Compiladores

Capítulo da Aula 2: Introdução aos Compiladores

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

Fevereiro de 2026

1 Objetivos

Este capítulo aprofunda os conceitos fundamentais sobre compiladores, estabelecendo a base teórica para o restante da disciplina. Os objetivos principais são:

- Definir formalmente o conceito de compilador e seus componentes.
- Diferenciar detalhadamente os modelos de execução: Compilação Pura (AOT), Interpretação Pura e Híbridos (JIT).
- Compreender a estrutura básica de T-Diagrams (Diagramas de Lápide) como ferramenta de modelagem.
- Visualizar o ciclo de vida da tradução de código através do modelo Análise-Síntese.

2 O que é um Compilador?

Em sua essência, um compilador é um **tradutor**. Diferente de um programa comum que processa dados para produzir resultados, um compilador processa *código* para produzir *código equivalente*.

2.1 Definição Formal

Formalmente, definimos um compilador C como uma função que mapeia um programa P_S escrito em uma linguagem fonte L_S para um programa semanticamente equivalente P_T em uma linguagem alvo L_T.

$$C : L_S \to L_T$$

A condição de equivalência semântica garante que, para qualquer entrada válida, ambos os programas produzam o mesmo resultado:

$$\forall input \in Inputs : Exec(P_S, input) \equiv Exec(P_T, input)$$

A propriedade fundamental é a **preservação da semântica**. O processo de compilação pode alterar a estrutura do código, otimizar loops e remover variáveis, mas nunca deve alterar o comportamento observável do programa original.

2.2 Por que compilar?

Existem três motivações principais para não programarmos diretamente em linguagem de máquina:

1. **Produtividade**: Linguagens de alto nível (Java, Python, C++) oferecem abstrações que tornam o desenvolvimento mais rápido e menos propenso a erros.
2. **Portabilidade**: O mesmo código fonte pode ser compilado para arquiteturas de hardware completamente diferentes (x86, ARM, RISC-V), abstraindo os detalhes do set de instruções.
3. **Otimização**: Compiladores modernos realizam transformações matemáticas complexas (como alocação de registradores por coloração de grafos) que seriam inviáveis de serem feitas manualmente em grandes projetos.

3 Modelos de Execução

A forma como um programa escrito em linguagem de alto nível chega à execução varia significativamente entre linguagens. Identificamos três modelos principais:

3.1 A) Compiladores Puros (Ahead-of-Time - AOT)

Neste modelo, o código fonte é totalmente traduzido para linguagem de máquina *antes* de qualquer execução ocorrer.

- **Exemplos**: C, C++, Rust, Go, Haskell.
- **Fluxo Típico**:

```
Fonte (.c) -> Compilador -> Objeto (.o) -> Linker -> Executável
```

- **Vantagens**: Máxima performance, pois o compilador tem tempo ilimitado para otimizar o código antes da execução; detecção antecipada de erros de tipos e sintaxe.
- **Desvantagens**: Ciclo de desenvolvimento (edit-compile-run) mais lento; o executável gerado é preso à plataforma (sistema operacional e arquitetura) específica.

3.2 B) Intérpretes Puros

Não existe uma fase de tradução para código de máquina persistente. Uma **Máquina Virtual** ou Intérprete lê o código fonte (ou uma representação interna leve como AST) e executa as ações em tempo real.

- **Exemplos**: Bash, versões antigas de PHP, Python (conceitualmente).
- **Fluxo Típico**:

```
Fonte (.py) -> Intérprete (Lê instrução -> Executa instrução)
```

- **Vantagens**: Flexibilidade extrema (código pode ser gerado e executado dinamicamente com 'eval'); portabilidade total, bastando ter o intérprete instalado; ciclo de desenvolvimento imediato.
- **Desvantagens**: Performance significativamente menor (frequentemente 10x a 100x mais lento que código compilado) devido ao overhead de decodificar e despachar instruções repetidamente.

3.3 C) Híbridos (Bytecode e JIT)

Este é o modelo dominante em linguagens modernas corporativas. O código fonte é compilado para uma **Linguagem Intermediária (IL)** padrão e portátil (blo Bytecode), que é então executada por uma Máquina Virtual eficiente.

- **Exemplos**: Java (JVM), C# (.NET CLR), JavaScript (V8/SpiderMonkey).
- **Fluxo Típico**:
  1. *Tempo de Compilação*: `Fonte (.java)` → `Bytecode (.class)`
  2. *Tempo de Execução*: `Bytecode` → `JVM` → `Código de Máquina`
- **Just-In-Time (JIT) Compiler**: A inovação crucial deste modelo. A VM monitora a execução. Se um método é executado frequentemente (é "hot"), o JIT o compila para código de máquina nativo *em tempo de execução*, armazenando-o em cache. Isso permite que linguagens como Java alcancem performance próxima à de C++.

4 Diagramas de Lápide (T-Diagrams)

T-Diagrams são uma notação gráfica fundamental para descrever compiladores e processos de *cross-compilation* e *bootstrapping*. Um T-Diagram possui três componentes dispostos em forma de "T":

1. **Topo Esquerdo (S)**: A linguagem Fonte (Source) que o compilador aceita.
2. **Topo Direito (T)**: A linguagem Alvo (Target) que o compilador gera.

3 **Base (I)**: A linguagem de Implementação na qual o compilador foi escrito e roda.

Isso se lê como: "Um compilador de S para T, escrito em I". Essa notação ajuda a visualizar como construímos compiladores complexos. Por exemplo, o processo de **Bootstrapping** descreve como o primeiro compilador de C foi escrito (provavelmente em Assembly) e como, subsequentemente, reescrevemos o compilador em C e usamos o binário anterior para compilar o novo código fonte.

5 A Anatomia Simplificada de um Compilador

Para fins de estudo e projeto, dividimos o compilador em duas grandes fases, conhecida como modelo **Análise-Síntese**.

5.1 Front-End (Análise)

O foco desta fase é compreender o programa fonte e validar sua correção.

1 **Análise Léxica**: "As palavras existem?" O fluxo de caracteres é convertido em tokens (palavras válidas da linguagem, como ‘if‘, ‘while‘, identificadores).

2 **Análise Sintática**: "A frase faz sentido?" Os tokens são organizados em uma estrutura hierárquica (Árvore Sintática ou AST) conforme a gramática da linguagem.

3 **Análise Semântica**: "O significado é válido?" Verificações de contexto, como tipagem (não somar inteiro com string) e escopo de variáveis.

Saída: Uma Representação Intermediária (IR) ou AST decorada.

5.2 Back-End (Síntese)

O foco desta fase é gerar o código para a máquina alvo da forma mais eficiente possível.

1 **Geração de Código Intermediário**: Criação de uma versão genérica do programa, independente da máquina alvo.

2 **Otimização**: Transformações para tornar o código mais rápido ou menor (eliminação de código morto, unrolling de loops).

3 **Geração de Código Alvo**: Emissão do código final, seja Assembly ou binário de máquina (x86, ARM, JVM Bytecode).

6 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. (Caps 1.1, 1.2)
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. (Cap 1)