Chapter 1

Geração de Código: Seleção, Escalonamento e Alocação

1.1 Objetivos

- Compreender a Geração de Código como a fase final do *Back-End*, que mapeia a Representação Intermediária (IR) para Assembly nativo.
- Entender o profundo e insolúvel **Phase Ordering Problem** (O Problema da Ordem das Fases) entre Seleção, Escalonamento e Alocação.
- Explorar a **Seleção de Instruções** via algoritmos de *Tree Covering* e *Maximal Munch*.
- Analisar o **Escalonamento de Instruções** para evitar *Hazards* e maximizar o uso do pipeline da CPU através de *List Scheduling*.
- Estudar matematicamente a **Alocação de Registradores**, a construção do Grafo de Interferência e as engenhosas heurísticas de Coloração de Chaitin-Briggs.

1.2 O Desafio do Back-End

A Geração de Código é o momento em que as abstrações terminam e o compilador encontra o silício. O objetivo desta fase é traduzir uma IR genérica, independente de máquina, para a linguagem de montagem (Assembly) nativa de uma arquitetura alvo específica (como x86, ARM ou RISC-V).

O trabalho do *Back-End* é garantir simultaneamente velocidade extrema, tamanho binário reduzido e precisão semântica absoluta. Para isso, ele é construído sobre três pilares, ou fases, extremamente complexas e independentes:

1 **Seleção de Instruções:** Quais instruções de máquina nativas deverão ser usadas para representar as operações abstratas?

2 **Escalonamento de Instruções:** Em qual ordem exata essas instruções devem ser despachadas para manter os circuitos do processador ocupados 100% do tempo?

3 **Alocação de Registradores:** Em quais registradores físicos, extremamente escassos, as infinitas variáveis virtuais do programa devem morar?

1.2.1 O Problema da Ordem das Fases (Phase Ordering Problem)

O maior desafio teórico no projeto de back-ends de compiladores é que essas três fases estão intimamente interligadas e, frequentemente, "brigam" entre si. Não existe uma ordem canônica e perfeita de execução.

Se escolhermos **Escalonar antes de Alocar**: o escalonador irá separar ao máximo as instruções vizinhas para explorar o paralelismo da CPU. Ao espalhar as instruções, as variáveis nelas utilizadas permanecerão "vivas" por muito mais tempo. Isso estoura a capacidade física de registradores, forçando o sistema a usar a RAM (*Spilling*), o que derruba a performance brutalmente.

Se escolhermos **Alocar antes de Escalonar**: o alocador tentará espremer as dezenas de variáveis locais no menor número de registradores físicos (como reutilizar o `RAX` várias vezes). Porém, ao reusar registradores, criam-se falsas dependências estruturais (ex: a linha 5 não pode rodar antes da 2, pois as duas calharam de cair no mesmo registrador `RAX`). Isso destrói o grafo de dependências livres e impede completamente o escalonador de reordenar o código!

Devido a este embate, compiladores modernos utilizam dezenas de heurísticas e iteram entre essas passagens.

1.3 Seleção de Instruções

Como escolher as melhores instruções de máquina para implementar uma instrução genérica da IR? Mapear operações afeta drasticamente a velocidade.

Considere a instrução matemática IR para acesso a uma matriz:

```
t1 = reg_base + (reg_index * 8) + constante
```

Uma abordagem ingênua traduziria isso no estilo 1-para-1:

```
// Código gerado (Abordagem Ingênua)
MUL R1, reg_index, 8
ADD R2, R1, reg_base
ADD R3, R2, constante
```

São três instruções separadas, ocupando espaço e ciclos preciosos.

No entanto, a arquitetura x86 possui a complexa instrução `LEA` (Load Effective Address) capaz de fazer cálculo de base+índice+escala+deslocamento nativamente em um único ciclo no hardware de endereço:

```
// Código gerado (Seleção de Instrução Ótima)
LEA RAX, [RBX + RCX*8 + const]
```

1.3.1 Tree Covering e Maximal Munch

Para encontrar sistematicamente a instrução `LEA`, a IR é convertida em Árvores Sintáticas de Operações. Cada instrução da arquitetura alvo é modelada como um "ladrilho" (*tile*) geométrico que pode cobrir um determinado formato da árvore. O objetivo se resume ao clássico problema de **Tree Covering**: cobrir toda a árvore de operações encaixando o menor número possível de ladrilhos, gastando o menor "custo" de ciclos de clock.

Um dos algoritmos clássicos para essa tarefa é o **Maximal Munch**. Ele é um algoritmo guloso que desce pela raiz da árvore e tenta sempre aplicar o "maior" e mais englobante ladrilho nativo disponível (como o do LEA). Se o ladrilho não servir, ele tenta um menor. Por ser extremamente rápido em tempo de execução, é o favorito para compiladores JIT (Just-in-Time).

1.4 Escalonamento de Instruções

Após escolher *quais* instruções usar, devemos escolher *quando* emiti-las. CPUs modernas (Superescalares) engolem e tentam executar dezenas de instruções ao mesmo tempo usando *Pipelining* (linha de montagem).

1.4.1 Hazards e Bolhas (Stalls)

O grande inimigo do *pipeline* é a dependência (Hazard). Observe o seguinte bloco de código:

```
1: LOAD r1, [endereco_a]
2: ADD  r2, r1, 1       // STALL: Esperando o r1 chegar da memória!
3: LOAD r3, [endereco_b]
4: ADD  r4, r3, 1       // STALL: Esperando o r3 chegar da memória!
```

A RAM é centenas de vezes mais lenta que a CPU. Ao chegar na linha 2, o dado `r1` ainda não chegou. A CPU precisa congelar todas as engrenagens, criando uma bolha inútil de dezenas de nanosegundos (Stall).

1.4.2 List Scheduling

O algoritmo para curar o código chama-se **List Scheduling**. Ele opera da seguinte forma:

1 Desenha um Grafo de Dependências.

2 Percebe que as linhas 1 e 2 são dependentes, mas a linha 3 independe completamente das outras duas.

3 Puxa instruções independentes para o topo para preencher as bolhas.

```
// Após Escalonamento:
1: LOAD r1, [endereco_a]
3: LOAD r3, [endereco_b]  // PREENCHEU O STALL: Executa independentemente!
2: ADD  r2, r1, 1         // Quando essa rodar, r1 já chegou da memória!
4: ADD  r4, r3, 1         // Quando essa rodar, r3 já chegou da memória!
```

O mesmo código agora executa perfeitamente sem atrasos.

1.5 Alocação de Registradores via Coloração

A linguagem intermediária de um compilador possui uma quantidade ilimitada de registradores virtuais (`v1, v2, ..., v1000`). Porém, o processador físico só possui alguns (ex: x86_64 tem 16 de propósito geral). O papel final do Back-end é colocar essas 1000 variáveis em apenas 16 buracos. Isso só é possível se as variáveis não coexistirem no tempo.

1.5.1 Liveness Analysis e Grafo de Interferência

Calcula-se o tempo de vida (*liveness*) para descobrir em quais pontos exatos do programa cada variável carrega um valor importante. Com isso em mãos, desenha-se um **Grafo de Interferência**:

- **Vértices:** Cada variável vira um nó.
- **Arestas:** Traçamos uma linha entre duas variáveis se elas estiverem "vivas" ao mesmo tempo. Se houver aresta, elas "interferem" e são proibidas de ser alocadas no mesmo registrador físico.

Atribuir registradores vira um problema estrito de Teoria dos Grafos: encontrar uma coloração para os vértices de forma que nenhum vértice adjacente tenha a mesma cor, usando apenas K cores (K sendo os registradores disponíveis). Esse problema matemático clássico é NP-Completo.

1.5.2 Heurísticas de Chaitin e Briggs

Sendo impossível resolver na força bruta em tempo hábil, **Gregory Chaitin** propôs uma heurística de sucesso que fundamenta a teoria de Compiladores:

> "Se um nó do grafo tem grau menor que K, e conseguirmos colorir todos os outros nós com K cores, garantidamente sobrará uma cor para este nó na hora em que voltarmos com ele para a estrutura."

A técnica envolve ir arrancando do grafo os vértices fáceis (grau $< K$) e os empilhando numa pilha. O pavor ocorre quando restam apenas nós pesados com grau $\geq K$. A abordagem original de Chaitin imediatamente decretava que aquele nó perdeu o direito de ficar num registrador, e era condenado a ir para a memória RAM (um **Spill**).

Anos depois, **Preston Briggs** publicou uma melhoria "otimista". Briggs postulou: não decrete o *spill* imediatamente! Ao invés disso, arranque e empilhe o nó problemático do mesmo jeito. O pulo do gato ocorre na fase de *desempilhar*: quando o vértice pesado voltar para o grafo, há a probabilidade (um "milagre") de que muitos dos seus dezenas de vizinhos tenham compartilhado acidentalmente a *mesma* cor. Se os 10 vizinhos tiverem a mesma cor, eles ocuparam apenas 1 cor global do conjunto K. Logo, sobrará tranquilamente uma cor para salvar o nó pesado sem precisar jogar sua variável na RAM!

A Alocação de Registradores de Chaitin-Briggs transformou o campo da computação de alto desempenho nos anos 80 e 90 e ainda sustenta grande parte dos alocadores modernos de *backends* robustos.

1.6 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Pearson. Capítulo 8.
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. Morgan Kaufmann.
- Chaitin, G. J. (1982). *Register allocation & spilling via graph coloring.* ACM SIGPLAN.
- Briggs, P. et al. (1994). *Improvements to Graph Coloring Register Allocation.* ACM TOPLAS.