Resumos das Unidades 3 e 4 de Compiladores

Unidade 3: Análise Semântica e Estruturação de Dados

Nesta fase, completamos o *front-end* do compilador. O foco deixa de ser apenas a estrutura gramatical e passa a ser o significado e a consistência das instruções.

1. A Tabela de Símbolos (TS)
A TS é a estrutura de dados central que permeia todas as etapas de compilação.

- **Definição e Uso:** É uma estrutura de par chave-valor que suporta inserção e busca eficiente. Embora comece a ser preenchida no analisador léxico (par token/lexema), é na análise sintática que atributos complexos (como tipos e escopo) são adicionados ou verificados.
- **Implementação:** Em Java, utiliza-se frequentemente a classe `Hashtable`. Para evitar colisões (quando chaves diferentes geram o mesmo hash), define-se uma função hash robusta e tratamento de colisões.
- **Integração JFlex e CUP:** Para o seu projeto, a integração é vital. O CUP (gerador de parser LALR) implementa a TS internamente. No arquivo de especificação do JFlex, deve-se incluir a diretiva `%cup` para garantir a compatibilidade dos tokens gerados com a classe `Symbol` do CUP.

2. Tradução Dirigida pela Sintaxe (TDS)
A TDS associa regras semânticas às produções gramaticais, permitindo verificar tipos e preparar a geração de código.

- **Conceito:** Conecta a definição de variáveis ao seu uso e verifica a correção de tipos em expressões.
- **Atributos:** Na árvore de derivação, os nós possuem atributos que podem ser **sintetizados** (derivados dos filhos) ou **herdados** (derivados dos pais ou irmãos).
- **Execução:** A árvore gramatical é percorrida e as regras semânticas são avaliadas. Essas regras podem gerar código, salvar dados na TS ou emitir erros.

Unidade 4: Geração de Código Intermediário, Otimização e Síntese

Esta unidade aborda o *back-end*, onde a representação interna é transformada em instruções executáveis.

1. Geração de Código Intermediário (RI)
Para evitar a complexidade de traduzir a árvore sintática diretamente para código de máquina de diversas arquiteturas ($M \times N$), gera-se um código intermediário independente da máquina alvo.

- **Tipos de RI:**
  - **Árvores Sintáticas Abstratas (AST):** Representação hierárquica onde nós são registros e ponteiros.
  - **Notação Pós-fixa:** Linear, dispensa parênteses e usa pilha para execução (ex: $ABC+*D/$ corresponde a A*(B+C)/D).
  - **Código de Três Endereços:** Instruções simples com no máximo três operandos (ex: $t1 = b + t0$). Facilita muito a otimização e a geração final.

2. Otimização de Código
O objetivo é melhorar a eficiência (tempo e espaço) preservando o significado original do programa.

- **Blocos Básicos e Grafos de Fluxo:** O código é dividido em blocos básicos (sequências de instruções sem desvios internos). O fluxo de controle entre esses blocos forma um grafo (CFG), essencial para análise e otimização.
- **Exemplo Prático:** Redução de operações de memória e desvios desnecessários dentro de loops, transformando o código em instruções mais diretas.

3. Geração de Código Alvo

A etapa final converte a RI para a linguagem da máquina de destino (como Bytecode JVM ou Assembly). Envolve três ações principais, segundo Cooper (2014):

1. **Seleção de instruções:** Mapear a RI para o *instruction set* da máquina (ex: mapear `a=1` para `dconst 1` e `dstore 1` na JVM).
2. **Escalonamento:** Definir a ordem de execução.
3. **Alocação de registros:** Gerenciar o uso limitado de memória rápida do processador.

4. Tendências Atuais

- **Compiladores Híbridos:** A maioria das linguagens modernas (Java, C#) usa uma abordagem híbrida (compilação para bytecode + interpretação/JIT), representando cerca de 34% das linguagens populares.
- **DSLs (Domain Specific Languages):** O conhecimento de compiladores é essencial hoje para criar pequenas linguagens focadas em problemas específicos, aumentando a expressividade do código.