Compiladores

Aula 03: Fases de um Compilador

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

Objetivos da Aula

- Revisar a atuação conjunta das fases de *Front-End* (Análise).
- Compreender o problema da tradução $N \times M$ e a solução baseada em **Representação Intermediária (IR)**.
- Aprofundar nas fases de *Back-End* (Geração de IR, Otimização e Geração de Código Alvo).
- Entender o papel contínuo da **Tabela de Símbolos** e do **Tratamento de Erros**.

Recapitulando: O Front-End (Análise)

Na aula anterior, destrinchamos a conversão do código fonte até a **Árvore Sintática (AST)**:

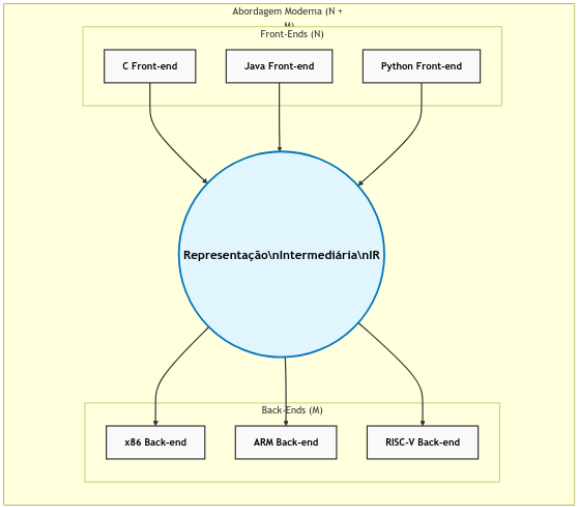
1. **Léxico**: `123`, `if`, `x1` (Agrupa caracteres $\rightarrow$ Tokens).
2. **Sintático**: Gramáticas Livres de Contexto validam a ordem estrutural (gera a AST).
3. **Semântico**: Garante a coerência lógica (tipos, escopos) usando a Tabela de Símbolos.

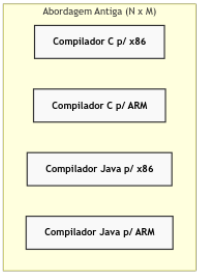
Pergunta central: Por que não traduzimos a Árvore Sintática validada diretamente para Assembly/Código de Máquina?

O Problema da Tradução $N \times M$

Considere N linguagens fonte (C, Java, Python) e M arquiteturas alvo (x86, ARM, RISC-V). Se cada compilador for uma "caixa-preta" que vai do Fonte direto ao Código de Máquina, precisaremos construir $N \times M$ compiladores inteiros.

Abordagem Monolítica ($N \times M$)





A Solução de Retargetabilidade ($N + M$)

Ao dividirmos o compilador, introduzimos uma fase de **Representação Intermediária (IR)**:

- **Front-Ends (N)**: Cada linguagem só precisa de 1 front-end para gerar a **IR** padrão.
- **Back-Ends (M)**: Cada arquitetura só precisa de 1 back-end que lê a **IR** padrão e gera Assembly.

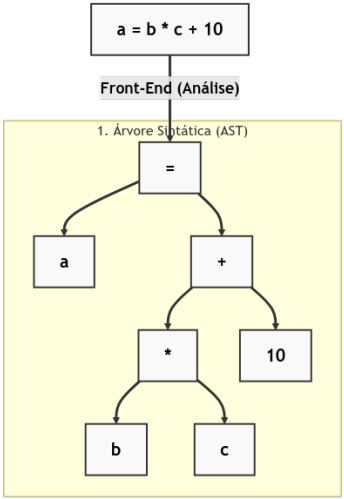
Total de módulos necessários cai drasticamente de $N \times M$ para $N + M$. Ferramentas como o **LLVM** dominam o mercado moderno exatamente por fornecerem uma IR extremamente rica e otimizada.

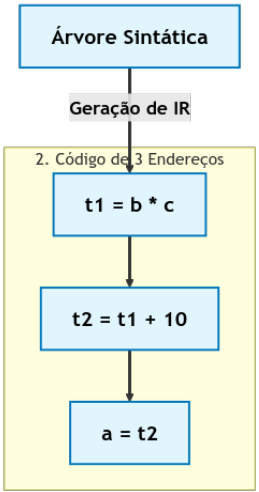
1. Geração de Código Intermediário (IR)

A IR age como um "Assembly genérico de máquina virtual infinita". A forma mais clássica de IR é o **Código de Três Endereços (TAC)**.

- Quebra equações complexas em várias operações simples.
- Usa variáveis temporárias ($t1, t2$).
- Regra de ouro: *No máximo três operandos por instrução*.

Exemplo: Código de Três Endereços





2. Otimização de Código

Com a IR em mãos, o compilador tenta melhorar o código foneticamente (tamanho ou velocidade) mantendo *exatamente* a mesma semântica.

- **Constant Folding**: Se o código diz `x = 10 * 2`, o compilador troca diretamente por `x = 20` em tempo de compilação.
- **Dead Code Elimination**: Remoção de variáveis declaradas e nuca usadas, ou blocos `if (false)` que jamais serão atingidos.
- **Loop Unrolling**: Desenrola pequenos loops para evitar o overhead computacional de recálculo de índices.

Nota: As otimizações são feitas na IR justamente para que todas as Linguagens Front-End se beneficiem do mesmo processo gratuitamente.

3. Geração de Código Alvo (Back-End final)

A última fase traduz a IR otimizada para o código nativo da plataforma.

- **Seleção de Instruções**: Escolhe qual comando Assembly específico usar (ex: no x86, usar `LEA` para otimizar matemática ponteiro).
- **Alocação de Registradores**: A CPU tem um número ínfimo de registradores ultrarrápidos (ex: 16). O gerador decide quais variáveis da IR merecem ficar neles e quais vão para a lenta Memória Principal (RAM).
- **Output**: Um arquivo ‘.s‘ (Assembly) ou um ‘.o‘ (Object file binário relocável).

Orquestração: Tabela de Símbolos e Tratamento de Erros

Todas essas 6 fases ocorrem sob a supervisão de duas entidades globais ativas:

1. **Tabela de Símbolos**: O "cartório" do compilador. Alimentada pelo Lexer e Parser, consultada pesadamente pelo Semântico, e usada até o final pelo Gerador de Código para saber a quantia de memória a alocar para as variáveis.
2. **Tratamento de Erros**: Compiladores modernos usam *Panic Mode Recovery*. Se um Lexer acha um símbolo inválido, ele não derruba a compilação; ele descarta o símbolo, tenta continuar escaneando e gera uma lista robusta de erros ao final (como o Clang/GCC modernos).

Referencias

- Aho, et al. *Compilers: Principles, Techniques, and Tools* (Livro do Dragao).
- Cooper & Torczon. *Engineering a Compiler*.

Obrigado!

Dúvidas?

Email: alessio@cefetmg.br

SIGAA: sistema acadêmico
Site: `https://alessiojr.com/disciplines/compiladores`
GitLab: `https://git.juninho.com.br/` **Maratona:** `https://maratona.alessiojr.com/`