

---

# Compiladores

## Capítulo da Aula 3: Fases de um Compilador

Prof. Aléssio Miranda Júnior  
alessio@cefetmg.br  
CEFET-MG - Campus Timóteo

Março de 2026

### 1 Objetivos

- Compreender a metáfora da "linha de montagem" e a estrutura de Pipeline dos compiladores modernos.
- Revisar a atuação conjunta das fases de *Front-End* (Análise) e a transição da sequência bruta de texto para a abstração da Árvore Sintática.
- Compreender o famoso problema da explosão combinatória  $N \times M$  e como a **Representação Intermediária (IR)** resolve definitivamente esse gargalo na indústria.
- Aprofundar o estudo das fases de *Middle-End* e *Back-End* (Síntese): Geração de IR, Passes de Otimização e Geração de Código Alvo nativo.
- Entender o papel contínuo e onipresente da **Tabela de Símbolos** e das estratégias avançadas de **Tratamento de Erros**.

### 2 A Metáfora da Linha de Montagem (O Pipeline)

Traduzir um programa de centenas de milhares de linhas escrito em C++ para código de máquina puro não é um problema que pode ser atacado de uma só vez. A engenharia de software ensina que problemas colossais devem ser fatiados em problemas menores, sequenciais e altamente coesos.

Um compilador opera exata e conceitualmente como uma **linha de montagem industrial (Pipeline)**. O código fonte entra bruto em uma das extremidades da esteira. À medida que avança, ele passa por "estações de trabalho" isoladas (as fases lógicas). Cada estação recebe o material da estação anterior, faz exclusivamente o seu trabalho (sem saber o que as outras estações fazem), refina a estrutura e repassa para a próxima. Se alguma estação detectar uma peça defeituosa (um erro no código), a esteira soa um alarme.

Historicamente, esse longo pipeline é fatiado ao meio em duas grandes fatias arquiteturais: o **Front-End** (A fase de Análise, que compreende a linguagem) e o **Back-End** (A fase de Síntese, que cospe o hardware).

---

## 3 O Front-End: A Fase de Análise

A missão exclusiva do Front-End é confirmar se o desenvolvedor escreveu algo com sentido matemático e sintático. O Front-End lê o arquivo 'c' ou 'java' e gera como saída uma **Árvore de Sintaxe Abstrata (AST)**.

Essa metamorfose de texto para árvore ocorre em três estações:

### 3.1 1. Análise Léxica (O Scanner)

O código fonte no disco rígido não passa de um longo "varal" unidimensional de caracteres ('i', 'n', 't', ' ', 'x', '=', '5', ';'). A Análise Léxica mastiga esse fluxo implacável e o fatia em "palavras válidas" conhecidas como **Tokens**. Além de agrupar, o Scanner atua como o *filtro de impurezas*: ele destrói agressivamente todos os espaços em branco, quebras de linha (tabs/enters) e ignora os comentários deixados pelo programador.

No jargão de compiladores, um identificador mal escrito (ex: 23variavel) aciona um **Erro Léxico**. Mecanicamente, Scanners operam lendo lotes da memória RAM (buffers) e aplicando algoritmos puramente baseados em **Autômatos Finitos Determinísticos (DFA)** gerados a partir de **Expressões Regulares**.

### 3.2 2. Análise Sintática (O Parser)

O Scanner garantiu que o vocabulário existe, mas não a gramática. A frase "bola o chutou menino" contém palavras válidas na língua portuguesa, mas não faz sentido gramatical.

O Parser recebe a fila linear de tokens e tenta encaixá-los na **Gramática Livre de Contexto (GLC)** da linguagem. Se a sequência for matematicamente derivável a partir das regras, o Parser transforma a lista unidimensional em um grafo hierárquico: a **Árvore de Sintaxe Abstrata (AST)**. Neste momento, a declaração  $a = b + c * 2$  não é mais lida da esquerda para a direita, mas sim do topo para a base, onde o nó superior = tem o filho esquerdo a e o filho direito +, garantindo na raiz da estrutura a sagrada precedência da multiplicação. Uma violação gramatical lança um **Erro Sintático**.

### 3.3 3. Análise Semântica (O Type-Checker)

Sintaxe correta não garante coerência lógica. A expressão `carro = 5 + "texto"` possui perfeita gramática de atribuição, mas uma semântica computacional inaceitável.

O Analisador Semântico navega pelos galhos da AST checando a coerência estática do programa. Suas responsabilidades máximas incluem:

- **Resolução de Escopo**: Garantir que as variáveis utilizadas realmente foram previamente declaradas e estão acessíveis dentro do nível atual de chaves {}.
- **Verificação de Tipos (Type Checking)**: Validar se as assinaturas de funções "casam" matematicamente com os argumentos passados, e checar cruzamento lógico indesejado (somar Booleans com Ponteiros de Memória).
- **Coerção Automática de Tipos (Type Casting)**: Se o usuário somar o float 3.14 com o int 2, o ASem silenciosamente injeta um nó novo na AST transformando o int 2 no float 2.0 antes da soma prosseguir para o Back-End.

---

## 4 A Revolução do Middle-End e o Problema $N \times M$

Quando os compiladores nasceram nos anos 70, o Analisador Semântico cuspiu a AST diretamente no Gerador de Código de Máquina. Isso funcionava porque só existia praticamente C para computadores PDP-11.

Considere o século 21: Temos  $N = 10$  linguagens fonte (C, C++, Java, Rust, Swift, Go, Python, Julia, Kotlin, Nim) e  $M = 6$  arquiteturas alvo vitais (x86\_64, ARM, RISC-V, MIPS, WebAssembly, PowerPC). Se mantivéssemos o design monolítico antigo, para dar cobertura total, o mundo Open Source teria que construir e testar  $10 \times 6 = 60$  **Compiladores Completos independentes**. Isso gera o temido problema da "Explosão Combinatória".

Para destruir esse problema lógico brutal, a engenharia de compiladores cortou a linha de montagem e inseriu um componente neutro no centro: a **Representação Intermediária Universal (IR)**.

- **A Salvação Matemática ( $N + M$ )**: Os desenvolvedores constroem 10 Front-Ends independentes. A única missão deles é entender as 10 linguagens diferentes e traduzi-las rigorosamente para a **mesma linguagem IR**.
- Simultaneamente, os fabricantes de hardware constroem 6 Back-Ends. Eles ignoram de qual linguagem o código veio; eles só sabem ler a **IR genérica** e traduzi-la para os registradores de seu hardware.

Isso reduz o esforço logístico da humanidade de 60 softwares para apenas **16 módulos conectáveis**. Esse é o exato design premiado internacionalmente adotado no ecossistema LLVM (**Low Level Virtual Machine**).

## 5 O Middle-End: O Otimizador de IR

Com o código perfeitamente fatiado e padronizado em IR, inserimos a fase que torna o código em C 100x mais veloz que o interpretado em Python: a Otimização agnóstica de hardware.

### 5.1 Código de Três Endereços (TAC)

A forma de IR mais legível para algoritmos clássicos é o **Three-Address Code (TAC)**. Neste padrão, instruções só podem carregar no máximo duas variáveis operantes, um operador, e um destino. Por exemplo, o código denso de matriz  $x = a[i] + b * c$ ; é brutalmente fatiado em:

```
t1 = b * c
t2 = i * 4      // (calculando deslocamento em array de inteiros)
t3 = a[t2]
x  = t3 + t1
```

## 5.2 Passes de Otimização Clássicos

O *Optimizer* (Middle-End) entra em cena varrendo as listas de instruções TAC milhares de vezes aplicando teoremas matemáticos chamados **Passes**:

- 1 Dobragem de Constantes (Constant Folding):** O compilador avalia expressões óbvias que podem ser resolvidas antes do cliente ligar o computador. `x = 60 * 60 * 24;` é removido do código final e substituído por `x = 86400;`.
- 2 Eliminação de Subexpressões Comuns (CSE):** Se o código for  
`a = (b+c)*10; y = (b+c)*20;`  
 O otimizador cria um TAC silencioso e altera o código para  
`t1 = b+c; a = t1*10; y = t1*20;`  
 Reduzindo a carga na CPU do usuário pela metade.
- 3 Remoção de Código Morto (DCE):** Variáveis guardadas e nunca lidas depois, funções importadas mas nunca chamadas, ou instruções dentro de `if(false)` são fisicamente apagadas da base do código. O usuário compila 50 mil linhas, mas o binário terá apenas as 20 mil essenciais.
- 4 Desenrolamento de Laços (Loop Unrolling):** O compilador analisa loops curtos `for(i=0;i<3;i++) a+=1;` e o troca internamente por instruções sequenciais brutas: `a+=1; a+=1; a+=1;`. Isso acaba com a lentidão mecânica causada pelos *\*jumps\** incondicionais nos circuitos da CPU.

## 6 O Back-End: A Fase de Síntese

Com a IR completamente purificada e enxuta, o compilador finalmente olha para as limitações do silício eletrônico específico da máquina do cliente.

### 6.1 1. Seleção de Instruções (Instruction Selection)

Processadores CISC (como Intel e AMD) possuem literalmente milhares de comandos (Assembly). Alguns desses comandos podem fazer o papel de três instruções TAC simultaneamente (ex: Instruções FMA que multiplicam e somam em um único pulso de clock). A seleção atua como um algoritmo de reconhecimento de padrões de Grafos (Tree-Matching), unificando nós genéricos do TAC na instrução de máquina mais robusta e barata possível.

### 6.2 2. Alocação de Registradores (Register Allocation)

Trata-se de um problema matemático classificado teoricamente como NP-Completo. Durante o TAC, o compilador presumiu ter "infinita memória" e criou 10.000 variáveis temporárias  $t_1 \dots t_{10000}$ . No mundo real, um processador AMD possui cerca de 16 registradores de alta velocidade de acesso na CPU. O compilador deve prever quais variáveis são as mais lidas em um dado microssegundo, mapeá-las para esses escassos 16 registradores, e decidir violentamente quais variáveis devem ser arremessadas (Spilled) para a distante e

morosa memória RAM. Isso é comumente resolvido usando o algoritmo de **Coloração de Grafos de Interferência**.

### 6.3 3. Escalonamento de Instruções (Instruction Scheduling)

Na reta final milimétrica, o compilador tenta ser mais inteligente que o arquiteto do processador. Certas instruções exigem aguardar acesso à memória externa. Para que a CPU não fique parada ("preso num gargalo" ou \*Pipeline Stall\*), o compilador reordena as instruções de Assembly fisicamente no arquivo texto gerado. Ele coloca operações matemáticas independentes entre uma requisição de memória lenta e a linha onde ela será usada, forçando a CPU a executar o que der em paralelo. A saída final produzida é o arquivo `.s` (Assembly) ou binário real (`.o`).

## 7 Orquestração: O Esqueleto e o Escudo do Compilador

Através dessas seis estações complexas, dois sistemas de apoio viajam onipresentes, de ponta a ponta na linha de montagem:

### 7.1 A Tabela de Símbolos

É uma estrutura de dados de banco de dados hiper-veloz (frequentemente Tabela Hash com suporte a Escopos Aninhados). Ela nasce no Analisador Léxico (para cadastrar nomes únicos) e morre apenas no Gerador de Código. Quando o Analisador Semântico precisa saber se a variável `salario` é um float local do loop atual ou uma variável global inteira, o Hash Table lhe responde em complexidade  $O(1)$ . Quando o Gerador de Código na Síntese precisa saber quantos bytes alocar de memória, a Tabela informa se `salario` necessita de 4 ou 8 bytes.

### 7.2 O Tratamento de Erros e o "Panic Mode"

Usuários cometem múltiplos erros sintáticos ou digitação errada continuamente. Se um compilador abortar a execução estritamente no primeiro ponto e vírgula esquecido (na linha 1), o usuário terá que rodar o processo compilação-aborto 5.000 vezes seguidas, inviabilizando qualquer desenvolvimento moderno em software pesado. O Tratamento de Erro atua como um escudo resiliênte. Ao detectar um *Syntax Error*, o compilador não para; ele emite um alerta na tela para o programador e entra em estado de recuperação de erro grave (conhecido como **Panic Mode Recovery**). Ele engole lixo sintático jogando dezenas de tokens fora silenciosamente, até que as engrenagens percebam algo "sólido" (como um caractere delimitador `}` ou a palavra `end`). A partir dali, o compilador se recupera do pânico e recomeça a análise fingindo que nada ocorreu, prosseguindo com a varredura do arquivo para garantir que a IDE mostre absolutamente \*todos\* os erros visíveis no código subjacente de uma só vez na tela final do desenvolvedor.

---

## 8 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Pearson. (Capítulos 1 e 8)
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. Morgan Kaufmann. (Capítulos 1 e 12)
- Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.