Resumo Acadêmico: Unidades 3 e 4 da Apostila de Compiladores

Este documento sintetiza os conceitos fundamentais apresentados nas Unidades 3 e 4 da apostila, abrangendo a conclusão da fase de análise (análise semântica) e toda a fase de síntese (geração de código intermediário, otimização e geração de código alvo).

Unidade 3: Tabela de Símbolos, Análise Semântica e Tradução Dirigida por Sintaxe

A Unidade 3 foca na verificação do *significado* e *contexto* de um programa, estabelecendo a ponte entre a análise sintática (que verifica a estrutura) e a geração de código.

Análise Semântica

A análise semântica é a etapa final da fase de análise do compilador. Sua principal função é verificar a coerência contextual e o significado do código-fonte, tarefas que o analisador sintático não consegue realizar.

As verificações centrais incluem:

- **Verificação de Tipos:** Assegura que os operandos em uma expressão sejam compatíveis (ex: não permitir a soma de um inteiro com uma string).
- **Verificação de Declaração:** Confere se variáveis e funções foram declaradas antes de serem utilizadas.
- **Verificação de Escopo:** Garante que um identificador está sendo acessado apenas dentro de seu escopo válido.
- **Compatibilidade de Argumentos:** Valida se uma chamada de função utiliza o número correto e os tipos adequados de parâmetros.

Tradução Dirigida pela Sintaxe (TDS)

Esta é a técnica fundamental utilizada para implementar a análise semântica. A TDS associa *atributos* (informações como tipo, valor, etc.) aos símbolos gramaticais (terminais e não-terminais) e *regras semânticas* (ações) às produções da gramática.

Quando o analisador sintático constrói a árvore de derivação, as regras semânticas são executadas para calcular os valores desses atributos, "decorando" a árvore com informações semânticas.

Existem dois tipos principais de atributos:

1. **Atributos Sintetizados:** O valor de um atributo em um nó-pai é determinado pelos valores dos atributos de seus nós-filhos. O fluxo de informação é de baixo para cima (bottom-up). (Ex: calcular o valor de uma expressão aritmética).

2. **Atributos Herdados:** O valor de um atributo em um nó é determinado pelos valores dos atributos de seu pai ou de seus irmãos. O fluxo de informação é de cima para baixo (top-down) ou lateral. (Ex: propagar o tipo de dado de uma declaração, como `int`, para todas as variáveis da lista).

Tabela de Símbolos

A Tabela de Símbolos (TS) é a estrutura de dados central que armazena informações sobre todos os identificadores (variáveis, funções, classes, etc.) utilizados no programa.

- **Função:** É consultada e atualizada por quase todas as fases do compilador. O analisador léxico insere os identificadores (lexemas) na tabela. Os analisadores sintático e semântico a consultam para verificar declarações, tipos e escopo. A fase de geração de código a utiliza para determinar os endereços de memória das variáveis.
- **Implementação:** A eficiência é crucial, pois a TS é acessada constantemente. A apostila destaca que a implementação mais comum e eficiente é através de **Tabelas Hash (Hashtable)**, que oferecem tempo de busca e inserção médio constante (O(1)).

Unidade 4: Geração de Código Intermediário, do Código Alvo e Otimização

A Unidade 4 aborda a fase de síntese (back-end) do compilador, focada na tradução do programa-fonte analisado para um código executável pela máquina.

Geração de Código Intermediário (RI)

Gerar código de máquina (código alvo) diretamente da árvore sintática é complexo e não-portável. Por isso, os compiladores modernos primeiro traduzem a árvore para uma **Representação Intermediária (RI)**.

- **Vantagens:**
    1. **Portabilidade:** Facilita a portabilidade do compilador. Para suportar uma nova arquitetura de máquina, basta escrever um novo back-end que traduza a RI (existente) para o novo código alvo.
    2. **Otimização:** A RI é projetada para ser fácil de otimizar, independentemente da máquina de destino.
- **Formas de RI:**
    1. **Notação Pós-fixa (Polonesa Reversa):** Utilizada por máquinas baseadas em pilha. (Ex: `a + b` torna-se `a b +`).
    2. **Código de Três Endereços (Three-Address Code):** A forma mais comum. As instruções possuem, no máximo, três operandos (um operador, dois operandos e um resultado). (Ex: `t1 = b + c`; `t2 = a * t1`).

3. **Bytecode (Ex: Java):** Um exemplo prático de RI executada por uma Máquina Virtual (JVM), que por sua vez traduz (via JIT) para o código nativo da máquina.

Otimização de Código

Otimização é o processo de transformar a Representação Intermediária (RI) para que ela produza um código alvo mais rápido, menor ou mais eficiente, sem alterar o seu significado (saída).

- **Análise de Fluxo de Controle:** Para otimizar, o compilador primeiro analisa a estrutura do programa.
    - **Blocos Básicos (BB):** Sequências de instruções de três endereços sem desvios para dentro ou para fora, exceto no início e no fim.
    - **Grafos de Fluxo de Controle (CFG):** Um grafo direcionado onde os nós são os Blocos Básicos e as arestas representam os saltos (desvios) entre eles.
- **Técnicas:** O CFG permite ao compilador identificar loops e aplicar otimizações (ex: mover cálculos redundantes para fora de um loop).

Geração de Código Alvo

Esta é a etapa final, onde a RI (otimizada) é traduzida para o código de máquina ou assembly da arquitetura de destino.

As três tarefas principais desta fase são:

1. **Seleção de Instruções:** Mapear as instruções da RI para as instruções de máquina específicas do processador (ex: `LOAD`, `ADD`, `STORE`).
2. **Alocação de Registros:** Decidir quais variáveis e resultados temporários devem ser mantidos nos registradores rápidos da CPU, minimizando acessos lentos à memória.
3. **Escalonamento de Instruções:** Reordenar as instruções de máquina para maximizar o desempenho do processador (ex: evitar que o processador fique ocioso esperando dados da memória).

Tópicos Adicionais

A unidade conclui discutindo a aplicação prática destes conceitos, como a criação de **DSLs (Domain Specific Languages)** — linguagens focadas em um problema específico, como SQL — e o uso avançado de **Expressões Regulares (REGEX)**, cuja teoria fundamenta a análise léxica.