MICRO C



APRENDENDO COMPILADORES

PEDRO HENRIQUE E BRIVALDO JUNIOR

Copyright © 2026 Pedro Henrique e Brivaldo Junior

ISBN: 978-65-02-42701-9

Licença Creative Commons. Esta obra está licenciada sob a licença Creative Commons Atribuição–Uso Não Comercial–Compartilhamento pela mesma Licença 3.0 Brasil (CC BY-NC-SA 3.0 BR). É permitido compartilhar e adaptar a obra, desde que seja atribuído o devido crédito aos autores, o uso não tenha finalidade comercial e as contribuições sejam distribuídas sob a mesma licença. O texto integral da licença está disponível em `https://creativecommons.org/licenses/by-nc-sa/3.0/br/`. A licença é não exclusiva: os autores permanecem livres para comercializar a obra e conceder autorizações separadas para outros usos.

Observação jurídica. Esta licença abrange somente os direitos autorais e conexos pertencentes aos autores. Direitos de terceiros eventualmente presentes na obra, bem como direitos de imagem, privacidade, marcas e dados pessoais, não são licenciados além do que estiver expressamente indicado. Os direitos morais dos autores e as exceções e limitações aos direitos autorais permanecem sujeitos à legislação brasileira, especialmente à Lei nº 9.610/1998. Esta observação não cria restrições adicionais às permissões concedidas pela CC BY-NC-SA 3.0 BR.

Primeira impressão, Dezembro, 2026



Sumário

I Começando a Jornada

1 Introdução 15

1.1 Tradutores de Linguagens: Compiladores e Interpretadores 17

1.2 Estrutura do Livro 18

II Caminhando pelo Código

2 Análise Léxica 23

2.1 Análise Léxica: A Primeira Validação 24

2.2 Do Código-fonte aos Lexemas e *Tokens* 25

2.3 Os *Tokens* no Compilador do *Micro C* 28

2.3.1 Vocabulário Completo: `tokens.h` 28

2.3.2 Atributos: O Valor por trás do *Token* 30

2.3.3 Tratamento de Erros Léxicos 32

2.4 Relação da Prática com a Teoria 34

2.4.1 Expressões Regulares para Especificação dos *Tokens* 35

2.4.2 O *Scanner* como um Autômato Finito Determinístico 35
2.4.3 O *Scanner* como um AFD Manual 38
2.5 O *Scanner* da Linguagem *Micro C* 39
2.5.1 Interface e Implementação do *Scanner* 39
2.5.2 A Função `proximo_token()` 39
2.5.3 Desafios Práticos e Soluções no Código 40
2.5.4 A Interação com as Fases Subsequentes 46
2.6 Sumário 47
2.7 Exercícios 48
2.8 Problemas 49

3 Análise Sintática 51

3.1 Expressões e Gramáticas 52
3.1.1 Hierarquia de Precedência em Expressões 52
3.1.2 Gramática como Especificação Formal da Linguagem 53
3.2 Tratamento de Erros na Análise Sintática 60
3.3 Leitura dos *Tokens* 61
3.3.1 Estrutura para Armazenamento de *Tokens* 62
3.3.2 Leitura do Arquivo `tokens.txt` 62
3.3.3 Conectando as Fases Léxica e Sintática 64
3.4 Gramática da Linguagem 65
3.4.1 Notações de Gramática 66
3.4.2 Relação entre Gramática e Analisador Sintático 67
3.5 Análise Sintática Descendente 68
3.5.1 Descida Recursiva 69
3.5.2 Problemas com Recursão à Esquerda 71
3.5.3 Implementação Sem Recursão 71
3.6 Conjuntos FIRST e FOLLOW 72
3.6.1 Cálculo dos Conjuntos 73
3.6.2 Gramáticas LL(1) 74
3.6.3 Gramáticas que não são LL(1) 74
3.6.4 Análise Preditiva 75
3.6.5 Recuperação de Erros 76

3.7 A Árvore Sintática Abstrata (ASA) 76
3.7.1 Estrutura de um Nó da ASA no *Micro C* 76
3.7.2 Construindo e Manipulando a Árvore 79
3.8 Analisador Sintático 80
3.8.1 Processamento de *tokens* esperados 81
3.8.2 Avanço para o próximo *token* 82
3.9 Sumário 83
3.10 Exercícios 84
3.11 Problemas 85

4 Análise Semântica 87
4.1 Além da Gramática: Sentido e Contexto 88
4.1.1 Verificação Estática vs. Dinâmica 88
4.1.2 Atributos, Regras Semânticas e a Tabela de Símbolos 89
4.2 Objetivos da Análise Semântica 90
4.3 O Papel da Tabela de Símbolos 91
4.3.1 Exemplos de Erros Semânticos 91
4.4 Estrutura da Tabela de Símbolos 92
4.4.1 Organização em Pilha de Escopos 93
4.4.2 Estrutura dos Símbolos e Tabelas 93
4.4.3 Inserção e Busca de Símbolos 95
4.4.4 Gerenciamento de Escopos 97
4.4.5 Processamento da Árvore Sintática e Validação 98
4.5 Sumário 99
4.6 Exercícios 100
4.7 Problemas 101

III Geração de Código Binário

5 Código Intermediário 105
5.1 A Transição da Análise para a Síntese 106
5.1.1 O que é um Código Intermediário? 106
5.1.2 *Front-End* vs. *Back-End*: Onde Traçar a Linha? 107

5.1.3 O Papel do *Back-End*: Do "O Quê" para o "Como" 108
5.1.4 A ASA e a Tabela de Símbolos 108
5.2 O Papel da Representação Intermediária 109
5.2.1 Abstração e a Simplificação do Problema 109
5.2.2 Portabilidade entre Arquiteturas 110
5.3 O Código de Três Endereços (CTE) 111
5.3.1 A Estrutura do CTE no compilador do *Micro C* 112
5.3.2 Formato das Instruções: Tradução para CTE 114
5.3.3 Variáveis Temporárias e Rótulos 118
5.4 Gerador de Código: ASA para RI 118
5.4.1 *Tree Walker* e a Interação com a Tabela de Símbolos 119
5.4.2 Traduzindo Expressões: A Função `gerar_ir_expr` 119
5.4.3 Traduzindo Instruções: função `gerar_ir_no` 120
5.5 Sumário 124
5.6 Exercícios 124
5.7 Problemas 126

6 Do Assembly ao Executável 127
6.1 A Plataforma Alvo: x86-64 e a Sintaxe AT&T 128
6.1.1 A Arquitetura x86-64 128
6.1.2 Sintaxe AT&T vs. Sintaxe Intel 129
6.2 O Modelo de Memória e Execução 131
6.2.1 Registradores: A Memória Rápida da CPU 131
6.2.2 O Registro de Ativação (Stack Frame) 132
6.2.3 Prólogo e Epílogo: Gerenciamento do Frame 133
6.2.4 Desafios de Implementação 134
6.3 Tradução da RI para *Assembly* 135
6.3.1 O Tradutor de Operandos 135
6.3.2 A Estratégia de Tradução: *Load-Operate-Store* 136
6.4 Convenções de Chamada e *Linking* 140
6.4.1 Convenção Interna do *Micro C* 140
6.4.2 O Desafio do print 141
6.5 Sumário 144

6.6 Exercícios 145
6.7 Problemas 147

IV Final de Jogo

A Linguagem do Micro C 151

B Gramática do Micro C 153

C Exemplos de Código 155
C.1 Fibonacci Recursivo 155
C.2 Bubble Sort 156
C.3 Fatorial de um Número 157

D Compilação Completa 159
D.1 Código Fonte: `soma.mcc` 159
D.2 Análise Léxica 160
D.3 Análise Sintática 161
D.4 Análise Semântica 161
D.5 Intercode 162
D.6 Assembly 162
D.7 Relação entre as Saídas Geradas 164

Bibliografia 165
Livros 165
Artigos 165
Outros 166

Índice 167

I Começando a Jornada

1 Introdução 15
1.1 Tradutores de Linguagens: Compiladores e Interpretadores
1.2 Estrutura do Livro

Prefácio

Com a palavra, o primeiro autor



Ao longo da minha vida — que ainda é breve — jamais imaginei que me tornaria autor de um livro, muito menos de um livro didático. Esta obra é a prova de que nunca sabemos exatamente quais caminhos a vida nos reserva.

A ideia de trabalhar com compiladores surgiu principalmente por incentivo do meu professor e amigo Brivaldo. Quando cursei a disciplina de Compiladores, ministrada pela professora Bianca, tive contato com conceitos que até então desconhecia completamente. Passei a compreender o processo de transformar o código que escrevemos em algo que a máquina consegue interpretar. Como tudo aquilo funcionava por trás? A disciplina me permitiu esclarecer muitas dúvidas e absorver diversos conhecimentos novos.

Ainda assim, fiquei com a sensação de que havia mais a explorar. Devido ao tempo limitado da disciplina, não foi possível aprofundar todos os tópicos relacionados a compiladores. Essa lacuna despertou em mim o desejo de ir além.

Com essa inquietação e com o incentivo do professor e orientador Brivaldo,

começamos a refletir sobre como poderíamos suprir essa necessidade. A melhor solução que encontramos foi criar um compilador do zero, detalhando cada etapa de forma modular e extraindo representações visuais que tornassem o processo mais compreensível. Foi assim que nasceu meu projeto de monografia.

No início, nem mesmo o nome estava definido. Ele mudou algumas vezes até chegarmos a Micro C Compiler, o que também originou a extensão dos arquivos `.mcc`. Durante o desenvolvimento do TCC, surgiu uma nova ideia: por que não transformar aquele material em um livro? A pergunta fez todo sentido. Enquanto implementávamos o compilador, eu escrevia sobre cada etapa na monografia, pensando em como aquele conteúdo poderia ser adaptado e expandido para se tornar uma obra didática mais completa.

No final de novembro de 2025, a monografia estava concluída e pude apresentá-la. Sou profundamente grato à banca avaliadora presente naquele quinto dia de dezembro: à professora Bianca, ao professor Diego Rubert e, especialmente, ao meu orientador, professor Brivaldo. Com suas sugestões e contribuições, foi possível aprimorar o trabalho e transformá-lo neste livro.

Com o TCC aprovado, iniciamos oficialmente a construção desta obra, que agora você começa a ler. Agradeço a todos que fizeram parte dessa jornada e espero que este material possa contribuir com todos que desejam aprender mais sobre compiladores e se aprofundar nesse fascinante universo.

E agora, o segundo autor



Escrever um livro sobre conceitos da computação é sempre um enorme desafio. Principalmente quando estamos falando de temas que envolvem diversos outros conhecimentos que, muitas das vezes, dependem de grande esforço dos alunos no seu aprendizado.

Compiladores é um tópico de interesse de muitos cientistas da computação e de várias outras áreas da própria computação. E o ser humano sempre busca formas, cada vez melhores, de capturar a imaginação e as ideias dos programadores em linguagens que os permitam construir programas (*softwares*) seguros e otimizados.

Quando estudei compiladores em meados de 2005, utilizamos o livro clássico de compiladores de Alfred V. Aho *et al.*, "O Livro do Dragão". E como eu, muitos acadêmicos utilizaram (ou continuam utilizando) o mesmo livro. Hoje temos outras referências, como a do livro "Crafting Interpreters" do Robert Nystrom e vários artigos e livros da área. Este livro, em especial, não chega nem perto e nem tem como objetivo ser superior aos livros clássicos e eu, pessoalmente, recomendo o estudo deles do ponto de vista científico, para aprofundar seus conhecimentos na área de compiladores.

Dito isso, se o *próprio autor* recomenda os livros clássicos, porquê você deveria usar o seu tempo lendo o nosso livro? É simples, esses livros clássicos vão abordar todas as formas, possibilidades e estruturas necessárias para desenvolver um compilador e isso, muitas das vezes, torna o aprendizado muito mais difícil.

Nosso objetivo com o compilador da linguagem *Micro C* foi desenvolver um compilador simples, com menos de 4 mil linhas de código, ou seja, algo em torno de 0,027% do `gcc`, para que qualquer acadêmico (ou entusiasta de tecnologia) conseguisse ler, alterar e ajustar o seu código e entender todas as etapas necessárias para a construção de um compilador.

Ou seja, com o *Micro C* você terá uma linguagem minimalista do próprio C, uma linguagem de programação desenvolvida por Dennis Ritchie e que é muito utilizada em várias áreas da computação, o que vai permitir certa facilidade no aprendizado e também na compreensão das estruturas básicas da própria linguagem. O que fizemos foi, com esse subconjunto minimalista, construir um compilador capaz de gerar código Assembly (formato da AT&T) do código fonte fornecido pelo programador para a arquitetura x86_64. Cada uma das etapas será abordada no livro e trechos de código serão utilizados para ilustrar como foram resolvidos problemas pontuais.

Além disso, serão propostas atividades para que você mesmo altere o compilador para dar suporte a novas funcionalidades simples e, com isso, entender, como uma pequena alteração na linguagem, altera várias etapas do próprio compilador. Como o nosso objetivo é a simplicidade, nós não vamos explicar conceitos fundamentais da computação. Ou seja, vamos apresentar sem rodeios quais foram as decisões tomadas durante a construção do compilador. Por exemplo, porque utilizar uma análise descendente de avaliação ao invés da ascendente? Esses conceitos você pode explorar, mais profundamente, nos livros clássicos. Aqui, seu objetivo é aprender como construir um compilador de ponta a ponta usando uma linguagem simples, passeando por todas as etapas necessárias.

Já aviso aos leitores que esse tipo de material pode ser **extremamente viciante** e não duvidaria de que alguns de vocês vão alterar o código do Micro C para dar suporte a novas estruturas de dados ou mesmo criar uma linguagem diferente e mais próxima a outras linguagens como, por exemplo, um *Micro Python*. É isso, espero que vocês possam se divertir programando seu próprio compilador e, dessa vez, compreendendo todas as etapas que um compilador executa para converter o código fonte em código de máquina.

Exercícios e Problemas

Uma das coisas que tivemos enorme satisfação (e muito trabalho), foi criar exercícios e problemas que fossem transversais no livro. O que isso significa? Que um exercício 2.1, no Capítulo 2, por exemplo, terá relação direta com o exercício 3.1 do Capítulo 3. Ou seja, no 2.1 você poderá ser instruído a adicionar uma nova instrução na análise léxica para introduzir o tipo `bool`, por exemplo, para incluir o suporte a variáveis booleanas (que podem ser verdadeiras ou falsas).

Já no capítulo seguinte, no exercício 3.1, será requisitado que você faça com que o compilador consiga realizar a análise sintática para o novo tipo gerado. Como o compilador do *Micro C* é todo segmentado, você será capaz de inserir uma etapa e testá-la para garantir que sua modificação foi implementada com sucesso naquela fase de compilação.

Agradecimentos

Este livro não teria sido desenvolvido se, ao longo dos anos na academia, eu não me incomodasse com a complexidade da literatura em torno da construção de um compilador. Mesmo quando meus professores há época (Paulo Pagliosa e Bianca Dantas) tentavam, da melhor maneira possível, destilar a complexidade do processo para que entendêssemos suas etapas. Este livro contou com a revisão e sugestões de hoje colegas como os professores Paulo Pagliosa, Bianca Dantas, Diego Rubert (meu colega de turma) e comentários e sugestões de outros colegas como o professor Amaury Junior.

E muito mais importante, do apoio e trabalho do meu aluno de graduação, hoje formado, Pedro Henrique, co-autor deste livro. Seu esforço na construção do compilador em um trabalho de conclusão de curso foi Hercúleo. Esperamos que vocês gostem e possam aproveitar o material desenvolvido para melhorar sua formação em computação e, em especial, em compiladores. Obrigado e boa leitura.



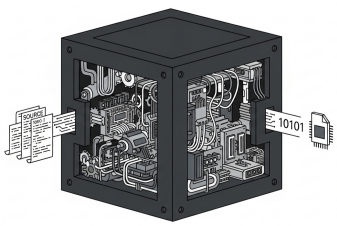
1. Introdução

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

Se olharmos por dentro dos sistemas que usamos todos os dias, encontraremos um compilador trabalhando silenciosamente. Apesar de sua importância, os compiladores modernos, como o GNU Compiler Collection (GCC) [8] e o *framework* LLVM (utilizado pelo Clang) [9, 10], são sistemas enormes e complexos. Eles possuem milhões de linhas de código e décadas de otimizações, o que os torna verdadeiras caixas-pretas para estudantes e iniciantes.

Do ponto de vista de quem está aprendendo sobre compiladores, são sistemas difíceis de serem compreendidos, estudados e entendidos em sua totalidade. Consequentemente, a maioria dos estudos acadêmicos e cursos desenvolvidos sobre compiladores fica restrita a algumas fases iniciais de compilação, como análise sintática e semântica, e tópicos teóricos como autômatos finitos, gramáticas livres de contexto e gerenciamento de pilha de compilação,



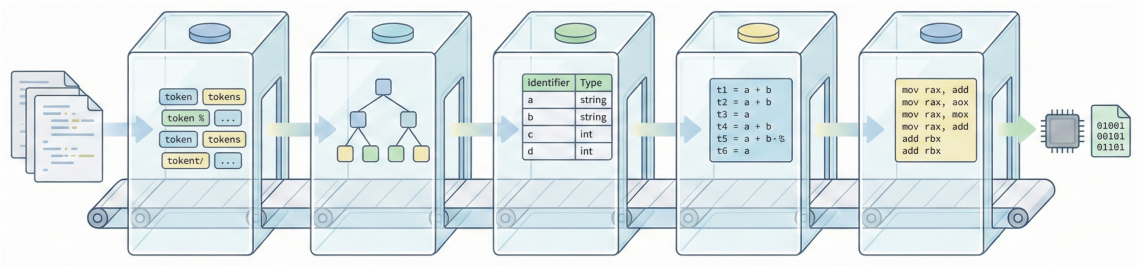
A complexidade opaca dos compiladores industriais, que funcionam como caixas-pretas.

deixando uma enorme lacuna sobre como essas peças se unem para gerar um programa executável real.

A construção de um compilador com fins acadêmicos é um desafio em vários níveis. Primeiro, escrever um compilador completo 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 livro nasce exatamente para preencher essa lacuna, focando na construção de compiladores com o objetivo de criar um material que permita que outros acadêmicos e estudantes possam compreender, de forma prática, como um compilador é construído do zero, os desafios de implementação e os conceitos teóricos envolvidos.

A linguagem *Micro C* é um subconjunto da linguagem C e, por isso, simplifica o aprendizado de programação e permite que o conhecimento seja aproveitado no futuro. Além disso, é a ferramenta ideal para o estudo de compiladores, pois sua representação é simples e reduzida. A Figura abaixo ilustra essa transparência. Em vez de uma caixa-preta incompreensível, o *Micro C* é estruturado como uma linha de montagem, dividida em módulos independentes.



A arquitetura do *Micro C*: o fluxo de compilação é quebrado em cinco etapas do código-fonte ao *Assembly*.

Como demonstrado na ilustração, o código-fonte entra no sistema e é processado em cinco etapas. Primeiro, passa pela *análise léxica* (onde o texto é agrupado em blocos chamados *tokens*); em seguida, pela *análise sintática* (que organiza a estrutura lógica em uma árvore de sintaxe) e pela *análise semântica* (que gerencia regras e tipos usando uma tabela de símbolos). Após as análises, o fluxo avança para a *geração de código intermediário* (representado por instruções matemáticas simples), até alcançar o último estágio de *geração de código Assembly* (onde surgem as instruções reais do processador, como `mov` e `add`), resultando no binário final.

Justamente por ter um código fonte reduzido e ter cada uma dessas etapas de compilação explícitas, o compilador do *Micro C* permite que seja compreendido exatamente como cada uma das transformações envolvidas na conversão do código fonte em linguagem de máquina foi desenvolvida.

É fundamental, no entanto, alinhar expectativas: o compilador do *Micro C* não está preocupado com performance ou otimização de código. Seus fins são estritamente 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, 2, 5] e utilizar apenas o estritamente necessário desses conceitos para implementar um compilador completo e com código acessível [16] publicamente sob a licença GPLv3.

Para colocar essa decisão em perspectiva, compiladores industriais aplicam dezenas de transformações complexas na representação do programa para extrair o máximo de desempenho da arquitetura alvo, um campo de estudo tão vasto e intrincado que exige literatura dedicada, como as obras de Kennedy e Allen [4] e Cooper et al. [6]. Ao abrir mão de técnicas avançadas de otimização, o *Micro C* garante que a jornada do código-fonte até a linguagem de máquina permaneça linear e inspecionável para o iniciante. Dessa forma, prioriza-se a clareza estrutural e pedagógica em detrimento da velocidade de execução do código gerado.

1.1 Tradutores de Linguagens: Compiladores e Interpretadores

Programas de *software* são escritos em linguagens de alto nível, como C ou Python, pois essa é a forma como programadores conseguem expressar conjuntos de instruções de maneira estruturada e humana. Contudo, um computador não compreende a lógica abstrata e muito menos as instruções textuais desses códigos de alto nível. O que a CPU (Unidade Central de Processamento) de um computador consegue executar são instruções binárias de baixo nível, conhecidas como código de máquina.

Para preencher essa lacuna, utilizamos tradutores. Na ciência da computação, existem duas abordagens principais para esta tradução: a compilação e a interpretação. Ambas têm o objetivo final de executar um conjunto de instruções para gerar um resultado (uma saída), mas adotam estratégias de tempo e montagem totalmente diferentes.

Um *compilador* é um programa que traduz uma representação de entrada, normalmente escrita em uma linguagem fonte de alto nível, para outra representação seman-

ticamente equivalente. Essa saída pode ser uma Representação Intermediária (RI), um *bytecode*, código *Assembly*, um arquivo objeto ou um executável. No fluxo adotado neste livro, o compilador do *Micro C* transforma o arquivo-fonte em *Assembly* x86-64; em seguida, ferramentas externas realizam a montagem e a ligação para produzir o executável. Portanto, a compilação não precisa ocorrer toda de uma só vez nem produzir diretamente o arquivo executável: essas são decisões da arquitetura de cada compilador.

Um *interpretador*, por outro lado, executa o programa por intermédio de outro programa, que lê o código-fonte ou uma representação intermediária e realiza as operações correspondentes durante a execução. Essa estratégia não implica, necessariamente, processamento linha a linha: implementações de linguagens como Python podem primeiro traduzir o código-fonte para *bytecode* e depois executá-lo em uma máquina virtual. Também existem modelos híbridos, como a compilação *just-in-time* (JIT), nos quais partes do programa são compiladas durante a execução.

Um interpretador pode atuar como uma máquina abstrata que executa diretamente o código-fonte ou uma forma intermediária. Python e Ruby possuem implementações predominantemente interpretadas, mas uma linguagem não está vinculada a uma única estratégia de execução. A portabilidade decorre da existência de uma implementação compatível do ambiente de execução para cada plataforma. O desempenho depende de vários fatores, como a representação executada, as otimizações, o custo das verificações dinâmicas e a eventual presença de compilação JIT; por isso, não deve ser atribuído apenas à tradução repetida de cada instrução.

Em resumo, compilação e interpretação são estratégias de implementação, não propriedades imutáveis de uma linguagem. A compilação antecipada para código nativo pode reduzir o trabalho realizado durante a execução, mas produz artefatos dependentes da arquitetura e da plataforma alvo. A interpretação e as máquinas virtuais podem favorecer portabilidade e inspeção dinâmica, embora introduzam custos próprios. Sistemas modernos frequentemente combinam essas estratégias.

1.2 Estrutura do Livro

Este material foi cuidadosamente organizado para guiar o leitor passo a passo por todas as etapas de construção de um compilador real. 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 especificação da linguagem. Encerrando, dessa forma, a construção do *front-end* do compilador.

O próximo passo nesta jornada 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 transição, 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 do *Micro C* 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 estudante inspecione e valide os resultados parciais de cada processo de compilação antes de prosseguir para a próxima fase. Além disso, a ferramenta também permite a **execução contínua de todas as etapas** de uma única vez para gerar exclusivamente o arquivo executável final diretamente, cumprindo assim o seu papel estrito de tradutor.

Como material de apoio para consulta, a obra conta ainda com quatro apêndices. O Apêndice A descreve toda a especificação da linguagem definida para o *Micro C*; o Apêndice B descreve a Gramática Livre de Contexto adotada; o Apêndice C apresenta diversos exemplos de códigos que podem ser escritos e compilados; e, finalmente, o Apêndice D demonstra um exemplo de compilação completa, passando por todas as etapas técnicas até a geração de um binário para o Linux.

II Caminhando pelo Código

2 Análise Léxica 23

2.1 Análise Léxica: A Primeira Validação
2.2 Do Código-fonte aos Lexemas e *Tokens*
2.3 Os *Tokens* no Compilador do *Micro C*
2.4 Relação da Prática com a Teoria
2.5 O *Scanner* da Linguagem *Micro C*
2.6 Sumário
2.7 Exercícios
2.8 Problemas

3 Análise Sintática 51

3.1 Expressões e Gramáticas
3.2 Tratamento de Erros na Análise Sintática
3.3 Leitura dos *Tokens*
3.4 Gramática da Linguagem
3.5 Análise Sintática Descendente
3.6 Conjuntos FIRST e FOLLOW
3.7 A Árvore Sintática Abstrata (ASA)
3.8 Analisador Sintático
3.9 Sumário
3.10 Exercícios
3.11 Problemas

4 Análise Semântica 87

4.1 Além da Gramática: Sentido e Contexto
4.2 Objetivos da Análise Semântica
4.3 O Papel da Tabela de Símbolos
4.4 Estrutura da Tabela de Símbolos
4.5 Sumário
4.6 Exercícios
4.7 Problemas



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

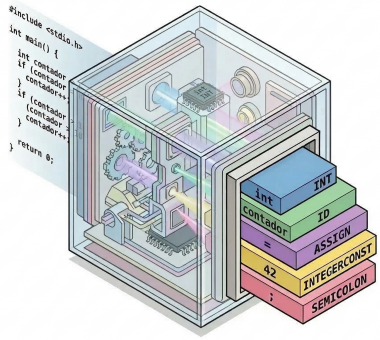
Neste capítulo, vamos abordar a análise léxica, a primeira etapa do fluxo de compilação. Ela tem o objetivo de ler os caracteres do código-fonte individualmente e identificar as sequências lógicas que correspondem a formatos esperados, ou seja, os *tokens* (símbolos ou elementos léxicos). Um *token* 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, que um programa possua linhas de código como $x = 5$ ou $y = 10$. Durante a análise léxica, o compilador lê essas linhas de código e identifica sequências de caracteres que representam operações ($=$), variáveis (x, y) ou outros elementos da linguagem. A essa sequência bruta lida do arquivo é dado o nome de **lexema**, que é a sequência real de caracteres encontrada no código-fonte. O compilador avalia o lexema e o associa a um *token* específico. Cada *token* possui um nome que indica sua função ou

tipo (por exemplo, uma palavra reservada, um número inteiro, um operador) 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.

O analisador léxico funciona como uma máquina de leitura e identificação de texto. Assim como o robô organiza as peças lidas em compartimentos específicos, o *scanner* isola os lexemas e os converte em blocos lógicos que são categorizados nas "prateleiras" apropriadas.



O padrão é a descrição do formato que os lexemas de um *token* devem seguir. Em outras palavras, é a regra 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.

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

2.1 Análise Léxica: A Primeira Validação

O analisador léxico realiza a triagem inicial do código-fonte. Ele valida se as cadeias de caracteres correspondem ao vocabulário da linguagem. Caso o *scanner* encontre um *padrão não previsto*, um erro léxico é imediatamente reportado, interrompendo o processo de análise.

Um erro léxico ocorre quando o *scanner* não consegue formar um *token* válido a

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

partir da entrada. Por exemplo, na linguagem do *Micro C*, o caractere `@` não pertence a nenhum padrão válido. Se encontrado, um erro é gerado.

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

Nesse exemplo, o *scanner* reconhece os lexemas `int`, `x`, `=` e `10`, mas ao encontrar o `@`, não consegue classificá-lo. O processo é interrompido e o compilador emite um erro indicando o caractere inválido.

2.2 Do Código-fonte aos Lexemas e *Tokens*

Já vimos que a análise léxica é o primeiro passo do processo de compilação, responsável por ler o arquivo fonte caractere por caractere, agrupando-os em elementos léxicos (*tokens*). Nós definimos que a linguagem do *Micro C* seria um subconjunto minimalista do C criado por Dennis Ritchie.

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 podem ser implementados como, por exemplo, *Bubble Sort*, Torre de Hanói ou Fibonacci (alguns exemplos estão no Apêndice C).

Para ilustrar a extração de lexemas e a categorização do vocabulário, vamos acompanhar como o *scanner* avança pela expressão `int numero = 45;`. A cada passo, o analisador foca em uma sequência de caracteres, isola o lexema e o classifica em sua respectiva classe léxica:

1. Palavras Reservadas

$$\underbrace{\texttt{int}}_{\textbf{INT}}\ \texttt{numero = 45;}$$

O primeiro lexema identificado é `int`, classificado com o *token* `INT`. Ele pertence à classe das palavras reservadas: termos que compõem a sintaxe fixa e estrutural da linguagem. Por possuírem significado pré-determinado, palavras reservadas nunca podem ser utilizadas pelo programador para nomear variáveis ou funções. No *Micro C*, esta categoria engloba tipos de dados (como `int` e `char`) e estruturas de controle de fluxo (como `if`, `else`, `for` e `return`).

2. Identificadores

$$\texttt{int}\ \underbrace{\texttt{numero}}_{\textbf{ID}}\ \texttt{= 45;}$$

Avançando no texto e ignorando o espaço em branco, o *scanner* isola o lexema `numero`, associando-o ao *token* genérico `ID`. Identificadores são os nomes arbitrários criados pelo programador para representar variáveis, funções e matrizes. O padrão léxico do *Micro C* exige que um identificador comece obrigatoriamente com uma letra ou *underscore*, podendo ser seguido por letras ou dígitos. Nesta fase, o analisador apenas extrai o nome; informações como o tipo da variável ou seu endereço de memória serão definidas futuramente pela análise semântica.

3. Operadores

$$\texttt{int numero}\ \underbrace{\texttt{=}}_{\textbf{ASSIGN}}\ \texttt{45;}$$

O próximo caractere é o símbolo =, classificado com o *token* `ASSIGN` (atribuição). Esta categoria abrange todos os símbolos matemáticos e lógicos responsáveis por realizar transformações, cálculos ou comparações em dados. A classe de operadores do *Micro C* reconhece operadores aritméticos (`+`, `-`, `*`, `/`) e operadores relacionais/lógicos (`==`, `!=`, `<`, `>`, `&&`, `||`).

4. Constantes Literais

$$\texttt{int numero =}\ \underbrace{\texttt{45}}_{\textbf{INTEGERCONST}}\ \texttt{;}$$

O *scanner* encontra a sequência numérica `45`, extraindo o lexema e mapeando-o para o *token* `INTEGERCONST`. Constantes ou literais são valores fixos embutidos diretamente no código-fonte. O compilador usa padrões distintos para classificar não apenas números inteiros, mas também caracteres únicos (`CHARCONST`, definidos entre aspas simples, como `'a'`) e cadeias de texto contínuas (`STRINGCONST`, definidas entre aspas duplas, como `"texto"`).

5. Delimitadores e Pontuação

```
int numero = 45 ;
                SEMICOLON
```

Por fim, o lexema `;` é isolado e associado ao *token* `SEMICOLON`. Os delimitadores formam uma classe de símbolos que não realizam operações matemáticas, mas ditam a estrutura e a organização do código. Eles incluem chaves para blocos de escopo (`{`, `}`), parênteses para parâmetros (`(`, `)`), vírgulas para separação (`,`) e o ponto e vírgula, que determina o final de uma instrução.

Ao final do processamento do *scanner*, todos os componentes foram classificados. A Figura 2.1 consolida visualmente o resultado desse escandimento, ilustrando a estrutura final gerada após cada fragmento de texto ter sido associado ao seu respectivo *token*.

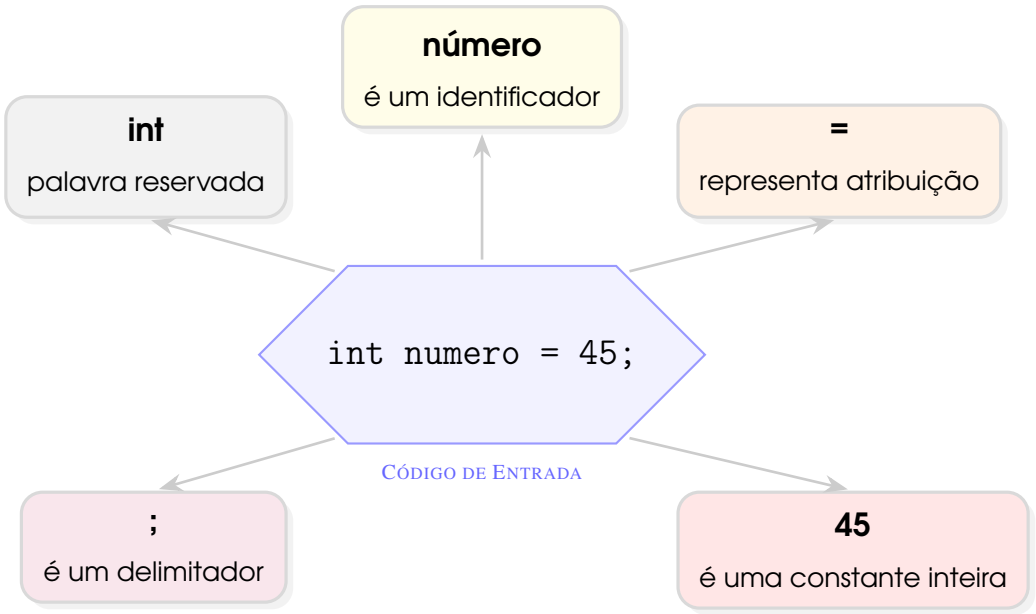


Figura 2.1: Uma expressão como `int numero = 45;` deve ser quebrada em vários *tokens* e cada um será interpretado de uma maneira diferente na análise léxica.

Quando o analisador léxico identifica um *token* do tipo `ID`, ele apenas registra o lexema correspondente (como o texto `numero`, classificado como `ID` na Figura 2.1). O *scanner* não associa 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 de conexão entre a análise léxica e as etapas seguintes do compilador.

Com a geração do fluxo de *tokens* pela análise léxica, encerra-se a primeira etapa da compilação. Ao transformar o texto bruto em unidades lógicas estruturadas, o analisador prepara o terreno para que as fases seguintes processem a gramática e a semântica 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 desses conceitos, mostrando como eles foram traduzidos para estruturas de dados no compilador. Esta seção detalha o *dicionário completo* do vocabulário do compilador do *Micro C* e como ele é implementado.

Para que essa transição entre o modelo abstrato e o código executável seja bem-sucedida, é fundamental estabelecer uma interface de comunicação interna no compilador. Todas as fases da compilação, em especial o analisador sintático, que será o consumidor primário desses dados, dependem de uma padronização dos elementos lidos. Sem uma base sólida para catalogar e trafegar essas entidades, o processamento do código-fonte seria inviável. Dessa forma, a modelagem desse vocabulário em linguagem C exige o agrupamento lógico de categorias por meio de enumerações (`enum`) e o uso de estruturas (`struct`) que empacotem a classe do *token* juntamente com metadados para a depuração, como o texto original lido e a linha exata em que ele foi identificado.

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 a manutenção. Cada valor dentro dessa enumeração representa um tipo diferente de *token* que o analisador léxico será capaz de identificar no código. Devido à sua extensão, dividiremos a apresentação desta estrutura em duas partes.

O primeiro trecho de código a seguir mostra a declaração inicial, contemplando os *tokens* fundamentais, literais e operadores aritméticos:

```
// arquivo: src/tokens/tokens.h (Parte 1.1: Tokens Fundamentais e Aritméticos)
#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,
```

Na sequência, a enumeração é concluída com os operadores relacionais, lógicos, símbolos de pontuação e as palavras reservadas da linguagem:

```
// arquivo: src/tokens/tokens.h (Parte 1.2:  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;
```

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, para que seja possível reportar erros mais completos e auxiliar o programador na identificação de falhas no código-fonte, apontando a localização mais próxima em que o erro ocorreu.

Esse comportamento é codificado na `struct Token`:

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

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[MAX_TOKEN_LEN]`: este é o atributo do *token*. Ele armazena o texto original do lexema. Como veremos na próxima subseção, este campo é útil para o **analisador semântico** (que o usará para inserir o nome da variável na Tabela de Símbolos) e para o **gerador de código** (que precisará saber o valor de uma constante como, por exemplo, `123`).
- `int linha`: Este campo armazena o número da linha em que o *token* foi encontrado. Sua finalidade é permitir que **qualquer fase do compilador** (léxica, sintática ou semântica) possa reportar erros que possam ajudar o programador a identificá-lo mais facilmente.

O campo `lexema` possui tamanho `MAX_TOKEN_LEN`, definido em `config.h`. As rotinas atuais contam todos os caracteres antes de gravá-los e retornam `UNDEF` com mensagens específicas quando identificadores, inteiros ou strings excedem seus limites. Assim, *uma entrada longa não é truncada* nem dividida em vários *tokens*.

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. Isso quer dizer que, ao identificar o `IF`, o analisador léxico já fornece toda a informação necessária apenas pelo tipo do *token*, sem precisar de um valor adicional no campo `lexema`. Diferentemente de identificadores ou números, cujo significado depende do valor específico reconhecido no código, palavras reservadas e delimitadores têm um significado fixo definido pela linguagem.

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 disso 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. Isso ocorre porque o tipo apenas informa a categoria léxica do *token* (por exemplo: identificador ou constante inteira), mas não contém informação suficiente sobre qual identificador específico ou qual valor numérico foi reconhecido. Nesses casos, o compilador precisa também do lexema ou de um atributo associado para determinar qual entidade do programa está sendo referenciada ou qual é o valor da constante.

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.

Esse exemplo mostra os usos complementares dos campos: `tipo` permite ao *parser* reconhecer a declaração, `lexema` distingue `x` de `y` e `10` de `20`, e `linha` localiza um eventual diagnóstico. A sequência de categorias descreve a estrutura gramatical; a estrutura `Token` completa preserva também a identidade e a localização dos elementos.

Isso exemplifica o porquê do `lexema` ser essencial. O fluxo de *tokens* 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 dessa declaração. Esse processo se repete para a variável `y`.

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 situações em que um lexema não corresponde a *nenhum token válido* (`UNDEF`).

Dessa forma, um *token* `UNDEF` pode ser gerado em dois cenários 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. Exemplos: uma string que não é fechada antes de uma quebra de linha ou um caractere que não é fechado corretamente. 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 `#`).

Nos dois cenários, `UNDEF` informa que o reconhecimento falhou, enquanto os demais campos preservam o motivo ou o caractere associado ao erro e a linha em que ele ocorreu. O programa principal pode, então, emitir o diagnóstico e interromper a compilação antes do *parser*. Os trechos seguintes mostram como essas informações são construídas.

A função atual `ler_string` ilustra o primeiro cenário. Ela consome a string diretamente no campo `lexema`, interpreta escapes e verifica cada condição de erro antes de retornar:

```
while (c != '"' && c != '\n' && c != EOF) {
    int valor = c;
    if (c == '\\') {
        int valido = 1;
        valor = decodificar_escape(prox_char(), &valido);
        if (!valido || valor == '\0')
            return token_erro("Escape invalido em string", linha);
    }
    if (tamanho + 1 >= sizeof(token.lexema) ||
        tamanho + 1 >= MAX_STR_LEN)
        return token_erro("String excede o limite", linha);
    token.lexema[tamanho++] = (char)valor;
    c = prox_char();
}
if (c != '"')
    return token_erro("String nao terminada", linha);
```

Ao sair do laço, `c != '"'` indica quebra de linha ou fim de arquivo antes das aspas finais. Nesse caso, `token_erro` produz um `UNDEF` seguro e preserva a linha inicial do literal.

O segundo cenário (caractere inválido) é tratado pelo caso `default` no final da cadeia de avaliações da função `proximo_token`. A seguir, um trecho que representa o *caso geral* (`default`) para qualquer caractere que não iniciou nenhum dos padrões previstos anteriormente:

```
// arquivo: src/scanner/scanner.c (trecho do final do switch)
    // Se nenhum padrão foi reconhecido, é um token indefinido
    default:
        token.lexema[0] = c;
        token.lexema[1] = '\0';
        token.tipo = UNDEF;
        break;
}
return token;
```

Notem que a lógica de parada não está no *scanner* em si, mas sim no código

principal do compilador (`main.c`). O `main` verifica cada *token* recebido pelo *scanner* e, se ele for classificado como `UNDEF`, interrompe imediatamente o processo de compilação. Essa abordagem obedece a um princípio fundamental de *design* de software: a separação de responsabilidades. O analisador léxico deve atuar estritamente como um fornecedor de elementos lógicos, sem tomar decisões de alto nível sobre o fluxo de execução em caso de falhas críticas. Ao simplesmente encapsular o caractere ofensor (como um `@` ou `#`) no campo `lexema` e repassá-lo adiante, o *scanner* fornece todas as evidências necessárias para que o orquestrador do compilador emita uma mensagem de diagnóstico precisa ao usuário e aborte a compilação com segurança. Essa delegação de controle mantém a máquina de estados do *scanner* muito mais enxuta, focada no reconhecimento contínuo de padrões.

```
// 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

Nas seções anteriores definimos o vocabulário da linguagem (`tokens.h`) e o papel do *scanner* em identificar lexemas e gerar *tokens*. O próximo passo é defini-las formalmente.

A lógica por trás da implementação do *scanner* não é arbitrária, mas a aplicação de conceitos clássicos de teoria da computação. E, embora não seja nosso 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 meta-caracteres 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**. A aplicação direta e a implementação estrutural desses padrões no código do compilador serão detalhadas na próxima seção.

2.4.2 O *Scanner* como um Autômato Finito Determinístico

O processo de reconhecimento do texto do código precisa de mecanismos capazes de capturar a relação entre os caracteres para a formação de um *token*. Isso pode ser feito usando **Autômatos Finitos Determinísticos (AFD)**. Um AFD pode ser descrito, empiricamente, como um jogo de tabuleiro. 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** (arestas direcionais entre os estados). Cada aresta é rotulada com um caractere ou classe de caracteres que precisam ser *processados* para que aquele caminho seja percorrido.

Trazendo esse conceito para a nossa implementação, o autômato finito determinís-

tico projetado para o *Micro C* reconhece *identificadores* seguindo uma regra léxica estrita, ou seja, eles devem iniciar *obrigatoriamente* com uma letra ou sublinhado (*underline*), podendo ser seguidos por uma sequência de letras, dígitos ou sublinhados.

A Figura 2.2 ilustra o processo de reconhecimento. Ele se inicia 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 é um *token* válido.

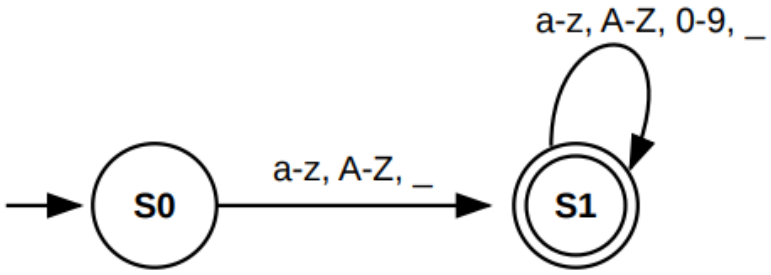
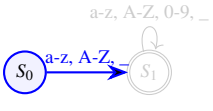


Figura 2.2: Autômato Finito Determinístico para reconhecimento de Identificadores.

A partir de S_1, existe uma transição de laço (*loop*) que permite ao autômato processar 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`.

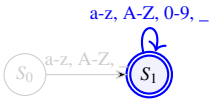
Para demonstrar essa mecânica passo a passo, considere o trecho de código: `int x1;`. Suponha que o *scanner* já tenha processado a palavra reservada `int` e o espaço em branco, e agora o cursor de leitura está posicionado no caractere `'x'`. Vamos acompanhar o autômato em ação. Na direita você pode acompanhar o estado atual do autômato:

$$[S_0] \xrightarrow{\text{lê 'x'}} \texttt{int}\ \underbrace{\texttt{x}}_{\textbf{lendo}}\ \texttt{1;}$$



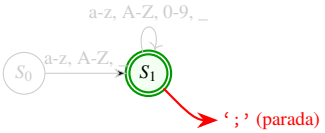
Passo 1: O autômato inicia no estado S_0. O caractere lido na fita de entrada é a letra `'x'`. Como a regra dita que variáveis podem começar com letras, existe uma transição válida de S_0 para S_1. O autômato processa o caractere `'x'` e avança para S_1.

$$[S_1] \xrightarrow{\text{lê `1'}} \texttt{int x } \underbrace{\texttt{1}}_{\textbf{lendo}} \texttt{ ;}$$



Passo 2: Agora no estado de aceitação S_1, o próximo caractere lido é o dígito `'1'`. O estado S_1 possui uma transição de laço (*loop*) que aceita letras ou dígitos adicionais. O autômato processa o `'1'` e permanece em S_1.

$$[S_1] \xrightarrow{\text{lê `;'}} \texttt{int x1 } \underbrace{\texttt{;}}_{\textbf{lendo}}$$



Passo 3 (Conclusão): O caractere lido é o ponto e vírgula (`;`). Estando em S_1, não há transição válida no AFD de identificadores para símbolos de pontuação. O processamento para. Como o autômato encerrou a leitura posicionado em S_1 (um estado de aceitação), a sequência `x1` é validada com sucesso e classificada como um *token* `ID`. O cursor do compilador então se prepara para avaliar o `';'` no próximo ciclo.

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 processar 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, existe **apenas 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 [4], além de gerar como saída o código em C que implementa o AFD. Não é o caso da 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, nós fizemos a implementação do *scanner* (`scanner.c`) manualmente para deixar o código do compilador do *Micro C* mais simples, independente e direto. Isso permite que o compilador fique, embora limitado a linguagem, 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()`, `case '=':`, etc.). 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).

Contudo, existem situações em que um token pode ter representações que seriam ambíguas entre si. Vejamos o exemplo a seguir que 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* (observando o próximo caractere antes de tomar uma decisão) 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 processar 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 dos *tokens* (arquivo `tokens.h`), possibilitam iniciar a implementação do analisador léxico. Nesta seção faremos uma análise do arquivo `scanner.c`.

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 é acessível em qualquer parte do compilador.

Essa interface é simples e possui apenas duas ações possíveis:

- `inicializar_scanner(const char* nome_arquivo)`: abre de um arquivo;
- `proximo_token()`: busca o próximo *token*.

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, significa 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: processar 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 a implementação do **autômato finito determinístico** do *Micro C*, o seja, é uma máquina de estados.

Ela 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`, 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 primeiro como um `ID`, 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

Vamos analisar agora a estrutura do arquivo `scanner.c` e as soluções implementadas para desafios práticos encontrados durante o processamento do código-fonte pelo analisador léxico.

Ruído no 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 é processar todos os espaços em branco e quebras de linha:

```
// arquivo: src/scanner/scanner.c (trecho 1)
static int ignorar_espacos_e_comentarios(void) {
    int c = prox_char();
    while (1) { //ignora espaços em branco
        while (c != EOF && isspace((unsigned char)c)) {
            if (c == '\n') linha_atual++;
            c = prox_char();
        }
//...
```

A função `isspace()` (da biblioteca `ctype.h`) é usada para processar 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 == '/') {
            int seguinte = prox_char();
//...
```

Ao verificar que o caractere é uma barra (`/`), o *scanner* precisa decidir se temos um comentário ou uma operação de divisão. Essa decisão depende do próximo caractere. Para isso, ocorre a avaliação do 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. 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 estado para capturar comentários em bloco.

```
//... (continuação)
            if (seguinte == '/') { // comentário de linha
                while ((c = prox_char()) != '\n' && c != EOF) { }
            } else if (seguinte == '*') { // comentário de bloco
                int anterior = 0;
                c = prox_char();
                while (c != EOF && !(anterior == '*' && c == '/')) {
                    if (c == '\n') linha_atual++;
                    anterior = c;
                    c = prox_char();
                }
```

A implementação atual usa `int` em `prox_char`, `c` e `seguinte`, preservando corretamente o valor especial `EOF`. Ao atingir o fim do arquivo dentro de um comentário de bloco, ela define `erro_comentario`. `proximo_token` transforma esse estado em um erro léxico com a mensagem "Comentario de bloco nao terminado".

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;
        }
    }
}
```

Notem que o `else` neste momento é muito importante. Se `next_c` não era `/` nem `*`, o *scanner* descobriu que se tratava então do operador de divisão. Nesse cenário, 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

O padrão formal para um identificador na linguagem é *uma letra ou underscore, seguido por zero ou mais letras, dígitos ou underscores*. Usando Expressões Regulares, definimos este *token* como:

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

Na prática, um dos desafios centrais do *scanner* é diferenciar identificadores que obedecem a essa regra (ex: `contador`) de palavras reservadas (ex: `if`), que também a obedecem. A abordagem utilizada no *Micro C* segue a estratégia clássica: 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 e corrige a definição.

Primeiro, o *consumidor* de identificadores atua. A verificação `if (isalpha(c) || c == '_')` garante que o caractere atual obedece à primeira parte da nossa Expressão Regular (o início válido). Em seguida, o laço `do-while` age como o fecho de Kleene (`*`), construindo o lexema no *buffer* enquanto os caracteres seguintes forem alfanuméricos ou `_`. O laço para ao ler um caractere que não pertence ao padrão, que é devolvido ao fluxo via `ungetc()`.

```
// 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';
//...
```

Com o lexema construído e devidamente armazenado no *buffer*, o analisador avança para a etapa de *classificação*. O autômato já garantiu que a sequência forma um identificador sintaticamente válido, restando verificar se ela é uma palavra reservada. Para manter a simplicidade didática, o *Micro C* resolve esse conflito com um bloco condicional sequencial e chamadas à função `strcmp`. Em compiladores de produção, o grande número de palavras reservadas torna essa busca um gargalo de desempenho.

A estratégia padrão consiste em utilizar Tabelas Hash para pré-inicializar essas strings, garantindo uma classificação rápida com complexidade de tempo constante $O(1)$ [1]. Ferramentas como o `gperf` também são frequentemente empregadas para gerar funções de *hashing* perfeito para essa finalidade [11].

Observe a implementação desta etapa no nosso analisador:

```
//... (continuação: O Classificador)
    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 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*

Para lidar com operadores que compartilham prefixos, como = e ==, o analisador utiliza a técnica de *lookahead* (olhar à frente). O trecho de código a seguir mostra como o *lookahead* simples 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 o padrão: o *scanner* lê o caractere =, mas *observa antecipa-*

damente (`prox_char()`) o seguinte. Se for outro `=`, ele gera o *token* `EQ`. Caso contrário, ele devolve o caractere (`ungetc()`) e gera `ASSIGN`.

Constantes Inteiras (INTEGERCONST)

Um `INTEGERCONST` é formado somente por um ou mais dígitos. O scanner sempre emite `MINUS` separadamente; a função `unary_expression` do parser decide se o operador representa negação aritmética. Portanto, `n-1`, `n - 1` e `-123` são analisados sem dependência de espaços.

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

O reconhecimento léxico é realizado por `ler_inteiro`. A função consome apenas dígitos, verifica o limite configurado e devolve ao fluxo o primeiro caractere que não pertence ao literal:

```
static Token ler_inteiro(int primeiro, int linha) {
    Token token = criar_token();
    size_t tamanho = 0;
    int c = primeiro;
    while (c != EOF && isdigit((unsigned char)c)) {
        if (tamanho + 1 < sizeof(token.lexema))
            token.lexema[tamanho] = (char)c;
        tamanho++;
        c = prox_char();
    }
    devolver_char(c);
    if (tamanho > (size_t)(MAX_INT_LEN - 2))
        return token_erro("Constante inteira excede o limite", linha);
    token.lexema[tamanho] = '\0';
    token.tipo = INTEGERCONST;
    return token;
}
```

Não há mais duas rotas léxicas para `-`. O caso correspondente no `switch` sempre produz `MINUS`; a distinção entre subtração binária e negação unária pertence à sintaxe.

Reconhecimento de Literais de Texto (Strings e Caracteres)

Uma string começa e termina com aspas duplas, não pode atravessar uma quebra de linha e admite os escapes `\n`, `\t`, `\\`, `\'`, e `\"`. O escape `\0` é rejeitado em strings porque introduziria um terminador nulo dentro do lexema.

```
"([^"\\\n]|\\[nt\\'"])*"
```

A função `ler_string` percorre o literal, chama `decodificar_escape` depois de cada barra invertida e grava no token os bytes já interpretados. Ela produz diagnósticos distintos para escape inválido, excesso de tamanho e string não terminada.

```
while (c != '"' && c != '\n' && c != EOF) {
    int valor = c;
    if (c == '\\') {
        int valido = 1;
        valor = decodificar_escape(prox_char(), &valido);
        if (!valido || valor == '\0')
            return token_erro("Escape invalido em string", linha);
    }
    if (tamanho + 1 >= MAX_STR_LEN)
        return token_erro("String excede o limite", linha);
    token.lexema[tamanho++] = (char)valor;
    c = prox_char();
}
```

Após o laço, a função aceita o token apenas se encontrou as aspas de fechamento. Uma quebra de linha ou `EOF` produz `UNDEF` com "String nao terminada".

2.5.4 A Interação com as Fases Subsequentes

O analisador léxico fornece a entrada da análise sintática. Essa interação pode ocorrer sob demanda, quando o parser solicita um *token* por vez, ou em lote, quando todos os *tokens* são produzidos antes da análise seguinte. O *Micro C* adota a segunda abordagem: os elementos são armazenados em um vetor dinâmico do tipo `TokenList`, mantido em memória e compartilhado diretamente com o parser.

No *Micro C*, a análise **léxica é executada integralmente antes da etapa seguinte**. A função `executar_analise_lexica()` inicia um laço de repetição que chama `proximo_token()` até encontrar o *token* `END_OF_FILE`. Cada *token* gerado é armazenado em uma estrutura de lista na memória (`TokenList`). Uma vez que todos os tokens foram extraídos e o arquivo foi totalmente processado, a estrutura `TokenList` é enviada para o analisador sintático (*parser*).

A estratégia em lote também facilita a inspeção da fase. O comando `make scan` executa o compilador no modo `-scan`, imprime a lista completa e salva uma cópia em `tokens.txt`. Esse arquivo é um artefato de depuração: na execução contínua, as fases

seguintes consomem a estrutura `TokenList` que permanece em memória, e não releem `tokens.txt`.

O trecho de código a seguir ilustra essa lógica no *Micro C*:

```
// arquivo: src/main.c

// 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);
```

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, como o uso da abordagem *em lote* ao invés de outras estratégias, que são comuns e, normalmente, utilizadas em compiladores comerciais.

Em seguida, discutiu-se a implementação do vocabulário (arquivo `tokens.h`) da linguagem e como os tipos de tokens são processados. A estrutura da forma como foi proposta e implementada, permite que erros comuns sejam informados ao programador apontando em qual linha de código o erro ocorreu.

Por fim, foram abordados conceitos importantes que foram utilizados na implementação do `scanner.c` como o uso de expressões regulares e autômatos finitos determinísticos. Isso tornou o *scanner* mais simples, mas ao mesmo tempo, dependente da linguagem definida.

2.7 Exercícios

Exercício 2.1 Explique a diferença entre token, lexema e padrão. Dê um exemplo de como esses três elementos aparecem ao processar a linha de código `int x = 10;`. ■

Exercício 2.2 Em muitas linguagens, a string 'if' pode ser tanto uma palavra reservada quanto o início de um identificador (como `if_contador`). Como o analisador léxico resolve esse conflito de *maior prefixo*? ■

Exercício 2.3 Por que as Expressões Regulares são a ferramenta de escolha para definir a estrutura léxica de uma linguagem? Quais são as limitações das expressões regulares nesse cenário? ■

Exercício 2.4 Descreva o processo de conversão de uma Expressão Regular para um AFD (Autômato Finito Determinístico). Por que, na prática, prefere-se utilizar AFDs em vez de AFNs (Autômatos Finitos Não-Determinísticos)? ■

Exercício 2.5 Compare as vantagens e desvantagens de utilizar uma ferramenta geradora de analisadores léxicos (como Lex ou Flex) em relação à escrita manual de um analisador baseado em um comando `switch-case` ou tabelas de transição. ■

Exercício 2.6 Como e em que momento o analisador léxico interage com a Tabela de Símbolos? É responsabilidade do analisador léxico inserir todos os identificadores nela ou apenas identificá-los? Justifique. ■

Exercício 2.7 O que caracteriza um erro léxico e quais são as estratégias de recuperação comuns (como o "modo pânico") que um analisador pode adotar para continuar o processamento sem interromper a compilação no primeiro caractere inválido? ■

Exercício 2.8 Discuta o impacto de linguagens que utilizam a indentação como parte da sintaxe (como Python) na fase de análise léxica. Como o analisador deve lidar com espaços em branco e quebras de linha nesses casos específicos? ■

Exercício 2.9 O analisador léxico lê o código-fonte caractere por caractere. Explique a técnica de *lookahead* e como o seu uso evita ambiguidades. ■

2.8 Problemas

Problemas que serão implementados em cada fase do compilador para estendê-lo.

Problema 2.1 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo booleanas que suportam dois tipos de valores: `true` ou `false`.

Problema 2.2 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo flutuante (com casas decimais após a vírgula).

Dica de Implementação

Para resolver estes problemas, você precisará alterar apenas dois arquivos: `tokens.h` (para definir as novas categorias de *tokens* na enumeração) e `scanner.c` (para ensinar o analisador a reconhecer os novos padrões).



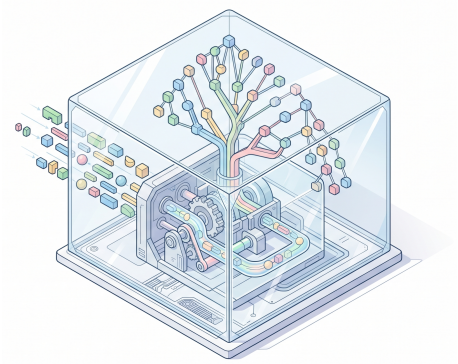
3. Análise Sintática

Trees sprout up just about everywhere in computer science.

– Donald E. Knuth

Neste capítulo será discutida a construção do analisador sintático do *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, tudo isso alinhado com uma gramática previamente definida.



Nesta fase da compilação é que a estrutura do programa começa a ganhar forma. Além de validar a gramática, está fase é responsável por *construir* uma estrutura de dados hierárquica que será usada por todas as fases subsequentes, conhecida como Árvore Sintática Abstrata (ASA). Está árvore é o resultado da análise sintática e serve como a representação da lógica do programa que será usada na análise semântica no Capítulo 4.

O analisador sintático recebe uma cadeia de *tokens* e verifica se ela pertence à linguagem definida pela gramática. Ao encontrar uma construção inválida, deve produzir um diagnóstico com o *token* encontrado, o elemento esperado e a linha aproximada. Compiladores de produção costumam tentar recuperar-se para relatar múltiplos erros, mas o parser didático do *Micro C* interrompe a execução no primeiro erro sintático.

Existem três abordagens principais para a construção de analisadores sintáticos: a abordagem *universal*, a *descendente* e a *ascendente* [1, 6, 5]. A abordagem universal é capaz de analisar qualquer gramática, mas seus algoritmos, como o **algoritmo de Earley [7]**, são pouco eficientes. Eles precisam explorar múltiplas árvores de derivação possíveis, um processo computacionalmente custoso (complexidade $O(n^3)$ em relação ao tamanho do programa) e, por isso, raramente são usados em compiladores de produção. 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. Veremos nas próximas seções como essas técnicas funcionam e como o analisador sintático trata erros e 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. Vamos explicar isso adiante.

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 que expressam essa hierarquia: `E` (para Expressão), `T` (para Termo) e `F` (para Fator). Esses símbolos

são também conhecidos pelo fato de não serem terminais, ou seja, servem para expandir expressões. Esse é um padrão clássico no projeto de compiladores garantindo, por exemplo, *a avaliação na ordem correta de operações matemáticas*. A estrutura lógica é construída da maior para a menor precedência. A seguir uma descrição de cada um desses níveis hierárquicos:

- **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]: $F \rightarrow (E) \mid \text{id} \mid \text{num}$.
- **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 na avaliação de expressões matemáticas.

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.

[1] A leitura desta notação, *adotada ao longo do livro*, 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.

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, a gramática é essencial para que o compilador possa interpretar corretamente o código fonte e avançar para as etapas seguintes da compilação. A definição completa da gramática aceita pelo *Micro C* está no Apêndice B.

Para ilustrar o papel da gramática, considere uma linguagem simples para definir expressões aritméticas e que permite somar números e agrupar operações entre parênteses. A linguagem está descrita a seguir 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. Agora vejamos a gramática completa para operações aritméticas, como as do exemplo anterior:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T \div F \mid F$$
$$F \rightarrow (E) \mid \textbf{id} \mid \textbf{num}$$

Embora a gramática acima pareça expressar de forma natural as 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:

$$E \rightarrow E + T$$

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:

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

Durante a execução do exemplo, acima vimos um novo símbolo, o $\Rightarrow$ (seta dupla). Este símbolo representa um *passo de derivação*, enquanto $\rightarrow$ (seta simples), uma produção estática de uma regra. Ou seja, a seta dupla 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:

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

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:

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid -TE' \mid \varepsilon \\
T &\to FT' \\
T' &\to *FT' \mid \div FT' \mid \varepsilon \\
F &\to (E) \mid \textbf{id} \mid \textbf{num}
\end{aligned}
$$

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:

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

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

$$E' \Rightarrow +TE'$$

O T seguinte processa $b * c$ por meio de $T \to FT'$ e $T' \to *FT'$. Ao final, a derivação $E' \Rightarrow \varepsilon$ encerra a expansão sem consumir outro *token*, pois ε representa a cadeia vazia. Assim, os operadores são agrupados segundo a precedência desejada e o analisador não entra em recursão infinita.

Contextualização da Implementação: Estruturas e Fluxo

Para compreender a implementação da análise sintática no *Micro C* é necessário entender primeiro como o analisador é estruturado e quais dados ele manipula. O *Micro C* utiliza a

abordagem **descendente recursiva**, 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.

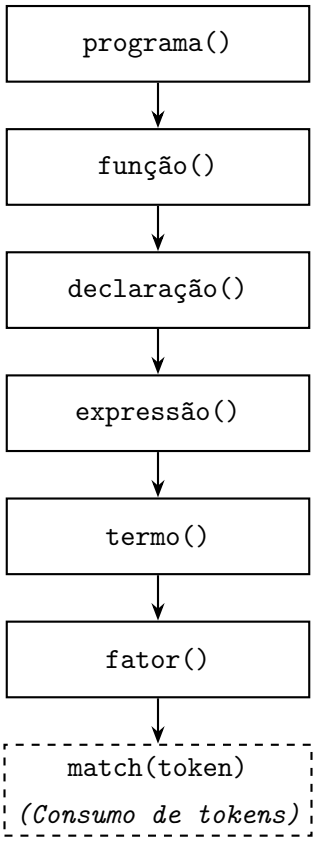


Figura 3.1: Fluxo de chamadas do Analisador Descendente Recursivo.

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). Por outro lado, a Árvore Sintática Abstrata (ASA) é efetivamente construída ***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 agrupados por nós operadores (pais) nas funções superiores. Esse processo é construído usando duas estruturas de dados:

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` e um ponteiro para o seu `proximo_irmao`. Isso permite que um nó pai tenha quantidades variáveis de filhos.

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* e um cursor para o elemento atual. Vejam o trecho de código a seguir retirado do compilador:

```
// --- 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;
```

Uma vez que conhecemos as estruturas, podemos analisar a implementação da função `arithmetic_expression`. No *Micro C*, realiza-se uma otimização em relação à gramática. É essencial distinguir que, ao eliminar a recursão à esquerda (que causa *loops* infinitos), a gramática 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 executando enquanto houver operadores ($+, -$), simulando a lógica 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

Erros sintáticos são detectados quando a sequência de *tokens* não corresponde a uma construção válida no ponto corrente da gramática. A implementação do *Micro C* encerra a análise no primeiro erro por meio da função `error`; as estratégias descritas a seguir são alternativas apresentadas para contextualização e não fazem parte do parser atual.

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;
```

O exemplo acima também contém entradas que podem falhar ainda na análise léxica, antes de alcançar o parser. Um exemplo estritamente sintático seria `int a = ;`. Em um parser com modo de pânico, os *tokens* poderiam ser descartados até o ponto e vírgula, usado como símbolo de sincronização. O *Micro C* não realiza esse descarte, ele informa o fator ausente e encerra a compilação.

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` $\Rightarrow$ `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 gramática é incrementada com produções para reconhecer construções incorretas comuns de erros dos programadores. Quando uma dessas produções ocorre, uma mensagem de erro personalizada pode ser gerada.

Exemplo: Se a gramática original não reconhece uma entrada como **abcd**, podemos introduzir 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 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.

A verificação de tipos não é uma estratégia de recuperação sintática. Ela pertence à análise semântica, apresentada no capítulo seguinte. Além disso, a versão atual do *Micro C* não possui literais de ponto flutuante nem a coerção implícita ilustrada anteriormente. Por isso, esse exemplo não representa uma entrada válida da linguagem documentada.

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. 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 como os *tokens* são armazenados e oferecidos ao analisador sintático. Na execução normal, o scanner adiciona cada elemento a um vetor dinâmico `TokenList`; o parser recebe um ponteiro para esse mesmo vetor e o percorre por índice. O arquivo `tokens.txt` é gerado para inspeção didática e depuração. A função `carregar_tokens` existe como utilitário, mas não é utilizada pelo fluxo principal definido em `main.c`.

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*.

```
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 = '%99[^']', 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; }
            lista->tokens = nova_lista;
        }
        //armazena o token na lista
        lista->tokens[lista->tamanho].tipo = (TokenType)tipo;
        snprintf(lista->tokens[lista->tamanho].lexema,
                 sizeof(lista->tokens[lista->tamanho].lexema),
                 "%s", lexema);
        lista->tokens[lista->tamanho].linha = linha_num;
        lista->tamanho++;
    } fclose(arquivo); return 0;
}
```

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 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 estrutura `TokenList` mantida em memória fornece ao parser uma entrada ordenada, sem que ele precise repetir o reconhecimento léxico. O arquivo textual contém apenas uma cópia desse estado para inspeção. A interface efetiva entre as fases é o vetor de estruturas `Token`, cujos elementos preservam tipo, lexema e linha.

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.

Cada construção sintática possui correspondência direta com uma função do *parser*. Na implementação atual, `unary_expression()` fica entre `term()` e `factor()` e reconhece recursivamente `-` e `!`.

Programa e funções: `program()`, `function()` e `param_list()`.

Declarações e comandos: `var_declaration()`, `statement()`, `assign()`, `if_statement()`, `for_statement()` e `print_statement()`.

Expressões: `expression()`, `logical_and_expression()`, `relational_expression()`, `arithmetic_expression()`, `term()`, `factor()` e `arg_list()`.

3.4.1 Notações de Gramática

Para evitar ambiguidades e facilitar a leitura, nós adotamos convenções simples 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*:
   - `programa` → representa o ponto de entrada do compilador, contendo uma sequência de funções.
   - `funcao` → definição completa de função, incluindo tipo de retorno, nome, parâmetros e corpo.
   - `tipo` → especificador de tipo (`int` ou `char`).
   - `var_declaracao` → declaração de variáveis locais ou parâmetros.
   - `expressao`, `expressao_aritmetica`, `termo`, `fator` → 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.
   - `declaracao` → 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 `programa`.

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 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 definidas pela gramática, confirmando que o programa obedece à estrutura sintática esperada. Caso a sequência não se encaixe nas regras, o analisador sintático gera 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.

Exemplo:

Considere a expressão aritmética a seguir:

$$3 + (5 + 2)$$

Suponha que a gramática que avalia expressões aritméticas seja a seguinte:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T \div F \mid F$$
$$F \rightarrow (E) \mid \textbf{id} \mid \textbf{num}$$

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.

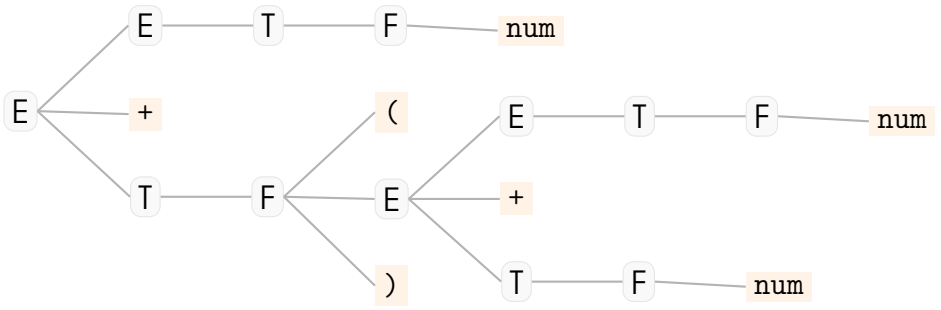


Figura 3.2: Árvore Sintática para a expressão `3 + (5 + 2)`.

Na Figura 3.2, 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. Dessa forma, é possível observar que a gramática orienta e guia o analisador sintático e a construção da árvore sintática. Isso estabelece uma relação entre as regras da linguagem e a representação 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 [3]), 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, 6, 4], especialmente porque muitas construções em linguagens de programação não exigem essa complexidade.

Para ilustrar, considere a regra $S \rightarrow abc \mid abd$ e a entrada `a b d`. O analisador tentaria primeiro *abc*. Ao falhar no `c`, ele é obrigado a desfazer a leitura anterior e tentar *abd*. Em gramáticas reais, esse retrocesso gera uma lentidão inaceitável para o compilador.

Exemplo:

Considere a seguinte gramática:

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

Suponha a cadeia de entrada: **x w z**. Ela será analisada utilizando a análise sintática descendente. Inicialmente, a descida começa do símbolo inicial S e aplica-se a produção $S \rightarrow xBz$. Em seguida, tenta-se expandir o não terminal B para que ele reconheça o símbolo intermediário da entrada.

Neste ponto, testa-se a primeira alternativa para B, que é $B \rightarrow y$. Como o símbolo y não corresponde ao próximo símbolo da entrada, que é `w`, ocorre um erro (Figura 3.3).

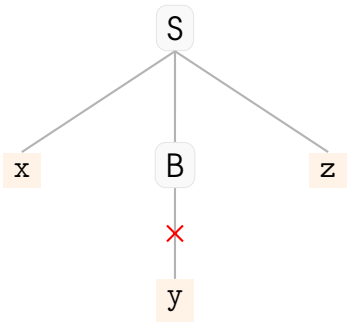


Figura 3.3: Árvore de derivação da gramática falhando ao avaliar parte da expressão.

O analisador então retrocede, retornando ao ponto em que B foi invocado e tenta a próxima alternativa. Agora, ao utilizar a derivação $B \rightarrow w$, ocorre uma validação bem-sucedida com a entrada, permitindo o avanço para o símbolo final z, completando a derivação com sucesso (Figura 3.4).

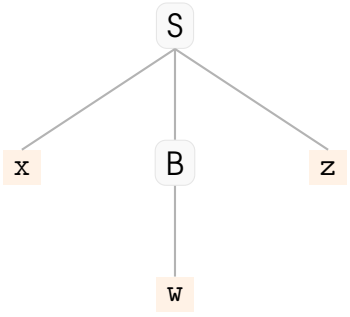


Figura 3.4: Árvore de derivação da gramática aceita.

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 processar nenhum símbolo da entrada, o *parser* pode entrar em um *loop* infinito, impedindo que a análise avance.

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 processar 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 descendentes preditivos, os conjuntos **FIRST** e **FOLLOW** permitem decidir qual produção aplicar a partir do próximo símbolo da entrada. Eles também aparecem como ferramentas auxiliares em outros métodos, mas sua aplicação direta neste capítulo é a construção e a compreensão de parsers LL.

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 útil em momentos de recuperação de erro e no tratamento de produções que podem derivar na 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 $\texttt{FOLLOW}(\text{S})$, em que 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 $\texttt{FIRST}(X) = \{X\}$.
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 $\texttt{FIRST}(Y_1)$ em $\texttt{FIRST}(X)$;
   - Se $\varepsilon \in \texttt{FIRST}(Y_1)$, então também inclua os símbolos não-ε de $\texttt{FIRST}(Y_2)$, e assim por diante;

- Se $\varepsilon \in \texttt{FIRST}(Y_i)$ para todo $i = 1, \ldots, k$, então $\varepsilon \in \texttt{FIRST}(X)$.

3. Se $X \to \varepsilon$ é uma produção, então $\varepsilon \in \texttt{FIRST}(X)$.

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

1. Inclua $ em FOLLOW(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 FOLLOW(B).
3. Se $\varepsilon \in \texttt{FIRST}(\beta)$ 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 Cálculo dos Conjuntos

Vamos avaliar 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.

Exemplo:

Considere a seguinte gramática sem recursão à esquerda:

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

As representações 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) = { *, +,), $ }

Ou seja, 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)

Para verificar se uma gramática admite um parser preditivo, utilizamos os conjuntos **FIRST** e **FOLLOW**. Eles permitem avaliar a complexidade da gramática e confirmar se ela pode ser analisada de forma eficiente.

Um parser preditivo escolhe a produção com base em uma quantidade limitada de *tokens* de antecipação, sem precisar desfazer decisões anteriores. Parsers com retrocesso tentam uma alternativa e retornam a um ponto anterior caso ela falhe; essa estratégia pode aumentar o custo e dificultar os diagnósticos, razão pela qual não foi adotada no *Micro C*.

Para que o parser seja preditivo sem retrocesso, a gramática deve permitir uma escolha inequívoca com o *lookahead* disponível. Em LL(1), o primeiro "L" indica leitura da esquerda para a direita, o segundo indica derivação mais à esquerda e o "1" informa que cada decisão examina um único *token* da entrada, não um caractere do arquivo-fonte.

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 $\texttt{FIRST}(\alpha) \cap \texttt{FOLLOW}(A) = \emptyset$

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 do `else`:

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

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:

$$E \to TE'$$
$$E' \to +TE' \mid \varepsilon$$
$$T \to \textbf{id}$$

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 $E' \to \varepsilon$.

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**, em que 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 do inglês, *Abstract Syntax Tree* (AST). A ASA é o principal resultado da análise sintática e é a estrutura de dados base para todas as fases subsequentes na compilação

Diferente da árvore de derivação, a ASA condensa a estrutura do programa, removendo nós intermediários e *tokens* que não carregam significados considerados essenciais, 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 do tipo `ASTNode` no arquivo `ast.h`. Cada nó na árvore representa um conceito gramatical como, por exemplo, uma declaração, uma expressão ou um laço, e precisa de uma *etiqueta* para que o compilador saiba o que ele representa. Esta etiqueta é utilizada nas fases

subsequentes (como a análise semântica e a geração de código) 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 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` 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_UNARY_OP,          // Operação unária (ex: -, !)
    NODE_ID,                // Um identificador
    NODE_INTEGER_CONST,     // Uma constante inteira
    //... outros tipos de nós
} NodeType;
```

Além da estrutura sintática, `ASTNode` contém `symbol`, preenchido pela análise semântica com o símbolo resolvido, e `inferred_type`, que registra o tipo calculado das expressões. Esses campos evitam novas buscas e transportam informação validada para a geração da RI (Representação Intermediária) do código antes da geração do Assembly.

A forma como os nós são conectados é um detalhe de implementação importante. Em vez de cada nó ter um número fixo de filhos (tornando a implementação menos flexível), 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. A Figura 3.5 mostra como as instruções são encadeadas em um bloco.

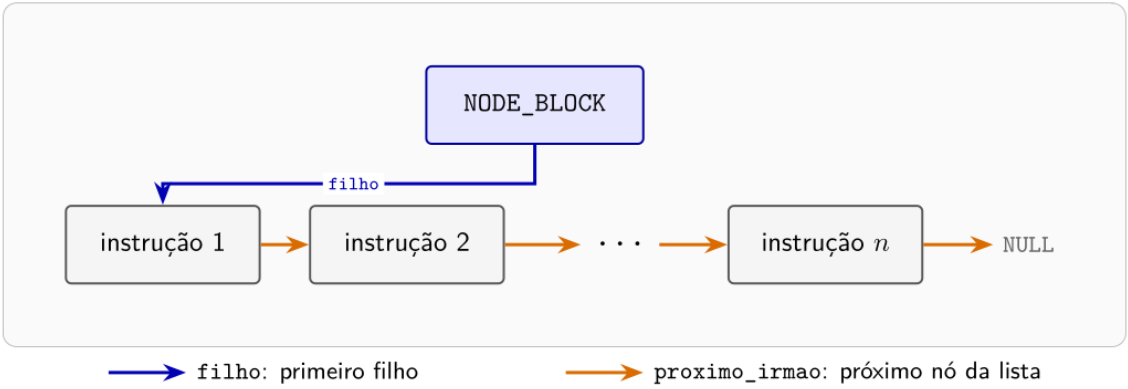


Figura 3.5: Nó com quantidade variável de filhos.

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;
    void* symbol;
    TokenType inferred_type;

    union {
        long int_value;
        char char_value;
        char* string_value;
        TokenType op_type; // operações binárias/unárias e tipos
    } 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;
    no->symbol = NULL;
    no->inferred_type = UNDEF;
    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.

Veja o trecho de código que implementa a função `adicionar_filho`:

```
//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;
    } ...
```

```
//arquivo: src/ast/ast.c (trecho)
   ... 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 trata se o *token* atual é o esperado:

```
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` compara o tipo de `parser->current_token` com o tipo esperado. Quando há correspondência, ela chama `advance`, que seleciona o próximo elemento do vetor de entrada; caso contrário, produz o erro sintático.

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*

Cada vez que um *token* esperado é consumido, o analisador sintático precisa avançar na sequência léxica. A função `advance` incrementa o índice mantido pelo parser e faz `current_token` apontar para o elemento correspondente no vetor dinâmico `TokenList`.

A implementação abaixo reproduz o código atual e também trata o fim do vetor, atribuindo `NULL` ao ponteiro quando não há outro elemento:

```
void advance(Parser *parser) {
    if (parser->current < parser->lista->tamanho) {
        parser->current++;
        parser->current_token = parser->current < parser->lista->tamanho
            ? &parser->lista->tokens[parser->current]
            : NULL;
    } else {
        parser->current_token = NULL;
    }
}
```

A função `advance` incrementa o índice `current` e atualiza `current_token` para o endereço do próximo elemento do vetor `lista->tokens`. Quando o índice alcança `tamanho`, o ponteiro passa a `NULL`. Não existe um campo `proximo` na estrutura `Token`; a sequência é armazenada de forma contígua em um vetor dinâmico.

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.

O final da entrada é representado por um *token* `END_OF_FILE`. Depois de consumi-lo, o índice alcança o tamanho do vetor e `current_token` torna-se `NULL`, evitando acessos além da área alocada.

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. Além disso, 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 é a análise de significado (semântica).

3.10 Exercícios

Exercício 3.1 Explique o papel do analisador sintático (parser) no processo de compilação e como ele se diferencia do analisador léxico. ■

Exercício 3.2 O que é uma Gramática Livre de Contexto (GLC) e por que ela é utilizada para especificar a sintaxe de linguagens de programação? ■

Exercício 3.3 Diferencie símbolos terminais de símbolos não terminais em uma gramática, fornecendo exemplos de cada um no contexto do *Micro C*. ■

Exercício 3.4 Dada a gramática $E \rightarrow E + T | T$, demonstre por que ela é considerada recursiva à esquerda e qual o problema que isso causa em *parsers* descendentes. ■

Exercício 3.5 Realize a eliminação da recursão à esquerda da gramática de expressões aritméticas clássica e apresente a nova gramática equivalente adequada para análise descendente. ■

Exercício 3.6 Defina o que é um passo de derivação e explique a diferença entre os símbolos $\rightarrow$ (seta simples) e $\Rightarrow$ (seta dupla). ■

Exercício 3.7 Descreva a hierarquia de precedência clássica em expressões (Fator, Termo e Expressão) e como a gramática força essa ordem de avaliação. ■

Exercício 3.8 O que são os conjuntos FIRST e FOLLOW e qual a importância deles para a construção de um analisador sintático preditivo? ■

Exercício 3.9 Explique as condições necessárias para que uma gramática seja classificada como $LL(1)$. ■

Exercício 3.10 Como o analisador sintático lida com o problema da ambiguidade durante a derivação da árvore? ■

Exercício 3.11 Diferencie a Árvore de Derivação (Sintática) da Árvore Sintática Abstrata (ASA). Por que a ASA é preferida para as fases posteriores da compilação? ■

Exercício 3.12 Explique o funcionamento da estratégia de recuperação de erros conhecida como *modo de pânico* na análise sintática. ■

Exercício 3.13 No contexto do *Micro C*, como as funções *match* e *advance* colaboram para o reconhecimento de tokens durante a análise? ■

Exercício 3.14 Descreva a estrutura de um nó da ASA e como o enumerador NodeType auxilia o compilador a identificar construções gramaticais. ■

Exercício 3.15 Exercício Prático (Derivação): Utilizando a gramática transformada (sem recursão à esquerda) para expressões, realize a derivação passo a passo da entrada **a + b * c**. ■

3.11 Problemas

Problemas que serão implementados em cada fase do compilador para estendê-lo.

Problema 3.1 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo booleanas que suportam dois tipos de valores: `true` ou `false`.

Problema 3.2 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo flutuante (com casas decimais após a vírgula).

Dica de Implementação

Para resolver estes problemas, será necessário alterar três arquivos: `ast.h` (para expandir a estrutura da Árvore Sintática Abstrata com os novos tipos de dados), `parser.c` (para adequar as regras gramaticais ao reconhecimento das novas declarações e literais) e `main.c` (para atualizar a rotina responsável pela impressão visual da árvore).



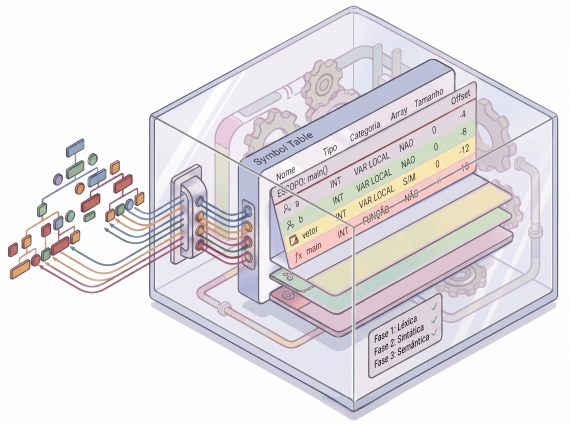
4. Análise Semântica

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 fase relacionada 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 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, o que resulta em um erro pelo analisador semântico.

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 $x \div y$, o compilador não pode garantir que o valor de `y` nunca será zero durante a execução.
- **Desreferência de ponteiro nulo:** é um exemplo clássico em linguagens que possuem ponteiros. A versão atual do *Micro C* não oferece ponteiros na sintaxe e, portanto, essa verificação não faz parte do compilador desenvolvido neste livro.

Linguagens como C e, por consequência, o *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, 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 o 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 uma operação binária: `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 minuciosa, 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 = "texto"; // Erro: atribuição de string a uma variável inteira
```

Esse trecho é sintaticamente válido: a atribuição `x = "texto";` está de acordo com a gramática da linguagem. No entanto, semanticamente ele está incorreto, pois em C não é permitido atribuir uma cadeia de caracteres diretamente a uma variável do tipo `int` sem conversão explícita. Esse erro é capturado 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.

Essa consulta também estabelece a ligação entre cada uso de um identificador e sua declaração. Como os escopos são aninhados, a busca começa no escopo ativo mais interno e prossegue em direção aos escopos externos. Desse modo, declarações locais podem ocultar nomes de escopos mais amplos, enquanto um nome ausente em todas as tabelas ativas produz o erro de variável não declarada exemplificado a seguir.

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) {
  return a + b;
}

int main() {
  soma(10); // Erro: número de argumentos incompatível
  return 0;
}
```

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 a forma. A análise semântica agora utiliza duas passagens. `registrar_funcoes` insere todas as assinaturas na tabela global antes de qualquer corpo ser examinado; em seguida, `analisar_funcao` verifica parâmetros, comandos, tipos e retornos. Assim, recursão direta e chamadas para funções definidas posteriormente são aceitas. A primeira passagem também exige `int main()` sem parâmetros.

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.

A Tabela de Símbolos precisa tratar os escopos criados por funções e blocos, permitindo declarações locais independentes e busca do escopo mais interno para o mais externo. A gramática atual não possui declaração de variável global; a tabela global armazena principalmente os símbolos das funções.

O objetivo desta seção é apresentar a organização da Tabela de Símbolos, suas

operações e sua integração com o analisador semântico. A solução usa uma pilha de tabelas para representar escopos aninhados; seus limites fixos e o gerenciamento de memória são discutidos adiante.

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;
    TabelaSimbolos *tabelas_alocadas[MAX_SIMBOLOS];
    int quantidade_tabelas;
} PilhaTabelasSimbolos;
```

`tabelas` e `topo` representam os escopos ativos. Outros dois campos registram a propriedade das tabelas criadas, inclusive das que já saíram do escopo: `tabelas_alocadas` armazena os ponteiros, e `quantidade_tabelas` informa quantos deles estão ocupados. Essa separação conserva os ponteiros usados pela ASA e pela RI e permite liberar cada alocação no encerramento do compilador.

O uso de uma pilha para gerenciar escopos é uma técnica clássica usada em compiladores [1, 2, 5], 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:

```
/* 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;
    int local_size;
    int frame_size;
} Simbolo;
```

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

Além dessas informações, símbolos de função agora possuem `local_size`, a quantidade de bytes ocupada pelas variáveis locais, e `frame_size`, o tamanho final do quadro após incluir temporários e aplicar alinhamento de 16 bytes. Variáveis e parâmetros continuam usando `memory_offset`.

Já a estrutura `TabelaSimbolos` gerencia o escopo individual por meio 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 mantém um conjunto independente de identificadores por escopo, mas não cresce dinamicamente: tanto a pilha de tabelas quanto cada vetor de símbolos possuem capacidade fixa definida por `MAX_SIMBOLOS`; assinaturas de funções também se limitam a `MAX_PARAMETROS`. A inserção deve rejeitar entradas que excedam esses valores.

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,
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 (um erro

que geraria ambiguidades de referência), o analisador semântico utiliza a função `buscar_simbolo_no_escopo_atual()`. Conforme detalhado no código abaixo, essa função realiza uma busca linear apenas na tabela que está no topo da pilha, validando os identificadores já registrados antes de prosseguir:

```
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` e `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 ||
        pilha->quantidade_tabelas >= 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_alocadas[pilha->quantidade_tabelas++] = nova_tabela;
    pilha->tabelas[++pilha->topo] = nova_tabela;
}
```

De forma simétrica, a função `desempilhar_tabela` é chamada ao sair do bloco. Na implementação atual, ela apenas decrementa o índice do topo, o que fecha logicamente o escopo e impede novas consultas aos seus símbolos. O trecho abaixo mostra exatamente esse comportamento.

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

O desempilhamento continua apenas lógico porque a ASA e a RI conservam ponteiros para símbolos. A lista `tabelas_alocadas` resolve o gerenciamento da memória.

Ao final, a função `destruir_pilha_tabelas()` percorre essa lista, libera todas as tabelas exatamente uma vez e, por fim, libera a pilha.

4.4.5 Processamento da Árvore Sintática e Validação

O analisador semântico continua percorrendo a ASA depois do parser, mas organiza o trabalho em duas passagens. A primeira registra funções globais; a segunda percorre cada corpo em profundidade, cria escopos, resolve identificadores, verifica tipos e atribui posições de memória sem reiniciar o contador nos blocos internos.

1. **Declaração de identificadores**: ao visitar nós de declaração, como `NODE_VAR_DECL`, o analisador verifica se o identificador já existe no escopo corrente antes de invocar `adicionar_simbolo()`.
2. **Uso de identificadores**: a função `buscar_simbolo_em_todos_escopos()` garante que a variável foi declarada e recupera seu tipo para a checagem de expressões e atribuições.
3. **Gerenciamento de escopos**: ao descer em um bloco ou função, o analisador invoca `empilhar_tabela()`. Depois de visitar os filhos, executa `desempilhar_tabela()`, garantindo o isolamento das variáveis locais.

A implementação foi dividida em rotinas menores. O trecho a seguir mostra o disparo das duas passagens e a verificação de retorno de uma função:

```
registrar_funcoes(raiz, pilha);
for (ASTNode* funcao = raiz->filho;
     funcao != NULL;
     funcao = funcao->proximo_irmao) {
    analisar_funcao(funcao, pilha);
}

// Dentro de analisar_funcao:
if (!instrucao_retorna(corpo)) {
    erro_semantico("Funcao pode terminar sem retornar um valor", id->linha);
}
```

A separação permite validar chamadas futuras e garante que toda função `int` ou `char` retorne um valor em todos os caminhos reconhecidos. Em um `if`, isso exige ramos `then` e `else` que retornem; em um bloco, basta alcançar uma instrução que garanta o retorno.

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 [3], sendo preferível o uso de *Tabelas Hash* com complexidade $O(1)$ [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 de *escopo* e *tipagem*.

4.5 Sumário

Neste capítulo, exploramos a terceira fase do processo de compilação, a análise semântica, responsável por verificar o significado e a coerência estática do código-fonte. Assim, embora um programa possa estar sintaticamente correto, ele ainda pode conter erros semânticos, como incompatibilidade de tipos ou uso de variáveis não declaradas. Erros de lógica, por sua vez, podem permanecer mesmo quando o programa é aceito por todas as fases do compilador.

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, foram 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 a 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*).

4.6 Exercícios

Exercício 4.1 Utilizando a analogia linguística da frase "ideias verdes incolores dormem furiosamente", explique a diferença fundamental entre a análise sintática e a análise semântica no processo de compilação. ■

Exercício 4.2 Diferencie verificação estática de verificação dinâmica. Cite dois exemplos de erros que linguagens como C (e o *Micro C*) não capturam em tempo de compilação, delegando-os para o tempo de execução. Qual é a principal justificativa técnica para essa decisão de *design*? ■

Exercício 4.3 O que são atributos e regras semânticas no contexto de um compilador? Explique como o analisador semântico determina o tipo resultante de um nó de operação binária (como `expr1 + expr2`) ao percorrer a Árvore Sintática Abstrata (ASA). ■

Exercício 4.4 Com base nos objetivos da análise semântica, liste pelo menos três tipos de erros comuns cometidos por programadores que não podem ser detectados pelo analisador sintático, mas que o módulo semântico deve obrigatoriamente capturar. ■

Exercício 4.5 Explique a lógica por trás do uso da estrutura `PilhaTabelasSimbolos` para o gerenciamento de escopos. Descreva as ações exatas que ocorrem na pilha (utilizando as funções `empilhar_tabela` e `desempilhar_tabela`) quando o compilador entra e sai do bloco de uma função. ■

Exercício 4.6 No *design* da Tabela de Símbolos do *Micro C*, qual é a diferença entre `buscar_simbolo_no_escopo_atual()` e `buscar_simbolo_em_todos_escopos()`? Especifique em quais cenários de validação (declaração de uma nova variável ou uso em uma expressão) cada função deve ser empregada e por quê. ■

Exercício 4.7 A implementação didática da Tabela de Símbolos do *Micro C* utiliza vetores com busca linear, resultando em uma complexidade assintótica $O(n)$. Qual estrutura de dados é o padrão na indústria para compiladores comerciais com o objetivo de reduzir o tempo de busca para $O(1)$? Justifique a escolha pedagógica de manter a busca linear neste projeto. ■

Exercício 4.8 O texto afirma que a Árvore Sintática Abstrata (ASA) atua como o "esqueleto" do programa, enquanto a Tabela de Símbolos serve como seu "cérebro e memória". Justifique essa metáfora explicando por que a estrutura da ASA, isoladamente, é incapaz de validar a instrução de atribuição `y = x + 5;`. ■

4.7 Problemas

Problemas que serão implementados em cada fase do compilador para estendê-lo.

Problema 4.1 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo booleanas que suportam dois tipos de valores: `true` ou `false`.

Problema 4.2 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo flutuante (com casas decimais após a vírgula).

Dica de Implementação

Para resolver estes problemas, será necessário alterar dois arquivos: `semantic.c` (para implementar as regras de checagem de tipos e compatibilidade envolvendo expressões booleanas e flutuantes) e `symbol_table.c` (para garantir o registro correto desses dados e incluir o reconhecimento dos novos tipos na impressão visual de depuração).

II Geração de Código Binário

5 Código Intermediário 105

5.1 A Transição da Análise para a Síntese
5.2 O Papel da Representação Intermediária
5.3 O Código de Três Endereços (CTE)
5.4 Gerador de Código: ASA para RI
5.5 Sumário
5.6 Exercícios
5.7 Problemas

6 Do Assembly ao Executável ... 127

6.1 A Plataforma Alvo: x86-64 e a Sintaxe AT&T
6.2 O Modelo de Memória e Execução
6.3 Tradução da RI para *Assembly*
6.4 Convenções de Chamada e *Linking*
6.5 Sumário
6.6 Exercícios
6.7 Problemas

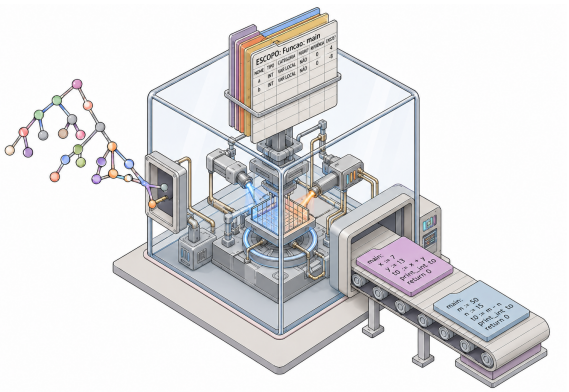


5. Código Intermediário

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 não é suficiente porque ela representa o programa em um formato hierárquico e de alto nível e não como um processador executa programas. Além disso, a análise semântica apenas confirmou que o programa *faz sentido*, mas não *como* ele deve ser executado.



Neste capítulo nós faremos 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, independentes da máquina, conhecida como Código Intermediário. Esta fase cria o elo 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-la 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 [2] 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, no ***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 lineares, explícitas e simplificadas.

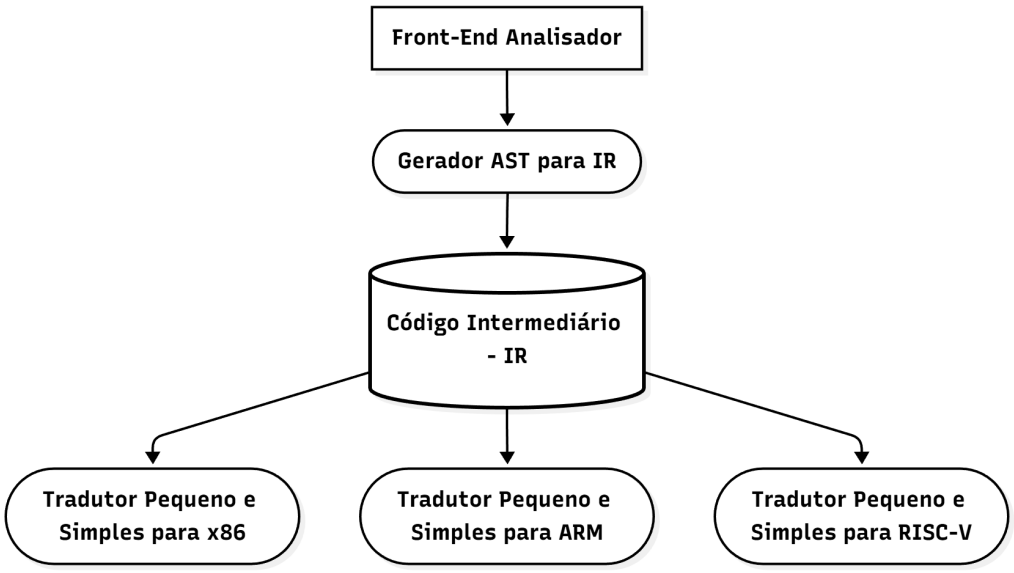


Figura 5.1: A abordagem modular, com Código Intermediário.

Dessa forma, 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.

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 [10]. Lattner *et al.* [10], 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. Essa especificação exige três componentes: 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 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])
    ...
} IROpcode;
```

Os Operandos: Os *Substantivos* da RI

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

```
// 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;
    int owns_label;
    union {
        Simbolo* symbol;
        int temp_id;
        int const_val;
        char* label_name;
    } data;
    char* string_content;
    int stack_offset;
} IROperand;
```

O campo `stack_offset` registra a posição de um `OPERAND_TEMP`. O campo `owns_label` controla a propriedade da memória dos rótulos. Por sua vez, `string_content` mantém o conteúdo associado a um rótulo de string.

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 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. Veja o trecho de código a seguir:

```
// 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 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

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.

Internamente, o mapeamento para a `struct IR_Instruction` reflete essa simplicidade: o ponteiro `result` aponta para `x` e `arg1` carrega a constante `10`. Como não há um segundo argumento nesta operação direta, o campo `arg2` permanece nulo (`NULL`), demonstrando a flexibilidade da estrutura para lidar com instruções de diferentes tamanhos.

O próximo exemplo mostra uma operação aritmética. Neste exemplo é possível observar a 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;` 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 atribuir 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 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.

Observe como a 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 posição de um vetor é decomposto em mais de uma operação. 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, acessando a informação diretamente, sem a necessidade de percorrer os elementos anteriores. Em seguida, o valor armazenado nesta posição é atribuído 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.

Diferente das expressões aritméticas, em que a execução ocorre de forma contínua, o controle de fluxo exige a quebra dessa sequencialidade. Para isso, o compilador introduz marcadores virtuais chamados de rótulos (*labels*) na lista encadeada da RI.

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:

- **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, verificar o resultado da expressão condicional `a < b`. O resultado da expressão condicional pode ser verdadeiro ou falso e 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, salte (`goto`) para o rótulo `L0`.

Por outro lado, 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`). Além disso, sempre que uma condição de estrutura condicional for inserida, é criado um **rótulo de saída**. O 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 você deve ter observado, o gerador de códigos adiciona entidades extras que não existiam no código original (variáveis temporárias e rótulos) para respeitar a estrutura definida pela CTE. Vamos ver agora dois tipos de entidades geradas com essa finalidade: as *variáveis temporárias* e os *rótulos*.

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.

O gerador ainda usa um contador para criar `t0`, `t1` e assim por diante, mas agora atribui a cada temporário um `stack_offset` logo abaixo da região das variáveis locais: `-(local_size + (temp_id + 1) * 4)`. Ao terminar a função, `frame_size` inclui locais e temporários e é arredondado para múltiplo de 16 bytes.

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 com 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 em uma Representação Intermediária (RI), 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 estruturas de dados produzidas pelo *front-end* (a ASA e a 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`.

No *Micro C*, a lógica envolvida no *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.

Dessa forma, expressões (como `a + b`) que são recursivas, podem ser aninhadas e devem sempre *retornar um valor* (ou o local onde o valor será armazenado). Por outro lado, instruções (ou *statements*, como `if` ou `for`) controlam o fluxo do programa e não retornam valores. As subseções seguintes detalham a implementação dessas funções.

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 invocou possa utiliza-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 operações como as atribuições. O exemplo mais simples 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
    return temp; // E. Retorna o local do resultado
}
```

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 (`x + 10`) está armazenado na variável `t0`.

`gerar_ir_logico` implementa curto-circuito com `IR_IF_FALSE`, rótulos e atribuições de 0 ou 1. Em `a && b`, `b` só é executado se `a` for verdadeiro; em `a || b`, `b` só é executado se `a` for falso. O backend ainda reconhece `IR_AND` e `IR_OR`, mas expressões produzidas pelo gerador seguem o fluxo com desvios.

Em chamadas, `gerar_ir_call` primeiro avalia todos os argumentos e guarda seus operandos em um vetor local. Somente depois emite a sequência `IR_PARAM` da chamada externa. Essa ordem impede que parâmetros de chamadas aninhadas sejam intercalados.

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 de **L_ELSE**) e outro para o fim da instrução `if` (o **L_FIM_IF**). No *Micro C* nós utilizamos a função `criar_operando_label_novo()` para criar nomes únicos 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 gerada. 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 gerado para *pular por cima* do bloco `else`.
5. **Tradução do Bloco 'Else':** o rótulo `L_ELSE:` é gerado, seguido pela tradução do bloco `B`.
6. **Rótulo Final:** por fim, o rótulo `L_FIM_IF:` é gerado, 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 gerar 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 e no seu 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 para uma representação intermediária pronta para a próxima etapa: a geração de código *Assembly*.

5.6 Exercícios

Exercício 5.1 Explique a diferença conceitual entre o *front-end* e o *back-end* adotada no compilador do *Micro C*. Por que a geração da Representação Intermediária (RI) é considerada o início do processo de síntese (*back-end*) neste projeto? ■

Exercício 5.2 Discuta duas desvantagens principais de se traduzir a Árvore Sintática Abstrata (ASA) diretamente para o código *Assembly* da máquina alvo sem passar por uma Representação Intermediária. ■

Exercício 5.3 O que é o Código de Três Endereços (CTE)? Explique o seu princípio fundamental de funcionamento e por que ele se assemelha a um "*Assembly* abstrato". ■

Exercício 5.4 Descreva a estrutura sintática da Representação Intermediária no *Micro C*, explicando a função e a relação entre `IROpcode`, `IROperand` e `IR_Instruction`. ■

Exercício 5.5 Como o compilador garante que a regra de "no máximo uma operação por instrução" do CTE seja respeitada ao traduzir uma expressão aritmética complexa contendo múltiplas operações lógicas e matemáticas? Qual é o papel das variáveis temporárias nesse processo? ■

Exercício 5.6 Explique como o acesso a uma posição de um vetor (tanto para leitura via `LOAD` quanto para escrita via `STORE`) é decomposto em instruções dentro do *Micro C*. ■

Exercício 5.7 Durante a tradução de uma estrutura de controle de fluxo condicional (como o `NODE_IF`), o gerador de código precisa adicionar novas entidades à RI que não existiam explicitamente no código fonte. Quais são essas entidades e por que elas são estritamente necessárias? ■

Exercício 5.8 O que é o padrão de projeto *Tree Walker* (caminhante da árvore) utilizado no *Micro C*? Descreva como ele interage simultaneamente com a Árvore Sintática Abstrata e a Tabela de Símbolos durante o processo de geração de código. ■

Exercício 5.9 Qual é a principal diferença de responsabilidade estrutural entre as funções `gerar_ir_expr` e `gerar_ir_no` na implementação do tradutor? ■

5.7 Problemas

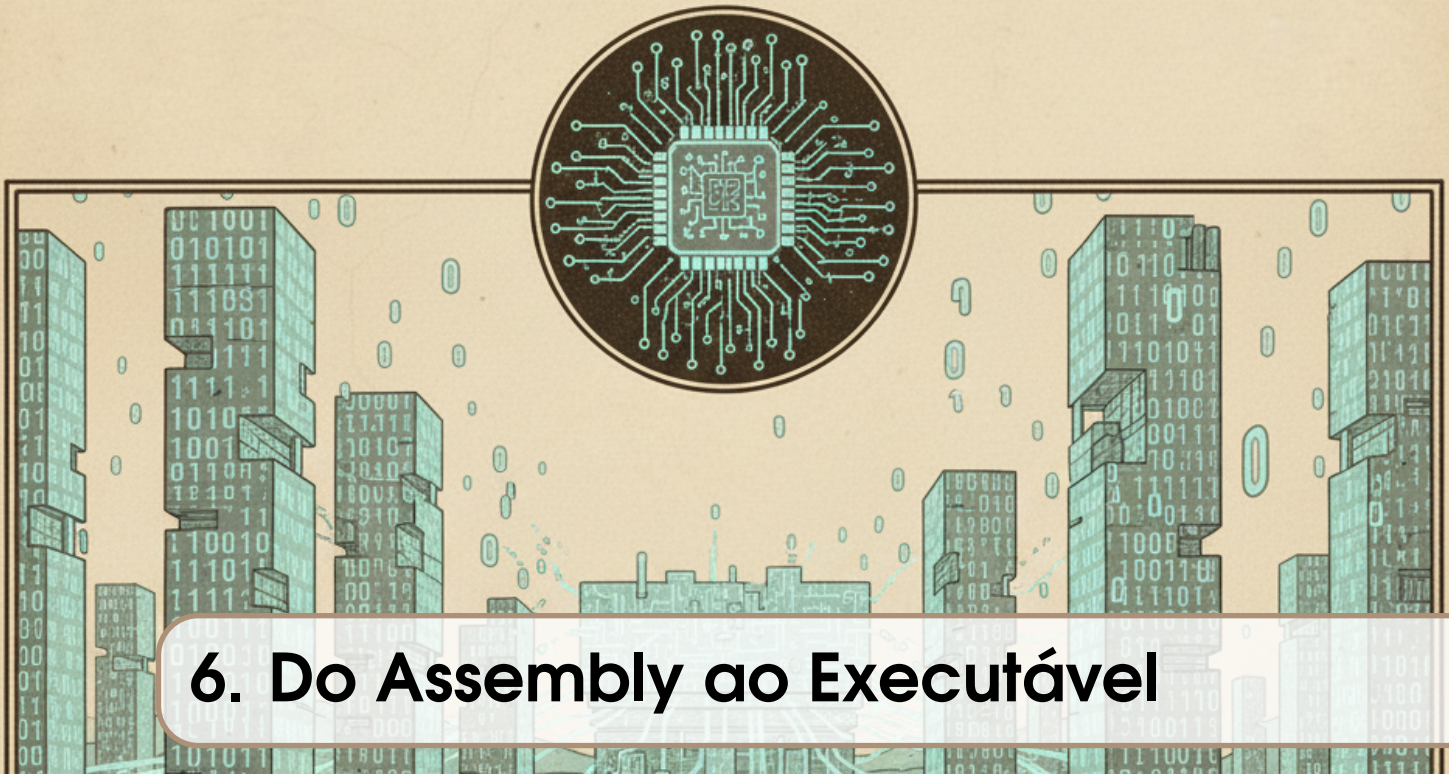
Problemas que serão implementados em cada fase do compilador para estendê-lo.

Problema 5.1 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo booleanas que suportam dois tipos de valores: `true` ou `false`.

Problema 5.2 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo flutuante (com casas decimais após a vírgula).

Dica de Implementação

Para resolver estes problemas nesta etapa do projeto, será necessário alterar três arquivos: `ir.h` (para expandir `IROpcode` com operações matemáticas para flutuantes e `OperandType` para suportar literais booleanos e reais), `intercode.c` (para atualizar o *Tree Walker* de forma que ele traduza os novos nós da ASA e emita os opcodes corretos com base no tipo estático) e, por fim, o `main.c` (para atualizar as funções de impressão e salvamento visual da RI, garantindo que o modo de depuração reconheça as novas instruções).

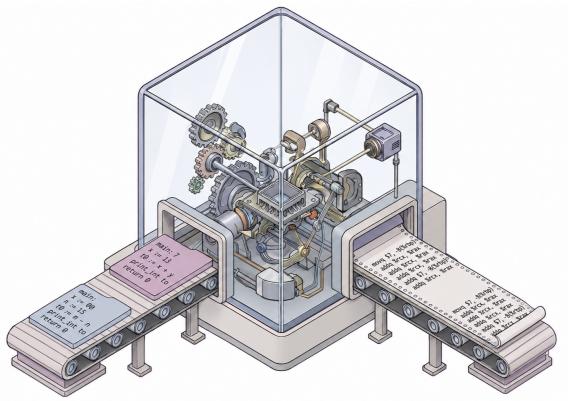


6. Do Assembly ao Executável

"Vision without execution is hallucination."

– Thomas Edison

Concluímos, no capítulo anterior, a primeira fase do *back-end* do compilador ao traduzir a Árvore Sintática Abstrata (ASA) para uma Representação Intermediária (RI). A RI, como vimos, é abstrata e independente de plataforma. Ela descreve o que o programa deve fazer, mas não como a máquina deve executá-lo.



Dando continuidade, neste capítulo exploramos a etapa de geração de código, o estágio final do processo de compilação. Aqui, a lógica da RI é traduzida em instruções nativas da arquitetura x86-64. O objetivo é transformar a lista de Código de Três Endereços (linear e explícita) em um arquivo de saída (`.s`) válido e funcional, o que consolida a transição do modelo conceitual para a execução em hardware real.

Este arquivo, por sua vez, serve de entrada para um montador (ou *assembler*), que

traduz o código *Assembly* em um código de objeto (`.o`). Por fim, esse código é processado por um ligador (ou *linker*), que o combina às bibliotecas do sistema para gerar o programa executável.

No *Micro C*, o `gcc` realiza a montagem do arquivo `.s` e a ligação que produz o executável. A instrução `print` não é independente de bibliotecas: o Assembly gerado chama `printf`, e a cópia de strings usa `strcpy`; ambas vêm da biblioteca C. Não há instruções customizadas de entrada e saída, mas chamadas externas que devem obedecer à ABI da plataforma.

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

A geração de código binário a partir do Assembly é dependente da arquitetura do processador e do sistema operacional. Enquanto a representação intermediária construída no Capítulo 5 é abstrata e universal, o hardware real é limitado: ele interpreta apenas um conjunto específico de instruções de máquina. Nesta seção vamos falar sobre a arquitetura x86-64 e a diferença entre as duas principais sintaxes da representação das instruções.

6.1.1 A Arquitetura x86-64

Para o compilador do *Micro C*, escolhemos a arquitetura x86-64 (AMD64). Ela é a evolução de 64 bits da antiga x86 de 32 bits. Por ser uma arquitetura comum, ela está em todo lugar. Seja em PCs, desktops ou servidores. Hoje, é a arquitetura que domina o mercado, equipando processadores como os da Intel e AMD. Além disso, o *Micro C* gera o código de três instruções o que facilita a construção de outros geradores para arquiteturas como ARM ou RISC-V.

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, em que 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 simples, usaremos apenas um pequeno subconjunto dessas instruções.

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 diferentes: a **Sintaxe Intel** e a **Sintaxe AT&T**. Embora ambas produzam o mesmo código de máquina, o formato textual das instruções é diferente.

Enquanto a **Sintaxe Intel** é utilizada, principalmente, pela Microsoft [13] (com montadores como MASM [12] ou NASM [14]), ambientes derivados do Unix, como Linux e macOS, utilizam a **Sintaxe AT&T** e, por isso, muitos projetos como GCC [8] e CLANG/LLVM [9] 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 na 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`. Por outro lado, 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

É comum, na sintaxe Intel, a inferencia do tamanho da operação (8, 16, 32 ou 64 bits) a partir dos operandos, já sintaxe AT&T, por outro lado, exige que o tamanho seja explícito no nome da instrução (o mnemônico):

- `movl` (*move long*, 32 bits) é usado nas operações com valores escalares de `int` e `char`, ambos armazenados em 4 bytes. Elementos de vetores de `char`, por sua vez, ocupam 1 byte e usam instruções específicas de byte.
- `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 combina operações de 8, 32 e 64 bits: bytes para elementos de vetores de `char`, 32 bits para inteiros e resultados aritméticos, e 64 bits para endereços e manutenção 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)`.

Essas foram as decisões de projeto e convenções que foram definidas para garantir a correta geração de código Assembly pelo *Micro C*. 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*. Vamos definir na próxima seção o modelo de memória utilizado para resolver este problema, o *stack frame* (registro de ativação), que permitirá mapear os símbolos 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 importanet compreender a execução de instruções em Assembly. Diferente de linguagens de alto nível, onde o gerenciamento de dados é transparente, no Assembly não existem variáveis automáticas. O que existe é apenas a memória que deve ser organizada manualmente por meio da pilha de execução (stack). Assim, a tradução da RI para o Assembly consiste em mapear conceitos como 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. 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).
- `%eax`, `%r10d` e `%ecx` são usados como registradores voláteis de rascunho. O gerador não usa mais `%rbx`, evitando a obrigação de salvar e restaurar esse registrador preservado pela função chamada na System V AMD64 ABI.
- `%rdi`, `%rsi`: são registradores para chamadas de funções externas. A Convenção de Chamada System V ABI [15] (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** (***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 aos parâmetros é feito de forma relativa ao ponteiro base `%rbp`. A Figura 6.1 ilustra o modelo utilizado pelo compilador. O analisador semântico atribui os deslocamentos dos parâmetros e das variáveis locais; depois, a geração da RI posiciona os temporários e calcula o tamanho final do quadro:

- **Offsets positivos (parâmetros):** o primeiro parâmetro é acessado em `16(%rbp)`, o segundo em `24(%rbp)` e assim por diante.
- **Offsets negativos (locais):** a função `atribuir_offset_local` mantém um contador único por função, inclusive nos blocos aninhados, e posiciona cada símbolo

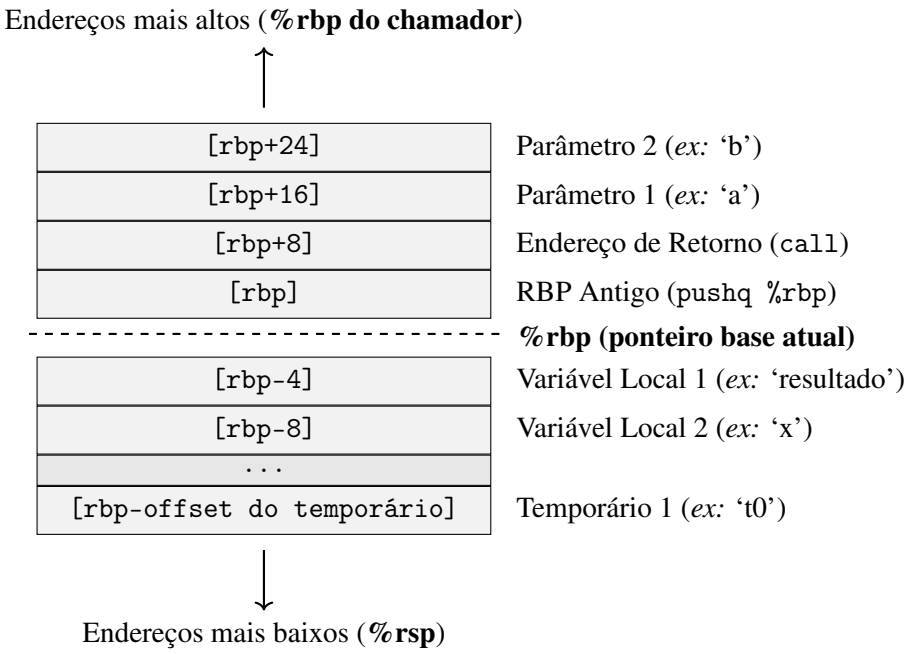


Figura 6.1: Estrutura dinâmica do Registro de Ativação (Stack Frame) no x86-64.

abaixo de `%rbp`.

- **Temporários:** cada temporário ocupa 4 bytes logo após a região das variáveis locais. Para um tamanho local L, `t0` recebe o deslocamento $-(L+4)$, `t1` recebe $-(L+8)$ e assim sucessivamente.

A soma do espaço dos locais e dos temporários é arredondada para um múltiplo de 16 e registrada como `frame_size`. Assim, o quadro não possui mais limite fixo de 256 bytes, e variáveis de blocos aninhados não se sobrepõem.

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 é encontrado um `IR_LABEL` que representa uma função (ex: `main:` ou `soma:`).

```
// Código gerado pela função asm_gen_prologue()
    pushq %rbp
    movq %rsp, %rbp
%    subq $256, %rsp
    subq $frame_size, %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 importante 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 $frame_size, %rsp`: abre exatamente o espaço calculado para variáveis locais e temporários, arredondado para preservar o alinhamento de 16 bytes. Se o tamanho for zero, essa instrução é omitida.

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 de uma só vez todo o espaço dinâmico do quadro, 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

A implementação atual separa o cálculo em duas responsabilidades. Durante a análise semântica, os parâmetros recebem deslocamentos positivos e `atribuir_offset_local` usa um único contador de bytes para todos os blocos de cada função. Ao final da geração da RI, a quantidade de temporários é conhecida; seus deslocamentos são posicionados abaixo

dos locais, e `frame_size` recebe a soma alinhada a 16 bytes. Essa divisão impede tanto a sobreposição entre escopos aninhados quanto a colisão entre variáveis e temporários.

```
// semantic.c: um contador por função
atribuir_offset_local(simbolo, contexto);

// ir.c: temporários abaixo dos locais
temp->stack_offset = -(local_size + (temp_id + 1) * 4);
frame_size = alinhar_16(local_size + temp_count * 4);
```

Na próxima seção vamos detalhar como a tradução é realizada na prática. Ou seja, 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` para a sintaxe de texto que o montador (`gcc`) espera. Vamos descrever os três casos de uso principais 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)

A função acessa o `memory_offset` (conexão com a analise semântica) 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

Os temporários não pertencem à Tabela de Símbolos, mas cada `IROperand` temporário recebe um `stack_offset` depois que o tamanho das variáveis locais é conhecido. O tradutor apenas formata esse deslocamento já calculado.

```
// IR: OPERAND_TEMP, data.temp_id = 0 (para 't0')
case OPERAND_TEMP:
    snprintf(asm_buffer, size, "%d(%%rbp)", op->stack_offset);
    break;
// Exemplo: com 8 bytes locais, t0 -> -12(%rbp)
```

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

Como o código de três endereços admite três operandos e muitas instruções x86-64 têm formato de dois operandos, o gerador usa a estratégia **Load-Operate-Store**, com `%eax` e o registrador volátil `%r10d` como rascunhos.

1. **Load (Carregar):** carrega as fontes para `%eax` e, quando necessário, `%r10d`.
2. **Operate (Operar):** executa a operação, por exemplo `addl %r10d, %eax`.
3. **Store (Armazenar):** grava o resultado no deslocamento dinâmico do destino.

Tradução de Operações Aritméticas e Atribuição

Os `cases` 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 atual: x -> -4(%rbp), t0 -> -8(%rbp)

// 1. Load: Carrega t0 (-8(%rbp)) para o registrador %eax
   movl -8(%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 -> -12(%rbp), a -> -4(%rbp), b -> -8(%rbp)
// 1. Load: Carrega 'a' (16(%rbp)) em %eax
   movl -4(%rbp), %eax
// 2. Load: Carrega 'b' em %r10d
   movl -8(%rbp), %r10d
// 3. Operate: Adiciona %r10d em %eax
   addl %r10d, %eax
// 4. Store: Armazena %eax no temporário t0
   movl %eax, -12(%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 em instruções 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

Esta instrução têm tradução direta para instruções de desvio de fluxo no x86-64.

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

IR_IF_FALSE

E o mesmo acontece com essa instrução, que também têm tradução direta para instruções de desvio de fluxo.

```
// 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 (*arrays*) 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)`. A seguir vamos ver como cada um deles é traduzido.

IR_LOAD

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

```
// Mapeamento: v -> -40(%rbp), i -> -44(%rbp), t0 -> -48(%rbp)
// 1. Carrega o Endereço Base de 'v' em %rax
    asm_instr("leaq -40(%%rbp), %%rax");
    asm_instr("movl -44(%%rbp), %%r10d");// índice em %r10d
// 3. Carrega o Valor [base + indice*4] em %ecx
    asm_instr("movl (%%rax, %%r10, 4), %%ecx");
// 4. Salva o Valor (em %ecx) no destino 't0'
    asm_instr("movl %%ecx, -48(%%rbp)");
```

A instrução `leaq` carrega o endereço-base do vetor. Para vetores de `int`, `IR_LOAD` e `IR_STORE` usam escala 4; para vetores de `char`, `IR_LOAD_BYTE` e `IR_STORE_BYTE` usam escala 1 e operações de byte.

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 %r10d
   asm_instr("movl -8(%%rbp), %%r10d");
// 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, %%r10, 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`.

A próxima seção aborda a interação com funções externas. As convenções de chamada, ou ABIs, determinam como argumentos e retornos são transportados e quais registradores devem ser preservados. No *Micro C*, `print` é traduzido para uma chamada a `printf`; portanto, sua implementação depende da biblioteca C e da System V AMD64 ABI.

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*) [15].

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 exigida por funções da biblioteca C.

6.4.1 Convenção Interna 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 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 e descrevemos a seguir.

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
```

A RI avalia primeiro todos os argumentos, inclusive chamadas aninhadas, e somente depois emite o grupo de `IR_PARAM` da chamada externa. O gerador os acumula e os empilha da direita para a esquerda. Quando a quantidade é ímpar, reserva antes 8 bytes de preenchimento para manter o alinhamento de 16 bytes no ponto de `call`.

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, o chamador descarta os 8 bytes de cada argumento e, quando presente, o preenchimento de alinhamento. Para dois parâmetros, `addq $16, %rsp` continua sendo suficiente; para uma chamada com um parâmetro, por exemplo, são removidos 16 bytes.

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** [15]. Esta convenção determina que os primeiros argumentos inteiros ou ponteiros sejam 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, `assembly.c` traduz esses opcodes para chamadas a `printf`. O quadro da função é múltiplo de 16, portanto a pilha já está alinhada no ponto da chamada. O gerador carrega `%rdi` e `%rsi`, zera `%eax` por se tratar de função variádica e chama `printf` sem ajuste adicional.

```
// 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. Chama com a pilha já alinhada
   call printf
```

No código é possível observar que a tradução para uma chamada externa é mais complexa do que para uma 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`.

O gerador produz um arquivo `.s` que o `gcc` monta e liga nos casos suportados. Ele não usa mais `%rbx`; chamadas internas com quantidade ímpar de argumentos inserem 8 bytes de preenchimento antes dos argumentos, removidos junto com eles após `call`. Essa convenção interna continua sendo didática e própria do *Micro C*, enquanto as chamadas externas seguem a System V AMD64 ABI.

Como um extra, no projeto do compilador do *Micro C* foi disponibilizado um `Makefile` que pode ser utilizado para facilitar a compilação de códigos fonte (extensão `.mcc`) por etapas. Muito embora, você possa executar essas etapas 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* usam 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 discutidas as convenções de chamada: a convenção interna baseada na pilha e a System V AMD64 ABI, necessária para invocar `printf`. Essa integração usa `%rdi` e `%rsi`, dados em `.rodata` e ligação com a biblioteca C; portanto, não é independente de bibliotecas externas.

6.6 Exercícios

Exercício 6.1 Explique as principais diferenças entre as sintaxes AT&T e Intel no desenvolvimento de código *Assembly*, destacando a ordem dos operandos e o uso de prefixos para registradores e valores imediatos. ■

Exercício 6.2 Na sintaxe AT&T, os mnemônicos frequentemente exigem sufixos (como `movl` e `movq`). Explique o significado desses dois sufixos específicos e em quais contextos o gerador de código do *Micro C* os utiliza. ■

Exercício 6.3 Qual é a diferença fundamental entre os registradores `%rbp` (Base Pointer) e `%rsp` (Stack Pointer) no gerenciamento da pilha de execução? ■

Exercício 6.4 Descreva a estrutura do *Stack Frame* (Registro de Ativação) e explique a razão pela qual os parâmetros de uma função recebem *offsets* positivos em relação ao `%rbp`, enquanto as variáveis locais e temporárias recebem *offsets* negativos. ■

Exercício 6.5 O que são o prólogo e o epílogo de uma função em *Assembly*? Detalhe as instruções que os compõem e o objetivo de cada uma delas no ciclo de vida do *Stack Frame*. ■

Exercício 6.6 A Representação Intermediária (CTE) opera em um formato de três endereços (ex: `t0 := a + b`), mas a arquitetura x86-64 tipicamente utiliza um formato de

dois endereços. Como a estratégia *Load-Operate-Store* soluciona essa incompatibilidade durante a tradução? ∎

Exercício 6.7 Como o modo de endereçamento indexado da sintaxe AT&T, descrito pela fórmula `offset(%base, %indice, escala)`, é utilizado para calcular o acesso em memória a uma posição específica de um vetor? ∎

Exercício 6.8 Diferencie a convenção de chamada interna criada para as funções do próprio *Micro C* da convenção externa **System V ABI**. Por que foi estritamente necessário adotar a System V ABI para o funcionamento da instrução `print`? ∎

6.7 Problemas

Problemas que serão implementados em cada fase do compilador para estendê-lo.

Problema 6.1 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo booleanas que suportam dois tipos de valores: `true` ou `false`.

Problema 6.2 Altere o código-fonte do compilador para que ele passe a suportar variáveis do tipo flutuante (com casas decimais após a vírgula).

Problema 6.3 Altere o código-fonte do compilador para que ele passe a suportar leitura de dados pelo usuário simulando a função `scanf` do C, como uma função simplificada chamada `scan` para o *Micro C*.

Problema 6.4 — DESAFIO. Reescreva o `assembly.c` para que o Assembly de saída seja compatível com a arquitetura ARM64. Altere o `Makefile` do projeto para permitir a geração de Assembly compatível com o x86-64 e ARM64 dependendo dos parâmetros utilizados.

Dica de Implementação

Para resolver estes problemas, será necessário alterar apenas um arquivo: `assembly.c` (para mapear os novos *opcodes* e operandos da Representação Intermediária para as instruções e registradores correspondentes da arquitetura x86-64, além de gerenciar a alocação na seção de dados para os novos literais).

IV

Final de Jogo

A Linguagem do Micro C 151

B Gramática do Micro C 153

C Exemplos de Código 155

C.1 Fibonacci Recursivo
C.2 Bubble Sort
C.3 Fatorial de um Número

D Compilação Completa 159

D.1 Código Fonte: `soma.mcc`
D.2 Análise Léxica
D.3 Análise Sintática
D.4 Análise Semântica
D.5 Intercode
D.6 Assembly
D.7 Relação entre as Saídas Geradas

Bibliografia 165

Livros
Artigos
Outros

Índice 167

A. Linguagem do Micro C

Este apêndice apresenta os tipos de token produzidos por `scanner.c`. Nem todo token reconhecido pelo scanner já possui tratamento completo nas fases seguintes; essas diferenças são indicadas explicitamente.

Literais

- **UNDEF** - Token indefinido (erro ou valor desconhecido).
- **ID** - Identificadores (variáveis, funções).
- **INTEGERCONST** - Constante inteira (ex: 42).
- **CHARCONST** – literal contendo exatamente um caractere ou um escape aceito: `'a'`, `'\n'`, `'\t'`, `'\0'`, `'\\'` ou `'\''`.
- **STRINGCONST** – cadeia delimitada por aspas duplas. O scanner interpreta `\n`, `\t`, `\\`, `\'` e `\"`; o escape `\0` é rejeitado dentro de strings.

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, `!`, reconhecido como operador unário pelo *parser*.

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.

Observação sobre números negativos: o scanner sempre produz **MINUS**; a função `unary_expression` do *parser* decide se ele representa negação aritmética ou subtração. Assim, `n-1`, `n - 1` e `-x` não dependem de espaços.

B. Gramática do Micro C

Este apêndice descreve a sintaxe reconhecida pelo analisador descendente recursivo de `parser.c`. A notação é EBNF: `?` indica uma ocorrência opcional, `*` indica repetição e `|` separa alternativas.

Estrutura, declarações e comandos

- `programa ::= função* END_OF_FILE`
- `função ::= tipo (ID | main) '(' parâmetros? ')' bloco`
- `parâmetros ::= declaração (',' declaração)*`
- `tipo ::= int | char`
- `declaração ::= tipo ID ('[' INTEGERCONST ']')? ('=' expressão)?`
- `bloco ::= '{' comando* '}'`
- `comando ::= bloco | declaração ';' | if | for | print ';' | atribuição ';' | chamada ';' | return`
- `if ::= if '(' expressão ')' comando (else comando)?`
- `for ::= for '(' (declaração | atribuição)? ';' expressão? ';' atribuição? ')' comando`
- `print ::= print '(' expressão ')'`
- `return ::= return expressão? ';'`
- `atribuição ::= ID ('[' expressão ']')? '=' expressão`
- `chamada ::= ID '(' argumentos? ')'`
- `argumentos ::= expressão (',' expressão)*`

Expressões por nível de precedência

- `expressão ::= conjunção ('||' conjunção)*`
- `conjunção ::= relação ('&&' relação)*`
- `relação ::= aritmética (('==' | '!=' | '<' | '>' | '<=' | '>=') aritmética)*`
- `aritmética ::= termo (('+' | '-') termo)*`
- `termo ::= unária (('*' | '/' | '%') unária)*`
- `unária ::= ('-' | '!')  unária | fator`
- `fator ::= INTEGERCONST | CHARCONST | STRINGCONST | ID | chamada | ID '[' expressão ']' | '(' expressão ')'`

A função `unary_expression` reconhece recursivamente `-` e `!`, permitindo inclusive composições como `!!x` e `-(-x)`. Não há comando vazio `;`. O retorno sem expressão é aceito sintaticamente, mas rejeitado semanticamente porque a linguagem não possui tipo `void`.

C. Exemplos de Código

A seguir são apresentados programas aceitos pela linguagem descrita nos Apêndices A e B. Eles utilizam alguns recursos importantes do compilador. Além disso, a função `print()` é mais simples que `printf()`, embora sua geração de Assembly invoque `printf` da biblioteca C.

C.1 Fibonacci Recursivo

O exemplo de Fibonacci executa duas chamadas recursivas em uma expressão e, para a entrada 8, a saída esperada é 21. Ele funciona como teste para geração de variáveis temporárias, chamadas e retornos.

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;
}
```

A condição `n <= 1` define os dois casos-base da recorrência: para 0 e 1, a função retorna o próprio argumento. Para valores maiores, cada invocação calcula

`fibonacci(n - 1)` e `fibonacci(n - 2)` e soma os resultados. O exemplo evidencia por que chamadas recursivas precisam manter separados os parâmetros, as variáveis locais e o retorno de cada iteração. Também exercita a manutenção de um resultado temporário enquanto a outra chamada da expressão é avaliada.

C.2 Bubble Sort

O *Bubble Sort* ordena o vetor local de cinco inteiros em ordem crescente. O programa explora a declaração e indexação de vetor, atribuições, comparações e laços `for` aninhados. Ele usa tamanho fixo porque a linguagem não permite passar vetores como parâmetros.

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;
}
```

Em cada iteração do laço interno, os elementos adjacentes `v[j]` e `v[j+1]` são comparados. Quando aparecem em ordem decrescente, a variável `temp` permite trocar suas posições sem perder nenhum valor. Uma passagem completa desloca os maiores elementos em direção ao fim do vetor, e o laço externo repete o processo. Esta versão conserva deliberadamente os limites fixos em todas as passagens e não introduz uma condição de parada antecipada, mantendo o exemplo concentrado nas construções reconhecidas pelo compilador.

C.3 Fatorial de um Número

O cálculo do fatorial possui um caso-base, selecionado por `if-else`, e um passo recursivo que multiplica `n` por `fatorial(n - 1)`. Para a entrada 5, o programa deve imprimir 120.

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;
}
```

D. Compilação Completa

Este apêndice acompanha um programa pelas cinco fases do compilador e mostra os artefatos de diagnóstico produzidos em cada modo. As fases comunicam-se em memória; os arquivos `tokens.txt`, `ast.txt`, `symbols.txt` e `ir.txt` registram resultados para inspeção e não são entradas obrigatórias da fase seguinte.

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 [16].

D.2 Análise Léxica

A primeira fase é a análise léxica. Para executá-la isoladamente e inspecionar os *tokens* produzidos, usamos o comando:

```
$ mcc --scan exemplos/soma.mcc
```

Ao executar esse comando, o scanner preenche uma `TokenList` em memória e também grava `tokens.txt` para depuração. O *parser* recebe diretamente essa lista; o fluxo principal não relê o arquivo.

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
```

Com `-parse`, o compilador executa novamente a análise léxica e entrega a lista de tokens em memória ao analisador sintático.

Fase 2: análise sintática

```
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, `ast.txt` contém uma impressão da ASA para inspeção. A análise semântica recebe o ponteiro para a árvore construída em memória e não reconstrói a ASA a partir desse arquivo.

D.4 Análise Semântica

Em seguida, a análise semântica verifica regras estáticas de tipos, declarações, escopos, chamadas e retornos. Ela não valida se o algoritmo implementa corretamente a intenção do programador.

```
$ mcc --semantic exemplos/soma.mcc
```

Se o programa satisfizer as regras semânticas implementadas, essa fase termina sem erro e também produz `symbols.txt` como relatório da tabela de símbolos. A saída abaixo foi abreviada para destacar as mensagens de conclusão.

Fase 3: análise semântica

```
Fase 1 (Lexica) concluida: 31 tokens gerados.
Fase 2 (Sintatica) concluida: AST gerada.
Fase 3 (Semantica) concluida: Codigo validado.

--- MODO DE ANALISE SEMANTICA CONCLUIDO ---
```

D.5 Intercode

A quarta fase é a geração do código intermediário:

```
$ mcc --gen-ir exemplos/soma.mcc
```

A saída dessa fase segue o formato de código de três endereços:

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.
```

A fase 4 mantém a lista de RI em memória para o gerador de Assembly e grava sua forma textual em `ir.txt`.

D.6 Assembly

Na quinta fase, a RI é traduzida para texto Assembly x86-64 em sintaxe AT&T.

O código de máquina só será produzido depois, quando um montador processar o arquivo `.s`:

```
$ mcc --gen-asm exemplos/soma.mcc
```

A execução produz o arquivo Assembly `soma.s`, que poderá ser entregue ao montador para a geração do código de máquina.

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 trecho a seguir corresponde à saída atual para o exemplo `soma.mcc`: o quadro possui 16 bytes, o temporário vem logo após as variáveis locais, o registrador volátil `%r10d` substitui `%ebx` e `printf` é chamado com a pilha já alinhada.

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 $16, %rsp
    movl $10, %eax
    movl %eax, -4(%rbp)
    movl $20, %eax
    movl %eax, -8(%rbp)
    movl -4(%rbp), %eax
    movl -8(%rbp), %r10d
    addl %r10d, %eax
    movl %eax, -12(%rbp)
    leaq .L.str.int(%rip), %rdi
    movl -12(%rbp), %esi
    xorl %eax, %eax
    call printf
    movl $0, %eax
    movq %rbp, %rsp
    popq %rbp
    ret
```

A listagem atual preserva os requisitos relevantes da System V AMD64 ABI: usa um registrador volátil como rascunho, mantém o quadro múltiplo de 16 e não altera a pilha imediatamente antes de `printf`.

O código final em Assembly pode ser transformado em binário usando o `gcc`:

```
$ gcc exemplos/soma.s -o soma
$ ./soma
30
```

O `gcc` monta o arquivo `.s`, liga a referência a `printf` com a biblioteca C e produz o executável `soma`. A saída 30 confirma o comportamento deste exemplo específico.

D.7 Relação entre as Saídas Geradas

As listagens deste apêndice mostram as saídas geradas por cada fase sobre o mesmo programa. A lista de *tokens* preserva categorias, lexemas e linhas. A ASA organiza esses elementos em uma hierarquia e omite pontuações que já cumpriram sua função sintática. A análise semântica associa declarações e tipos aos nós. E, na sequência, a RI transforma a hierarquia validada em uma sequência linear de operações. Por fim, o gerador de Assembly substitui operandos abstratos por endereços, registradores, rótulos e instruções x86-64.

A expressão `a + b` permite acompanhar essa continuidade. Na saída léxica, ela aparece como os *tokens* `ID`, `PLUS` e `ID`. Na ASA, torna-se um nó de operação binária com dois identificadores. E na RI, produz `t0 := a + b`. A listagem Assembly carrega `a` e `b` a partir de seus deslocamentos no frame, executa `addl` e armazena o resultado no espaço destinado a `t0`. A chamada a `printf` processa esse valor, e a saída 30 permite comparar o comportamento observado com o cálculo do programa-fonte.

Essa relação não significa que uma fase releia o arquivo textual produzido pela anterior. Como mostrado ao longo do apêndice, `tokens.txt`, `ast.txt`, `symbols.txt` e `ir.txt` são saídas geradas e que servem de apoio ao usuário na avaliação de cada etapa. Durante a compilação, as estruturas equivalentes permanecem em memória e são utilizadas diretamente pela fase seguinte.

Bibliografia

[Aho+08] Alfred V. Aho et al. *Compiladores: Princípios, Técnicas e Ferramentas*. Portuguese. Traduzido por Daniel Vieira. 2ª edição. Pearson Addison-Wesley, 2008. ISBN: 978-85-88639-24-9 (ver páginas 17, 44, 52, 69, 71, 93, 99, 107).

[CT12] Keith D. Cooper e Linda Torczon. *Engineering a Compiler*. English. 2ª edição. Morgan Kaufmann, 2012. ISBN: 978-0-08-092509-3 (ver páginas 17, 93, 107).

[Cor+22] Thomas H Cormen et al. *Introduction to Algorithms*. MIT press, 2022 (ver páginas 69, 99).

[KA01] Ken Kennedy e John R Allen. *Optimizing Compilers for Modern Architectures: a Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., 2001 (ver páginas 17, 37, 69).

[Nys21] Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021 (ver páginas 17, 52, 93).

[CST02] Keith D Cooper, Devika Subramanian e Linda Torczon. "Adaptive Optimizing Compilers for the 21st Century". Em: *The Journal of Supercomputing* 23.1 (2002), páginas 7–22 (ver páginas 17, 52, 69).

[Ear70] Jay Earley. "An Efficient Context-Free Parsing Algorithm". Em: *Communications of the ACM* 13.2 (1970), páginas 94–102. DOI: 10.1145/362007.362035 (ver página 52).

[Sta02] Richard M Stallman. “GNU Compiler Collection Internals”. Em: *Free Software Foundation* 46 (2002), página 765 (ver páginas 15, 129).

[Lat08] Chris Lattner. “LLVM and Clang: Next Generation Compiler Technology”. Em: *The BSD conference*. Volume 5. 2008, páginas 1–20 (ver páginas 15, 129).

[LA04] Chris Lattner e Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. Em: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. IEEE Computer Society, 2004, páginas 75–86. ISBN: 0-7695-2102-X (ver páginas 15, 111).

[Sch90] Douglas C Schmidt. “Gperf: A Perfect Hash Function Generator”. Em: *Proceedings of the 2nd USENIX C++ Conference*. USENIX Association. 1990, páginas 87–100 (ver página 44).

[Cor25] Microsoft Corporation. *Microsoft Macro Assembler Reference*. [Online; acessado em 06-11-2025]. 2025 (ver página 129).

[Int25] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Manual oficial de desenvolvedor de software da Intel. Order Number: 325462-089US. Outubro de 2025 (ver página 129).

[Tat+25] Simon Tatham et al. *NASM - The Netwide Assembler*. 3.01. [Online; acessado em 11-11-2025]. outubro de 2025 (ver página 129).

[Mat+13] Michael Matz et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. Jul. de 2013 (ver páginas 132, 140, 142).

[Ped25] Pedro H. F. Carvalho and Brivaldo A. da Silva Jr. *Compilador Micro C*. `https://github.com/WorksFacom/mcc`. [Online; acessado em 11-11-2025]. 2025 (ver páginas 17, 159).

Índice Remissivo

A

Análise Léxica 23
Análise Semântica 87
Análise Sintática 51

C

Compilação Completa 159
Código intermediário 105

E

Exemplos de Código 155
Exercícios 48, 84, 100, 124, 145

G

Geração de Assembly 127
Gramática do Micro C 153

L

Linguagem do Micro C 151

P

Problemas 49, 85, 101, 126, 147

MICRO C
APRENDENDO COMPILADORES

Um compilador deixa de ser uma caixa-preta quando acompanhamos, passo a passo, a transformação de um programa. Neste livro, a linguagem minimalista *Micro C* conduz o leitor por todas as etapas desse percurso: análise léxica, análise sintática, construção da árvore, verificação semântica, geração de código intermediário e tradução para *Assembly* x86-64.

Com explicações diretas, representações visuais, trechos da implementação e atividades progressivas, a obra aproxima a teoria da prática. O objetivo é permitir que estudantes e entusiastas compreendam um compilador de ponta a ponta, experimentem suas decisões de projeto e tenham uma base concreta para ampliar a linguagem ou criar seus próprios compiladores.



Pedro Henrique

"Agradeço a todos que fizeram parte dessa jornada e espero que este material possa contribuir com todos que desejam aprender mais sobre compiladores e se aprofundar nesse fascinante universo."

"Espero que vocês possam se divertir programando seu próprio compilador e, dessa vez, compreendendo todas as etapas que um compilador executa para converter o código fonte em código de máquina."



Brivaldo Junior

ISBN 978-65-02-42701-9

Obrigado por chegar até aqui.

Esperamos que esta leitura desperte novas perguntas, novos experimentos e a vontade de construir linguagens que ainda não existem.