

Análise Sintática I

Eduardo Ferreira dos Santos

Ciência da Computação
Centro Universitário de Brasília – UniCEUB

Abril, 2017

Sumário

- 1 Introdução
- 2 Derivações
- 3 Ambiguidade
- 4 Análise sintática descendente
- 5 Análise sintática ascendente

- 1 Introdução
- 2 Derivações
- 3 Ambiguidade
- 4 Análise sintática descendente
- 5 Análise sintática ascendente

Fases

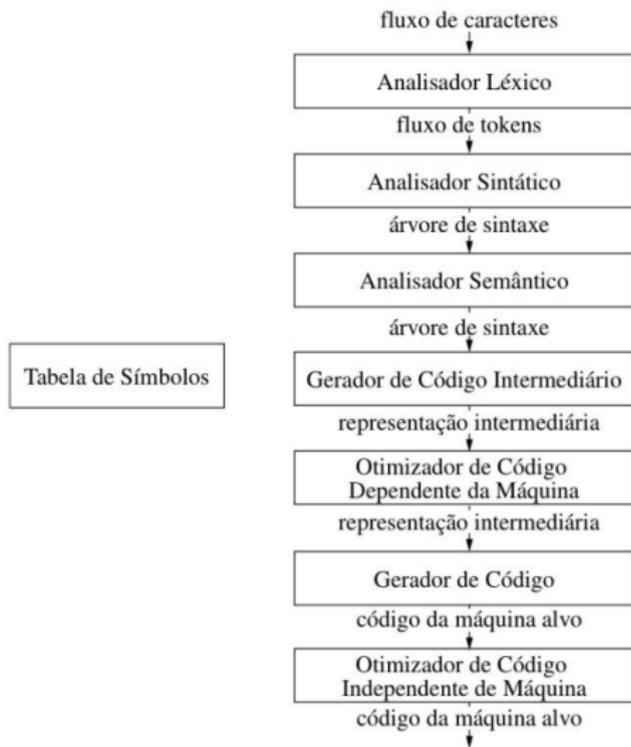


Figura 1.1: Fases do compilador [Aho et al., 2007]

Tokens, padrões e lexemas

Token Dupla formada pelo **nome** e **valor de atributo** (opcional)

- Nome do token: símbolo abstrato que representa um **tipo de unidade léxica** [Aho et al., 2007];
- Também é um **símbolo de entrada** para o analisador sintático.

Padrões Descrição da **forma** que os lexemas de um token podem assumir:

- Palavra-chave como token;
- Casamento de sequências de caracteres.

Lexemas Sequência de caracteres no programa fonte que casa com o padrão para um token.

- Identificado como **instância** desse token.

Papel do analisador sintático

- O analisador sintático recebe do analisador léxico a **cadeia de tokens**;
- Verifica se a cadeia pertence à **linguagem** gerada pela gramática;
- Deve emitir mensagens de erro e recuperar-se desses erros;
- Caso ocorra uma falha, deve continuar processando o programa.

Métodos de análise

- Técnicas de análise sintática [Aho et al., 2007]:
 - 1 Universal;
 - 2 Ascendente (*bottom up*);
 - 3 Descendente (*top down*).
- A abordagem universal é considerada muito ineficiente para ser utilizada em compiladores [de Alencar Price and Toscani, 2000];
- Os métodos utilizados em compiladores utilizam a abordagem ascendente ou descendente:

Métodos descendentes Constroem a árvore de derivação de cima (raiz) para baixo (folhas);

Métodos ascendentes Realizam a análise no sentido inverso, começando nas folhas e seguindo até a raiz.

Execução

A saída do analisador sintático é alguma representação da árvore de derivação para a cadeia de tokens reconhecidos pelo analisador léxico. [Aho et al., 2007]

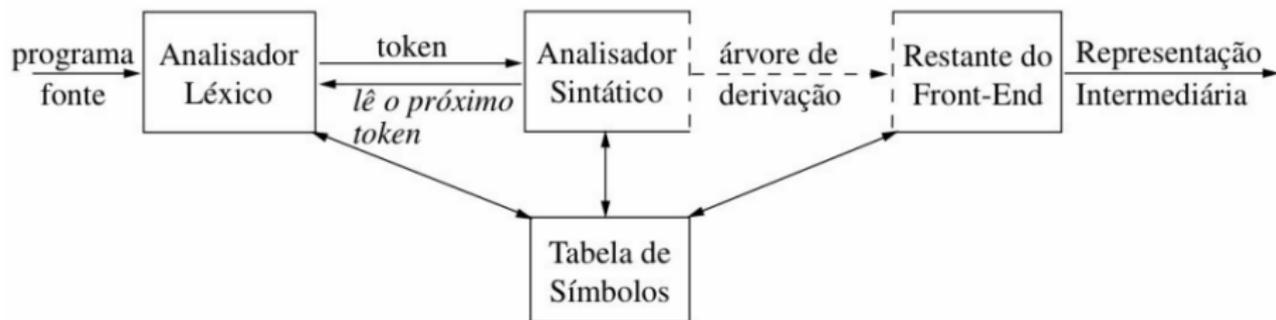


Figura 1.2: Posição do analisador sintático no modelo de compilador [Aho et al., 2007]

Gramáticas representativas

- As palavras-chave, como **while** ou **if** são fáceis de analisar;
- Precisamos representar claramente as **expressões**;
- A gramática da Figura 9 representa análise sintática **descendente**, pois possui **recursividade à esquerda**.

$$\begin{array}{l}
 E \rightarrow E + T \mid T \\
 T \rightarrow T * F \mid F \\
 F \rightarrow (E) \mid \mathbf{id}
 \end{array}$$

Figura 1.3: Gramática que possui associatividade e precedência [Aho et al., 2007]

Análise descendente

- É possível remover a recursividade do exemplo da Figura 9;
- Ao remover a recursividade, aplicamos a **análise sintática descendente**.

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$f \rightarrow (E) \mid \mathbf{id}$$

Figura 1.4: Variação da gramática da Figura 9 sem recursividade à esquerda [Aho et al., 2007]

Produzindo uma análise

- A cada passo da análise descendente, o problema é determinar a **produção** a ser aplicada em um **não-terminal**;
- Ao escolher uma produção, o restante do processo consiste em “casar” os símbolos dos terminais do corpo da produção com a cadeia de entrada;
- **Análise sintática de descida recursiva**: pode ser necessário retroceder na cadeia para encontrar a produção a ser aplicada;
- **Reconhecimento sintático preditivo**: caso de análise de recursiva de descida onde não é necessário nenhum retrocesso;
- Um analisador **preditivo** pode escolher a melhor produção lendo o **próximo símbolo** de entrada.
- Análise sintática descendente não-recursiva: utilização de pilhas para evitar chamadas recursivas.

Tratamento de erros

- O compilador deve auxiliar o **rastreio de erros**;
- Alguns tipos de erro mais comuns:
 - Erro léxico** Erro de ortografia em palavras-chave, identificadores ou operadores;
 - Erro sintático** Incluem ponto e vírgula mal colocado e chave faltando. Geram uma árvore de parsing incorreta;
 - Erro semântico** Erro entre operador e operando, por exemplo;
 - Erro lógico** Mais difícil de detectar, acontece por erro do programador. Um exemplo é a utilização de = no lugar de ==.

Recuperação de erro

- Algumas estratégias para recuperação de erros:

Modo pânico Descarta um símbolo de entrada de cada vez até que um **delimitador de sincronismo** seja encontrado;

Nível de frase Aplica uma correção local ao restante da entrada, como por exemplo adição de ; ou substituição de , por ;

Global Tenta alterar a árvore de parsing até obter uma entrada correta.

- 1 Introdução
- 2 Derivações
- 3 Ambiguidade
- 4 Análise sintática descendente
- 5 Análise sintática ascendente

Definição

- **Definição:** representação gráfica de uma derivação que filtra a ordem na qual as produções são aplicadas.
- Cada nó interior da árvore de derivação representa a **aplicação de uma produção**;
- Pequeno algoritmo:
 - 1 O nó interior é rotulado como o não terminal A do lado esquerdo da produção;
 - 2 Dos filhos dos nós são rotulados, da esquerda para a direita, pelos símbolos do corpo da produção pelo qual A foi substituído.

Exemplo de derivação

$$E \rightarrow -E \rightarrow -(E) \rightarrow -(E + E) \rightarrow -(id + E) \rightarrow -(id + id) \quad (1)$$

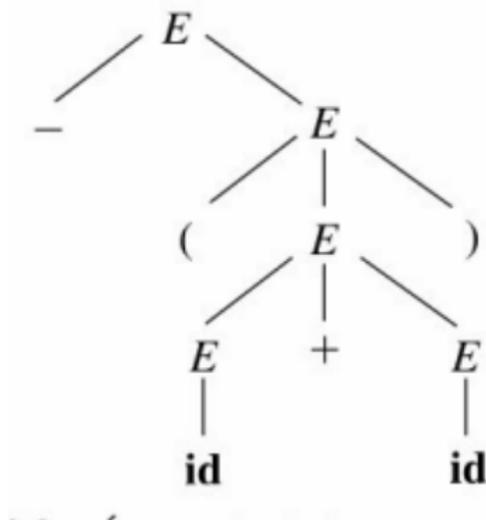


Figura 2.1: Árvore de derivação para a cadeia $-(id + id)$ do exemplo 1 [Aho et al., 2007]

Ambiguidade

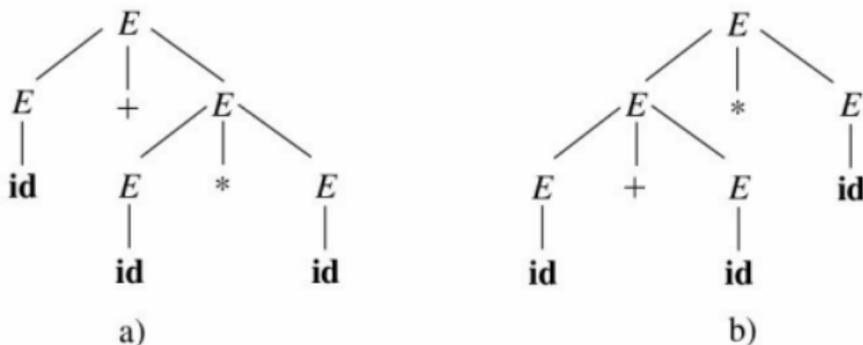


Figura 2.2: Árvore de derivação para a cadeia $id + id * id$ do exemplo da Figura 9 [Aho et al., 2007]

- 1 Introdução
- 2 Derivações
- 3 Ambiguidade**
- 4 Análise sintática descendente
- 5 Análise sintática ascendente

Gramáticas e ambiguidade

- Uma gramática é **ambígua** se existem múltiplas derivações para uma cadeia de entrada [Amarasinghe and Rinard, 2010];
- Múltiplas derivações implicam em múltiplas **árvores de parsing**;
- As derivações e árvore de parsing normalmente refletem a **semântica do programa**;
- Em uma gramática, a ambiguidade normalmente reflete ambiguidade na **semântica da linguagem**, o que não é desejável.

Exemplo

Op = +|-|*|/

Int = [0-9] [0-9]*

Open = <

Close = >

Start

Expr

Expr Op Expr

Open Expr Close Op Expr

Open Expr Op Expr Close Op Expr

Open Int Op Expr Close Op Expr

Open Int Op Expr Close Op Int

Open Int Op Int Close Op Int

1) *Start* → *Expr*

2) *Expr* → *Expr Op Expr*

3) *Expr* → Int

4) *Expr* → Open *Expr* Close

< 2 - 1 > + 1

Figura 3.1: Primeiro exemplo de derivação [Amarasinghe and Rinard, 2010]

Exemplo de ambiguidade

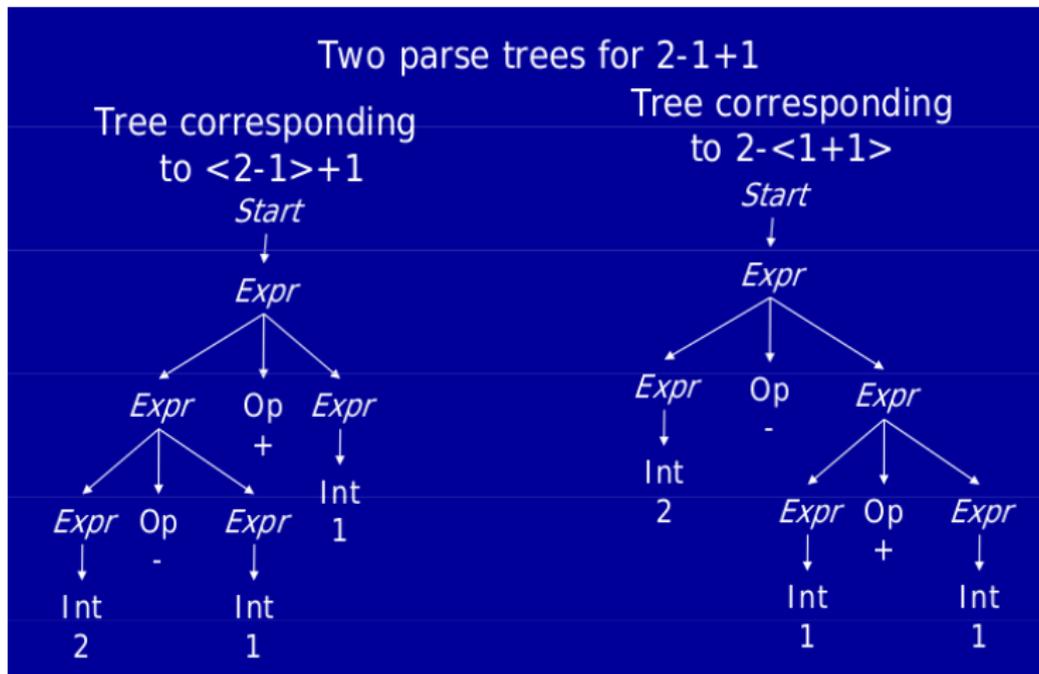


Figura 3.2: Exemplo de ambiguidade para a gramática
[Amarasinghe and Rinard, 2010]

Eliminando a ambiguidade

- Vamos aplicar um *hack* na gramática;
- Todos os operadores vão sofrer associatividade à esquerda.

Original Grammar	Hacked Grammar
$Start \rightarrow Expr$	$Start \rightarrow Expr$
$Expr \rightarrow Expr Op Expr$	$Expr \rightarrow Expr Op Int$
$Expr \rightarrow Int$	$Expr \rightarrow Int$
$Expr \rightarrow Open Expr Close$	$Expr \rightarrow Open Expr Close$

Figura 3.3: Eliminando a ambiguidade [Amarasinghe and Rinard, 2010]

Árvore de parsing

- Agora existe somente uma árvore de parsing para a cadeia 2-1+1

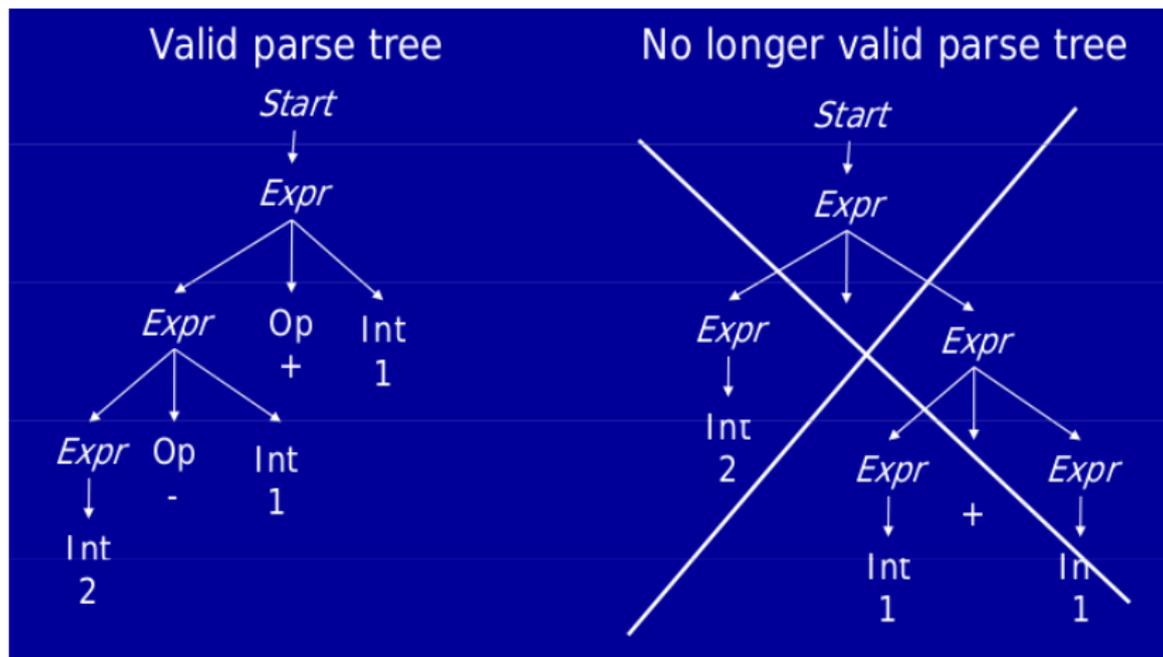


Figura 3.4: Eliminando a ambiguidade [Amarasinghe and Rinard, 2010]

Precedência

Original Grammar	Hacked Grammar
$Op = + - * /$	$AddOp = + -$
$Int = [0-9] [0-9]^*$	$MulOp = * /$
$Open = <$	$Int = [0-9] [0-9]^*$
$Close = >$	$Open = <$
	$Close = >$
$Start \rightarrow Expr$	$Start \rightarrow Expr$
$Expr \rightarrow Expr Op Int$	$Expr \rightarrow Expr AddOp Term$
$Expr \rightarrow Int$	$Expr \rightarrow Term$
$Expr \rightarrow Open Expr Close$	$Term \rightarrow Term MulOp Num$
	$Term \rightarrow Num$
	$Num \rightarrow Int$
	$Num \rightarrow Open Expr Close$

Figura 3.5: Aplicação da precedência de operadores
[Amarasinghe and Rinard, 2010]

Nova árvore

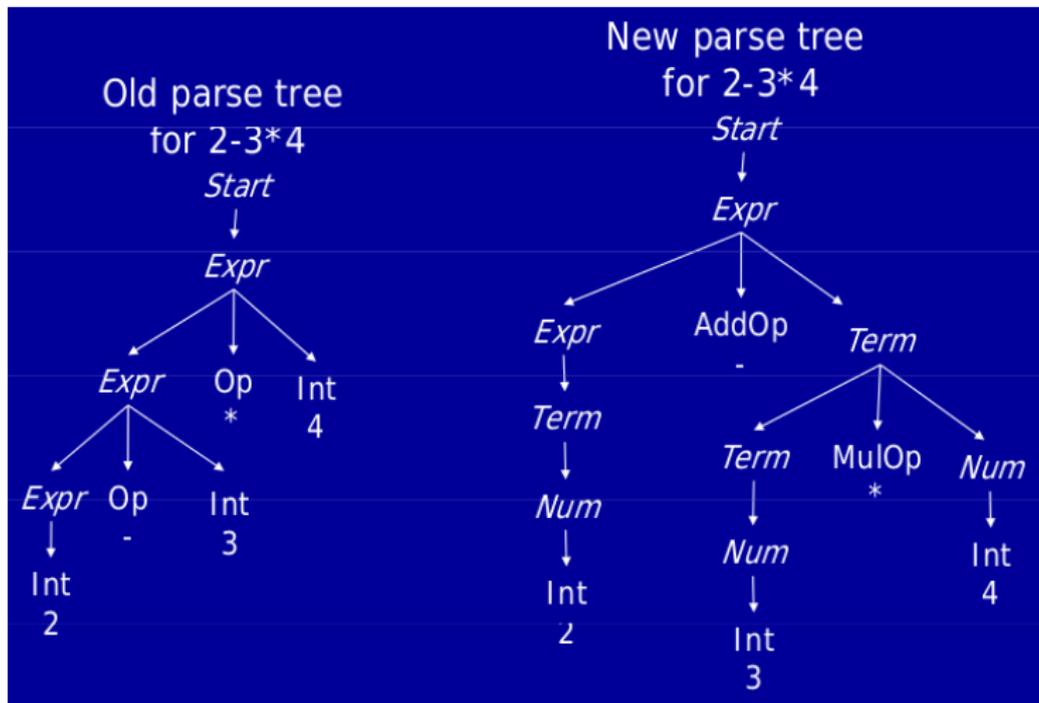


Figura 3.6: Alteração na árvore após aplicação da precedência
[Amarasinghe and Rinard, 2010]

Princípios gerais [Amarasinghe and Rinard, 2010]

- Agrupar operadores em níveis de precedência:
 - * e / são mais fortes, de maior nível;
 - + e - são o próximo nível.
- Um não terminal para cada nível de precedência:
 - *Term* é não terminal para * e /
 - *Expr* é não terminal para + e -
- Utilização de associatividade à esquerda em cada um dos níveis;
- Generalização para níveis arbitrários de precedência.

Parser

- Converte o programa em uma árvore de parsing;
- Pode ser escrito à mão ou por um parser automático:
 - Aceita uma gramática como entrada;
 - Produz um parser como saída.
- Deve ser levado em consideração o **nível de abstração** do problema.

- 1 Introdução
- 2 Derivações
- 3 Ambiguidade
- 4 Análise sintática descendente**
- 5 Análise sintática ascendente

Conceitos

- Constrói a cadeia de cima para baixo;
- **Problema principal**: determinar a próxima produção a ser aplicada para um não terminal;
- Trazemos de volta o exemplo da figura 29.

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$f \rightarrow (E) \mid \mathbf{id}$$

Figura 4.1: Variação da gramática sem recursividade à esquerda [Aho et al., 2007]

Exemplo

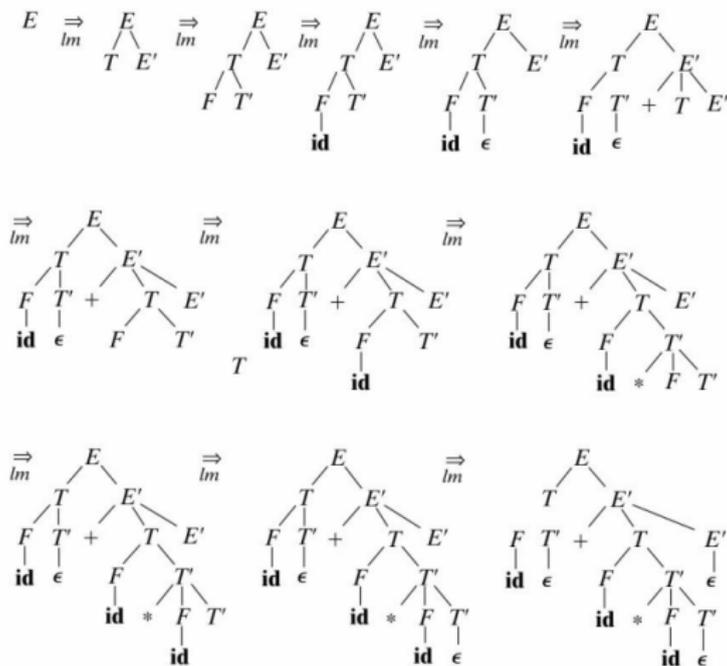


Figura 4.2: Análise sintática descendente para $\text{id} + \text{id} * \text{id}$ [Aho et al., 2007]

Análise sintática de descida recursiva

- Algoritmo simples:
 - 1 Escolhe uma produção para o primeiro caractere que casar;
 - 2 Analisa o próximo símbolo até acabar a entrada;
 - 3 Informa o sucesso se conseguir reconhecer toda a entrada.
- Pode ser **não determinista** por exigir **retrocesso**;
- Acabam não sendo muito eficientes.

Exemplo

```

void A() {
1)   Escolha uma produção-A,  $A \rightarrow X_1 X_2 \dots X_k$ ;
2)   for (  $i = 1$  até  $k$  ) {
3)       if (  $X_i$  é um não-terminal )
4)           ative procedimento  $X_i()$ ;
5)       else if (  $X_i$  igual ao símbolo de entrada  $a$  )
6)           avance na entrada para o próximo símbolo terminal;
7)       else /* ocorreu um erro */;
    }
}

```

Figura 4.3: Exemplo de analisador descendente [Aho et al., 2007]

O modelo é *não recursivo*. Como introduzir a recursividade?

Modelos preditivos

- Um analisador **preditivo** escolhe a próxima produção examinando o próximo símbolo de entrada;
- Necessita de alguma **estrutura de dados** para armazenar as cadeias que já foram analisadas;
- Utilização de **memória** (pilha);
- Algoritmo do tipo FIRST e FOLLOW.

Analizador preditivo

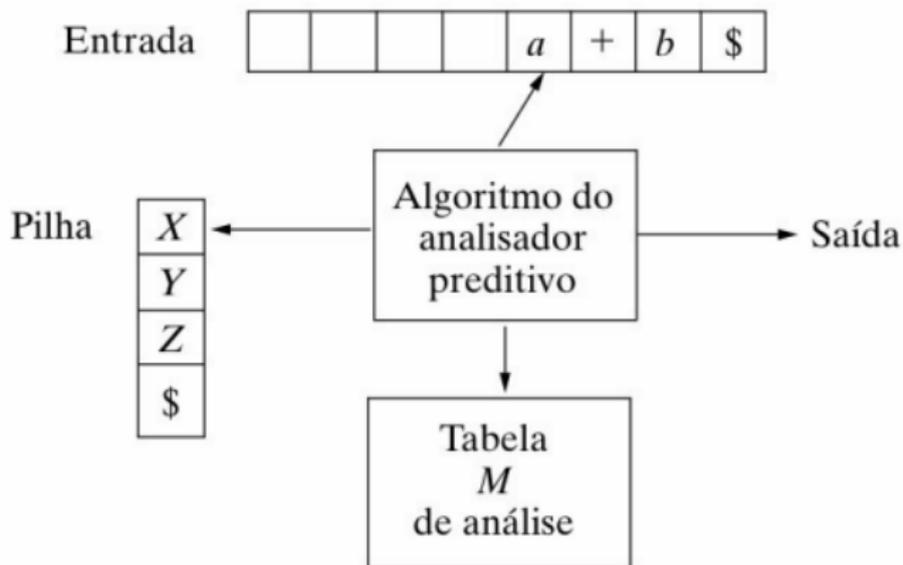


Figura 4.4: Modelo de analisador sintático dirigido por tabela [Aho et al., 2007]

- 1 Introdução
- 2 Derivações
- 3 Ambiguidade
- 4 Análise sintática descendente
- 5 Análise sintática ascendente

Conceitos

- A análise ascendente faz a construção da árvore a partir das folhas (raiz);
- Utilização de um método geral chamado **shift-reduce**.

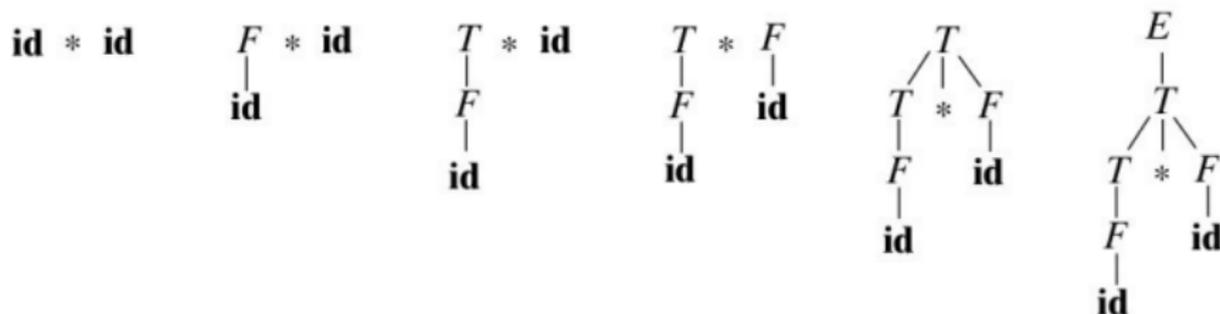


Figura 5.1: Uma análise ascendente para **id * id** [Aho et al., 2007]

Autômato pushdown

- **Definição:** autômato finito que utiliza uma pilha como memória;
- Realiza uma das três operações:
 - Shift** Move o símbolo atual para a pilha (memória);
 - Reduce** Realize uma produção, se houver. A partir daí, executa uma sequência de ações:
 - Executa pop nos símbolos para fora da pilha;
 - Dá um push no símbolo para dentro da pilha.
 - Aceita** Aceita a cadeia de caracteres.
- O autômato *pushdown* executa o algoritmo *shift-reduce* para produzir a árvore de parsing.

Implementação

- Reconstrói a árvore a partir da palavra de entrada;
- Lê a entrada da esquerda para a direita;
- Constrói uma árvore a partir da abordagem **descendente**;
- Utiliza uma pilha como memória de sequências de terminais e não terminais que ainda estão pendentes.

Exercício I

Prove que a gramática é ambígua e construa uma solução para essa ambiguidade.

$$Expr \rightarrow Expr Op Expr$$
$$Expr \rightarrow (Expr)$$
$$Expr \rightarrow - Expr$$
$$Expr \rightarrow \text{num}$$
$$Op \rightarrow +$$
$$Op \rightarrow -$$
$$Op \rightarrow *$$

Figura 5.2: Exemplo de gramática com conflito [Aho et al., 2007]

Método de análise

- É possível resumir o método de análise e síntese estudado nos seguintes passos:
 - 1 Defina uma gramática;
 - 2 Dada uma gramática produza um parser (análise léxica);
 - 3 Use a gramática e a técnica de *shift-reduce* para criar uma **tabela de parsing**;
 - 4 Use a tabela de parsing para produzir **gerador de código**.
- A tabela de parsing é a entrada para a **análise semântica**.

OBRIGADO!!!
PERGUNTAS???

-  Aho, A., Lam, M., Sethi, R., and Ullman, J. (2007). *Compiladores—Princípios Técnicas e Ferramentas*. Pearson, 2a. edition.
-  Amarasinghe, S. and Rinard, M. (2010). Computer language engineering. Disponível em <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/> Acessado em 02/08/2016.
-  de Alencar Price, A. M. and Toscani, S. S. (2000). *Implementação de linguagens de programação: compiladores*. Sagra-Luzzatto.