrata `IROperand` (da RI) para a sintaxe de texto que o montador (`gcc`) espera. Ela é implementada como um `switch` que analisa o `op->type` e formata uma string. Os três casos de uso principais são descritos a seguir.

Valores Imediatos (Constantes) Para constantes numéricas, o *Assembly* AT&T exige um prefixo `$`.

```
// IR: OPERAND_CONST, data.const_val = 10
case OPERAND_CONST:
    sprintf(asm_buffer, "$%d", op->data.const_val);
    break;
// Saída: "$10"
```

Símbolos (Variáveis e Parâmetros) Este é o ponto de conexão crucial com o analisador semântico. A função acessa o `memory_offset` (calculado na Fase 3) e o formata na sintaxe de endereçamento relativo à base da pilha.

```
// IR: OPERAND_SYMBOL, data.symbol->memory_offset = -4 (para 'x')
// IR: OPERAND_SYMBOL, data.symbol->memory_offset = 16 (para 'a')
case OPERAND_SYMBOL:
    sprintf(asm_buffer, "%d(%%rbp)", op>data.symbol->
    memory_offset);
    break;
// Saída para 'x': "-4(%rbp)"
// Saída para 'a': "16(%rbp)"
```

Temporários Como os temporários (ex: `t0`, `t1`) não existem na Tabela de Símbolos, são atribuímos a eles um espaço de memória próprio. O `case` de cada um deles

calcula um offset negativo a partir de um endereço base (no caso, -128), garantindo que eles não colidam com as variáveis locais.

```
// IR: OPERAND_TEMP, data.temp_id = 0 (para 't0')
case OPERAND_TEMP: {
    int temp_offset = -128 - (op->data.temp_id * 4);
    sprintf(asm_buffer, "%d(%%rbp)", temp_offset);
    break;
}
// Saída para 't0': "-128(%rbp)"
// Saída para 't1': "-132(%rbp)"
```

6.3.2 A Estratégia de Tradução: *Load-Operate-Store*

A RI implementada no compilador do *Micro C* utiliza o mecanismo de *Código de Três Endereços* (CTE). Isso significa que uma instrução típica tem três operandos como, por exemplo, `t0 := a + b`. Por outro lado, a maioria das instruções aritméticas do x86-64 (como `addl`) opera em um formato de *dois endereços*, onde o destino é também um dos operandos fonte (`destino = destino + fonte`). Por isso, é utilizada a estratégia de tradução **Load-Operate-Store** (Carregar-Operar-Armazenar), que utiliza registradores de rascunho (como `%eax` e `%ebx`) para dividir a operação. A seguir a descrição de cada uma dessas operações:

1. **Load (Carregar):** carrega os operandos fonte da pilha de memória (ex: `-4(%rbp)`) para registradores de rascunho (ex: `%eax`, `%ebx`).
2. **Operate (Operar):** executa a operação (ex: `addl %ebx, %eax`).
3. **Store (Armazenar):** move o resultado do registrador de rascunho (ex: `%eax`) de volta para o destino na pilha (ex: `-128(%rbp)`).

Tradução de Operações Aritméticas e Atribuição

Os `case`s para `IR_ASSIGN` e `IR_ADD` são exemplos desta estratégia.

IR_ASSIGN A atribuição (ex: `x := t0`) é um **Load-Store** de duas etapas:

```
// IR: x := t0
// Mapeamento: x -> -4(%rbp), t0 -> -128(%rbp)

// 1. Load: Carrega t0 (-128(%rbp)) para o registrador %eax
    movl -128(%rbp), %eax
// 2. Store: Armazena %eax na variável x (-4(%rbp))
    movl %eax, -4(%rbp)
```

IR_ADD A adição (ex: `t0 := a + b`) é o exemplo completo de **Load-Load-Operate-Store**:

```
// IR: t0 := a + b
// Mapeamento: t0 -> -128(%rbp), a -> 16(%rbp), b -> 24(%rbp)

// 1. Load: Carrega 'a' (16(%rbp)) em %eax
    movl 16(%rbp), %eax
// 2. Load: Carrega 'b' (24(%rbp)) em %ebx
    movl 24(%rbp), %ebx
// 3. Operate: Adiciona %ebx em %eax (resultado fica em %eax)
    addl %ebx, %eax
// 4. Store: Armazena %eax no temporário t0 (-128(%rbp))
    movl %eax, -128(%rbp)
```

Esta mesma estratégia é usada para todas as operações binárias, como `IR_SUB`, `IR_MUL`, `IR_LT` e `IR_EQ`.

Tradução de Controle de Fluxo

Estruturas de alto nível, como o `if`, foram *compactadas* pelo gerador de RI em um conjunto de rótulos e desvios. O gerador de *Assembly* simplesmente traduz essas instruções de controle de baixo nível.

IR_LABEL A tradução de um rótulo (ex: `L0:` ou `main:`), por outro lado, precisa de uma lógica especial para diferenciar um rótulo de função (que precisa de um prólogo) de um rótulo de desvio.

```
// IR: main:
case IR_LABEL: {
    // 1. Busca o label na Tabela de Símbolos
    Simbolo* s = buscar_simbolo_em_todos_escopos(pilha, ...);

    // 2. Verifica se é uma função
    if (s != NULL && s->is_function) {
        // Se for, gera o prólogo completo
        asm_print(".globl %s", s->nome);
        asm_print("%s:", s->nome);
        asm_gen_prologue();
    } else {
        // Se for um label comum (ex: L0), apenas imprime o rótulo
        asm_print("%s:", get_operand_asm(instr->result));
    }
    break;
}
```

IR_GOTO e IR_IF_FALSE Estas instruções têm traduções diretas para instruções de salto do x86-64.

```
// IR: goto L1
case IR_GOTO:
    asm_instr("jmp L1");
    break;

// IR: if_false t1 goto L0
// Mapeamento: t1 -> -132(%rbp)
case IR_IF_FALSE:
    // 1. Compara o booleano 't1' com 0 (falso)
    asm_instr("cmpl $0, -132(%rbp)");
    // 2. "Jump if Equal" (Pula se for igual a 0, ou seja, falso)
    asm_instr("je L0");
    break;
```

Tradução de Acesso a Arrays (LOAD e STORE)

A tradução de acesso a vetores é a operação mais complexa, pois requer o cálculo de um endereço de memória em tempo de execução. Ela é resolvida usando o modo de **endereçamento indexado** do x86-64. A fórmula para encontrar um elemento `v[i]` é: `Endereço = EndereçoBase(v) + (Indice(i) * TamanhoDoElemento)`. Na sintaxe da AT&T, isso é escrito como: `offset(%base, %indice, escala)`.

IR_LOAD Para a leitura de `t0 := v[i]`:

```
// IR: t0 := v[i]
// Mapeamento: v -> -40(%rbp), i -> -8(%rbp), t0 -> -128(%rbp)

// 1. Carrega o Endereço Base de 'v' em %rax
    asm_instr("leaq -40(%%rbp), %%rax");
// 2. Carrega o Índice 'i' em %rbx
    asm_instr("movl -8(%%rbp), %%ebx");
// 3. Carrega o Valor [base + indice*4] em %ecx
    asm_instr("movl (%%rax, %%rbx, 4), %%ecx");
// 4. Salva o Valor (em %ecx) no destino 't0'
    asm_instr("movl %%ecx, -128(%%rbp)");
```

Neste trecho de código é possível observar que a instrução `leaq` (Load Effective Address), carrega o *endereço* de `v` (ou seja, `-40(%rbp)`), e não o valor contido nele. A instrução `movl (%rax, %rbx, 4), %ecx` implementa a fórmula de acesso, usando 4 como a escala (isso acontece porque no *Micro C* `int`'s e `char`'s ocupam 4 bytes).

IR_STORE Para a escrita `v[i] := x`, o processo é o inverso, com o endereço indexado no lado do *destino*:

```
// IR: v[i] := x
// Mapeamento: v -> -40(%rbp), i -> -8(%rbp), x -> -4(%rbp)

// 1. Carrega o Endereço Base 'v' em %rax
    asm_instr("leaq -40(%%rbp), %%rax");
// 2. Carrega o Índice 'i' em %rbx
    asm_instr("movl -8(%%rbp), %%ebx");
// 3. Carrega o Valor 'x' em %ecx
    asm_instr("movl -4(%%rbp), %%ecx");
// 4. Salva o Valor (em %ecx) no Endereço [base + indice*4]
    asm_instr("movl %%ecx, (%%rax, %%rbx, 4)");
```

Com a implementação da estratégia *Load-Operate-Store* e a tradução dos opcodes de atribuição, operações aritméticas, controle de fluxo e acesso a vetores, o gerador de *Assembly* está completo. Agora é possível traduzir o corpo de qualquer função do *Micro C* em um conjunto de instruções x86-64.

No entanto, um programa funcional é mais do que uma coleção de funções isoladas; elas precisam se comunicar. Até agora, não foram implementadas as instruções que permitem essa comunicação: `IR_PARAM`, `IR_CALL` e `IR_RETURN`. Mais importante, ainda não foi tratada a complexidade relacionada a invocação de funções *externas*, como o `printf`, que não fazem parte do projeto de implementação do compilador do *Micro C*.

A próxima seção abordará este desafio final. Serão detalhadas as regras e protocolos, conhecidas como **Convenções de Chamada (ABIs)**, que ditam como os parâmetros são passados, como os valores são retornados e como o código se conecta com bibliotecas externas para criar um executável.

6.4 Convenções de Chamada e *Linking*

A geração de código não termina ao emitir as instruções de uma única função. Um programa funcional é composto por múltiplas funções que precisam interagir entre si e, o mais importante, com o sistema operacional e suas bibliotecas. Essas interações são realizadas por um conjunto estrito de regras conhecidas como **Convenção de Chamada** (ou ABI, *Application Binary Interface*) [12].

A convenção dita como os parâmetros são passados para as funções, como os valores são retornados e quem é responsável pela limpeza da pilha. No compilador do *Micro C*, são utilizadas duas convenções: uma convenção interna simplificada para as funções do próprio *Micro C*, e a convenção externa formal exigida por funções da biblioteca C.

6.4.1 Convenção Interna: Chamando Funções do *Micro C*

Para a invocação de funções que existem dentro do próprio código-fonte (como a `main` ou outras funções declaradas internamente), é utilizada uma convenção de

interna simplificada. Esta convenção não utiliza os registradores para a passagem de argumentos, optando por passar todos os parâmetros, exclusivamente, usando a pilha. Essa é uma abordagem mais simples de implementar e depurar. O processo de invocação é dividido em três etapas que são mapeadas a partir das instruções da RI:

IR_PARAM (Empilhando Argumentos) Para cada argumento de uma função, o `intercode.c` emite uma instrução `IR_PARAM`. O gerador de *Assembly* traduz isso em código que move o valor do argumento para um registrador de rascunho e, em seguida, *empurra* esse valor para a pilha.

```
// IR: param x
// Mapeamento: x -> -4(%rbp)

    // 1. Carrega o valor do argumento 'x' em %eax
    movl -4(%rbp), %eax
    // 2. Empurra o valor (como 64 bits) na pilha
    pushq %rax
```

Este processo é repetido para cada parâmetro, construindo-os na pilha para a próxima função consumir.

IR_CALL (Executando a Chamada) A instrução `IR_CALL` é o ponto central. Ela realiza a chamada, limpa a pilha e armazena o resultado.

```
// IR: t2 := call soma, 2
// Mapeamento: t2 -> -136(%rbp)

// 1. Pula para o label da função 'soma'
    call soma
// 2. Limpa os 2 argumentos (2 * 8 bytes = 16) da pilha
    addq $16, %rsp
// 3. Pega o valor de retorno (em %eax) e salva em 't2'
    movl %eax, -136(%rbp)
```

Quando a instrução `call` é executada, o processador automaticamente empilha o endereço de retorno. Em seguida, a função invocada (neste exemplo, `soma`) executa seu prólogo. O código *Assembly* padrão para este prólogo (Subseção 6.2.3) estabelece seu próprio *stack frame*. Dentro de `soma`, os parâmetros `a` e `b` são acessados usando os offsets positivos previamente calculados (ex: `16(%rbp)` e `24(%rbp)`).

Após o retorno da função, a convenção determina que a função que fez a invocação (`main`) é responsável por limpar a pilha. A instrução `addq $16, %rsp` realiza essa ação de forma eficiente, simplesmente movendo o ponteiro da pilha para cima, *descartando* os 16 bytes que tinham sido alocados para os dois parâmetros.

6.4.2 O Desafio do print

A convenção interna criada para o *Micro C* funciona corretamente para funções controladas pelo compilador, ou seja, de escopo local ao programa. No entanto, ela não serve para cenários que necessitam de interação com códigos externos ao programa ou mesmo ao compilador do *Micro C*. A função `print`, que é uma implementação exclusiva para o *Micro C*, é uma imitação limitada da função `printf` da biblioteca C padrão. O objetivo dessa função é fornecer ao programador a capacidade de visualizar saídas dos seus programas e conferir se eles estão corretos, por exemplo.

O `printf` é uma função externa, pré-compilada, que não foi escrita para o compilador do *Micro C*. Ela não tem como saber sobre a convenção estrita de repassar os parâmetros *todos pela pilha*. Em vez disso, a função espera que os argumentos sejam enviados seguindo a convenção de chamadas do sistema operacional.

O Problema dos Argumentos (Convenção System V ABI)

Como o ambiente de desenvolvimento utiliza o `gcc` em um ambiente Unix-*like*, a função `printf` espera que seus argumentos sigam a convenção **System V AMD64 ABI** [12]. Esta convenção determina que os primeiros argumentos inteiros ou ponteiros são passados usando registradores específicos, não da pilha. Para o `printf`, em especial, os dois primeiros argumentos são:

- **Argumento 1 (Formato):** um ponteiro para a string de formato (ex: `"%d\n"`). Deve ser colocado no registrador `%rdi`.
- **Argumento 2 (Valor):** o valor a ser impresso (ex: o valor de `z`). Deve ser colocado no registrador `%rsi`.

Para resolver isso, na implementação do compilador (`intercode.c`), primeiro é detectado o tipo de argumento (usando `get_expr_static_type`) que vai gerar `opcodes` (códigos de operação) especializados:

- `IR_PRINT_INT`
- `IR_PRINT_CHAR`
- `IR_PRINT_STRING`

Em seguida, o compilador (`assembly.c`) traduz esses opcodes usando a convenção correta. A tradução para `IR_PRINT_INT`, detalhada abaixo, serve como o principal exemplo de como a ABI System V é implementada. O processo exige o carregamento dos argumentos nos registradores corretos (`%rdi` e `%rsi`), o zeramento do `%eax` (um requisito para funções com número variável de argumentos como `printf`), e o alinhamento da pilha antes da chamada:

```
// IR: print_int z
// Mapeamento: z -> -12(%rbp)

// 1. Carrega o ponteiro da string de formato "%d\n" em %rdi
    leaq .L.str.int(%rip), %rdi
// 2. Carrega o valor de 'z' em %esi (os 32 bits
inferiores de %rsi)
    movl -12(%rbp), %esi
// 3. Define %eax como 0 (necessário para funções
variádicas como printf)
    xorl %eax, %eax
// 4. Alinha a pilha (requisito da ABI) e chama
    subq $8, %rsp
    call printf
    addq $8, %rsp
```

Como é possível observar no código, a tradução para uma chamada externa é, significativamente, mais complexa do que uma para chamada interna.

O Problema dos Dados (A Seção `.rodata`)

A convenção ABI resolve como *passar* os argumentos, mas acaba desencadeando em um outro problema: o primeiro argumento, `%rdi`, deve conter o *endereço* da string de formato (ex: `"%d\n"`). Essa string precisa estar fisicamente armazenada na memória do programa para que o ponteiro possa referência-la.

A solução foi criar uma seção de dados no topo do arquivo *Assembly* gerado pelo compilador com uma chamada `.section .rodata` (*Read-Only Data*). Para preencher esta seção, o compilador (`assembly.c`) realiza uma **pré-varredura** na lista de RI *antes* de traduzir o código.

```
// Código gerado no topo do arquivo teste.s

.section .rodata
.L.str.int: .string "%d\n"
.L.str.char: .string "%c\n"
.L.str.str: .string "%s\n"

// ... código da pré-varredura para strings literais ...
// Ex: L0: .string "Meu Compilador"

.text
.globl main
//... (resto do código)
```

Esta pré-varredura procura por instruções `IR_PRINT_STRING` e define seus labels (ex: `L0`) com o conteúdo da string. A instrução `leaq .L.str.int(%rip), %rdi` então usa o *RIP-relative addressing* (endereçamento relativo ao ponteiro de instrução) para carregar o endereço daquela string de formato no registrador correto,

completando os requisitos para a chamada a função `printf`.

Com a implementação das convenções de chamada, tanto as internas (para funções como `soma`) quanto as externas (para a função `printf` da biblioteca C), o gerador de *Assembly* está agora completo. O resultado da compilação gera um arquivo com extensão `.s` que não é apenas uma tradução literal da RI, mas um arquivo de *Assembly* válido que obedece às regras da arquitetura x86-64 e do Sistema Operacional.

No entanto, este arquivo `.s` ainda é apenas texto; ele não é um programa executável. A etapa final, e a conclusão de todo o processo é a **montagem** e ***linking***. No projeto do compilador do *Micro C* foram disponibilizados um `Makefile` que pode ser utilizado para facilitar a compilação de novos códigos fonte (extensão `.mcc`), embora as mesmas etapas possam ser realizadas manualmente utilizando o próprio binário do compilador (`mcc`). O Apêndice D mostra um exemplo de compilação gerando cada uma das fases de compilação do código fonte exemplo até a geração do seu, respectivo, código binário.

6.5 Sumário

Este capítulo encerra a construção do compilador para a linguagem *Micro C* ao implementar a fase final do *back-end*, o **gerador de código *Assembly***. Esta etapa foi responsável por traduzir a Representação Intermediária (RI) descrita no Capítulo 5, que é abstrata e independente de plataforma, em código *Assembly*.

Inicialmente foi definida a arquitetura **x86-64** e a **Sintaxe AT&T** que seria utilizada para gerar código em *Assembly*. A opção foi pelo formato esperado pelo `GCC`, que é a ferramenta utilizada para a montagem e *linking*(conversão do *Assembly* gerado pelo compilador do *Micro C* para binário).

Em seguida, foram explorados os modelos de memória necessários para montar o código em *Assembly*, explicando o papel de registradores (como `%rbp`, `%rsp` e `%rax`) e a estrutura do **Registro de Ativação (Stack Frame)**. Além disso, foram explicados como o prólogo e o epílogo ('pushq %rbp' / 'popq %rbp') criam e destroem esse *frame*, e como ajustes na fase semântica do compilador do *Micro C* foram necessárias para corrigir o mapeamento das variáveis locais para *offsets* negativos (ex: `-4(%rbp)`) e parâmetros para *offsets* positivos (ex: `16(%rbp)`)

Com as estruturas devidamente organizadas, foram detalhados como o gerador de *Assembly* (`assembly.c`) do *Micro C* usa a função `get_operand_asm`, que atua como o *dicionário* tradutor de operandos, e a estratégia *Load-Operate-Store* usada para converter instruções de três endereços (como `IR_ADD`) em sequências de *Assembly* (como `movl`, `movl`, `addl`, `movl`). Além disso, foram analisadas a tradução de instruções-chave, desde `IR_ASSIGN` e `IR_IF_FALSE` até o complexo endereçamento indexado de `IR_LOAD` e `IR_STORE`.

Por fim, foram detalhados os desafios para realizar a interação com código externo. Por exemplo, foram detalhas as **Convenções de Chamada**, fazendo um contraste entre a convenção de pilha interna definida para o *Micro C* (baseada em pilha, para a função `soma`) com a convenção externa **System V ABI**, que foi necessária para implementar e invocar a função `printf` sem a necessidade de bibliotecas externas. Isso incluiu o uso de registradores (`%rdi` e `%rsi`) para argumentos e a definição de dados na seção `.rodata` do código *Assembly*.

Capítulo 7

Conclusão

"It always seems impossible until it's done."

– Nelson Mandela

Este trabalho teve como objetivo o projeto, a implementação e a validação de um compilador para a linguagem *Micro C*, que é um subconjunto minimalista da linguagem de programação C. Ao longo dos capítulos, detalhou-se cada fase do complexo *pipeline* de compilação de um programa, do seu código fonte até geração de código binário.

7.1 Sumário do Compilador do *Micro C*

O desenvolvimento do compilador da linguagem *Micro C* foi extenso e complexo. Cada capítulo abordou uma fase do processo de compilação e está associada diretamente ao software que foi desenvolvido, o compilador da linguagem *Micro C*. A seguir serão sintetizadas as contribuições de cada uma das etapas de construção do compilador, por fase de compilação.

7.1.1 O Front-End: Análise e Compreensão

O *front-end* do compilador é responsável pela fase de **análise**. A implementação inicia com a Análise Léxica (Capítulo 2), onde o `scanner.c` converte o arquivo-fonte do *Micro C* em uma sequência linear de *tokens*. Em seguida, a Análise Sintática (Capítulo 3) consome esses tokens, valida a gramática da linguagem e constrói a Árvore Sintática Abstrata (ASA), a representação hierárquica do programa. Por fim, a Análise Semântica (Capítulo 4) percorre a ASA para realizar a verificação de tipos e escopos, utilizando a Tabela de Símbolos para armazenar o contexto (como tipos de variáveis) e informações para o *back-end*, como os *offsets* de memória.

7.1.2 O *Back-End*: Síntese e Geração

O *back-end* é o responsável pela fase de **síntese** da compilação (Capítulo 5 e Capítulo 6). Esta fase inicia com a geração de código intermediário (`intercode.c`), que percorre a ASA e a traduz (ou *compacta*) para uma Representação Intermediária (RI) linear: o Código de Três Endereços (CTE). Esta RI é independente da plataforma. Na etapa final, o compilador (`assembly.c`) traduz a RI instrução por instrução para código em *Assembly x86-64* (sintaxe AT&T), resolvendo o acesso à memória (o *stack frame*) e implementando as convenções de chamada (ABI) necessárias para interagir com chamadas de funções externas, como o `printf`. O arquivo `.s` resultante pode ser montado e *linkado* pelo `gcc` para produzir um código binário executável.

7.2 Resultados Obtidos e Validação

A implementação teórica de um compilador não é suficiente sem uma validação prática. Esta seção detalha os resultados obtidos ao submeter o *Compilador Micro C* a um conjunto diversificado de algoritmos de teste, tanto de sucesso quanto de falha. O sucesso na compilação e a subsequente execução correta dos testes provam que o código *Assembly* gerado (Capítulo 6), é funcional e que toda a *pipeline* de síntese gera, corretamente um programa binário válido.

O conjunto de testes de sucesso foi dividido em três categorias principais, cada uma validando um subconjunto de funcionalidades do compilador:

- **Testes de Funcionalidade Básica:** verificação das operações aritméticas fundamentais (`+`, `-`, `*`, `/` e `%`), todos os operadores relacionais (como `<`, `==`, `!=`, `>=`), a estrutura condicional `if-else`, e a capacidade de invocar a função `print` com os três tipos de dados suportados (`int`, `char` e `string`).
- **Testes de Controle e Memória:** que realizam operações de laço complexas e acesso a dados. Isso incluiu a implementação de algoritmos como a **Peneira de Eratóstenes** (para encontrar números primos) e o **Bubble Sort** (para ordenação). Esses testes validaram intensivamente os laços `for` aninhados e o correto acesso a arrays (operações `IR_LOAD` e `IR_STORE`).
- **Testes do *Stack Frame*:** validação do gerenciamento da pilha em chamadas recursivas. Isso incluiu a **Recursão Simples** (Fatorial) e a **Recursão Múltipla** (Máximo Divisor Comum e **Fibonacci**). O teste de Fibonacci foi importante, por exemplo, por validar a capacidade do compilador de lidar com instruções `return` que contêm expressões complexas (ex: `return fibonacci(n - 1) + fibonacci(n - 2);`). Isso mostra que o gerador de RI trata corretamente múltiplas chamadas `IR_CALL` aninhadas dentro de uma única expressão aritmética.

Além da validação de sucesso, o compilador também foi testado quanto à sua capacidade detectar erros. Testes de falha foram criados para garantir que cada fase

do *front-end* capturasse corretamente códigos inválidos:

- **Erros Léxicos:** caracteres inválidos (ex: `@`) são corretamente capturados pelo `scanner.c`.
- **Erros Sintáticos:** Gramática incorreta (ex: falta de `;`) é corretamente capturada pelo `parser.c`.
- **Erros Semânticos:** Uso de variáveis não declaradas ou incompatibilidade de tipos são corretamente capturados pelo `semantic.c`.

A capacidade do compilador em aceitar programas complexos e válidos, produzindo a saída correta, ao mesmo tempo em que rejeita programas inválidos com mensagens de erro claras, demonstra a integridade e a funcionalidade de todas as fases implementadas.

7.3 Desafios de Implementação e Soluções

O desenvolvimento do compilador da linguagem *Micro C* não foi um processo linear. A transição da teoria para uma implementação revelou diversos desafios. Esta seção detalha os principais desafios de engenharia encontrados tanto no *front-end* (análise), quanto no *back-end* (síntese) e as soluções aplicadas.

7.3.1 Desafios do *Front-End*: Ambiguidade e Estrutura

No *front-end*, os desafios centraram-se em resolver ambiguidades no código-fonte e em traduzir regras gramaticais recursivas para um parser funcional.

Desambiguação Léxica (*Scanner*): o `scanner.c` enfrentou o desafio de diferenciar operadores de comentários, que compartilham o mesmo prefixo (o caractere ‘/‘). Uma tradução simples falharia em distinguir a expressão `a / b` de um comentário `//comentário`. A solução foi implementar uma lógica de *lookahead* (olhar à frente), em que o *scanner observa* o próximo caractere ao `/` (usando `prox_char()`) e, caso não seja um comentário, devolve o caractere ao fluxo de entrada (usando `ungetc()`) para ser processado corretamente como um operador.

Precedência e Recursão à Esquerda (Parser): o desafio do `parser.c` foi implementar a precedência de operadores (ex: `*` antes de `+`) sem usar uma gramática com recursão à esquerda (ex: $E \rightarrow E + T$), que causaria um laço infinito em um parser recursivo descendente. A solução foi implementar a precedência por meio da própria estrutura das chamadas de função: a gramática foi dividida em `expression`, `term` e `factor`, onde `expression` (nível mais baixo de precedência) deriva em `term`, que por sua vez deriva `factor` (nível mais alto), fazendo com que a árvore a fosse construída na ordem correta.

Ambiguidade Sintática (Parser): o `parser.c` falhou inicialmente ao tentar analisar chamadas de função usadas como uma instrução independente. O `case ID:` na função `statement` assumia que todo `ID` (como `minha_funcao`) era o início de uma atribuição (ex: `x = ...`) e gerava um erro ao encontrar o token `LPAREN` (representando o caractere `'('`). A solução foi implementar uma função `peek` para `observar` o token seguinte. O `case ID:` agora verifica: se o próximo token for `LPAREN`, ele chama a rotina de análise de expressão (que sabe processar uma chamada de função); caso contrário, ele chama a rotina de análise de atribuição.

7.3.2 Desafios do Back-End: Memória e Convenções

No *back-end*, os desafios foram, principalmente, no gerenciamento de memória e na adesão às convenções de baixo nível da arquitetura.

Gerenciamento de Memória na RI (O *Bug* do `double free`): O primeiro *bug* crítico ocorreu na Fase 4 (Capítulo 5). A depuração revelou que a reutilização de ponteiros de `IROperand` (ex: `t0` ou `L1`) em múltiplas instruções causava um `double free` em `liberar_ir`. A solução foi refatorar a RI, adicionando a flag `owns_label` e a função `copiar_operando`, garantindo que cada instrução possuía cópias únicas de seus operandos.

Cálculo de *Offsets* de Pilha (O *Bug* do `0(%rbp)`): Após corrigir o *bug* anterior, o *Assembly* gerado ainda era inválido, pois acabava mapeando todas as variáveis para `0(%rbp)`. O problema estava no `semantic.c`: a função `analisar_no` chamava `calcular_offsets` em Pré-Ordem (antes de os símbolos serem adicionados à tabela). A solução foi reestruturar `analisar_no` para uma lógica de Pós-Ordem, garantindo que os *offsets* fossem calculados apenas *após* a tabela do escopo estar completa.

Convenções de Chamada (Os *Bugs* do `printf` e da Recursão): A implementação de chamadas de função (`IR_CALL`) revelou dois *bugs* de convenção de chamada:

1. **Convenção Externa:** a chamada a função `printf` falhava porque ele espera argumentos nos registradores (`%rdi`, `%rsi`) e dados na seção `.rodata`, seguindo a ABI *System V* [12]. O `assembly.c` foi corrigido para tratar `IR_PRINT` como um caso especial que segue esta convenção.
2. **Convenção Interna:** chamadas de função recursivas estavam causando um estouro de pilha. A depuração revelou que o `assembly.c` estava empilhando os parâmetros na ordem errada (da esquerda para a direita), em vez de seguir a convenção do C (da direita para a esquerda). Isso fazia com que a função recursiva fizesse a leitura de seus próprios argumentos de forma incorreta, levando a um laço infinito. A solução foi criar um *buffer* de parâmetros, permitindo ao `IR_CALL` empilhá-los na ordem correta (da direita para a esquerda).

7.4 Limitações e Trabalhos Futuros

Um projeto de compilador é, normalmente, limitado pela linguagem definida e pelos objetivos dos projetistas. A versão atual do compilador do *Micro C*, embora funcional e capaz de compilar algoritmos complexos, foi desenvolvida para fins didáticos. Dessa forma, várias funcionalidades não foram implementadas e o escopo da linguagem foi restrito ao máximo, para tornar o programa desenvolvido simples de aprender ou expandir.

7.4.1 Limitações Atuais

A seguir serão descritas algumas de suas limitações de implementação. Elas se concentram em três áreas:

- **Ausência de Suporte a Ponteiros:** esta é a limitação mais significativa da linguagem *Micro C*. A gramática, o analisador semântico e os geradores de código não reconhecem a sintaxe de ponteiros (como `*` ou `&`). A consequência mais direta disso é a incapacidade de passar vetores como parâmetros para funções. Isso impede a implementação de algoritmos que dependem dessa funcionalidade, como o *Quick Sort Recursivo*.
- **Geração de Código Não Otimizado:** o código *Assembly* gerado é simples e correto, mas não se preocupa com a performance. A estratégia *Load-Operate-Store*, que move valores constantemente entre a pilha e os registradores de rascunho, é lenta. Além disso, o prólogo de cada função aloca um espaço fixo de 256 bytes na pilha, independentemente de quantas variáveis a função realmente necessita, desperdiçando espaço.
- **Tipos de Dados Primitivos:** o compilador suporta apenas os tipos básicos `int` e `char`. Faltam outros tipos fundamentais do C, como `float`, `double` ou mesmo qualificadores como `unsigned`.
- **Gramática Incompleta:** A gramática da *Micro C* é um subconjunto estrito do C. Faltam diversas estruturas de controle de fluxo essenciais, como os laços `while` e `do-while`, a seleção `switch-case`, e tipos de dados agregados, como `structs`.

7.4.2 Propostas de Trabalhos Futuros

Cada limitação listada acima representa uma oportunidade para a evolução deste projeto. As propostas de trabalhos futuros podem transformar o Compilador *Micro C* de uma ferramenta didática para um compilador de propósito mais geral implementando algumas estruturas adicionais como, por exemplo:

- **Implementação de uma Fase de Otimização:** adição de uma nova fase de otimização *dentro* do *back-end*, a ser executada entre a geração da RI e a geração do *Assembly*. Esta fase operaria sobre a RI para realizar otimizações,

como eliminação de código desnecessário, propagação de constantes e, o mais importante, alocação de registradores para reduzir o tráfego de memória da estratégia *Load-Operate-Store*.

- **Suporte a Ponteiros e Alocação Dinâmica:** expansão da sintaxe e da semântica para incluir ponteiros. Isso não só permitiria a passagem de arrays, mas também abriria caminho para a implementação de funções de alocação de memória (como `malloc`), aproximando a linguagem do poder total do C.
- **Expansão do Sistema de Tipos:** adicionar suporte para tipos de ponto flutuante (`float` e `double`), o que exigiria a utilização dos registradores FPU/SSE (como `%xmm0`) e novas instruções de *Assembly*.
- **Geração de *Back-Ends* Adicionais:** criação de novos geradores de código. Um *back-end* para arquiteturas **ARM** ou **RISC-V** poderia ser implementado, reutilizando todo o *front-end* e o gerador de RI existentes.
- **Independência de Ferramentas Externas:** substituir a dependência atual do `gcc`. Isso envolveria a implementação das duas fases finais da compilação: um **Montador** (*Assembler*) próprio, capaz de traduzir o arquivo `.s` em um arquivo objeto binário (`.o`), e, subsequentemente, um ***Linker*** (*Ligador*) capaz de ligar o arquivo `.o` com as bibliotecas do sistema para produzir o executável final.

7.5 Considerações Finais

A construção de um compilador é um exercício que relaciona teorias e conceitos de ciência da computação com a construção de um software complexo. Foi projetado, implementado e validado um compilador para a linguagem *Micro C*, que funciona do código fonte a geração de código *Assembly* para o executável binário.

Além disso, do ponto de vista da implementação, este trabalho vai facilitar o aprendizado acadêmico, porque cria um compilador para uma linguagem minimalista com um tamanho acessível (em linhas de código) para que outros estudantes possam entender e modificar, de acordo com suas necessidades de aprendizado, o compilador. Por exemplo, o `gcc`, um compilador muito utilizado em ambientes Unix-*like*, possui mais de 15 milhões de linhas de código (informação de 2019), por outro lado, o compilador do *Micro C*, possui pouco menos de 4 mil linhas de código (o que representa em torno de 0,027% do `gcc`). Isso demonstra que é possível implementar um compilador funcional, mesmo que não seja o compilador mais eficiente.

O código fonte completo do projeto, documentado nesta monografia, está disponível online sob a licença GPLv3 [14].

Referências Bibliográficas

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compiladores: Princípios, Técnicas e Ferramentas*. Pearson Addison-Wesley, 2 edition, 2008.

[2] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 3(5):299–314, 1960.

[3] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

[4] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2 edition, 2012.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2022.

[6] M. Corporation. *Microsoft Macro Assembler Reference*, 2025. [Online; acessado em 06-11-2025].

[7] J. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[8] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Outubro 2025. Manual oficial de desenvolvedor de software da Intel. Order Number: 325462-089US.

[9] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., 2001.

[10] C. Lattner. LLVM and Clang: Next Generation Compiler Technology. In *The BSD conference*, volume 5, pages 1–20, 2008.

[11] C. Lattner and V. S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86. IEEE Computer Society, 2004.

[12] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V Application Binary Interface: AMD64 Architecture Processor Supplement, July 2013.

[13] R. Nystrom. *Crafting Interpreters*. Genever Benning, 2021.

[14] Pedro H. F. Carvalho and Brivaldo A. da Silva Jr. Compilador Micro C. `https://github.com/WorksFacom/mcc`, 2025. [Online; acessado em 11-11-2025].

[15] R. M. Stallman. GNU Compiler Collection Internals. *Free Software Foundation*, 46:765, 2002.

[16] S. Tatham, J. Hall, H. P. Anvin, and N. D. Team. *NASM - The Netwide Assembler*, 3.01 edition, outubro 2025. [Online; acessado em 11-11-2025].

Apêndice A

Linguagem do *Micro C*

Abaixo estão descritos os operadores e demais elementos do subconjunto da linguagem de programação C utilizada pelo compilador Micro C. São definidos os tipos literais, operadores aritméticos, relacionais, lógicos e de atribuição. Além disso, são definidos os delimitadores palavras reservadas e o token utilizado para representar o final do arquivo. Os elementos definidos são os utilizados em todas as fases da compilação mas em momentos diferentes.

Literais

- **UNDEF** - Token indefinido (erro ou valor desconhecido).
- **ID** - Identificadores (variáveis, funções).
- **INTEGERCONST** - Constante inteira (ex: 42).
- **CHARCONST** - Constante de caractere (ex: 'a').
- **STRINGCONST** - Constante de string (ex: "a35er").

Operadores Aritméticos

- **PLUS** - Adição "+".
- **MINUS** - Subtração "-".
- **MUL** - Multiplicação "*".
- **DIV** - Divisão "/".
- **MOD** - Módulo "%".

Operadores Relacionais

- **EQ** - Igualdade "==".
- **NEQ** - Desigualdade "!=".
- **LT** - Menor que "<".

- **GT** - Maior que "`>`".
- **LEQ** - Menor ou igual que "`<=`".
- **GEQ** - Maior ou igual que "`>=`".

Operadores Lógicos

- **AND** - Operador lógico E "`&&`".
- **OR** - Operador lógico OU "`||`".
- **NOT** - Operador lógico NÃO "`!`".

Operadores de Atribuição

- **ASSIGN** - Atribuição "`=`".

Delimitadores

- **SEMICOLON** - Ponto e vírgula "`;`".
- **COMMA** - Vírgula "`,`".
- **LPAREN** - Parêntese esquerdo "`(`".
- **RPAREN** - Parêntese direito "`)`".
- **LBRACE** - Chave esquerda "`{`".
- **RBRACE** - Chave direita "`}`".
- **LBRACKET** - Colchete esquerdo "`[`".
- **RBRACKET** - Colchete direito "`]`".

Palavras Reservadas

- **MAIN** - Função principal `main`.
- **IF** - Estrutura condicional `if`.
- **ELSE** - Estrutura condicional `else`.
- **PRINT** - Comando de Imprimir `print`.
- **FOR** - Laço de repetição `for`.
- **RETURN** - Comando de retorno `return`.
- **INT** - Tipo de dado `int`.
- **CHAR** - Tipo de dado `char`.

Fim de Arquivo

- **END_OF_FILE** - Fim do arquivo.

Apêndice B

Gramática do *Micro C*

Neste Apêndice, apresentamos a especificação formal da sintaxe da linguagem *Micro C*. A estrutura é definida por meio de uma Gramática Livre de Contexto (GLC), utilizando uma notação baseada em BNF (*Backus-Naur Form*) [2].

A gramática descreve as regras de produção para a estrutura do programa, comandos, expressões e operadores suportados pela linguagem do *Micro C*.

1. Estrutura e Declarações

$$
\begin{aligned}
\langle programa \rangle &\rightarrow \textbf{int main ()}\ \langle bloco \rangle \\
\langle bloco \rangle &\rightarrow \{\ \langle declarações \rangle\ \langle comandos \rangle\ \} \\
\langle declarações \rangle &\rightarrow \langle declaração \rangle\ \langle declarações \rangle \mid \varepsilon \\
\langle declaração \rangle &\rightarrow \langle tipo \rangle\ \textbf{ID}\ ; \mid \langle tipo \rangle\ \textbf{ID}\ [\ \textbf{INTEGERCONST}\ ]\ ; \\
\langle tipo \rangle &\rightarrow \textbf{int} \mid \textbf{char}
\end{aligned}
$$

2. Comandos

$$
\begin{aligned}
\langle comando \rangle &\rightarrow \langle if \rangle \mid \langle for \rangle \mid \langle atribuição \rangle \mid \langle return \rangle \mid \langle print \rangle \mid \langle bloco \rangle \mid ; \\
\langle if \rangle &\rightarrow \textbf{if}\ (\ \langle expressão \rangle\ )\ \langle comando \rangle\ [\ \textbf{else}\ \langle comando \rangle\ ] \\
\langle for \rangle &\rightarrow \textbf{for}\ (\ \langle atribuição\_interna \rangle\ ;\ \langle expressão \rangle\ ; \\
&\qquad\qquad \langle atribuição\_interna \rangle\ )\ \langle comando \rangle \\
\langle return \rangle &\rightarrow \textbf{return}\ [\ \langle expressão \rangle\ ]\ ; \\
\langle print \rangle &\rightarrow \textbf{print}\ (\ \langle conteúdo\_print \rangle\ )\ ; \\
\langle atribuição \rangle &\rightarrow \langle atribuição\_interna \rangle\ ; \\
\langle atribuição\_interna \rangle &\rightarrow \textbf{ID}\ =\ \langle expressão \rangle \mid \textbf{ID}\ [\ \langle expressão \rangle\ ]\ =\ \langle expressão \rangle
\end{aligned}
$$

3. Expressões e Operadores

$$
\begin{aligned}
\langle conteúdo\_print \rangle &\rightarrow \textbf{STRINGCONST} \mid \langle expressão \rangle \\
\langle expressão \rangle &\rightarrow \langle expressão\_lógica \rangle \\
\langle expressão\_lógica \rangle &\rightarrow \langle expressão\_relacional \rangle \ \{ \ (\&\& \mid ||) \ \langle expressão\_relacional \rangle \ \} \\
\langle expressão\_relacional \rangle &\rightarrow \langle expressão\_aritmética \rangle \ [\ \langle operador\_relacional \rangle \\
&\qquad \langle expressão\_aritmética \rangle \ ] \\
\langle operador\_relacional \rangle &\rightarrow == \mid != \mid < \mid > \mid <= \mid >= \\
\langle expressão\_aritmética \rangle &\rightarrow \langle termo \rangle \ \{ \ (+ \mid -) \ \langle termo \rangle \ \} \\
\langle termo \rangle &\rightarrow \langle fator \rangle \ \{ \ (* \mid / \mid \%) \ \langle fator \rangle \ \} \\
\langle fator \rangle &\rightarrow (\ \langle expressão \rangle \ ) \mid \textbf{ID} \mid \textbf{ID} \ [\ \langle expressão \rangle \ ] \mid \\
&\quad \textbf{INTEGERCONST} \mid \textbf{CHARCONST} \mid ! \ \langle fator \rangle
\end{aligned}
$$

Apêndice C

Exemplos de Código

A seguir, temos exemplos de implementação de códigos utilizando a linguagem reconhecida pelo compilador *Micro C*, de acordo com a gramática definida no Apêndice B e com seus respectivos *tokens* definidos no Apêndice A. As implementações demonstram que, mesmo uma linguagem simples, é capaz de resolver vários problemas computacionais. No repositório do código [14] existem outros exemplos.

C.1 Fibonacci Recursivo

O código do Fibonacci recursivo demonstra como o compilador é capaz de tratar corretamente a invocação recursiva de chamadas a mesma função, empilhando corretamente as chamadas e gerando o resultado esperado.

fibonacci.mcc

```
int fibonacci(int n) {
    int resultado;

    if (n <= 1) {
        resultado = n;
        return resultado;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    print("Calculando fibonacci(8)...");
    print(fibonacci(8)); // Deve imprimir 21
    return 0;
}
```

C.2 Bubble Sort

O *Bubble Sort* é um algoritmo de ordenação de vetores que recebe como entrada um vetor não ordenado e retorna como saída o vetor com os elementos ordenados. O uso de várias variáveis, alocação de vetores, e laços aninhados funciona como esperado.

bubble.mcc

```
int main() {
    int v[5];
    int i;
    int j;
    int temp;
    v[0] = 50;
    v[1] = 20;
    v[2] = 40;
    v[3] = 10;
    v[4] = 30;

    print("--- Array Desordenado ---");
    for (i = 0; i < 5; i = i + 1) {
        print(v[i]);
    }

    // --- algoritmo Bubble Sort ---

    for (i = 0; i < 5; i = i + 1) {
        //passa pelo array comparando pares
        for (j = 0; j < 4; j = j + 1) {
            if (v[j] > v[j+1]) {
                //Troca
                temp = v[j];
                v[j] = v[j+1];
                v[j+1] = temp;
            }
        }
    }

    print("--- Array Ordenado ---");
    for (i = 0; i < 5; i = i + 1) {
        print(v[i]);
    }

    return 0;
}
```

C.3 Fatorial de um Número

Diferente do exemploa da sequência de Fibonacci, o calculo do fatorial de um número utiliza sequências comparativas determinísticas (`if-else` com o fluxo do código variando entre atribuir um valor a uma variável ou calcular recursivamente o fatorial do número anterior e receber o resultado agregado final.

fatorial.mcc

```
int fatorial(int n) {
    int resultado_recursivo;
    int resultado_final;

    if (n < 2) {
        resultado_final = 1;
    } else {
        resultado_recursivo = fatorial(n - 1);
        resultado_final = n * resultado_recursivo;
    }
    return resultado_final;
}

int main() {
    int valor;

    print("--- Teste de Recursao ---");

    //calcular o fatorial de 5 (5! = 120)
    print("O resultado de 5! e:");
    print(fatorial(5));

    return 0;
}
```

Um detalhe importante é que, diferente da linguagem C original, o *Micro C* implementa uma função de impressão `print()` simplificada que foi criada apenas para fins de depuração. Ela não utiliza os mesmos parâmetros da função `printf()` da biblioteca padrão do C.

Apêndice D

Compilação Completa

Neste apêndice será mostrado um exemplo de código e quais os códigos intermediários gerados pelo compilador para cada uma das fases de compilação.

D.1 Código Fonte: `soma.mcc`

O exemplo a seguir apresenta uma soma simples de duas variáveis inteiras declaradas. As variáveis recebem, cada uma, um valor de atribuição e sua soma é impressa no terminal.

soma.mcc

```
int main()
{
  int a;
  int b;

  a = 10;
  b = 20;

  print(a + b);

  return 0;
}
```

Este exemplo se encontra no diretório `/exemplos` no repositório do código fonte do compilador.

D.2 Análise Léxica

A primeira fase é a análise léxica. Para gerar os *tokens* pela análise léxica usamos o comando:

```
$ mcc --scan exemplos/soma.mcc
```

ao executar este comando, o compilador irá realizar a análise léxica e gerar o arquivo `tokens.txt` que será utilizado nas fases posteriores.

Fase 1: análise léxica

```
Fase 1 (Lexica) concluida: 31 tokens gerados.
--- LISTA DE TOKENS GERADOS ---
Token: tipo = 33, lexema = 'int', linha = 1
Token: tipo = 28, lexema = 'main', linha = 1
Token: tipo = 22, lexema = '(', linha = 1
Token: tipo = 23, lexema = ')', linha = 1
Token: tipo = 24, lexema = '{', linha = 2
Token: tipo = 33, lexema = 'int', linha = 3
Token: tipo = 1, lexema = 'a', linha = 3
Token: tipo = 20, lexema = ';', linha = 3
Token: tipo = 33, lexema = 'int', linha = 4
Token: tipo = 1, lexema = 'b', linha = 4
Token: tipo = 20, lexema = ';', linha = 4
Token: tipo = 1, lexema = 'a', linha = 6
Token: tipo = 19, lexema = '=', linha = 6
Token: tipo = 2, lexema = '10', linha = 6
Token: tipo = 20, lexema = ';', linha = 6
Token: tipo = 1, lexema = 'b', linha = 7
Token: tipo = 19, lexema = '=', linha = 7
Token: tipo = 2, lexema = '20', linha = 7
Token: tipo = 20, lexema = ';', linha = 7
Token: tipo = 35, lexema = 'print', linha = 9
Token: tipo = 22, lexema = '(', linha = 9
Token: tipo = 1, lexema = 'a', linha = 9
Token: tipo = 5, lexema = '+', linha = 9
Token: tipo = 1, lexema = 'b', linha = 9
Token: tipo = 23, lexema = ')', linha = 9
Token: tipo = 20, lexema = ';', linha = 9
Token: tipo = 32, lexema = 'return', linha = 11
Token: tipo = 2, lexema = '0', linha = 11
Token: tipo = 20, lexema = ';', linha = 11
Token: tipo = 25, lexema = '}', linha = 12
Token: tipo = 36, lexema = 'EOF', linha = 13
---------------------------------
Arquivo 'tokens.txt' gerado com sucesso.
```

D.3 Análise Sintática

O próximo passo é executar a análise sintática.

```
$ mcc --parse exemplos/soma.mcc
```

Executando o compilador com o parâmetro `-parse`, o analisador sintático processa a sequência de *tokens* para validar se a estrutura do código obedece às regras gramaticais definidas para o *Micro C*:

Fase 2: análise sinática

```
Fase 1 (Lexica) concluida: 31 tokens gerados.
Fase 2 (Sintatica) concluida: AST gerada.
--- ARVORE SINTATICA ABSTRATA (AST) ---
NO_PROGRAMA
  NO_DEFINICAO_FUNCAO
    NO_TIPO (INT)
    NO_ID (Nome: main)
    NO_BLOCO
      NO_DECLARACAO_VARIAVEL
        NO_TIPO (INT)
        NO_ID (Nome: a)
      NO_DECLARACAO_VARIAVEL
        NO_TIPO (INT)
        NO_ID (Nome: b)
      NO_ATRIBUICAO
        NO_ID (Nome: a)
        NO_CONST_INT (Valor: 10)
      NO_ATRIBUICAO
        NO_ID (Nome: b)
        NO_CONST_INT (Valor: 20)
      NO_PRINT
        NO_OP_BINARIA (PLUS)
          NO_ID (Nome: a)
          NO_ID (Nome: b)
      NO_RETORNO
        NO_CONST_INT (Valor: 0)
----------------------------------------
Arquivo 'ast.txt' gerado com sucesso.
```

ao final da análise sintática, é gerado o arquivo `ast.txt` que é a árvore sintática abstrata que será utilizado na análise semântica.

D.4 Análise Semântica

Em seguida, é realizada a análise semântica para validar a lógica implementada pelo programador e calcular os endereços de memória das variáveis.

```
$ mcc --semantic exemplos/soma.mcc
```

Se a lógica implementada no código estiver correta, o compilador valida a estrutura e exibe a **Tabela de Símbolos** gerada. Nela, é possível visualizar os escopos (Global e Local), os tipos das variáveis e, principalmente, o *Offset* (deslocamento) de memória calculado para a geração de código:

Fase 3: análise semântica e tabela de símbolos

```
Fase 1 (Lexica) concluida: 31 tokens gerados.
Fase 2 (Sintatica) concluida: AST gerada.
Fase 3 (Semantica) concluida: Codigo validado.

ESCOPO: Funcao: main
NOME                | TIPO       | CATEGORIA  | ARRAY?     | TAMANHO  | OFFSET
--------------------+------------+------------+------------+----------+--------
a                   | INT        | VAR LOCAL  | NAO        | 0        | -4
b                   | INT        | VAR LOCAL  | NAO        | 0        | -8
--------------------------------------------------------------------------------

ESCOPO: GLOBAL
NOME                | TIPO       | CATEGORIA  | ARRAY?     | TAMANHO  | OFFSET
--------------------+------------+------------+------------+----------+--------
main                | INT        | FUNCAO     | NAO        | 0        | 0
--------------------------------------------------------------------------------

Arquivo 'symbols.txt' gerado com sucesso.
```

D.5 Intercode

A quarta fase é a geração do código intermediário:

```
$ mcc --gen-ir exemplos/soma.mcc
```

A saída da geração intermediária são códigos seguindo o padrão CTE descrito no Capítulo 5. O resultado, que é o arquivo `ir.txt` contendo o código intermediário, dessa fase está a seguir:

Fase 4: intercode

```
Fase 1 (Lexica) concluida: 31 tokens gerados.
Fase 2 (Sintatica) concluida: AST gerada.
Fase 3 (Semantica) concluida: Codigo validado.
Fase 4 (Geracao de IR) concluida.

--- MODO DE GERACAO DE CODIGO INTERMEDIARIO ---
--- CODIGO INTERMEDIARIO (IR) ---
main:
    a := 10
    b := 20
    t0 := a + b
    print_int t0
    return 0
---------------------------------
Arquivo 'ir.txt' gerado com sucesso.
```

D.6 *Assembly*

Na última fase, o código intermediário é convertido em linguagem *Assembly*. Para executar a geração de código em *Assembly*:

```
$ mcc --gen-asm exemplos/soma.mcc
```

A execução gera o código de saída em *Assembly* `soma.s`, contendo todas as instruções necessárias para o GCC montar e *linkar* o binário compatível com a arquitetura Intel/AMD x86-64.

Fase 5: geração do *assembly*

```
Fase 1 (Lexica) concluida: 31 tokens gerados.
Fase 2 (Sintatica) concluida: AST gerada.
Fase 3 (Semantica) concluida: Codigo validado.
Fase 4 (Geracao de IR) concluida.
Fase 5 (Geracao de Assembly) concluida: 'exemplos/soma.s' gerado.
--- MODO DE GERACAO DE ASSEMBLY CONCLUIDO ---
```

O código *Assembly* gerado:

exemplo.s

```
.section .rodata
.L.str.int: .string "%d\n"
.L.str.char: .string "%c\n"
.L.str.str: .string "%s\n"

.text
.globl main
main:
    pushq %rbp
    movq %rsp, %rbp
    subq $256, %rsp
    movl $10, %eax
    movl %eax, -4(%rbp)
    movl $20, %eax
    movl %eax, -8(%rbp)
    movl -4(%rbp), %eax
    movl -8(%rbp), %ebx
    addl %ebx, %eax
    movl %eax, -128(%rbp)
    leaq .L.str.int(%rip), %rdi
    movl -128(%rbp), %esi
    xorl %eax, %eax
    subq $8, %rsp
    call printf
    addq $8, %rsp
    movl $0, %eax
    movq %rbp, %rsp
    popq %rbp
    ret
```

O código final em *Assembly* pode ser transformado em binário usando o compilador de *Assembly* do próprio `gcc` com o comando:

```
$ gcc exemplos/soma.s -o soma
$ ./soma
30
```

Com isso, se encerra o processo de compilação e execução do código binário gerado pelo compilador do *Micro C*.