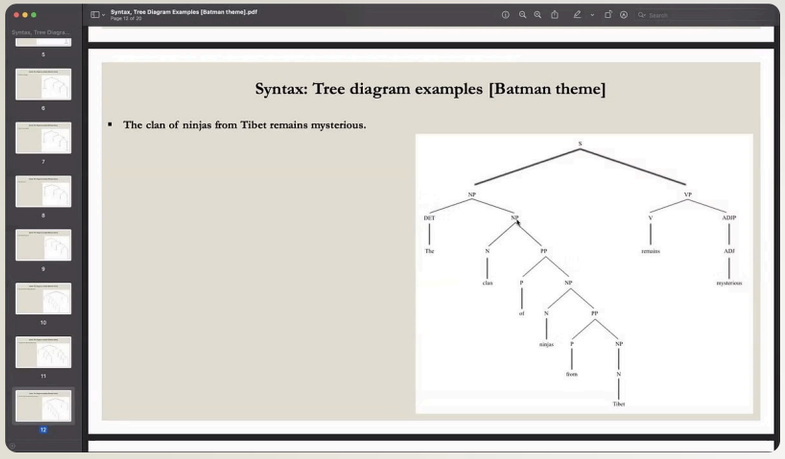
Análise Sintática



O Papel do Analisador Sintático

Verificação Estrutural

O analisador sintático verifica se o programa fonte está estruturalmente correto, seguindo as regras gramaticais da linguagem de programação.

Construção da Árvore Sintática

Gera uma representação hierárquica (árvore sintática) do programa, essencial para as fases subsequentes de compilação.

Detecção de Erros

Identifica e reporta erros sintáticos no código fonte, auxiliando os programadores na correção de problemas estruturais.

Preparação para Análise Semântica

Fornece uma estrutura organizada que facilita a análise semântica e a geração de código intermediário.

Detecção de erros - Tipos de erros

- Ordem incorreta dos tokens;
  - soma = a b +;
- Falta de símbolos;
  - Falta de ponto e vírgula
- Abertura e fechamento de blocos incorretos.
  - Quantidade menor de colchete ou chaves

Estratégias de tratamento de erros

- Panic Mode (Modo pânico) : Descarta os símbolos de entrada até achar um token de sincronismo;
  - Exemplos de tokens de sincronismo: "}" ou ";"
- Recuperação a nível de frase: inclui, exclui ou substitui um prefixo da entrada por outro que permite a continuação da análise;
  - Não muda o código. Faz internamente apenas para seguir a análise.
- Produções de erro: incluir como produções da gramática os erros mais comuns

Gramática Livres de Contexto

- Símbolos Terminais
  - Os símbolos a partir dos quais as cadeias são formadas (tokens)
- Símbolos Não Terminais
  - representam um conjunto de cadeias
- Um símbolo de partida;
  - Um símbolo não terminal usado como raiz da árvore
- Produções
  - Especificam como os terminais e não terminais se combinam para formar as cadeias

Análise Sintática Descendente

1

Início

A análise começa pelo símbolo inicial da gramática, decompondo-o em seus componentes.

2

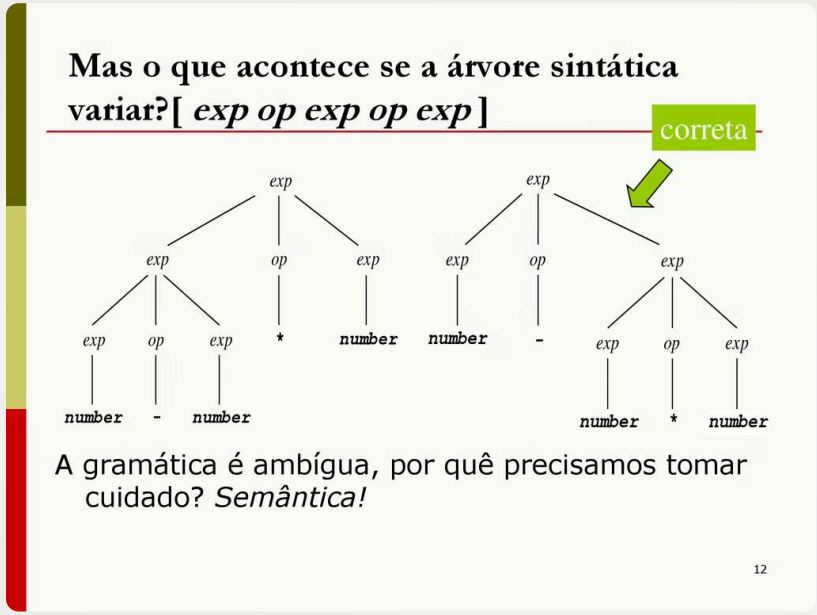
Expansão

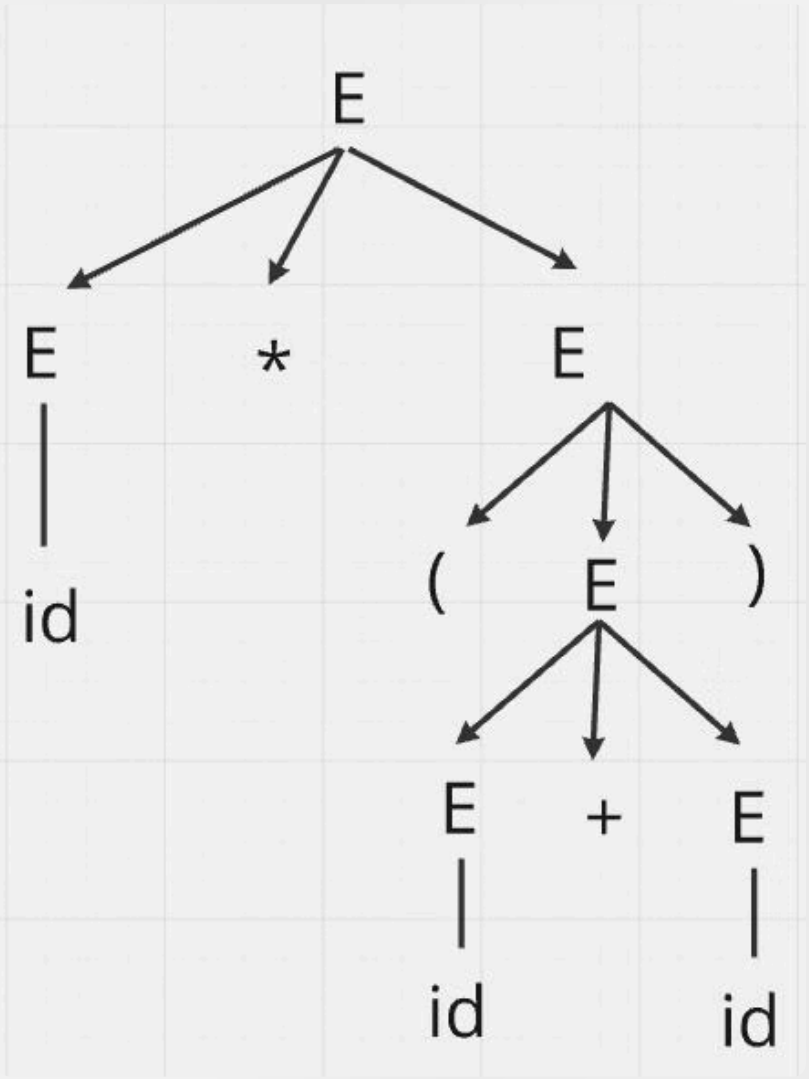
O analisador expande cada símbolo não-terminal, seguindo as regras de produção da gramática.

3

Correspondência

Os símbolos terminais são comparados com os tokens de entrada para verificar a correspondência.





Exemplo de Análise Descendente

Gramática:

E → E + E

E → E * E

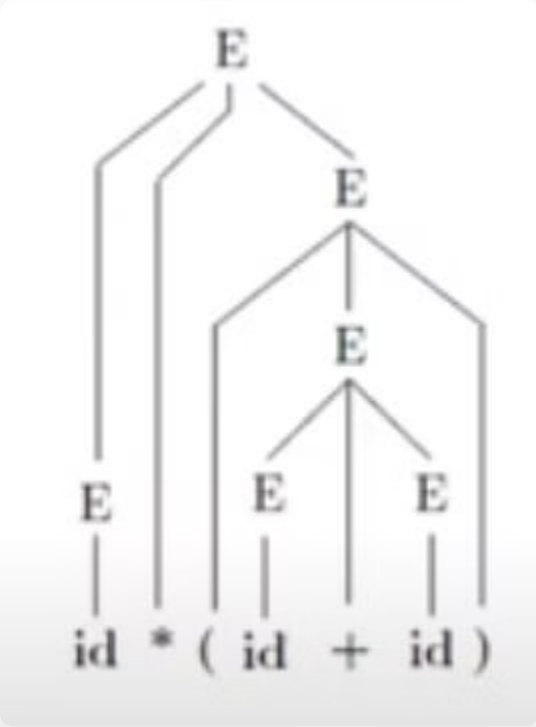
E → (E)

E → id

CADEIA:

id * (id + id)

Análise Sintática Ascendente



1

Início na Base

A análise começa com os tokens de entrada, construindo a árvore sintática de baixo para cima.

2

Shift

O analisador move (shift) tokens da entrada para a pilha de análise.

3

Reduce

Aplica reduções quando uma sequência na pilha corresponde ao lado direito de uma produção.

4

Aceitação

O processo continua até que toda a entrada seja consumida e o símbolo inicial seja alcançado.

Análise LL(k) vs. LR(k)

LL(k)

- Análise descendente predictiva
- Olha k símbolos à frente
- Mais fácil de implementar manualmente
- Limitada a gramáticas LL(k)

LR(k)

- Análise ascendente
- Olha k símbolos à frente
- Geralmente implementada com tabelas
- Cobre uma classe maior de gramáticas

Comparação

- LR(k) é mais poderoso que LL(k)
- LL(k) é mais intuitivo para gramáticas simples
- LR(k) lida melhor com ambiguidades

Códigos Intermediários

1 Representação de Três Endereços

Uma forma comum de código intermediário que utiliza no máximo três operandos por instrução, facilitando a otimização e a geração de código final.

2 Árvore Sintática Abstrata (AST)

Uma representação em árvore da estrutura sintática do código fonte, removendo detalhes desnecessários e focando na estrutura lógica do programa.

3 Grafo de Fluxo de Controle (CFG)

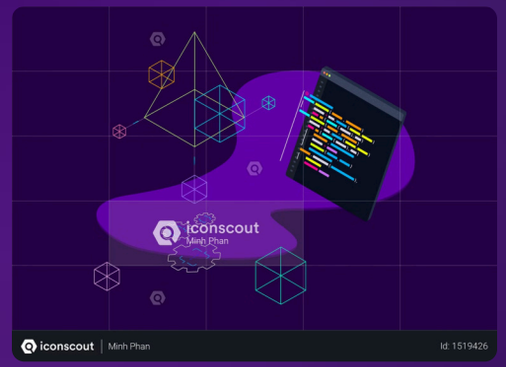
Representa a estrutura de controle do programa, mostrando todos os caminhos possíveis que a execução pode seguir, crucial para muitas otimizações.

```
/**
 * @dev Transfers `tokenId` from `from` to `to`.
 *
 * Requirements:
 *
 * - `from` cannot be the zero address.
 * - `to` cannot be the zero address.
 * - `tokenId` token must be owned by `from`.
 * - If the caller is not `from`, it must be approved to move this token
 * by either {approve} or {setApprovalForAll}.
 *
 * Emits a {Transfer} event.
 */
function transferFrom(
    address from,
    address to,
    uint256 tokenId
) public payable virtual override {
    uint256 prevOwnershipPacked = _packedOwnershipOf(tokenId);

    if (address(uint160(prevOwnershipPacked)) != from) revert TransferFromIncorrectOwner();

    (uint256 approvedAddressSlot, address approvedAddress) = _getApprovedSlotAndAddress(tokenId);

    ......
    // Underflow of the sender's balance is impossible because we check for
    // ownership above and the recipient's balance can't realistically overflow.
    // Counter overflow is incredibly unrealistic as `tokenId` would have to be 2**256.
    unchecked {
        // We can directly increment and decrement the balances.
        --ERC721AStorage.layout()._packedAddressData[to]; // Updates: `balance -= 1`.
        ++ERC721AStorage.layout()._packedAddressData[from]; // Updates: `balance += 1`.
    ......
}
```



Otimização de Código: Visão Geral

1 2 3 4

Análise

O compilador analisa o código intermediário para identificar oportunidades de melhoria, como operações redundantes ou código não utilizado.

Transformação

Aplicação de técnicas de otimização para melhorar o desempenho ou reduzir o tamanho do código, preservando a semântica original do programa.

Verificação

Garantia de que as otimizações não alteraram o comportamento do programa, muitas vezes através de análises formais ou testes extensivos.

Iteração

O processo de otimização é frequentemente iterativo, com múltiplas passagens aplicando diferentes técnicas até que não sejam mais possíveis melhorias significativas.

Otimizações Locais

Eliminação de Subexpressões Comuns

Identifica e remove cálculos redundantes dentro de um bloco básico, armazenando o resultado em uma variável temporária para reutilização.

Propagação de Constantes

Substitui variáveis que têm valores constantes conhecidos em tempo de compilação por suas constantes literais, simplificando expressões.

Dobramento de Constantes

Avalia expressões constantes em tempo de compilação, reduzindo a carga computacional em tempo de execução.

Eliminação de Código Morto

Remove instruções que não afetam o resultado do programa, como variáveis não utilizadas ou código inacessível.