

---

# Compiladores

## Aula 11: Análise Sintática Ascendente (LR)

Prof. Aléssio Miranda Júnior  
alessio@cefetmg.br  
CEFET-MG - Campus Timóteo

Fevereiro de 2026

### 1 Objetivos

- Compreender os conceitos fundamentais da Análise Sintática Ascendente (*Bottom-Up*).
- Entender a mecânica essencial do algoritmo **Shift-Reduce** e o conceito matemático de *Handle*.
- Estudar o funcionamento do motor genérico de Análise LR (Pilha, Fita e Matriz de Ações).
- Analisar a natureza dos Conflitos Sintáticos (*Shift/Reduce* e *Reduce/Reduce*).
- Explorar a hierarquia da família LR (LR(0), SLR(1), LALR(1) e LR(1)).
- Construir pontes práticas para o uso de ferramentas geradoras de *parsers* (Bison, ANTLR).

### 2 Introdução: A Visão Bottom-Up

Ao contrário da análise Top-Down (vista nos capítulos anteriores, que tenta adivinhar qual produção irá gerar o código a partir da raiz  $S$ ), a Análise Sintática Ascendente (*Bottom-Up*) opera como uma linha de montagem. Ela lê as “folhas” (os *tokens* gerados pelo Analisador Léxico) e tenta **reduzi-las** em blocos lógicos cada vez maiores, até colapsar todo o programa de volta no Símbolo Inicial  $S$ .

Matematicamente, esta técnica é equivalente a descobrir passo-a-passo a **derivação mais à direita em sentido reverso** (*Rightmost Derivation in Reverse*).

---

- Teoria

**Por que LR domina a indústria?** A família LR aceita uma classe de gramáticas consideravelmente mais ampla que a família LL. Em particular, parsers LR suportam **recursão à esquerda** nativamente, sem exigir reescritas manuais da gramática. Por esse motivo, o parsing LR é o motor principal de compiladores de C, C++, Java e Python, e é a base dos geradores automáticos Yacc, GNU Bison e Java CUP.

## 3 O Mecanismo Shift-Reduce

A maioria dos analisadores ascendentes opera sob o modelo **Shift-Reduce**. O ambiente possui duas estruturas fundamentais:

- 1 Uma **Pilha (Stack)**: Memória LIFO que armazena os símbolos processados e o estado atual do autômato.
- 2 Uma **Fita de Entrada (Input)**: Contendo o fluxo de tokens a serem consumidos, terminado pelo marcador \$.

Em cada ciclo, o parser executa uma das quatro ações possíveis:

- **Shift (Empilhar)**: Move o próximo token da fita para o topo da pilha. O ponteiro da fita avança.
- **Reduce (Reduzir)**: Identifica que os elementos no topo da pilha formam o lado direito de uma produção  $A \rightarrow \alpha$ . Remove esses elementos e empilha o não-terminal  $A$ . Esta é a operação central da análise.
- **Accept (Aceitar)**: A fita atingiu \$ e a pilha contém apenas  $S$ . O programa é sintaticamente válido.
- **Error (Erro)**: Nenhuma ação válida é possível. O parser reporta erro de sintaxe.

### 3.1 Exemplo Completo: Trace de Shift-Reduce

Considere a gramática não-ambígua para expressões aritméticas:

1.  $E \rightarrow E + T$
2.  $E \rightarrow T$
3.  $T \rightarrow T * F$
4.  $T \rightarrow F$
5.  $F \rightarrow \text{num}$

Para a entrada `num + num * num $`, o trace completo é:

#	Pilha	Entrada	Ação
1		num + num * num \$	Shift
2	num	+ num * num \$	Reduce 5: $F \rightarrow \text{num}$
3	F	+ num * num \$	Reduce 4: $T \rightarrow F$
4	T	+ num * num \$	Reduce 2: $E \rightarrow T$
5	E	+ num * num \$	Shift
6	E +	num * num \$	Shift
7	E + num	* num \$	Reduce 5: $F \rightarrow \text{num}$
8	E + F	* num \$	Reduce 4: $T \rightarrow F$
9	E + T	* num \$	Shift (NÃO reduz $E \rightarrow E + T$ !)
10	E + T *	num \$	Shift
11	E + T * num	\$	Reduce 5: $F \rightarrow \text{num}$
12	E + T * F	\$	Reduce 3: $T \rightarrow T * F$
13	E + T	\$	Reduce 1: $E \rightarrow E + T$
14	E	\$	<b>Accept</b>

Table 1: Trace de Shift-Reduce para num + num \* num

**△ Importante**

**Observe o passo 9:** A pilha contém E + T e o próximo token é \*. Neste momento, o parser **não** reduz  $E \rightarrow E + T$ , porque o \* tem maior precedência que +. Ele faz Shift do \* para processar a multiplicação primeiro. Esta é a decisão crucial que o autômato LR resolve corretamente.

**3.2 O Conceito de Handle**

O segredo do parsing ascendente está em saber **exatamente quando e como** acionar um Reduce. O **Handle** (*Alça*) é formalmente definido como:

A subsequência no topo da pilha que corresponde ao corpo de uma produção e cuja redução representa o único passo correto na derivação à direita reversa.

Se o parser reduzir o handle errado, a análise diverge e não consegue mais atingir o símbolo inicial. Todos os autômatos LR são projetados especificamente para identificar o handle correto de forma determinística.

**▷ Exemplo**

**Exemplo:** Com a pilha E + T \* F e entrada \$:

- Candidato 1: F — reduzir por  $T \rightarrow F$ ? **Errado!** Ignora o \*.
- Candidato 2: T \* F — reduzir por  $T \rightarrow T * F$ ? **Correto!** Este é o handle.

---

## 4 O Motor Universal de Análise LR

Independentemente do tipo de parser (LR(0), SLR, LALR ou LR(1)), todos rodam sobre a mesma máquina virtual genérica. O que os diferencia é apenas a **qualidade das tabelas** construídas. O motor é composto de:

- Uma **Pilha** contendo pares de (Estado, Símbolo).
- A tabela **ACTION** $[s, a]$ : Dado o estado  $s$  no topo da pilha e o token  $a$  na fita, retorna Shift  $s'$ , Reduce  $A \rightarrow \alpha$ , Accept ou Error.
- A tabela **GOTO** $[s, A]$ : Dado o estado  $s$  e o não-terminal  $A$  (recém-empilhado após um Reduce), retorna o próximo estado.

### 4.1 Algoritmo LR Genérico

O algoritmo opera em loop:

- 1 Inicializa a pilha com o estado 0.
- 2 Observa o estado  $s$  no topo da pilha e o token  $a$  na fita.
- 3 Consulta **ACTION** $[s, a]$ :
  - **Shift**  $s'$ : Empilha o token  $a$  e o estado  $s'$ . Avança a fita.
  - **Reduce**  $A \rightarrow \alpha$ : Remove  $|\alpha|$  pares do topo. O estado que “sobreviveu” é  $s_{antigo}$ . Empilha  $A$  com o estado **GOTO** $[s_{antigo}, A]$ .
  - **Accept**: Análise concluída com sucesso.
  - **Error**: Reporta erro de sintaxe.
- 4 Repete até Accept ou Error.

## 5 Conflitos Sintáticos

Quando a gramática é ambígua ou não pertence à classe LR correspondente, a construção das tabelas resulta em **conflitos**: uma mesma célula da tabela **ACTION** contém mais de uma ação possível.

### 5.1 Conflito Shift/Reduce

Ocorre quando, em um determinado estado com um certo lookahead, o parser não sabe se deve empilhar (Shift) ou reduzir (Reduce).

O caso clássico é o **Dangling-Else**:

```
if (A) if (B) C else D
```

Quando o parser lê o **else**, ele pode:

- **Shift** o `else`: associa-o ao `if(B)` mais próximo.
- **Reduce** o `if(B) C`: associa o `else` ao `if(A)` externo.

A maioria dos compiladores e ferramentas (Bison/YACC) resolve priorizando o Shift, o que associa o `else` ao `if` mais próximo — comportamento desejado.

## 5.2 Conflito Reduce/Reduce

É o tipo mais grave. Ocorre quando dois reduces diferentes são possíveis no mesmo estado. Indica geralmente uma **ambiguidade fundamental** na gramática que precisa ser corrigida via refatoração ou diretivas de precedência.

# 6 A Família LR

A hierarquia dos parsers ascendentes forma uma cadeia de inclusão:

$$\text{LR}(0) \subset \text{SLR}(1) \subset \text{LALR}(1) \subset \text{LR}(1)$$

## 6.1 LR(0)

É a base teórica (Knuth, 1965). O parser decide **sem olhar** nenhum token à frente (lookahead = 0). A tabela mapeia estados usando “itens” da forma  $A \rightarrow \alpha.\beta$ , onde o ponto marca o progresso de leitura.

Quando atinge um item de redução ( $A \rightarrow \alpha.$ ), o LR(0) marca a ação Reduce para **todos** os terminais, gerando conflitos massivos. Quase nenhuma linguagem prática é puramente LR(0).

## 6.2 SLR(1) — Simple LR

O SLR(1) reaproveita os mesmos estados do LR(0), mas filtra as ações de Reduce usando o conjunto **FOLLOW**. Quando encontra um item de redução  $A \rightarrow \alpha.$ , só autoriza o Reduce nas colunas cujo terminal pertence a  $\text{FOLLOW}(A)$ .

Essa simples melhoria elimina cerca de 90% dos conflitos do LR(0). Porém, como FOLLOW é um cálculo **global** (independente do contexto específico), o SLR pode ser permissivo demais para gramáticas complexas.

## 6.3 LR(1) — Canonical LR

O LR(1) resolve a imprecisão do SLR acoplando o **lookahead exato** dentro de cada item:  $[A \rightarrow \alpha.\beta, \{a\}]$ . As reduções tornam-se perfeitamente precisas.

O custo é uma **explosão de estados**: uma gramática que gera 300 estados em SLR pode gerar 3.000+ estados em LR(1), tornando as tabelas imensas.

## 6.4 LALR(1) — Look-Ahead LR

O LALR(1) é o compromisso engenhoso que dominou a indústria. Ele **funde** estados LR(1) que possuem o mesmo núcleo (mesmos itens LR(0)) mas diferem apenas nos conjuntos de lookahead.

O resultado é um autômato com o **mesmo número de estados que o SLR(1)**, porém com a **precisão de decisão** herdada do LR(1). Essa combinação de compactação e poder fez do LALR(1) o algoritmo padrão dos compiladores comerciais e das ferramentas geradoras como Bison e YACC.

Tipo	Lookahead	Nº Estados	Observação
LR(0)	0	Poucos	Base acadêmica, impraticável.
SLR(1)	FOLLOW (global)	= LR(0)	Bom para gramáticas simples.
LR(1)	Contexto exato	×10	Máxima precisão, custo de memória.
LALR(1)	Fusão de LR(1)	= SLR(1)	<b>Padrão da indústria.</b>

Table 2: Comparação da família LR

## 7 Engenharia Prática: Ferramentas Geradoras

As tabelas LALR(1) para uma gramática completa (como C ou Java) geram milhares de entradas. Construí-las manualmente é inviável. Por isso, compiladores ascendentes modernos são automatizados com ferramentas geradoras:

- **GNU Bison / YACC (C/C++):** O programador especifica a gramática em formato BNF declarativo. A ferramenta calcula as tabelas LALR(1), reporta conflitos interativamente, e gera o código-fonte do parser.
- **Java CUP (Java):** Equivalente ao Bison para o ecossistema Java.
- **ANTLR 4 (Java):** Embora use o algoritmo LL(\*) (descendente), o ANTLR suporta recursão à esquerda via reescrita interna, combinando a expressividade LR com a abordagem LL. É a ferramenta utilizada no nosso projeto do Mini-Pascal.

### ► Prática

**Lição prática:** Ninguém constrói tabelas LALR à mão em ambiente de produção. A teoria é fundamental para **entender os conflitos** quando a ferramenta os reporta, e para **refatorar a gramática** de forma informada. Sempre que o Bison indicar “1 shift/reduce conflict”, você saberá exatamente o que está acontecendo.

---

## 8 Exercício Guiado

Considere a gramática:

1.  $S \rightarrow S + S$
2.  $S \rightarrow S * S$
3.  $S \rightarrow \text{num}$

### Tarefas:

- 1 Esta gramática é ambígua? Justifique.
- 2 Faça o trace de Shift-Reduce para a entrada `num + num * num $`. Em que momento ocorre o conflito?
- 3 Se a ferramenta priorizar Shift no conflito, qual a árvore de parse resultante? A precedência de `*` sobre `+` é respeitada?
- 4 Reescreva a gramática usando a versão não-ambígua com  $E$ ,  $T$  e  $F$  (como no exemplo deste capítulo) e refaça o trace.

## 9 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. 2ª Edição. Cap. 4 (Seções 4.5–4.7).
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. 2ª Edição. Cap. 3 (Seção 3.4).
- Grune, D. et al. (2012). *Modern Compiler Design*. 2ª Edição. Cap. 9.
- Levine, J. R. (2009). *flex & bison: Text Processing Tools*. O'Reilly Media.