Análise Sintática II

Eduardo Ferreira dos Santos

Ciência da Computação
Centro Universitário de Brasília – UniCEUB

Outubro, 2016

Sumário

Fases

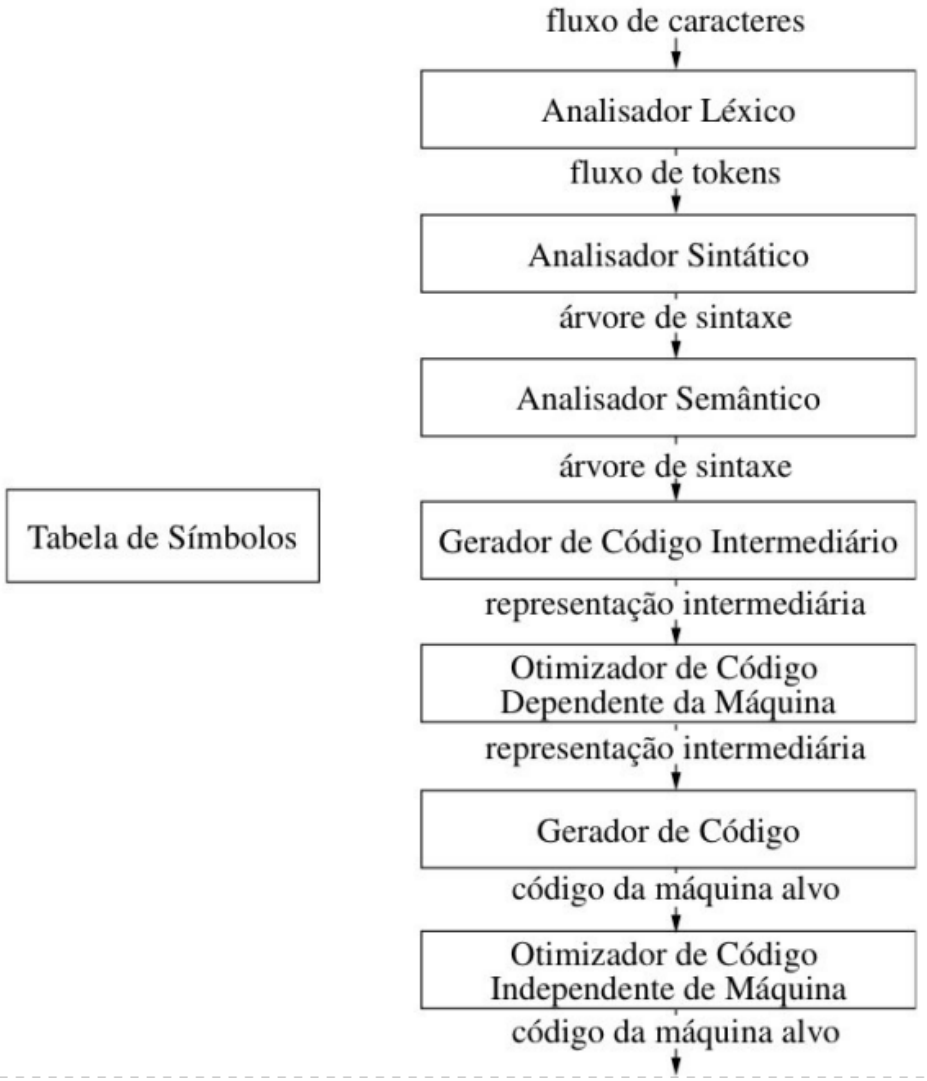

Figura 1.1: Fases do compilador [Aho et al., 2007]

Árvore de sintaxe

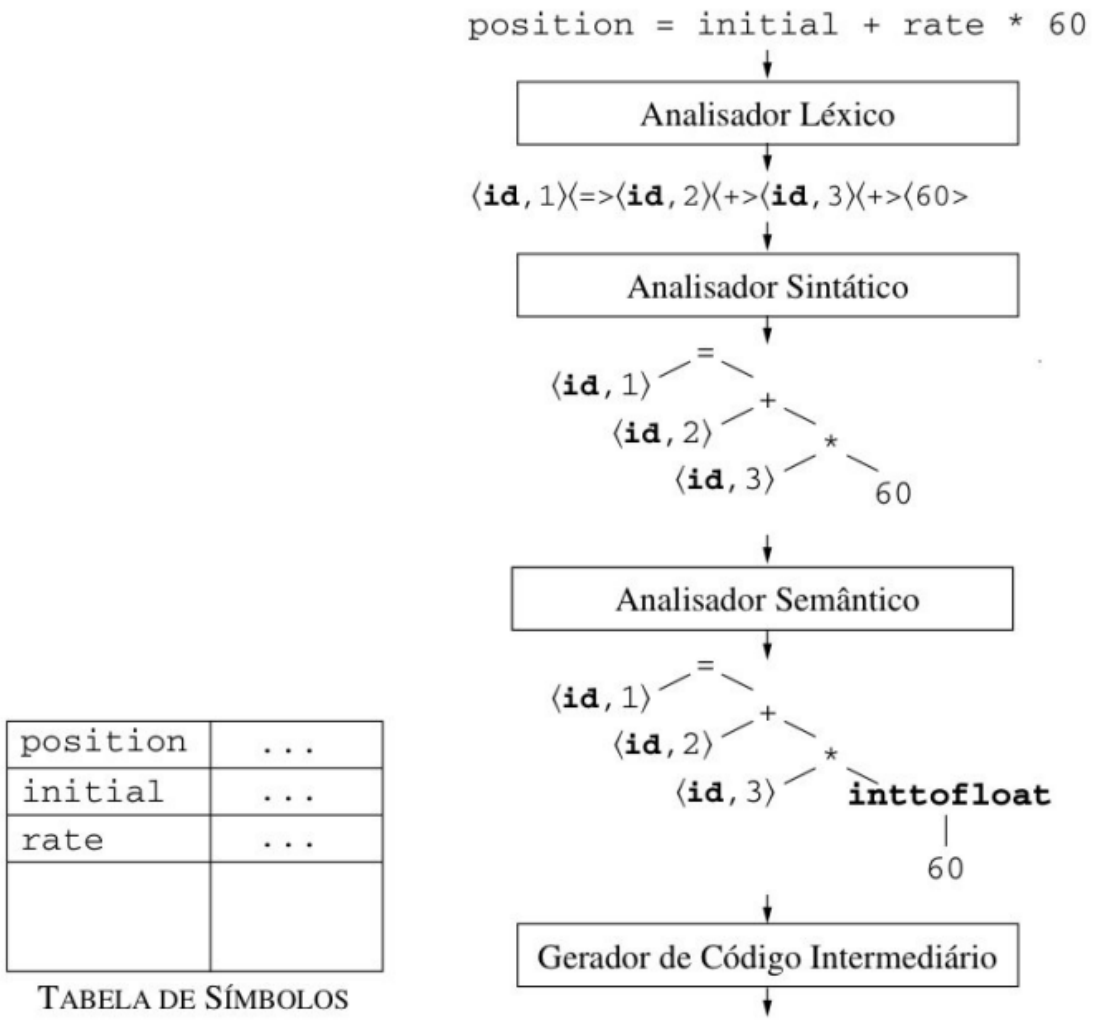

Figura 1.2: Modelo de tradução e atribuição [Aho et al., 2007]

Métodos de análise

- Técnicas de análise sintática [Aho et al., 2007]:
  1. Universal;
  2. Ascendente (*bottom up*);
  3. Descendete (*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]

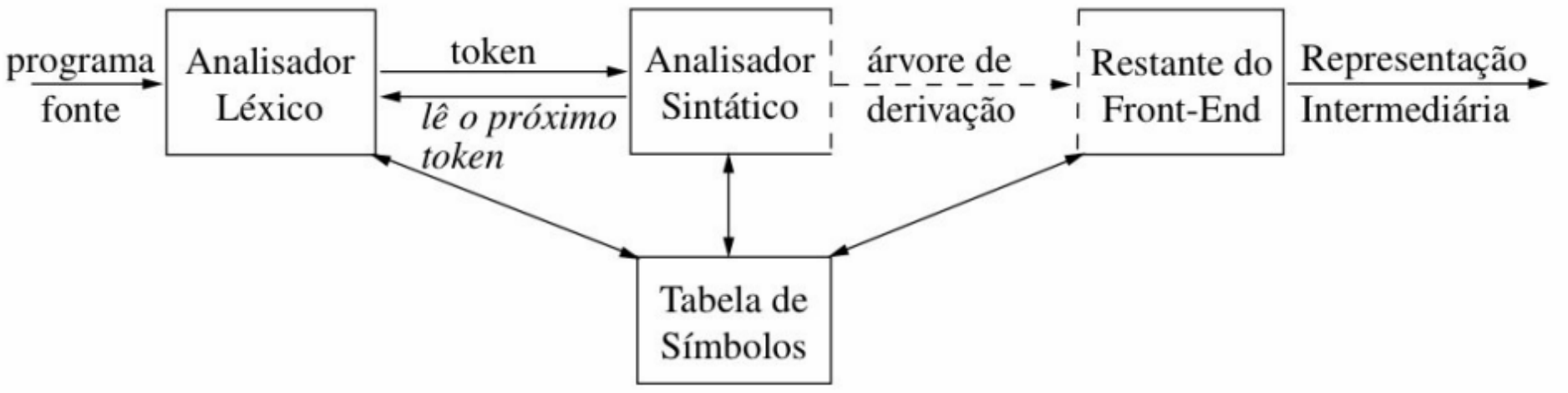

Figura 1.3: Posição do analisador sintático no modelo de compilador [Aho et al., 2007]

Análise sintática descendente

- Voltemos ao exemplo da gramática que sofre remoção de recursividade;
- Ao remover a recursividade, aplicamos a análise sintática descendente.

$$
\begin{aligned}
E &\rightarrow T\,E' \\
E' &\rightarrow +\,T\,E' \mid \epsilon \\
T &\rightarrow F\,T' \\
T' &\rightarrow *\,F\,T' \mid \epsilon \\
f &\rightarrow (\,E\,) \mid \mathbf{id}
\end{aligned}
$$

Figura 1.4: Variação da gramática 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.

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 = >

1) *Start* → *Expr*
2) *Expr* → *Expr* Op *Expr*
3) *Expr* → Int
4) *Expr* → Open *Expr* 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
< 2 - 1 > + 1

Figura 2.1: Primeiro exemplo de derivação [Amarasinghe and Rinard, 2010]

Exemplo de ambiguidade

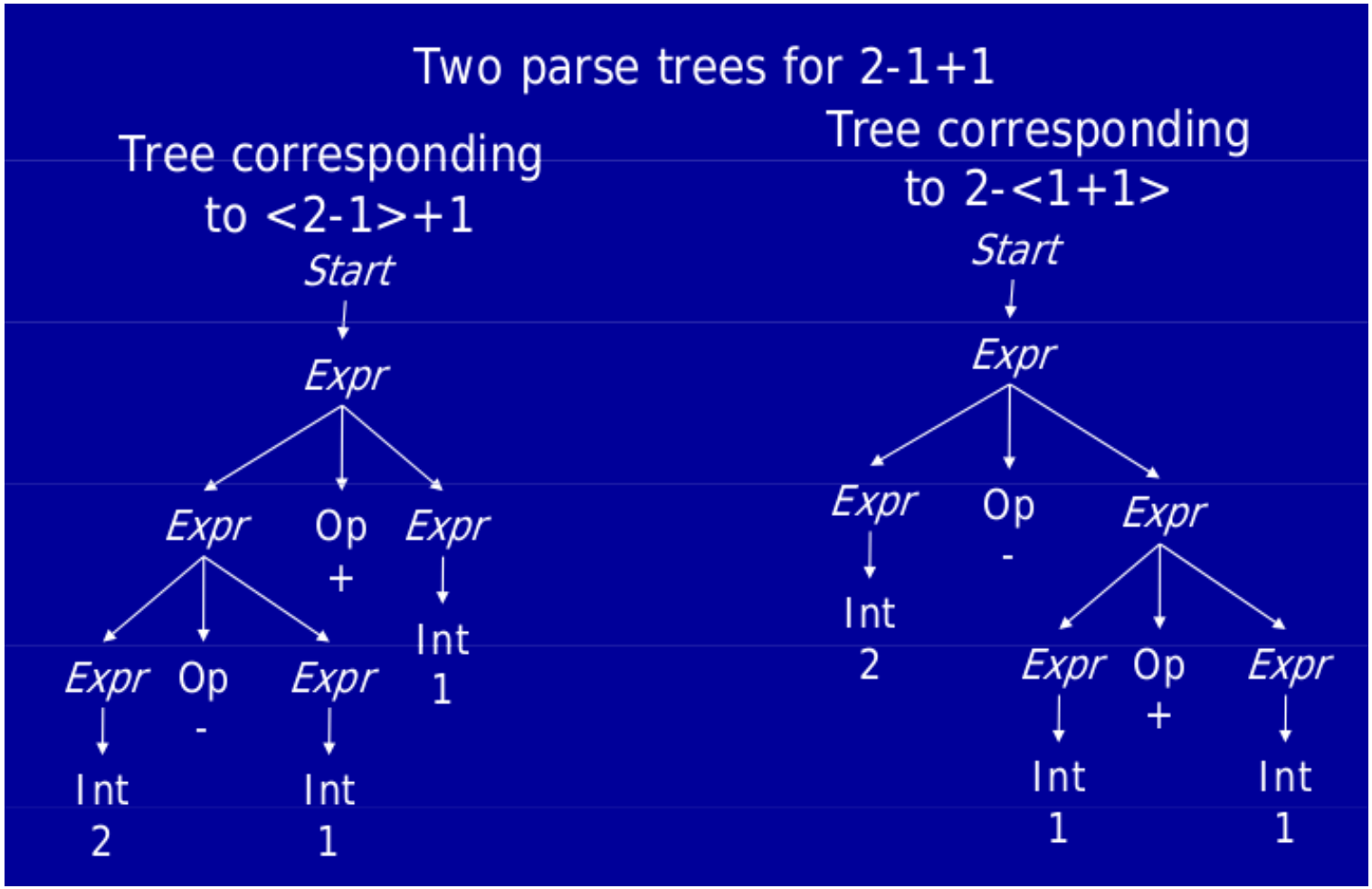

Figura 2.2: Exemplo de ambiguidade para a gramática [Amarasinghe and Rinard, 2010]

Eliminando a ambiguidade

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

Original Grammar	Hacked Grammar
Start → *Expr*	*Start* → *Expr*
Expr → *Expr* Op *Expr*	*Expr* → *Expr* Op Int
Expr → Int	*Expr* → Int
Expr → Open *Expr* Close	*Expr* → Open *Expr* Close

Figura 2.3: Eliminando a ambiguidade [Amarasinghe and Rinard, 2010]

Árvore de parsing

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

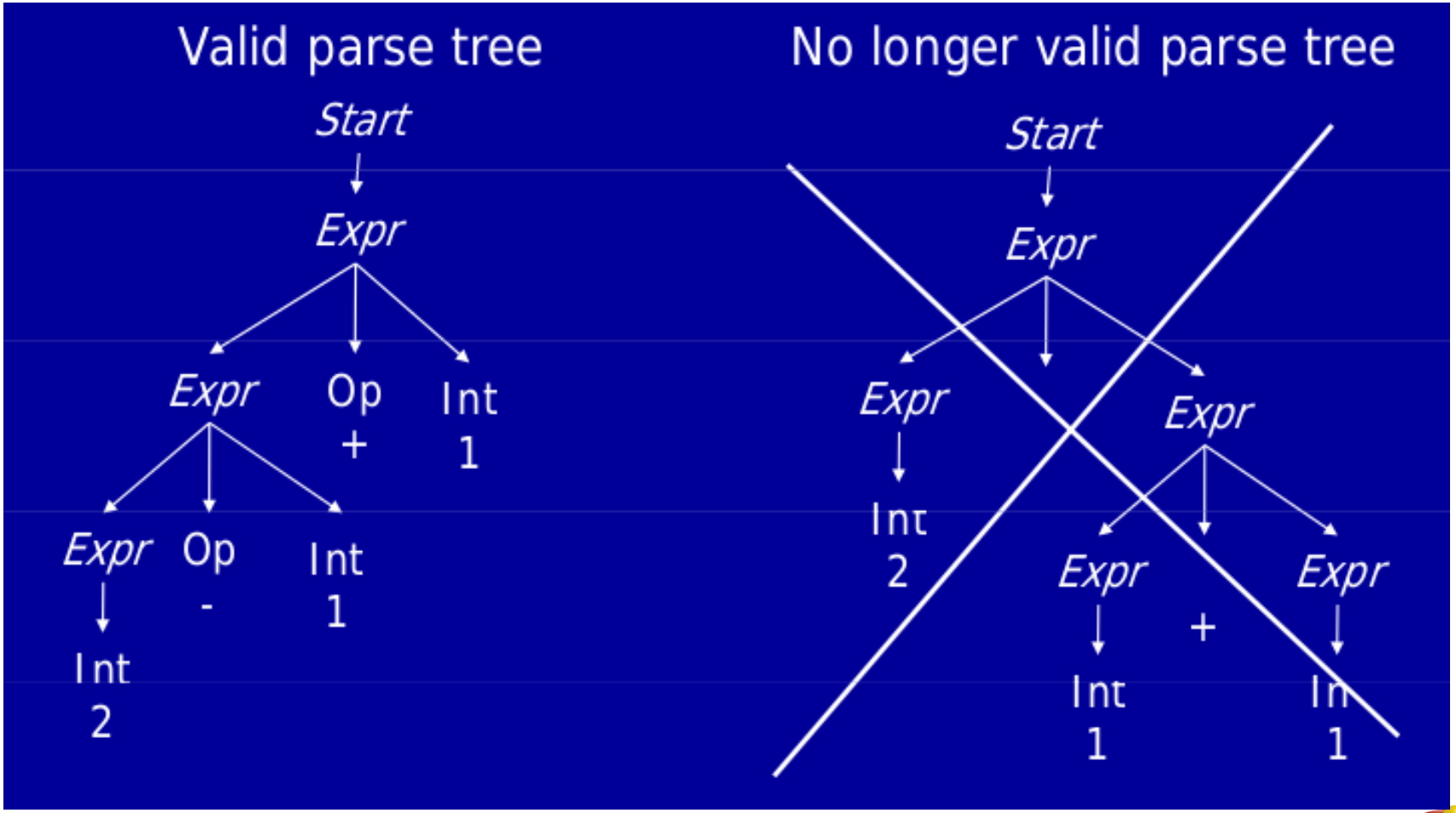

Figura 2.4: Eliminando a ambiguidade [Amarasinghe and Rinard, 2010]

Precedência

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

Figura 2.5: Aplicação da precedência de operadores [Amarasinghe and Rinard, 2010]

Nova árvore

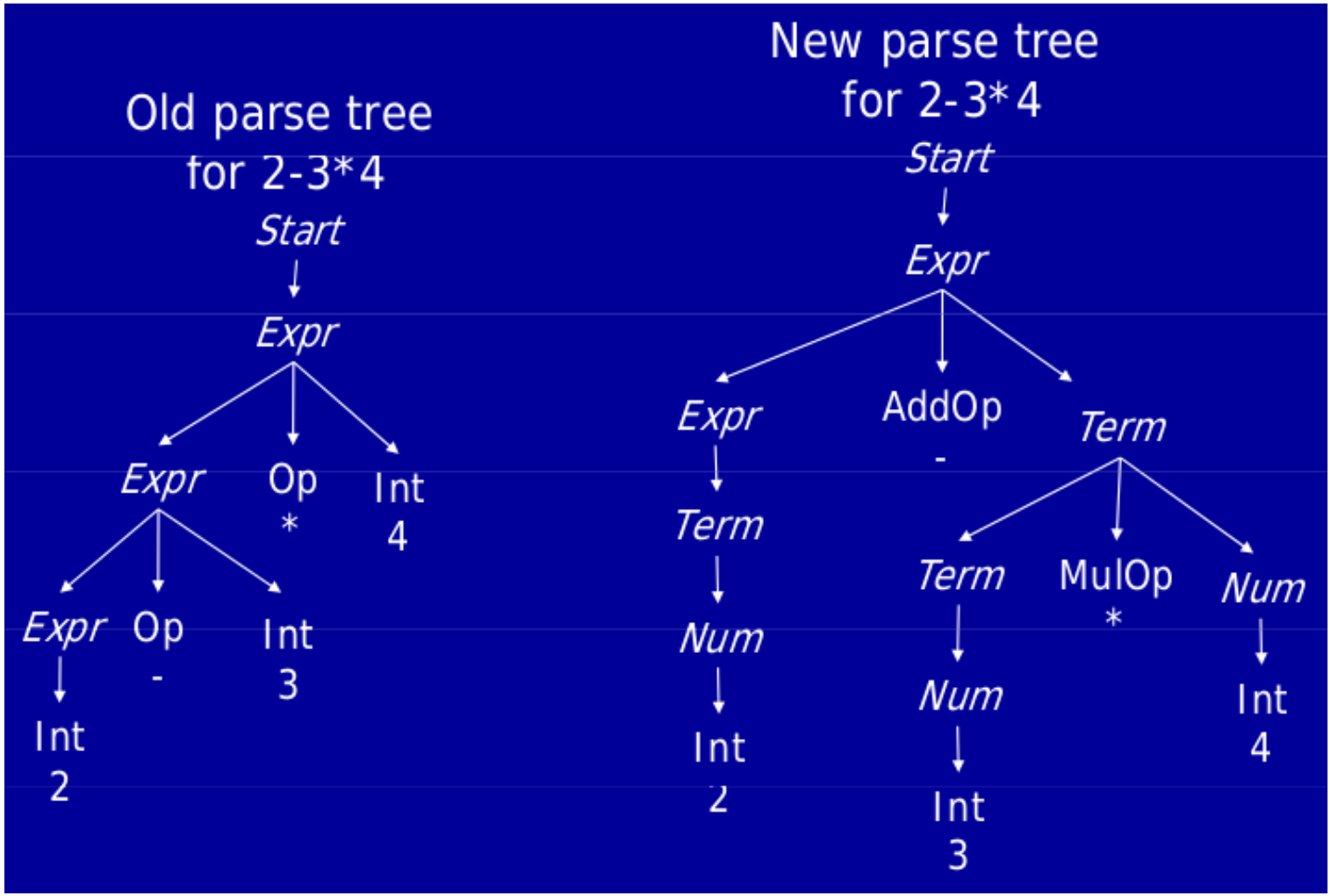

Figura 2.6: Alteração na árvore após aplicação da precedência [Amarasinghe and Rinard, 2010]

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.

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 21.

$$
\begin{aligned}
E &\rightarrow T\,E' \\
E' &\rightarrow +\,T\,E' \mid \epsilon \\
T &\rightarrow F\,T' \\
T' &\rightarrow *\,F\,T' \mid \epsilon \\
f &\rightarrow (\,E\,) \mid \mathbf{id}
\end{aligned}
$$

Figura 3.1: Variação da gramática sem recursividade à esquerda [Aho et al., 2007]

Exemplo

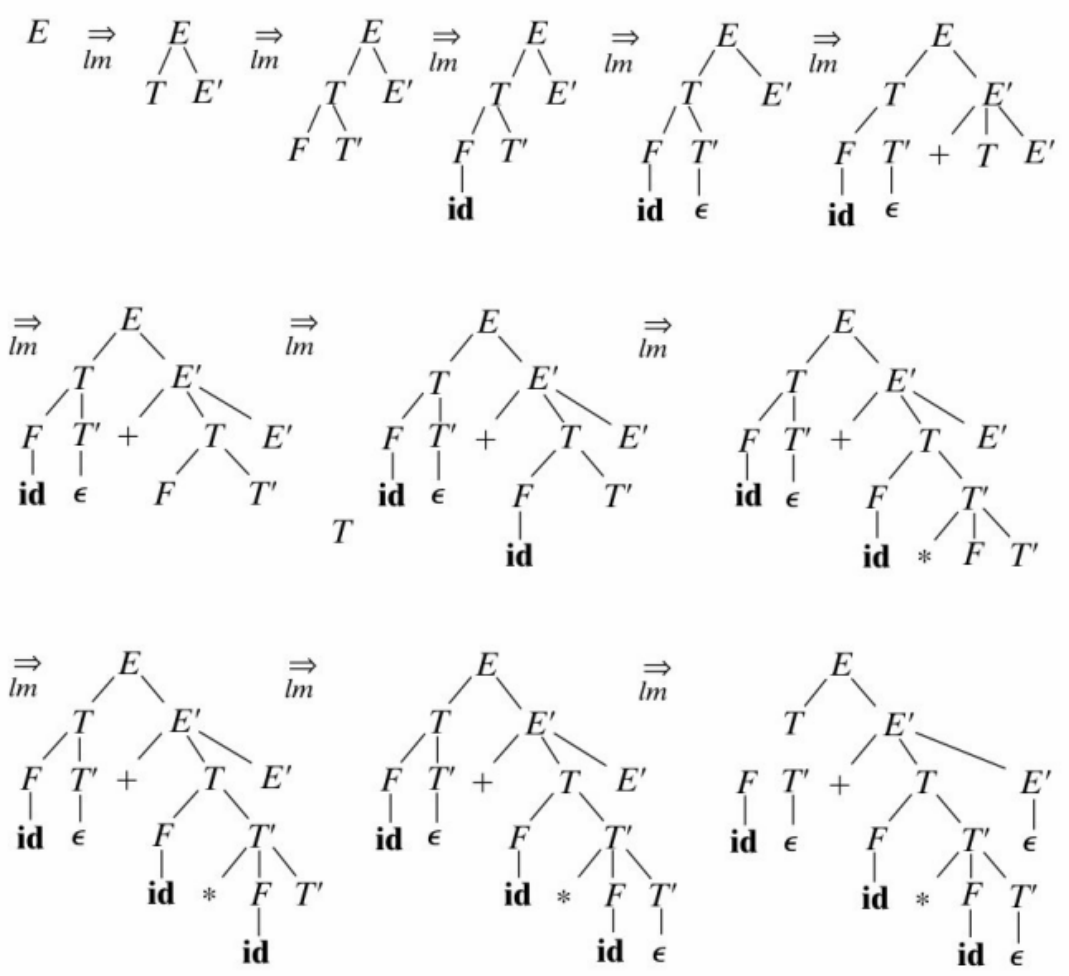

Figura 3.2: Análise sintática descendente para **id** + **id** * **id** [Aho et al., 2007]

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 → X1 X2 ··· Xk;
2)          for ( i = 1 até k ) {
3)                  if ( Xi é um não-terminal )
4)                          ativa procedimento Xi();
5)                  else if ( Xi igual ao símbolo de entrada a )
6)                          avance na entrada para o próximo símbolo terminal;
7)                  else /* ocorreu um erro */;
            }
        }
```

Figura 3.3: 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.

Analisador preditivo

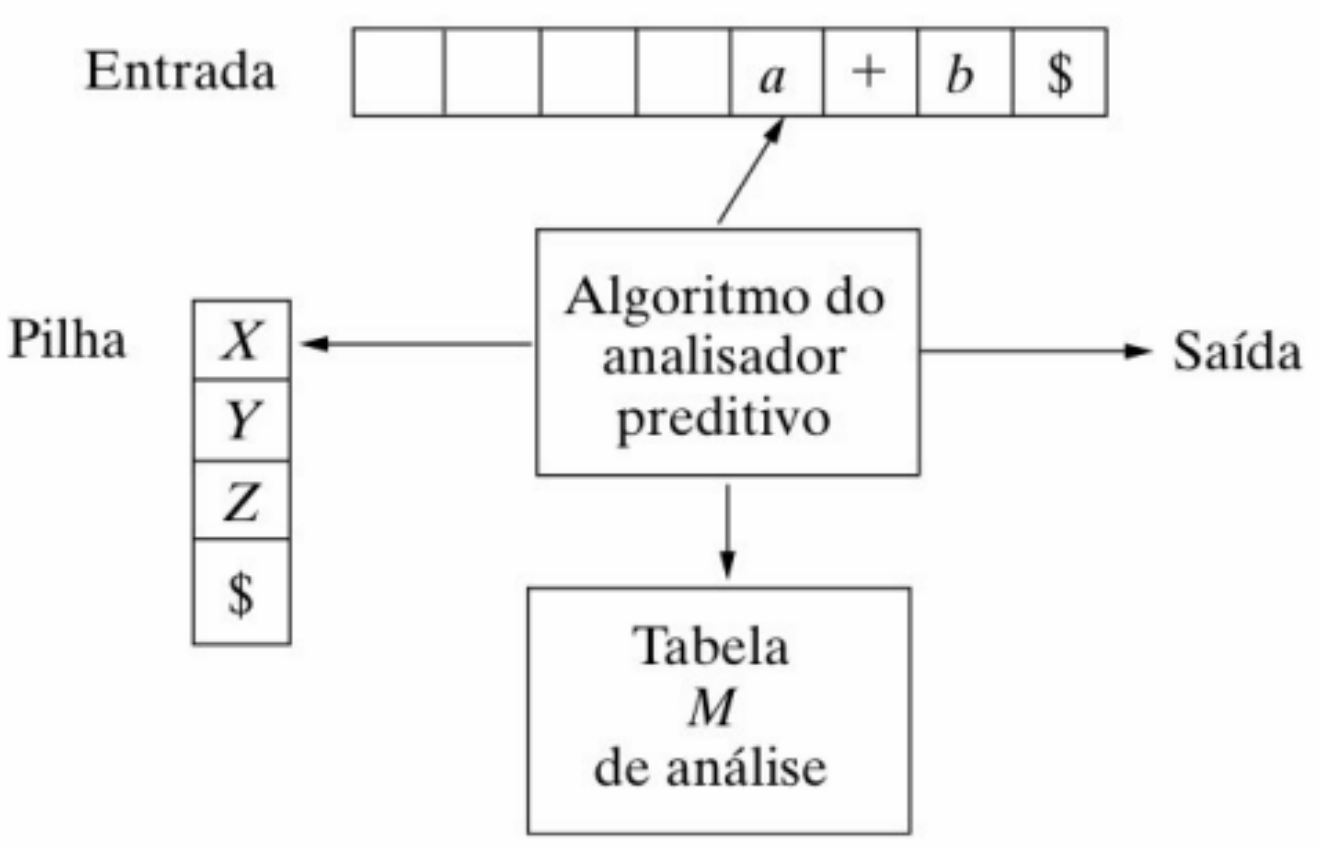

Figura 3.4: Modelo de analisador sintático dirigido por tabela [Aho et al., 2007]

Conceitos

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

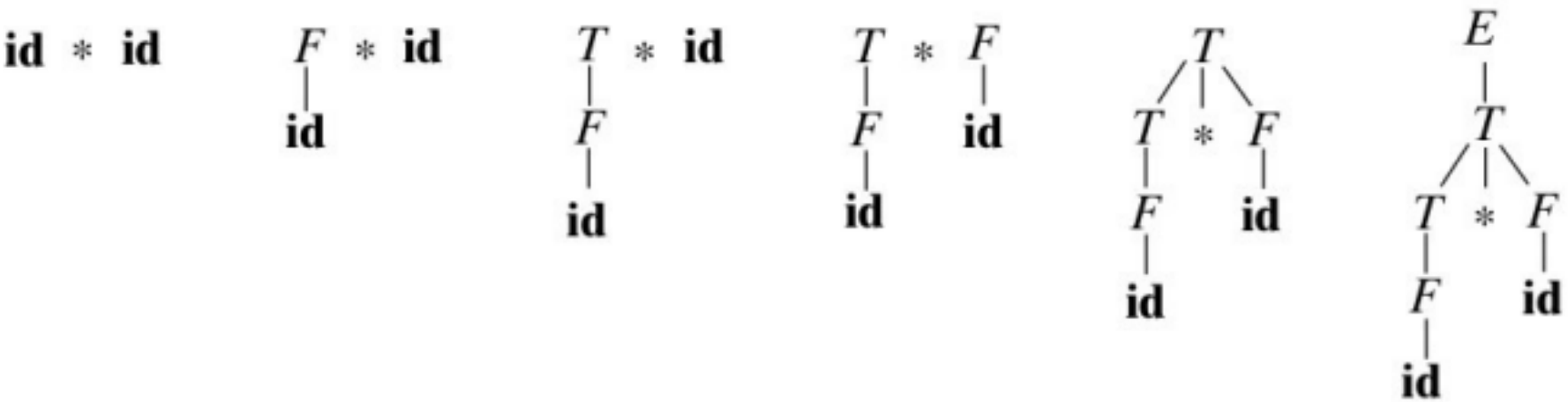

Figura 4.1: Uma análise ascendente para **id * id** [Aho et al., 2007]

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.

$$
\begin{aligned}
Expr &\rightarrow Expr\ Op\ Expr \\
Expr &\rightarrow (\ Expr\ ) \\
Expr &\rightarrow -\ Expr \\
Expr &\rightarrow num \\
Op &\rightarrow + \\
Op &\rightarrow - \\
Op &\rightarrow *
\end{aligned}
$$

Figura 4.2: 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–Princıpios 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.