COMPILADORES
ANÁLISE SINTÁTICA

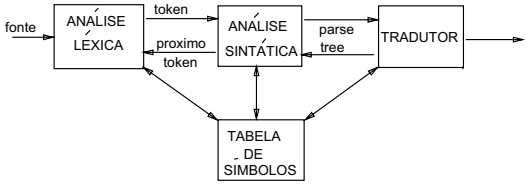
Roberto S. Bigonha e Mariza A. S. Bigonha

UFMG

11 de agosto de 2011

ANÁLISE SINTÁTICA

- O papel do reconhecedor:



- Estratégias:
  - Ascendente $\Leftrightarrow$ bottom-up
  - Descendente $\Leftrightarrow$ top-down

TRATAMENTO DE ERRO

Tipos de Erro

- Léxico
- Sintático
- Semântico
- Lógico

Objetivos do Tratamento de Erro

- Informar a presença de erros de forma clara e precisa.
- Recuperar-se de cada erro com rapidez suficiente para detectar erros subseqüentes.
- Acrescentar um custo mínimo no processamento de programas corretos.

Estratégias de Recuperação de Erro

- Pânico.
- Nível de frases.
- Produções de erro.
- Correção global.

CONVENÇÕES E DEFINIÇÕES

4.2 Gramática Livre do Contexto (CFG)

Uma gramática livre de contexto consiste em terminais, não-terminais, um símbolo inicial e produções.

1. Terminais são os símbolos básicos a partir dos quais as cadeias são formadas. O termo nome de token é um sinônimo para terminal e freqüentemente será usada a palavra token em vez terminal quando estiver claro que nos referimos apenas sobre o nome do token.
2. Não-terminais são variáveis sintáticas que representam conjuntos de cadeias. Os não-terminais impõem uma estrutura hierárquica sobre a linguagem que é a chave para a análise sintática e tradução.
3. Em uma gramática, um não-terminal é distinguido como o símbolo inicial, e o conjunto de cadeias que ele representa é a linguagem gerada pela gramática. Por convenção, as produções para o símbolo inicial são listadas primeiro.

··· Gramática Livre do Contexto (CFG)

4. As produções de uma gramática especificam a forma como os terminais e os não-terminais podem ser combinados para formar cadeias. Cada produção consiste em:
   - 4.1. Um não-terminal chamado de cabeça ou lado esquerdo da produção; essa produção define algumas das cadeias representadas pela cabeça.
   - 4.2. O símbolo "$\rightarrow$". Às vezes, "::=" é usado no lugar da seta.
   - 4.3. Um corpo ou lado direito da produção consiste em zero ou mais terminais e não-terminais. Os componentes do corpo descrevem uma forma como as cadeias do não-terminal do lado esquerdo da produção podem ser construídas.

··· Gramática Livre do Contexto (CFG)

Exemplo:

$$
\begin{aligned}
\text{expression} &\rightarrow \text{expression} + \text{term}\\
\text{expression} &\rightarrow \text{expression} - \text{term}\\
\text{expression} &\rightarrow \text{term}\\
\text{term} &\rightarrow \text{term} * \text{factor}\\
\text{term} &\rightarrow \text{term} / \text{factor}\\
\text{term} &\rightarrow \text{factor}\\
\text{factor} &\rightarrow \texttt{(}\ \text{expression}\ \texttt{)}\\
\text{factor} &\rightarrow \mathrm{id}
\end{aligned}
$$

Símbolos terminais: id "+" "-" "*" "/" "(" ")"

Símbolos não-terminais: `expression`, `term` e `factor`, e `expression` é o símbolo inicial da gramática.

Convenções de Notação

Convenções de notação para as gramáticas para evitar ter de dizer "estes são os terminais", "estes são os não-terminais":

1. Símbolos terminais:
   - 1.1. Letras minúsculas do início do alfabeto, como `a, b, c`.
   - 1.2. Símbolos operadores como "+", "*", etc.
   - 1.3. Símbolos de pontuação como parênteses, vírgula etc.
   - 1.4. Dígitos `0,1,...,9`.
   - 1.5. Cadeias em negrito como id ou if, cada um representando um único símbolo terminal.

··· Convenções de Notação

2. Não-Terminais:
   - 2.1. Letras maiúsculas do início do alfabeto, como `A, B, C`.
   - 2.2. A letra S, que, quando aparece, normalmente é o símbolo inicial da gramática.
   - 2.3. Nomes em minúsculas e itálico, como *expr* ou *stmt*.
   - 2.4. Nas construções das linguagens de programação, letras maiúsculas podem ser usadas para representar não-terminais. Por exemplo, os não-terminais para expressões, termos e fatores normalmente são representados por `E, T` e `F`, respectivamente.

··· Convenções de Notação

3. **Cadeias de terminais:** letras minúsculas do fim do alfabeto, `u`, `v`, ..., `z`.

4. **Cadeias de símbolos da gramática:** letras gregas minúsculas, por exemplo β, α e γ. Assim: $A \to \alpha$

   A = não-terminal
   = cabeça ou lado esquerdo da produção

   α **= string de símbolos da gramática.**
   = corpo ou lado direito da produção.

··· Convenções de Notação

5. **Símbolos da gramática, ou seja, não-terminais ou terminais: letras maiúsculas do fim do alfabeto, como** `X`, `Y`, `Z`.

6. $A \to \alpha_1$, $A \to \alpha_2$, $\cdots$,
   $A \to \alpha_k \equiv$ **produções de A.**
   $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_k$
   $\alpha_1 | \alpha_2 | \cdots | \alpha_k \equiv$ **alternativas de A.**

Derivações

Processo inverso do reconhecimento.

Exemplo: $E \to E + E \mid E * E \mid (E) \mid -E \mid \mathrm{id}$
$E \Rightarrow E + E \Rightarrow E + (E)$
$E \stackrel{*}{\Longrightarrow} \mathrm{id} + \mathrm{id}$

$\alpha A \beta \Rightarrow \alpha\gamma\beta$ **se** $A \to \gamma$ **é uma produção**
α **e** β **são quaisquer strings de símbolos da gramática.**

$\Longrightarrow$ **deriva em um passo**
$\stackrel{*}{\Longrightarrow}$ **deriva em zero ou mais passos.**
$\stackrel{+}{\Longrightarrow}$ **deriva em um ou mais passos.**
(1) $\alpha \stackrel{*}{\Longrightarrow} \alpha$ **para qualquer string** α.
(2) $\alpha \stackrel{*}{\Longrightarrow} \beta$ **e** $\beta \Longrightarrow \delta$ **então** $\alpha \stackrel{*}{\Longrightarrow} \delta$.

··· Derivações

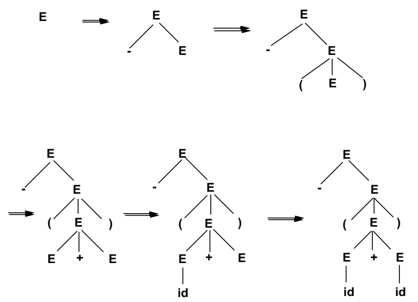
- **Dada uma gramática G com o símbolo de partida S, usamos** $\stackrel{+}{\Longrightarrow}$ **para definir L(G), a linguagem gerada por G.**
- **w está em L(G)** se e somente se **S** $\stackrel{+}{\Longrightarrow}$ **w**
  w $\equiv$ sentença de G.
- **Uma linguagem que pode ser gerada por uma gramática linguagem livre do contexto é dita ser linguagem livre do contexto.**
- **Se duas gramáticas geram a mesma linguagem, as gramáticas são equivalentes.**
- **S** $\stackrel{*}{\Longrightarrow} \alpha$**, onde** α **pode conter não-terminais, então dizemos que** α é uma forma sentencial de G.
- **Uma sentença é uma forma sentencial SEM não-terminais.**

Árvores de Derivação e Derivações

$-(\mathrm{id} + \mathrm{id})$ - **sentença da gramática mostrada porque existe uma derivação:**

$\mathbf{E} \Rightarrow -\mathbf{E} \Rightarrow -(\mathbf{E}) \Rightarrow -(\mathbf{E}+\mathbf{E}) \Rightarrow (\mathrm{id}+\mathbf{E}) \Rightarrow -(\mathrm{id} + \mathrm{id})$

Seqüência da construção da árvore de derivação:



Derivação Mais à Esquerda ou à Direita

A cada passo da derivação: **(1) escolher o não-terminal para substituir**
(2) escolher a alternativa para o não-terminal

Derivação mais à esquerda: **substitui-se o não terminal mais a esquerda em qualquer forma sentencial a cada passo.**

Exemplo:
wA$\gamma \stackrel{lm}{\Longrightarrow}$ **w**$\delta\gamma$ **onde,**
w somente terminais
A $\rightarrow \delta \equiv$ **produção aplicada**
$\gamma \equiv$ **string de símbolos da gramática.**

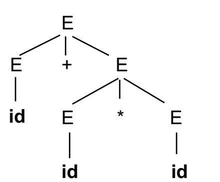
Para enfatizar o fato de que α **deriva** β **usando derivação mais à esquerda, escreve-se** $\alpha \stackrel{*}{\Longrightarrow}_{lm} \beta$.

Se S $\stackrel{*}{\Longrightarrow}_{lm} \alpha$, α **é a forma sentencial mais à esquerda da gramática em análise.**

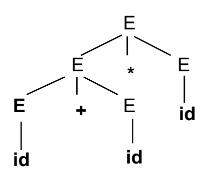
Ambigüidade

Exemplo: $\mathrm{id} + \mathrm{id} * \mathrm{id}$

$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E}$
$\Rightarrow \mathrm{id} + \mathbf{E}$
$\Rightarrow \mathrm{id} + \mathbf{E} * \mathbf{E}$
$\Rightarrow \mathrm{id} + \mathrm{id} * \mathbf{E}$
$\Rightarrow \mathrm{id} + \mathrm{id} * \mathrm{id}$



$\mathbf{E} \Rightarrow \mathbf{E} * \mathbf{E}$
$\Rightarrow \mathbf{E} + \mathbf{E} * \mathbf{E}$
$\Rightarrow \mathrm{id} + \mathbf{E} * \mathbf{E}$
$\Rightarrow \mathrm{id} + \mathrm{id} * \mathbf{E}$
$\Rightarrow \mathrm{id} + \mathrm{id} * \mathrm{id}$



Gramáticas Livres do Contexto Versus Expressões Regulares

- **Toda construção que pode ser descrita por uma expressão regular também pode ser descrita por uma gramática CFG.**

- **Expressão Regular: (a | b)* abb**

- **A Gramática Livre do Contexto:**

$A_0 \rightarrow \mathbf{a}A_0 \mid \mathbf{b}A_0 \mid \mathbf{a}A_1$
$A_1 \rightarrow \mathbf{b}A_2$
$A_2 \rightarrow \mathbf{b}A_3$
$A_3 \rightarrow \mathcal{E}$

descreve a mesma linguagem, o conjunto de *cadeias* **de** a**s e** b**s seguidas por** abb.

Gramáticas Livres do Contexto Versus Expressões Regulares

É possível construir mecanicamente uma gramática para reconhecer a mesma linguagem que um NFA usando a abordagem:

1. Para cada estado i do NFA, crie um não-terminal A_i.
2. Se o estado i possuir uma transição para o estado j lendo a, inclua a produção $A_i \rightarrow aA_i$.

   Se o estado i vai para o estado j lendo a entrada $\mathcal{E}$, inclua a produção $A_i \rightarrow A_j$.
3. Se i é um estado de aceitação, inclua $A_i \rightarrow \mathcal{E}$.
4. Se i é o estado inicial, faça A_i o símbolo inicial da gramática.

4.3 Escrevendo uma Gramática

Tudo o que pode ser descrito por uma expressão regular também pode ser descrito por uma gramática livre de contexto.

Por que usar expressões regulares para definir a sintaxe léxica de uma linguagem?

Análise Léxica versus Análise Sintática

Por que usar expressões regulares para definir a sintaxe léxica de uma linguagem?

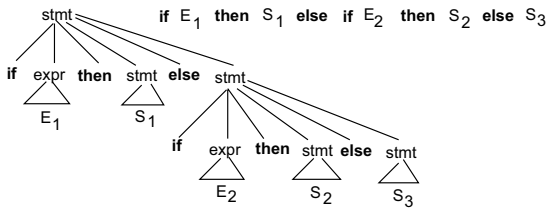
1. Separar a estrutura sintática de uma linguagem em partes léxica e não léxica provê uma forma conveniente de particionar o front-end de um compilador em dois módulos de tamanho administrável.
2. As regras léxicas de uma linguagem são freqüentemente muito simples e, para descrevê-las, não precisamos de uma notação tão poderosa quanto as gramáticas livre de contexto.
3. As expressões regulares geralmente oferecem uma notação mais fácil de entender e mais concisa para tokens do que as gramáticas.
4. Analisadores léxicos mais eficientes podem ser construídos automaticamente a partir de expressões regulares do que por meio de gramáticas arbitrárias.

Eliminando a Ambigüidade

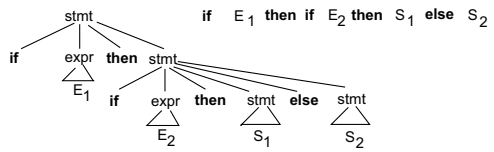
Eliminando ambigüidade

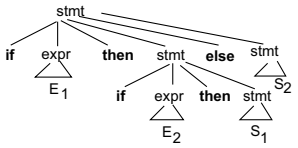
stmt → if **expr** then **stmt**
| if **expr** then **stmt** else **stmt**
| other

Árvore para comando condicional composto:



Duas árvores para uma sentença ambígua





stmt → **matched-stmt**
| **unmatched-stmt**

matched-stmt → if **expr** then **matched-stmt** else **matched-stmt** | other

unmatched-stmt → if **expr** then **stmt**
| if **expr** then **matched-stmt** else **unmatched-stmt**

Eliminação da Recursão à Esquerda

$$\mathbf{A} \rightarrow \mathbf{A}\ \alpha \mid \beta,$$

onde α e β são seqüências de terminais e não-terminais que não começam com A.

· · · Eliminação da Recursividade à Esquerda

- Solução Imediata: **re-estruturar a gramática.**
  A → **A** α | β torna-se: **A** → β **R**
  R → α **R** | $\mathcal{E}$
- Assim a gramática:

  expr → **expr** + **term** | **term**
  onde: A = expr
  α = + **term**
  β = **term**

  torna-se:
  expr → **term R**
  R → + **term R** | $\mathcal{E}$

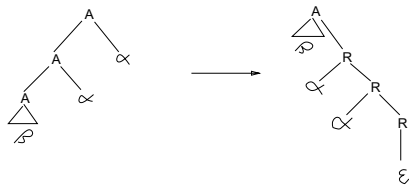
· · · Eliminação de Recursividade à Esquerda

Imediata

$$\mathbf{A} \rightarrow \mathbf{A}\ \alpha \mid \beta,$$

onde α e β são seqüências de terminais e não-terminais que não começam com A.

Transforma-se em: **A** → β **R**
R → α **R** | $\mathcal{E}$



· · · Eliminação de Recursividade à Esquerda Imediata

1. **expr** $\rightarrow$ **expr** + **term** | **term**
2. **term** $\rightarrow$ **term** $\star$ **factor** | **factor**
3. **factor** $\rightarrow$ **(expr)** | id

1. **expr** $\rightarrow$ **expr** + **term** | **term**

$\Downarrow$

expr $\rightarrow$ **term** $expr^{'}$
$expr^{'} \rightarrow$ + **term** $expr^{'} \mid \mathcal{E}$

2. **term** $\rightarrow$ **term** $\star$ **factor** | **factor**

$\Downarrow$

term $\rightarrow$ **factor** $term^{'}$
$term^{'} \rightarrow \star$ **factor** $term^{'} \mid \mathcal{E}$

Então:

expr $\rightarrow$ **term** $expr^{'}$
$expr^{'} \rightarrow$ + **term** $expr^{'} \mid \mathcal{E}$
term $\rightarrow$ **factor** $term^{'}$
$term^{'} \rightarrow \star$ **factor** $term^{'} \mid \mathcal{E}$
factor $\rightarrow$ **(expr)** | id

· · · Eliminação de Recursividade à Esquerda

- **Generalização**

A $\rightarrow$ **A**α_1 | **A**α_2 | **A**α_m |
$\beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ **onde** β_i **não começa com A (1** $\leq$ **i** $\leq$ **n)**

$\Downarrow$

A $\rightarrow \beta_1 A^{'} \mid \beta_2 A^{'} \mid \cdots \mid \beta_n A^{'}$
$A^{'} \rightarrow \alpha_1 A^{'} \mid \alpha_2 A^{'} \mid \cdots \mid \alpha_m A^{'} \mid \mathcal{E}$

- **Em mais de um passo:**

A $\stackrel{n}{\Longrightarrow}$ **A**α, **n** $\geq$ **2**

· · · Eliminação de Recursividade à Esquerda

Algoritmo

(funciona se não $\exists$ **A** $\stackrel{+}{\Longrightarrow}$ **A e não** $\exists$ **produções da forma: A** $\rightarrow \mathcal{E}$**).**

1. **coloque os não-terminais em ordem** A_1, A_2, $\cdots$, A_n
2. for **i** := **1** to **n** do
   begin for **j** := **1** to **i**−**1** do
   2.1. **"substitui cada produção:** $A_i \rightarrow A_j\ \gamma$ **por** $A_i \rightarrow \delta_1\ \gamma \mid \delta_2\ \gamma \mid \cdots$ $\delta_k\ \gamma$ **onde A**$_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \delta_k$ **| são as produções de** A_j**.**
   2.2. **elimina as recursões esquerdas imediatas nas produções:** A_i**.**

end

Eliminação de Recursividade à Esquerda em Ciclos

1. **S** $\rightarrow$ **Aa** | **b**
   A $\rightarrow$ **Ac** | **Sd** | $\mathcal{E}$

2. **A** $\rightarrow$ **Ac** | $\underline{Aa}$**d** | $\underline{b}$**d** | $\mathcal{E}$

3. **A** $\rightarrow$ **bdR** | **R**
   R $\rightarrow$ **cR** | **adR** | $\mathcal{E}$

Resultado:
S $\rightarrow$ **Aa** | **b**
A $\rightarrow$ **bdR** | **R**
R $\rightarrow$ **cR** | **adR** | $\mathcal{E}$

$\underline{\mathbf{S}} \Longrightarrow$ **Aa** $\Longrightarrow$ $\underline{\mathbf{S}}$**da**

Fatoração à Esquerda

Produções da Forma: $\mathbf{A} \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \alpha\ \beta_n \mid \gamma$
Não serve para *Reconhecedor por Descida Recursiva*!!!

Solução:
Adiar a decisão de qual alternativa a ser usada via fatoração.

$\mathbf{A} \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \alpha\ \beta_n \mid \gamma$

$\Downarrow$

$\mathbf{A} \rightarrow \alpha\ A' \mid \gamma$
$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

Exemplo:

S → **if E then S** |
if E then S else S |
simple-stmt

$\Downarrow$

S → **if E then S** S' | **simple-stmt**
S' → **else S** | $\mathcal{E}$

4.4 ANÁLISE SINTÁTICA DESCENDENTE

Análise Sintática Descendente ("Top-down")

Derivação mais a esquerda para uma *cadeia* **de entrada.**

- **Análise Sintática de Descida Recursiva**

  retrocesso (*backtracking***)**

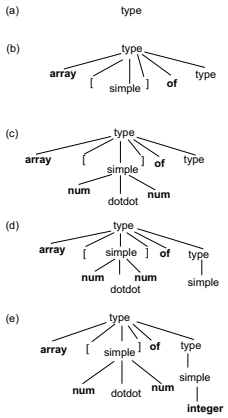
- **Analisador Sintático Preditivo (Predictive Parser)**

  não tem retrocesso (*backtracking***)**

··· Análise Sintática Descendente ("Top-down")

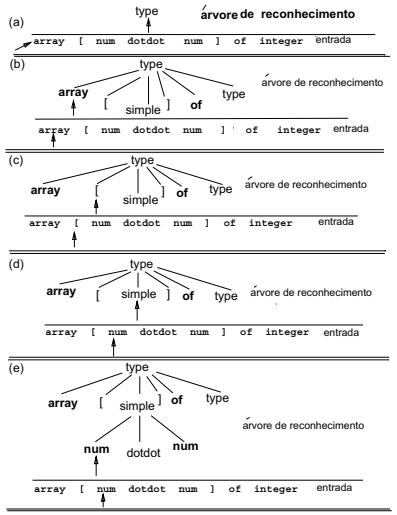
type → **simple** |
↑ id |
array [simple] of type
simple → integer |
char |
num **dotdot** num

· · · Análise Sintática Descendente - Construção da Árvore



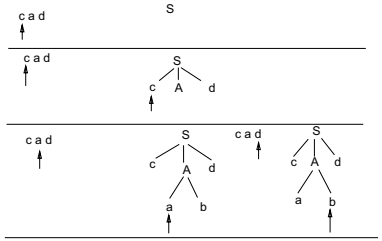
string: array [num dotdot num] of integer

· · · Análise Sintática Descendente - Reconhecimento

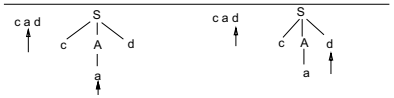


Análise Sintática de Descida Recursiva

- Considere a entrada $\underline{w = cad}$ e a gramática: **S** $\rightarrow$ **cAd**
  A $\rightarrow$ **ab** | **a**



- **Volta para A.**
- **Restaura apontador de entrada para posição 2.**



· · · Análise Sintática de Descida Recursiva

Analisador por descida recursiva

é um método "top-down" de análise sintática no qual é executado um conjunto de procedimentos recursivos para processar o string de entrada.

Um procedimento recursivo é associado a cada não-terminal da gramática.

Análise Gramatical Predictiva

- **caso especial**
- **não há "backtracking"**
- **lookahead**

Dado símbolo de entrada "a" e o não-terminal "A" para ser expandido qual das alternativas de A $\rightarrow \alpha_1, \alpha_2, \cdots$ é a única alternativa correta que deriva uma *cadeia* **começando com "a".**

· · · Análise Sintática de Descida Recursiva

Dada a Gramática:

type → **simple** |
↑ id |
array **[** **simple** **]** of **type**
simple → integer |
char |
num **dotdot** num

Os procedimentos chamados são:

`type`, `simple`

e um procedimento adicional `match`

- Código da função `type`:

match (array);
match ("[");
⟹ **simple;**
match ("]");
match (of);
⟹ **type**

- Código da Função `simple`:

match (num); **match** (dotdot); **match** (num);

Procedimento `type`

```
procedure type
    begin
        if lookahead in {integer, char, num}
        then simple
        else if lookahead = ↑
            then begin match (↑) match (id) end
            else if lookahead = array
                then begin match (array);
                           match ([) ;
                           simple;
                           match (]) ;
                           match (of) ;
                           type ;
                     end else "erro"  end
```


Procedimento `simple`

```
procedure simple begin
      if lookahead = integer then match (integer)
      else if lookahead = char
            then match (char)
            else if lookahead = num
                  then begin match (num);
                             match (..); match (num) ;
                       end
                       else "erro"
end
```


Procedimento `match`

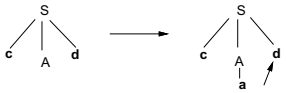
```
procedure match(t: token)
begin
      if lookahead = t then lookahead := nextoken
      else error
end
```

Dificuldades com o Reconhecedor por Descida Recursiva

- Recursividade à esquerda: $\mathbf{A} \stackrel{+}{\Longrightarrow} \mathbf{A}\ \alpha$
  loop infinito expr → expr + term nenhuma leitura é feita em chamadas recursivas.

- Backtracking: caro desfazer o que foi feito. **(a) tabela de símbolos**
  (b) código gerado

- **Ordem de escolha de alternativas pode afetar a linguagem aceita.** **S → cAd**
  A → ab | a



cabd
↑
erro na expansão: S → cAd!??? . Não se sabe exatamente onde o erro ocorreu.

RECONHECEDOR PREDITIVO

Funções FIRST e FOLLOW

Definição: **FIRST(α), onde** $\alpha \in V^*$

Conjunto de terminais que iniciam um "string" derivado de α. Se $\alpha \stackrel{*}{\Longrightarrow} \mathcal{E}$, inclui-se $\mathcal{E}$ em FIRST(α).

Definição: **FOLLOW(A), onde A** $\in (V_N \mid V_T)$

Conjunto de terminais a que podem aparecer imediatamente à direita de A em alguma forma sentencial.

Se A pode ser o símbolo mais à direita de alguma forma sentencial, inclue $\mathcal{E}$ em FOLLOW(A) (ou $ na pratica).

FOLLOW(A) = $\{\ \mathbf{a} \in V_T \mid \mathbf{S} \stackrel{*}{\Longrightarrow} \alpha\mathbf{A}\mathbf{a}\beta\} \cup \{\mathcal{E} \mid \mathbf{S} \stackrel{*}{\Longrightarrow} \alpha\mathbf{A}\gamma$ **e** $\gamma \stackrel{*}{\Longrightarrow} \mathcal{E}\}$.

··· Funções FIRST e FOLLOW

FIRST(α), onde A → α é uma produção do tipo:

type → **simple** |
↑ id |
array **[simple]** of **type**
simple → integer | char | num **dotdot** num

FIRST(simple) = { integer, char, num }

FIRST(↑ id**) =** { ↑ }

FIRST(array **[simple]** of **type) =** { array }

FIRST(type) = FIRST(simple) ∪ { array, ↑ }

... Função FIRST

• Tratamento de produções A $\rightarrow \mathcal{E}$

Se **A** $\rightarrow$ **BC** e **B** $\rightarrow \alpha_1$
B $\rightarrow \alpha_2$
B $\rightarrow \mathcal{E}$

então para todo A $\rightarrow$ BC, FIRST(C) é incluído em FIRST(A).

Se C também gera $\mathcal{E}$ então FIRST(BC) e FIRST(A) contém $\mathcal{E}$.

• Quando usar produções $\mathcal{E}$?
no caso das outras não casarem com o símbolo lookahead.

Exemplo: **stmt** $\rightarrow$ `begin` **opt-stmts** `end`
opt-stmts $\rightarrow$ **stmt-list** | $\mathcal{E}$

Cálculo de FIRST

Como gerar FIRST(X)?

1. **Se X** $\in V_T$, **FIRST(X) = { X }**

2. **Se X** $\in V_N$ **e** $\exists$ **X** $\rightarrow$ **a**α, **inclua a em FIRST(X).**
   Se X $\rightarrow \mathcal{E}$, **inclua** $\mathcal{E}$ **em FIRST(X)**

3. **Se X** $\in V_N$ **e** $\exists$ **X** $\rightarrow Y_1\ Y_2 \cdots Y_k$ então
   3.1. **Se todos FIRST(Y_1), FIRST(Y_2), $\cdots$ FIRST(Y_{i-1}) e FIRST(Y_j) contêm $\mathcal{E}$para j = 1, 2, $\cdots$, i − 1, então inclua em FIRST(X) todos os a $\neq \mathcal{E}$ de FIRST(Y_i).**

   3.2. **Se todos FIRST(Y_1), FIRST(Y_2), $\cdots$ FIRST(Y_k), contêm $\mathcal{E}$, então inclua $\mathcal{E}$ em FIRST(X) .**

··· Cálculo de FIRST - EXEMPLO

1. **E** $\rightarrow$ **TE'**
2. **E'** $\rightarrow$ **+TE'** | $\mathcal{E}$
3. **T** $\rightarrow$ **FT'**
4. **T'** $\rightarrow$ ***FT'** | $\mathcal{E}$
5. **F** $\rightarrow$ **(E)** | id

Regra 2	**F** $\rightarrow$ **(E)** | id $\Longrightarrow$	**FIRST(F)**	= { (, id }
Regra 3 i = 1	**T** $\rightarrow$ **F**T' $\Longrightarrow$	**FIRST(T)**	= **FIRST(F)** $\equiv$ { (, id}
Regra 3 i = 1	**E** $\rightarrow$ **T**E' $\Longrightarrow$	**FIRST(E)**	= **FIRST(T)** $\equiv$ { (, id}
Regra 2	E' $\rightarrow$ **+T**E' | $\mathcal{E}$ $\Longrightarrow$	**FIRST(E')**	= { +, $\mathcal{E}$}
Regra 2	T' $\rightarrow$ $\star$**F**T' | $\mathcal{E}$ $\Longrightarrow$	**FIRST(T')**	= { *, $\mathcal{E}$}

··· Cálculo de FIRST - OUTRO EXEMPLO

A $\rightarrow$ **BCD**
B $\rightarrow$ **b** | $\mathcal{E}$
C $\rightarrow$ **c** | $\mathcal{E}$
D $\rightarrow$ **d** | **e** | **EFG**
E $\rightarrow$ **e** | $\mathcal{E}$
F $\rightarrow$ **f** | $\mathcal{E}$
G $\rightarrow$ **g** | $\mathcal{E}$
FIRST(A) = { **b,c,d,e,f,g,**$\mathcal{E}$}
FIRST(B) = { **b,**$\mathcal{E}$}
FIRST(C) = { **c,**$\mathcal{E}$}
FIRST(D) = { **d,e,f,g,**$\mathcal{E}$}
FIRST(E) = { **e,**$\mathcal{E}$}
FIRST(F) = { **f,**$\mathcal{E}$}
FIRST(G) = { **g,**$\mathcal{E}$}

Cálculo do FOLLOW

Como gerar FOLLOW(A)

1. Inclua $ em FOLLOW(S)
   Repita as regras seguintes até que nenhum símbolo possa ser acrescentado a algum FOLLOW.

2. Se $\exists$ A $\rightarrow \alpha$Bβ, inclua FIRST(β) exceto $\mathcal{E}$ em FOLLOW(B).

3. Se $\exists$ A $\rightarrow \alpha$B ou A $\rightarrow \alpha$Bβ, onde FIRST(β) contêm $\mathcal{E}$ ($\beta \stackrel{*}{\Longrightarrow} \mathcal{E}$), inclua FOLLOW(A) em FOLLOW(B).

··· Cálculo de FOLLOW - EXEMPLO

1. E $\rightarrow$ TE'
2. E' $\rightarrow$ +TE' | $\mathcal{E}$
3. T $\rightarrow$ FT'
4. T' $\rightarrow$ *FT' | $\mathcal{E}$
5. F $\rightarrow$ (E) | id

FOLLOW(E) = {), $ }
FOLLOW(E') = {), $ }
FOLLOW(T) = { +,), $ }
FOLLOW(T') = { +,), $ }
FOLLOW(F) = { +, *,), $ }

··· Cálculo de FOLLOW - OUTRO EXEMPLO

A $\rightarrow$ BCD FIRST(A) = { b,c,d,e,f,g,$\mathcal{E}$}
B $\rightarrow$ b | $\mathcal{E}$ FIRST(B) = { b,$\mathcal{E}$}
C $\rightarrow$ c | $\mathcal{E}$ FIRST(C) = { c,$\mathcal{E}$}
D $\rightarrow$ d | e | EFG FIRST(D) = { d,e,f,g,$\mathcal{E}$}
E $\rightarrow$ e | $\mathcal{E}$ FIRST(E) = { e,$\mathcal{E}$}
F $\rightarrow$ f | $\mathcal{E}$ FIRST(F) = { f,$\mathcal{E}$}
G $\rightarrow$ g | $\mathcal{E}$ FIRST(G) = { g,$\mathcal{E}$}

FOLLOW(A) = { $ }
FOLLOW(B) = { c,d,e,f,g,$ }
FOLLOW(C) = { d,e,f,g,$ }
FOLLOW(D) = { $ }
FOLLOW(E) = { f,g,$ }
FOLLOW(F) = { g,$ }
FOLLOW(G) = { $ }

GRAMÁTICAS LL(1)

Gramáticas LL(1)

Analisadores sintáticos preditivos podem ser construídos para uma classe de gramáticas chamada LL(1), onde,

1^o $L \equiv$ *varre a entrada da esquerda para a direita,* $(L =$ *Left-to-right*$)$,

2^o $L \equiv$ *derivação mais à esquerda,* $(L =$ *Leftmost*$)$ **e**

"1" pelo uso de um símbolo à frente na entrada utilizado em cada passo para tomar as decisões quanto a ação de análise.

A classe de gramáticas LL(1) é rica o suficiente para reconhecer a maioria das construções presentes nas linguagens de programação, MAS nenhuma gramática com recursão à esquerda ou ambígua pode ser LL(1).

· · · Gramáticas LL(1)

Uma gramática G é LL(1) se e somente se, sempre que $A \to a|b$ forem duas produções distintas de G, as seguintes condições são verdadeiras:

1. **Para um terminal a, tanto α quanto β não derivam cadeias começando com a.**

2. **No máximo um dos dois, α ou β, pode derivar a cadeia vazia.**

3. **Se $\beta \Rightarrow \epsilon$, então α não deriva nenhuma cadeia começando com um terminal em $FOLLOW(A)$.**

   De mesma forma, se $\alpha \Rightarrow \epsilon$, então β não deriva qualquer cadeia começando com um terminal em $FOLLOW(A)$.

· · · Gramáticas LL(1)

Analisadores preditivos podem ser construídos para gramáticas LL(1), pois é possível selecionar a produção apropriada a ser aplicada para um não-terminal examinando-se apenas o símbolo corrente da entrada restante.

- **Dado o símbolo corrente "a" da entrada e o não-terminal a ser expandido A, só existe uma alternativa em: A** $\to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ **que** deriva uma cadeia que começa com "a" .

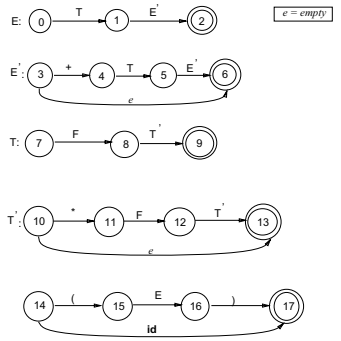
stat $\to$ **if C then stat else stat**
$\mid$ **while C do stat**
$\mid$ **begin stat-list** end

`if`, `while` **e** `begin` **indicam qual é a alternativa usada.**

- **Eficiente e fácil de implementar se a linguagem usada implementa chamada de procedimento eficientemente.**

Diagrama de Transição para Analisadores Sintáticos Preditivos

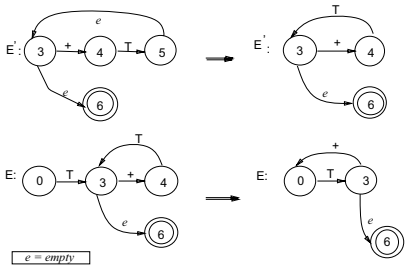
Úteis para visualizar analisadores preditivos.



predictive parser: **acabar com o não-determinísmo. Exemplo:**

Diagrama de Transição Simplificado

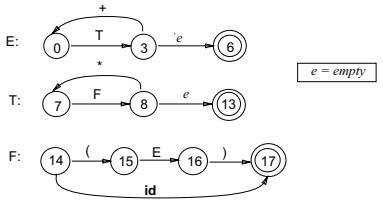
E → TE'
E' → +TE' | $\mathcal{E}$
T → FT'
T' → *FT' | $\mathcal{E}$
F → (E) | id



observação: $\mathcal{E} \equiv e$ nas figuras.

Mais Diagramas de Transição Simplificados

E → TE'
E' → +TE' | $\mathcal{E}$
T → FT'
T' → *FT' | $\mathcal{E}$
F → (E) | id



Execução: 20 a 25% mais rápida.

Projetando um "Predictive Parser"

Programa consistindo de um procedimento para cada NT, onde cada procedimento faz o seguinte:

1. **Ele decide que produção usar olhando o símbolo lookahead.**
   Usa-se a produção com o lado direito α se o símbolo lookahead está no FIRST(α).

   OBS.: Conflitos entre dois lados direitos para qualquer símbolo lookahead → não pode usar este método de reconhecimento para a gramática.

2. **Procedimento usa uma produção interpretando o seu lado direito.**
   Um NT resulta em uma chamada para o procedimento para o NT.
   Casamento do *token* com o símbolo *lookahead* faz com que o próximo símbolo seja lido.

3. ***token* da produção não casa com lookahead → erro de sintaxe.**

Exemplo da Tabela do Analisador Sintático Preditivo

Dada a gramática abaixo e a entrada: id + id * id

E → TE' **E' → +TE' | $\mathcal{E}$** **T' → *FT' | $\mathcal{E}$** **T → FT'** **F → (E) | id**

Não-terminal	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E' →$\mathcal{E}$	E'→$\mathcal{E}$
T	T→FT'			T→FT'		
T'		T'→$\mathcal{E}$	T' →*FT'		T'→$\mathcal{E}$	T'→$\mathcal{E}$
F	F→ id			F→(E)		

Construção da Tabela para o Reconhecedor Preditivo

O Algoritmo 4.31 coleta as informações dos conjuntos $FIRST$ e $FOLLOW$ em uma tabela de reconhecimento sintático preditivo $M[A, a]$, um arranjo bidimensional, onde A é um não-terminal e a é um terminal ou o símbolo \$, o marcador de fim de entrada.

- Suponha A $\rightarrow \alpha$ é uma produção com "a" em FIRST(α).
- O reconhecedor irá expandir A em α quando o símbolo de entrada for "a".
- Complicação:
  $\alpha = \mathcal{E}$ ou $\alpha \stackrel{*}{\Longrightarrow} \mathcal{E}$
- Expande A se o símbolo de entrada está em FOLLOW(A) ou se \$ da entrada foi alcançado e \$ está no FOLLOW(A).

··· Construção da Tabela para o Reconhecedor Preditivo

Algoritmo 4.31: Construção da tabela para o reconhecedor sintático preditivo.

ENTRADA: Gramática G.
SAÍDA: Tabela de análise M.
MÉTODO: Para cada produção $A \rightarrow \alpha$ da gramática, faça o seguinte:

1. Para cada terminal $a \in$ FIRST(α), inclua A $\rightarrow \alpha$ em $M[A, a]$.
2. Se $\mathcal{E}$ pertence a FIRST(α), inclua A $\rightarrow \alpha$ em $M[A, b]$ para cada $b \in V_T$, b em $FOLLOW(A)$.
3. Se $\mathcal{E}$ pertence a FIRST(α) e \$ pertence a FOLLOW(A), inclua A $\rightarrow \alpha$ em $M[A, \$]$.
4. Entradas de M não definidas $\rightarrow$ "erro".

··· Construção da Tabela para o Reconhecedor Preditivo

1. E $\rightarrow$ TE'
2. E' $\rightarrow$ +TE' | $\mathcal{E}$
3. T $\rightarrow$ FT'
4. T' $\rightarrow$ *FT' | $\mathcal{E}$
5. F $\rightarrow$ (E) | id

FIRST(E) = { (, id }	FOLLOW(E) ={), \$ }
FIRST(E') = { +, $\mathcal{E}$}	FOLLOW(E') ={), \$ }
FIRST(T) = { (, id }	FOLLOW(T) ={ +,), \$ }
FIRST(T') = { *, $\mathcal{E}$}	FOLLOW(T') ={ +,), \$ }
FIRST(F) = { (, id }	FOLLOW(F) ={ +, *,), \$ }

··· Construção da Tabela para o Reconhecedor Preditivo

- E $\rightarrow$ TE' e FIRST(T$E^{'}$) = { (, id } :
  M[E,(] = E $\rightarrow$ TE' (2)
  M[E,id] = E $\rightarrow$ TE'
- E' $\rightarrow$ +TE' e FIRST(+TE') = { + } :
  M[E',+] = E' $\rightarrow$ +TE' (2)
- E' $\rightarrow \mathcal{E}$ e FIRST($\mathcal{E}$) = { $\mathcal{E}$} FOLLOW(E') = {), \$ }:
  M[E',)] = E' $\rightarrow \mathcal{E}$ porque $\mathcal{E}$ está em FIRST($\mathcal{E}$) e ")" está em FOLLOW(E') (3)
  M[E',\$] = E' $\rightarrow \mathcal{E}$ porque $\mathcal{E}$ está em FIRST($\mathcal{E}$) e \$ em FOLLOW(E') (4)

Construção da Tabela para o Reconhecedor Preditivo

- T → FT' e FIRST(FT') = { (, id } :
  M[T,(] = T → FT' (2)
  M[T,id] = T → FT'

- T' → ⋆FT' e FIRST(⋆FT) = { ⋆ } :
  M[T',⋆] = T' → ⋆FT (2)

- T' → $\mathcal{E}$ e FIRST($\mathcal{E}$) = { $\mathcal{E}$}
  FOLLOW(T') = { +,), $ }:
  M[T',+] = T' → $\mathcal{E}$ porque $\mathcal{E}$ está em
  FIRST($\mathcal{E}$) e "+" está em FOLLOW(T') (3)

Construção da Tabela para o Reconhecedor Preditivo

M[T',)] = T' → $\mathcal{E}$ porque $\mathcal{E}$ está em
FIRST($\mathcal{E}$) e ")" está em FOLLOW(T') (3)

M[T',$] = T' → $\mathcal{E}$ porque $\mathcal{E}$ está em
FIRST($\mathcal{E}$) e $ em FOLLOW(T') (4)

- F → (E) e FIRST((E)) = { (} :
  M[F,(] = F → (E) (2)

- F → id e FIRST(id) = { id }
  M[F,id] = F → id (2)

Exemplo de Gramática que NÃO é LL(1)

1. S → if E then S S' | a
2. S' → else S | $\mathcal{E}$
3. E → b

FIRST(S) = { if, a }
FIRST(S') = { else, $\mathcal{E}$}
FIRST(E) = { b }

FOLLOW(S) = { $, else }
⟹ FOLLOW(S') = { $, else }
FOLLOW(E) = { then }

NT	a	b	else	if	then	$
S	S → a			S → if E then S S'		
S'			S' →$\mathcal{E}$ S' → else S			S' → $\mathcal{E}$
E		E → b				

Outro Exemplo de Gramática LL(1)

1 S → "("X
2 S → E "]"
3 S → F ")"
4 X → E ")"
5 X → F "]"
6 E → A
7 F → A
8 A → *empty*

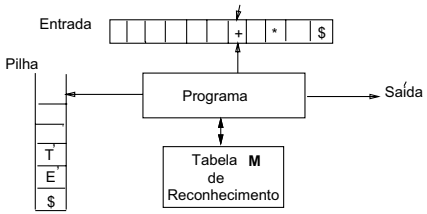
empty representa a cadeia vazia.

First(S) = {(,],) }	Follow(S) = { $ }
First(X) = {),] }	Follow(X) = { $ }
First(E) = { *empty* }	Follow(E) = {],) }
First(F) = { *empty* }	Follow(F) = {),] }
First(A) = { *empty* }	Follow(A) = {),] }

	(	)	]	$
S	S → (X	S → F)	S → E]	
X		X → E)	X → F]	
E		E → A	E → A	
F		F → A	F → A	
A		A → *empty*	A → *empty*	

ANALISADOR PREDITIVO SEM RECURSÃO

Analisador Preditivo sem Recursividade (LL(1))



- Apenas uma maneira mais eficiente de implementar o reconhecedor por descida recursiva.
- Pilha contém símbolos $\in V_T \cup V_N$ e a marca de fundo de pilha \$. No início a pilha contém \$S, onde S = axioma.
- *entrada* contém o ***string*** a ser analisado seguido da marca \$.
- Tabela do reconhecedor: M[A,a], A $\in V_N$ e a V_T

··· Analisador Preditivo sem Recursividade (LL(1))

Analisador Preditivo sem Recursividade é controlado por um programa.
O símbolo do topo da pilha **X** e o símbolo de entrada **a** determinam a ação do reconhecedor.

Algoritmo 4.34: Reconhecimento preditivo dirigido por tabela.

ENTRADA: Uma cadeia w de entrada e uma tabela M de análise para a gramática G.
SAÍDA: A derivação mais à esquerda de w, se w estiver em L(G); caso contrário, uma indicação de erro.
MÉTODO: Inicialmente, o analisador está em uma configuração com w no buffer de entrada e o símbolo inicial S de G no topo da pilha, acima de \$.

··· Analisador Preditivo sem Recursividade (LL(1))

··· **Algoritmo 4.34:** Reconhecimento preditivo dirigido por tabela.

1. se X = a = \$ então aceita.

2. se X = a ≠ \$ então desempilha X da pilha e avança ponteiro para o próximo símbolo da entrada.

3. se X é um não-terminal então consulta entrada M[X,a] da tabela do reconhecedor.
   Entrada pode ser: uma produção X ou um erro.

   Se M[X,a] = {X → UVW } o reconhecedor substitui X do topo da pilha por WVU. U no topo da pilha.

··· Analisador Preditivo sem Recursividade (LL(1))

··· **Algoritmo 4.34:** Reconhecimento preditivo dirigido por tabela.

define ip para que aponte para o primeiro símbolo de w;
define X para ser o símbolo no topo da pilha;
while ($X \neq \$$) { /* pilha não está vazia */
  if (X is a) desempilha e avança ip;
  else if (X é um terminal) $error()$;
  else if ($M[X,a]$ é uma entrada de erro) $error()$;
  else if ($M[X,a] = X \to Y_1 Y_2 \cdots Y_k$) {
    imprime a produção $X \to Y_1 Y_2 \cdots Y_k$;
    desempilha X;
    empilha $Y_k, Y_{k-1}, \ldots, Y_1$ na pilha, com Y_1 no topo;
  }
  define X como o símbolo no topo da pilha;
}

- Observação: o algoritmo é fixo. A tabela M é que muda.

Movimentos feitos pelo Analisador Sintático Preditivo para a Entrada id + id * id

Casamento	Pilha	Entrada	Ação
	$E\$$	id + id * id$	
	$TE'\$$	id+id*id$	imprime $E \to TE'$
	$FT'E'\$$	id+id*id$	imprime $T \to FT'$
	$id\ T'E'\$$	id+id*id$	imprime $F \to$ id
id	$T'E'\$$	$+ id * id\$$	match id
id	$E'\$$	$+ id * id\$$	imprime $T' \to \epsilon$
id	$+\ TE'\$$	$+ id * id\$$	imprime $E' \to + TE'$
$id +$	$TE'\$$	$id * id\$$	match $+$
$id +$	$FT'E'\$$	$id * id\$$	imprime $T \to FT'$
$id +$	$id\ T'E'\$$	$id * id\$$	imprime $F \to$ id
$id + id$	$T'E'\$$	$* id\$$	match id
$id + id$	$* FT'E'\$$	$* id\$$	imprime $T' \to * FT'$
$id + id *$	$FT'E'\$$	$id\$$	match $*$
$id + id *$	$id\ T'E'\$$	$id\$$	imprime $F \to$ id
$id + id * id$	$T'E'\$$	$\$$	match id
$id + id * id$	$E'\$$	$\$$	imprime $T' \to \epsilon$
$id + id * id$	$\$$	$\$$	imprime $E' \to \epsilon$

Recuperação de Erros em Analisador Sintático Preditivo

Situações de Erro

- Terminal na pilha difere da entrada.
- M[A,a] vazia.

Modo Pânico "panic-mode"

Em caso de erro, ignora entradas até encontrar um token para sincronização.

··· Recuperação de Erros em Analisador Sintático Preditivo

O que fazer quando M[A,a] estiver vazia?

1. Colocar tokens em FOLLOW(A) no conjunto de sincronização de A. Em caso de erro, desempilhar A e ignorar entradas até sincronização.

2. O item 1 pode não ser eficiente.

   Exemplo: ";" pode estar no conjunto de sincronização de uma expressão. Um ";" faltando na entrada resulta em um grande trecho de programa ignorado.
   Então, adiciona-se a FOLLOW(A) quando A é uma construção "menor" (exemplo, expressão) os tokens que podem iniciar uma construção maior (exemplo, comandos).

Recuperação de Erros em Analisador Sintático Preditivo

3. Adicionando-se tokens em FIRST(A) ao conjunto de sincronização de A permite reassumir reconhecimento de A.

O que fazer quando um terminal na pilha difere da entrada?

Desempilha terminal; emite mensagem de erro dizendo que terminal foi removido e continua.

Recuperação de Erros em Analisador Sintático Preditivo

Dada a gramática abaixo e a entrada: * id * + id

E→TE'
T→FT'
E'→+TE'|$\mathcal{E}$
T'→ *FT'|$\mathcal{E}$
F→(E)|id

NT	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E' →$\mathcal{E}$	E'→$\mathcal{E}$
T	T→FT'	synch		T→FT'	synch	synch
T'		T'→$\mathcal{E}$	T' →*FT'		T'→$\mathcal{E}$	T'→$\mathcal{E}$
F	F→ id	synch	synch	F→(E)	synch	synch

Recuperação de Erros em Analisador Sintático Preditivo

Pilha	Entrada	Comentário
$E\,\$$	$)\,id*+id\,\$$	erro, pula)
$E\,\$$	$id*+id\,\$$	id está em FIRST(E)
$TE'\,\$$	$id*+id\,\$$	
$FT'E'\,\$$	$id*+id\,\$$	
$id\;T'E'\$$	$id*+id\,\$$	
$T'E'\,\$$	$*+id\,\$$	
$*\,FT'E'\,\$$	$*+id\,\$$	
$FT'E'\,\$$	$+id\,\$$	erro, $M[F,+]$ = synch
$T'E'\,\$$	$+id\,\$$	F foi desempilhado
$E'\,\$$	$+id\,\$$	
$+\,TE'\,\$$	$+id\,\$$	
$TE'\,\$$	$id\,\$$	
$FT'E'\,\$$	$id\,\$$	
$id\,T'E'\,\$$	$id\,\$$	
$T'E'\,\$$	$\$$	
$E'\,\$$	$\$$	
$\$$	$\$$	

Considerações Finais sobre Predictive Parser

• Maior Dificuldade

Escrever uma gramática para uma linguagem fonte de tal forma que um reconhecedor deste tipo possa ser construido a partir da gramática dada.

• Eliminação de recursividade a esquerda e fatoração à esquerda – fáceis de fazer , ***contudo*** a gramática resultante é difícil de ler e difícil de usar.

• Organização Comum

Usar *Analisador Sintático Preditivo* para estruturas de controle.
Usar *Precedência Operadora* para expressões.
Usar *geradores de analisadores sintáticos LR* .

"Obter todos os benefícios do *Predictive Parser* e *Precedência Operadora* automaticamente".

4.5 RECONHECIMENTO ASCENDENTE

Reconhecimento Ascendente

1. **Descendentes** *TOP-DOWN*
   - Por Descida Recursiva com retrocesso (*backtracking*)
   - Analisador Sintático Preditivo
   - Analisador Sintático Preditivo sem Recursão (Gramáticas LL(1))
2. **Ascendentes** *BOTTOM-UP (Shift-Reduce)*
   - LR(k)
   - LALR(k)
   - SLR(k)

··· Reconhecedores Sintáticos Ascendentes

- **Método:**
  Tenta construir a àrvore de derivação a partir das ***folhas*** em direção à ***raiz*** da àrvore.
- **Redução**
  Substituição do lado direito pelo lado esquerdo de uma produção em uma forma sentencial.
- O processo de reconhecimento "bottom-up" corresponde a uma derivação mais a direita ao inverso.
- **Problemas:**
  (1) Determinar o ***string a ser reduzido.***
  (2) Determinar a produção a ser usada.

··· Reconhecedores Sintáticos Ascendentes

Exemplo

Considere a gramática: S → aABe
A → Abc | b
B → d

A sentença ***abbcde*** pode ser reduzida para S da seguinte forma:

a$\underline{b}$bcde
a$\underline{Abc}$de
aA$\underline{d}$e
$\underline{aABe}$
S

S $\overset{rm}{\Longrightarrow}$ aA$\underline{B}$e $\overset{rm}{\Longrightarrow}$ a$\underline{A}$de $\overset{rm}{\Longrightarrow}$ a$\underline{Abc}$de $\overset{rm}{\Longrightarrow}$ abbcde

Handle ≡ Redutendo

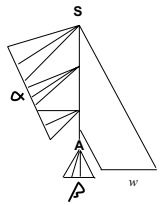
- **Handle de um *string* é um *substring* que casa o lado direito de uma produção, e cuja redução para o não-terminal do lado esquerdo representa um passo da derivação mais à direita ao inverso.**
- **Ou seja, Handle de uma forma sentencial direita γ gerada por derivação mais à direita é uma produção $A \to \beta$ e uma posição em γ onde β pode ser localizado e substituido por A.**
- **Se $S \stackrel{*}{\Longrightarrow}_{rm} \alpha A w \Longrightarrow_{rm} \alpha\ \beta\ w$, $w \in V_T^*$, então $A \to \beta$ e a posição que segue α é 1 handle de $\alpha\beta w$.**

··· Handle

- **Exemplos:**
  1. **abbcde é uma forma sentencial mais à direita cujo *handle* é A → b na posição 2.**
  2. **Para a forma sentencial mais à direita aAbcde o *handle* é A → Abc na posição 2.**

··· Handle - Outro Exemplo

O handle A → β na árvore para $\alpha\beta w$



Dada a gramática: E → E+E | E*E | (E) | id

Duas possíveis derivações mais a direita

E	$\Longrightarrow_{rm}$	$\underline{E + E}$	ou	E	$\Longrightarrow_{rm}$ $\underline{E * E}$
	$\Longrightarrow_{rm}$	$E + \underline{E * E}$			$\Longrightarrow_{rm}$ $E * \underline{id_3}$
	$\Longrightarrow_{rm}$	$E + E * \underline{id_3}$			$\Longrightarrow_{rm}$ $\underline{E + E} * id_3$ ⟸
	$\Longrightarrow_{rm}$	$E + \underline{id_2} * id_3$			$\Longrightarrow_{rm}$ $E + \underline{id_2} * id_3$
	$\Longrightarrow_{rm}$	$\underline{id1} + id_2 * id_3$ ⟸			$\Longrightarrow_{rm}$ $\underline{id_1} + id_2 * id_3$

Poda do Handle

- **Uma forma sentencial mais à direita em reverso pode ser obtida por meio do poda do handle (*handle prunning*), ou seja, começamos com uma *cadeia* de terminais w que desejamos analisar.**
- **Se w é uma sentença da gramática então w = Y_n onde Y_n é a n-ésima forma sentencial mais a direita de alguma derivação mais a direita *ainda não conhecida*.**

$$S = Y_0 \Longrightarrow_{rm} Y_1 \Longrightarrow_{rm} \cdots Y_{n-1} \Longrightarrow_{rm} Y_n \Longrightarrow_{rm} = w$$

· · · Handle Prunning

- Para reconstruir esta derivação em ordem inversa, localizamos o ***handle*** β_n em Y_n e substituimos β_n pelo lado esquerdo de alguma produção $A_n \rightarrow \beta_n$ para obter a (n−1) forma sentencial mais à direita "Y_{n-1}".

- Note que ainda não sabemos como encontrar o ***handle***.

- Repetimos este processo, ou seja, localizamos o ***handle*** β_{n-1} em Y_{n-1} e reduzimos este ***handle*** para obter a forma sentencial mais à direita Y_{n-2}.

- Se continuarmos e conseguirmos alcançar o símbolo inicial da gramática paramos e anunciamos sucesso no reconhecimento.

· · · Handle Prunning

- Exemplo: Dada a entrada: id_1 * id_2 e a gramática:

E → E+E | E*E | (E) | id

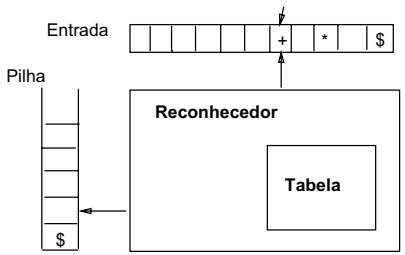
Forma Sentencial à Direita	Handle	Produção de Redução
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$
E		

NOTA: derivação mais à direita ao inverso é freqüentemente chamada de ***seqüência de reduções canônicas***.

Analisador Sintático *Shift-Reduce*

- **Reconhecedor:**
  Estrutura de Dados: pilha e entrada.

- **Convenções:**
  Inicialmente pilha contem $, $ não faz parte do vocabulário.
  A entrada termina com $.



· · · Analisador Sintático *Shift-Reduce*

Operações

- **TRANSFERE** (*SHIFT*)
  Transfere o próximo símbolo da entrada para o topo da pilha.

- **REDUZ** (*REDUCE*)
  Substituir o *handle* , que está no topo da pilha, pelo não-terminal que o produz. *Handle* é o β de A → β.

- **ACEITA** (*ACCEPT*)
  Tudo terminou bem !!!

- **ERRO** (*ERROR*)
  Erro de sintaxe.

··· Analisador Sintático *Shift-Reduce*

Diferença entre os Analisadores Sintáticos Ascendentes

"A diferença básica entre os vários $\underline{shift-reduce}$ $\underline{parsers}$ situa-se no método usado para determinar o *handle* ."

··· Analisador Sintático *Shift-Reduce* - EXEMPLO

Dada a entrada: $id_1 + id_2 * id_3$ e a gramática:

E $\rightarrow$ E+E
| E*E
| (E)
| id

E $\Longrightarrow_{rm}$ $\underline{E + E}$
$\Longrightarrow_{rm}$ E + $\underline{E * E}$
$\Longrightarrow_{rm}$ E + E * $\underline{id_3}$
$\Longrightarrow_{rm}$ E + $\underline{id_2}$ * id_3
$\Longrightarrow_{rm}$ $\underline{id1}$ + id_2 * id_3

Pilha	Entrada	Ação
(1) $	$id_1 + id_2 * id_3$ $	transfere (*shift*)
(2) $$\underline{id_1}$$	$+ id_2 * id_3$ $	reduz segundo E $\rightarrow$ **id**
(3) $E	$+ id_2 * id_3$ $	transfere
(4) $E +	$id_2 * id_3$ $	transfere
(5) $E + $\underline{id_2}$	$* id_3$ $	reduz segundo E $\rightarrow$ **id**
(6) $E + E	$* id_3$ $	transfere
(7) $E + E *	id_3 $	transfere
(8) $E + E * $\underline{id_3}$	$	reduz segundo E $\rightarrow$ **id**
(9) $E + $\underline{E * E}$	$	reduz segundo E $\rightarrow$ E * E
(10) $$\underline{E + E}$$	$	reduz segundo E $\rightarrow$ E + E
(11) $E	$	aceita

··· Analisador Sintático *Shift-Reduce*

O uso de uma pilha durante a análise sintática *shift-reduce* é justificado por um fato importante: "o *handle* sempre aparece do topo para baixo na pilha do reconhecedor."

Formas de derivação:

(1) S $\stackrel{*}{\Longrightarrow}_{rm}$ αAz $\Longrightarrow_{rm}$ $\alpha\beta$Byz $\Longrightarrow_{rm}$ $\alpha\beta\gamma$yz

Considerando em reverso:

Pilha	Entrada	Comentários
$\$\alpha\beta\gamma$	yz$	
$\$\alpha\beta$B	yz$	handle não pode estar abaixo de B na pilha
$\$\alpha\beta$By	z$	
$\$\alpha$A		

··· Analisador Sintático *Shift-Reduce*

··· O uso de uma pilha durante a análise sintática *shift-reduce* é justificado por um fato importante: "o handle sempre aparece do topo para baixo na pilha do reconhecedor."

Formas de derivação:

(2) S $\stackrel{*}{\Longrightarrow}_{rm}$ αBxAz $\Longrightarrow_{rm}$ α Bxyz $\Longrightarrow_{rm}$ $\alpha\gamma$xyz

Considerando em reverso:

Pilha	Entrada
$\$\alpha\gamma$	xyz$
$\$\alpha$Bxy	z$
$\$\alpha$BxA	

Construção da Árvore de Derivação

- Use uma pilha adicional paralela à pilha sintática.

1. Shift a, a $\in V_T$:

   Crie um vértice contendo "a" e coloque o apontador para o vértice na pilha 2.

2. Reduce $X_1\ X_2 \cdots X_n$ para A:

   Crie um vértice para A cujos filhos são as árvores $X_1\ X_2 \cdots X_n$.

Observação: *reduce* $\mathcal{E}$ a A:
Crie vértice A com filho $\mathcal{E}$.

Conflitos Durante a Análise Sintática "shift-reduce"

stmt → if expr then stmt
| if expr then stmt else stmt
| other

Para a configuração:

Pilha	Entrada
··· if expr then stmt	else ··· $

Temos um conflito shift-reduce .

··· Conflitos Durante a Análise Sintática "shift-reduce"

(1)	*stmt*	→ id (*parameter_list*)
(2)	*stmt*	→ *expr* := *expr*
(3)	*parameter_list*	→ *parameter_list* , *parameter*
(4)	*parameter_list*	→ *parameter*
(5)	*parameter*	→ id ⟸
(6)	*expr*	→ id (*expr_list*)
(7)	*expr*	→ id ⟸
(8)	*expr_list*	→ *expr_list* , *expr*
(9)	*expr_list*	→ *expr*

Para a entrada: A(I,J)

Pilha	Entrada
··· id (id	, id) ··· $

reduz para: parameter → id ou expr → id
???

RECONHECEDORES LR

Reconhecedores LR(k)

"L": representa a escansão da entrada da esquerda para a direita.

"R": representa a construção das derivações mais à direita ao inverso.

"k": representa os símbolos à frente no fluxo de entrada que auxiliam nas decisões de análise.

SLR(K)

LALR(K)

LR(K) Canônico

Porque Reconhecedores Sintáticos LR?

Vantagens dos Reconhecedores LR(K)

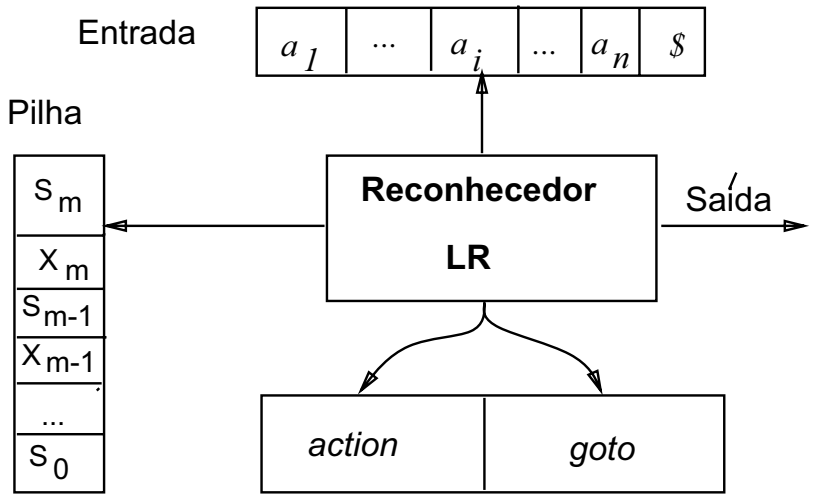
1. Aplicáveis à grande maioria das linguagens de programação.
2. Mais geral que outros métodos.
3. Podem ser implementados com o mesmo grau de eficiência (espaço e tempo) que outros "shift-reduce".
4. Detecção de erros tão cedo quanto possível (propriedade do prefixo correto).

··· Porque Reconhecedores Sintáticos LR?

Desvantagens dos Reconhecedores LR(K)

- Necessário o uso de computador para geração do analisador.

Arquitetura dos Reconhecedores LR



Estrutura da Tabela de Análise

ACTION - função de ação de análise.

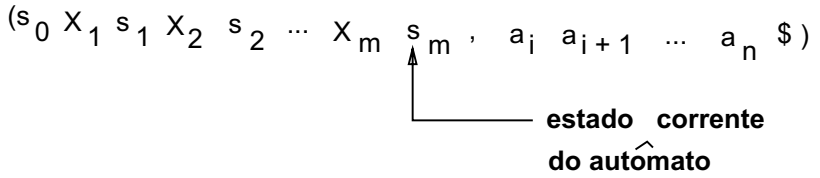
GOTO - função de transição.

1. **ACTION**[S_m,a_i]
   - 1.1. *Shift S*
   - 1.2. *Reduce* $A \rightarrow \beta$
   - 1.3. *Accept*
   - 1.4. *Error*

2. *GOTO* dado um estado e um símbolo da gramática produz um novo estado.

Configurações do Analisador LR

- Configuração de um reconhecedor LR: é um par

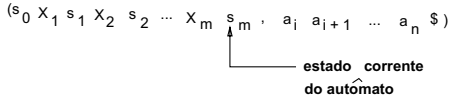


- e representa a forma sentencial mais à direita:

$$X_1\ X_2 \cdots X_m\ a_i\ a_{i+1} \cdots a_n\ \$$$

Comportamento dos Reconhecedores LR

- Dada a configuração de um reconhecedor LR:



- if ACTION[s_m , a_i] = shift s
  Nova Configuração: ($s_0\ X_1\ s_1\ X_2\ s_2 \cdots X_m\ s_m\ a_i$ **s** , $a_{i+1} \cdots a_n$ \$)
  onde **s** = GOTO[s_m , a_i] .

- if ACTION[s_m , a_i] = reduce A $\rightarrow \beta$
  Nova Configuração:($s_0\ X_1\ s_1\ X_2\ s_2 \cdots X_{m-r}\ s_{m-r}$ **A s** , $a_i\ a_{i+1} \cdots a_n$ \$)
  onde **s** = GOTO[s_{m-r} , **A**] , r = comprimento de β.

- if ACTION[s_m , a_i] = accept – reconhecimento acabou.
- if ACTION[s_m , a_i] = error – ativa recuperador de erros.

··· Comportamento dos Reconhecedores LR

Algoritmo 4.44 algoritmo de análise LR

ENTRADA: Uma cadeia de entrada w e uma tabela de análise LR com funções $ACTION$ e $GOTO$ para uma gramática G.

SAÍDA: Se w está em $L(G)$, os passos de redução de uma análise ascendente para w; caso contrário, uma indicação de erro.

MÉTODO: Inicialmente, o analisador sintático possui w no buffer de entrada e S_0 em sua pilha, onde S_0 representa o estado inicial.

Comportamento dos Reconhecedores LR - Algoritmo 4.44

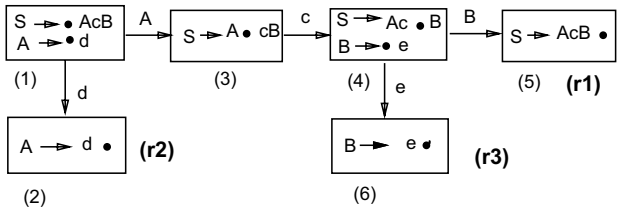
seja a o primeiro símbolo de $w\$$;
while(1) { /* repita indefinidamente */
  seja S o estado no topo da pilha;
  if (ACTION[S,a] = shift t) {
    empilha t na pilha;
    seja a o próximo símbolo da entrada;
  } else if (ACTION[S,a] = reduce $A \to \beta$) {
    desempilha símbolos $|\beta|$ da pilha;
    faça o estado t agora ser o topo da pilha;
    empilhe GOTO[t,A] na pilha;
    imprima a produção $A \to \beta$;
  } else if (ACTION[S,a] = accept) pare; /* análise terminou */
  else chame uma rotina de recuperação de erro;
}

Filosofia dos Reconhecedores LR

G = ({S, A, B }, {c, d, e}, P, S)

P: 1. S → A c B
   2. A → d
   3. B → e

S ⟹ AcB ⟹ Ace ⟹ dce



Reconhecimento LR de id * id + id

(1) E → E+ T
(2) | T
(3) T → T * F
(4) | F
(5) F → (E)
(6) | id

Pilha	Entrada	Ação
(1) 0	id * id + id $	shift
(2) 0 id 5	* id + id $	reduce F → id
(3) 0 F 3	* id + id $	reduce T → F
(4) 0 T 2	* id + id $	shift
(5) 0 T 2 * 7	id + id $	shift
(6) 0 T 2 * 7 id 5	+ id $	reduce F → id
(7) 0 T 2 * 7 F 10	+ id $	reduce T→ T*F
(8) 0 T 2	+ id $	reduce E → T
(9) 0 E 1	+ id $	shift
(10) 0 E 1 + 6	id $	shift
(11) 0 E 1 + 6 id 5	$	reduce F → id
(12) 0 E 1 + 6 F 3	$	reduce T → F
(13) 0 E 1 + 6 T 9	$	E → E + T
(14) 0 E 1	$	accept

Estado	Ação						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Gramáticas LRs

- Definição:
  Uma gramática G é LR se podemos construir, a partir de G, uma tabela de reconhecimento na qual toda entrada é univocamente definida.
- Existem gramáticas livres do contexto que não são LRs.
- Gramáticas ambíguas não são LRs.
- Gramáticas LRs não são ambíguas.

· · · Gramáticas LRs

- Muitas vezes, tem-se que olhar K símbolos à frente no fluxo de entrada para complementar a informação fornecida pelo estado no topo da pilha $\Longrightarrow$ Gramáticas LR(K).
- Na prática, k = 0 ou k = 1 é suficiente, porque toda linguagem LR(k) é LR(1).
- Reconhecedor LR é capaz de determinar o *handle* quando este aparece no topo da pilha.
- Existe um DFA que pode determinar que *handle* está no topo da pilha (se ele existir) lendo os símbolos da gramática presentes na pilha a partir fundo até o topo da pilha..

· · · Gramáticas LRs

- Em geral, o estado do topo da pilha incorpora toda a informação necessária para reconhecer o *handle*.
- Este estado é o estado que o DFA estaria se ele lesse os símbolos da gramática presentes na pilha a partir do fundo até o topo da pilha.

4.6 INTRODUÇÃO À ANÁLISE LR SIMPLES: SLR

Itens e o Autônomo LR(0)

Definição 1: Gramática Aumentada G' é uma gramática G com um novo símbolo de partida S' e a produção S' $\rightarrow$ S.

Definição 2: Um item LR(0) é uma produção com um "•" no seu lado direito.

A produção A $\rightarrow$ XYZ possui 4 itens:

$$A \rightarrow \bullet XYZ$$
$$A \rightarrow X \bullet YZ$$
$$A \rightarrow XY \bullet Z$$
$$A \rightarrow XYZ \bullet$$

A produção B $\rightarrow \mathcal{E}$ possui apenas um item: B $\rightarrow$ •

Fechamento de Conjuntos de Itens

Definição 3: Uma coleção canônica de itens LR(0) é uma coleção de conjuntos de itens LR(0).

Exemplo:
{{A $\rightarrow$ •XYZ,B $\rightarrow$ w•α},$\cdots$, {X $\rightarrow$ •Z,$\cdots$,Y $\rightarrow$ Z•Y, $\cdots$}, $\cdots$}.

Definição 4: *CLOSURE(I)*, onde ***I*** é um conjunto de itens:

1. Todo item em $\boldsymbol{I} \in$ *CLOSURE(I)*.

2. Se [A $\rightarrow$ α•Bβ] $\in$ *CLOSURE(I)* e B $\rightarrow$ γ é uma produção, então [B $\rightarrow$ •γ] $\in$ *CLOSURE(I)*.

   Este processo conhecido como *fechamento do estado*.

$\cdots$ Fechamento de Conjuntos de Itens

Observações:

Observe que, se uma `produção-B` for incluída no fechamento de I com o ponto mais à esquerda no seu lado direito, então todas as `produções-B` serão igualmente incluídas no fechamento.

Portanto não é necessário, em algumas circunstâncias, realmente listar os itens $B \rightarrow \cdot\gamma$ incluídos em I pela função $CLOSURE$.

Uma lista de não-terminais B, cujas produções foram incluídas dessa forma, é suficiente.

$\cdots$ Fechamento de Conjuntos de Itens

$\cdots$ Observações:

Dividimos todos os conjuntos de itens de interesse em duas classes:

1. Itens de base (*kernel*): o item inicial, $S' \rightarrow \cdot S$, e todos os itens cujos pontos não estão mais à esquerda em seus lados direitos.

2. Itens que não são de base (*Nonkernel*): todos os itens com seus pontos mais à esquerda de seus lados direitos, exceto para $S' \rightarrow \cdot S$.

Cada conjunto de itens de interesse é formado a partir do fechamento de um conjunto de *itens de base*, portanto, é possível representar os conjuntos de itens em que realmente estamos interessados com pouca memória se desprezarmos todos os itens que não são de base.

Computação do Fechamento I

```
SetOfItems CLOSURE(I) {
  J = I;
  repeat
    for ( cada item A → α·Bβ em J)
      for ( cada produção B → γ of G )
        if ( B → ·γ não está em J )
          adicione B → ·γ em J;
  until mais nenhum item seja adicionado a J em um
    passo do loop
  return J;
}
```

A Função de Transição

Definição 5: *GOTO(I,X)*,
onde *I* é um conjunto de itens e X $\in V_N \cup V_T$:

- **Se $[A \rightarrow \alpha \bullet X \beta] \in I$ então os elementos de** $CLOSURE([A \rightarrow \alpha X \bullet \beta]) \in GOTO(I,X)$.

Cálculo dos Conjuntos de Itens LR(0)

```
void itens(G') {
      C := { CLOSURE({S' → •S })}
      repeat
            for (cada conjunto de itens I ∈ C)
                  for (cada símbolo de gramática X ∈ V_N ∪ V_T)
                        if ( GOTO(I,X) ∉ C e GOTO(I,X) ≠ φ
                              inclua GOTO(I,X) em C
      until nenhum novo conjunto de itens seja incluído em C
}
```

C = conjunto de conjunto de itens
I = conjunto de itens.

Construção de Conjunto de Itens LR(0)

I_0: { $E' \rightarrow \bullet E$ | $\xrightarrow{E} I_1$
E → •E+T |
E → •T | $\xrightarrow{T} I_2$
T → •T*F |
T → •F | $\xrightarrow{F} I_3$
F → •(E) | $\xrightarrow{(} I_4$
F → •id } $\xrightarrow{id} I_5$

I_1: { $E' \rightarrow E\bullet$
E → E•+T } $\xrightarrow{+} I_6$

I_2: { E → T•
T → T•*F } $\xrightarrow{*} I_7$

I_3: { T → F• }

I_4: { F → (•E) | $\xrightarrow{E} I_8$
E → •E+T |
E → •T | $\xrightarrow{T} I_2$
T → •T*F |
T → •F | $\xrightarrow{F} I_3$
F → •id } | $\xrightarrow{id} I_5$

I_5: { F → id• }

I_6: { E → E+•T | $\xrightarrow{T} I_9$
T → •T*F |
T → •F | $\xrightarrow{F} I_3$
F → •(E) | $\xrightarrow{(} I_4$
F → •id } | $\xrightarrow{id} I_5$

I_7: { T → T*•F | $\xrightarrow{F} I_{10}$
F → •(E) | $\xrightarrow{(} I_4$
F → •id } $\xrightarrow{id} I_5$

I_8: { F → (E•) | $\xrightarrow{)} I_{11}$
E → E•+T } $\xrightarrow{+} I_6$

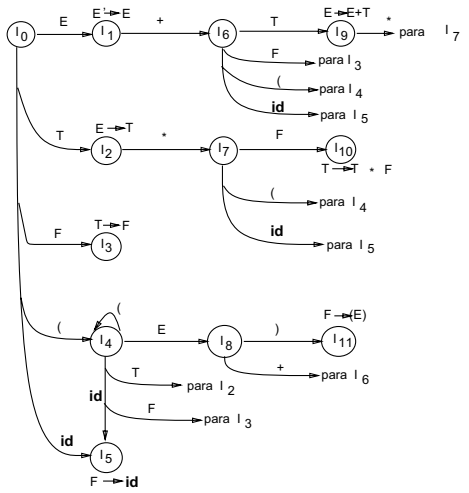
I_9: { E → E+T•
T → T•*F } $\xrightarrow{*} I_7$

I_{10}: { T → T*F• }

I_{11}: { F → (E)• } }

Observação: *Itens de base* (*kernel*) representados na cor azul. Demais itens, na cor preta, representam os *itens que não são de base.*

Autômato de Estados Finitos Determinístico



Uso do Autômoto LR(0)

Análise de id * id

Linha	Pilha	Símbolos	Entrada	Ação
(1)	0	$	id * id $	empilha 5 e avança
(2)	0 5	$ id	* id $	reduz segundo $F \rightarrow$ id
(3)	0 3	$ F	* id $	reduz segundo $T \rightarrow F$
(4)	0 2	$ T	* id $	empilha 7 e avança
(5)	0 2 7	$ T *	id $	empilha 5 e avança
(6)	0 2 7 5	$ T * id	$	reduz segundo $F \rightarrow$ id
(7)	0 2 7 10	$ T * F	$	reduz segundo $T \rightarrow T * F$
(8)	0 2	$ T	$	reduz segundo $E \rightarrow T$
(9)	0 1	$ E	$	aceitar

Construção da Tabela do Reconhecedor SLR(1) - Algoritmo 4.46:

ENTRADA: Uma gramática estendida G'.

SAÍDA: As funções $ACTION$ e $GOTO$ da tabela de análise SLR para G'.

MÉTODO:

1. Construa C = { I_0, I_1, $\cdots$, I_n } coleção de conjuntos de itens LR(0)

2. Seja: 0, 1, $\cdots$, n estados do reconhecedor. Estado i é construído a partir de I_i.

3. Seja a $\in V_T$ e A $\in V_N$

$\cdots$ Construção da Tabela do Reconhecedor SLR(1) - Algoritmo 4.46:

4. As ações (ACTION) do reconhecedor para o estado i são determinadas da seguinte forma:

   - se [A $\rightarrow \alpha \bullet$a$\beta$] $\in I_i$ e GOTO[I_i,a] = Ij
     então faça ACTION[i,a] = "shift j". "a" tem que ser terminal.

   - se [A $\rightarrow \alpha \bullet$] $\in I_i$ então faça ACTION[i,a] = "reduce A $\rightarrow \alpha$" $\forall$ a $\in$ FOLLOW(A).
     A não pode ser S' aqui.

   - se [$S' \rightarrow$ S$\bullet$] $\in I_i$ então faça ACTION[i,$] = "accept".

$\cdots$ Construção da Tabela do Reconhecedor SLR(1) - Algoritmo 4.46:

5. As transições GOTO para o estado i são construídas para todos os não-terminais A usando a regra: se GOTO(I_i, A) = I_j então GOTO(i,A) = j.

6. O estado que contém [$S' \rightarrow \bullet$S] é o estado inicial.

7. Posições vagas em ACTION ou GOTO indicam erros.

Se alguma ação conflitante for gerada pelas regras acima, a gramática não é SLR(1).

Exemplo da Construção da Tabela SLR(1)

ESTADO	ACAO						GOTO		
	id	+	*	(	)	$	E	T	F
0	(S5)			(S4)			(1)	(2)	(3)
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									

I_0: { $E' \rightarrow \bullet E$ | $\xrightarrow{E} I_1$
$E \rightarrow \bullet E+T$ |

$E \rightarrow \bullet T$ | $\xrightarrow{T} I_2$
$T \rightarrow \bullet T*F$ |

$T \rightarrow \bullet F$ | $\xrightarrow{F} I_3$
$F \rightarrow \bullet(E)$ | $\xrightarrow{(} I_4$
$F \rightarrow \bullet$ **id** } | $\xrightarrow{\text{id}} I_5$

··· Exemplo da Construção da Tabela SLR(1)

ESTADO	ACAO						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2									
3									
4									
5									
6									
7									
8									
9									
10									

I_1: { $E' \rightarrow E\bullet$

$E \rightarrow E\bullet+T$ } | $\xrightarrow{+} I_6$

··· Exemplo da Construção da Tabela SLR(1)

ESTADO	ACAO						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		(r3)	(S7)		(r3)	(r3)			
3		(r5)	(r5)		(r5)	(r5)			
4									
5									
6									
7									
8									
9									
10									

I_2: { $E \rightarrow T\bullet$ *regra 3* FOLLOW(E) = { $,+,) }

$T \rightarrow T\bullet*F$ } $\xrightarrow{*} I_7$ FOLLOW(T) = { +,),$,* }

I_3: { $T \rightarrow F\bullet$} *regra 5*

··· Exemplo da Construção da Tabela SLR(1)

ESTADO	ACAO						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		r3	S7		r3	r3			
3		r5	r5		r5	r5			
4	(S5)			(S4)			(8)	(2)	(3)
5									
6									
7									
8									
9									
10									

I_4: { $F \rightarrow (\bullet E)$ | $\xrightarrow{E} I_8$
$E \rightarrow \bullet E+T$ |

$E \rightarrow \bullet T$ | $\xrightarrow{T} I_2$
$T \rightarrow \bullet T*F$ |

$T \rightarrow \bullet F$ | $\xrightarrow{F} I_3$
$F \rightarrow \bullet(E)$ | $\xrightarrow{(} I_4$
$F \rightarrow \bullet$**id** } | $\xrightarrow{\text{id}} I_5$

··· Exemplo da Construção da Tabela SLR(1)

ESTADO	ACAO						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		r3	S7		r3	r3			
3		r5	r5		r5	r5			
4	S5			S4			8	2	3
5		(r7)	(r7)		(r7)	(r7)			
6									
7									
8									
9									
10									

I_5: { F → id• } *regra 7* FOLLOW(F) = { +,),*,$ }

Exemplo: ¬ SLR(1)

Conjunto itens LR(0):
(0) $S' \rightarrow S$
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → id
(5) R → L

FOLLOW(S') = { $ }
FOLLOW(S) = { $ }
FOLLOW(R) = { $, = } ⟸
FOLLOW(L) = { $, = }

States	Action			Goto
	···	=	···	···
	···		···	···
2	···	S6/r5	···	···
	···		···	···

I_0: $S' \rightarrow \bullet S$, $\xrightarrow{S} I_1$
S → •L=R $\xrightarrow{L} I_2$
S → •R, $\xrightarrow{R} I_3$
L → •*R $\xrightarrow{*} I_4$
L → •id $\xrightarrow{id} I_5$
R → •L $\xrightarrow{L} I_2$

⇒ I_2: S → L•=R $\xrightarrow{=} I_6$
R → L•

I_4: L → *•R $\xrightarrow{R} I_7$
R → •L $\xrightarrow{L} I_8$
L → •*R $\xrightarrow{*} I_4$
L → •id $\xrightarrow{id} I_5$

I_1: $S' \rightarrow S\bullet$

I_3: S → R•

I_5: L → id•

I_6: S → L=•R $\xrightarrow{R} I_9$
R → •L $\xrightarrow{L} I_8$
L → •⋆R $\xrightarrow{*} I_4$
L → •id $\xrightarrow{id} I_5$

I_7: L → *R•
I_8: R → L•
I_9: S → L=R•

Considerações sobre a Gramática (¬SLR(1))

• **A gramática abaixo não é ambígua. Porque aparece este conflito *shift-reduce*?**

Conjunto itens LR(0):
(0) $S' \rightarrow$ **S**
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → id
(5) R → L

Ele aparece porque o método de construção do reconhecedor SLR não é o suficientemente poderoso para guardar o contexto à esquerda para decidir que ação o reconhecedor deve tomar com a entrada "=" tendo visto um *string* dedutível para L.

Os reconhecedores LR canônico e LALR conseguem reconhecer esta gramática.

··· Considerações sobre Gramáticas ¬ SLR(1)

• **No método SLR, o estado i chama por uma redução Se [A → α•] ∈ I_i e a ∈ FOLLOW(A)**

então ACTION[i, a] = "reduz A → α"

• **Contudo em algumas situações, suponha que:**
  – **o prefixo viável $\beta\alpha$ é o "string" corrente na pilha (α no topo)**
  – **aδ o trecho do fluxo de entrada ainda não lido**

  então é possível que βA não possa ser seguido por aδ em uma forma sentencial mais à direita. Ou seja, βAaδ não pode ser uma forma sentencial da gramática dada.

PORTANTO a ∈ FOLLOW(A) não garante que A possa ser seguido por a em qualquer contexto.

Reconsidere o Exemplo Anterior (¬ SLR(1))

$S' \to$ S
S $\to$ L = R
S $\to$ R
L $\to$ *R
L $\to$ **id**
R $\to$ L

FOLLOW(R) = { $, = }

States	Action			Goto
	...	=	...	...
	...		...	...
2	...	S6/r5	...	...
	...		...	...

Não há nenhuma forma sentencial nesta gramática que comece com $\mathbf{R} = \cdots$.
Note que **R** pode ser seguido de "=" somente se estiver precedido de "*":

$S' \Longrightarrow$ L = R $\Longrightarrow$ * R = R

$S' \overset{*}{\Longrightarrow}$ R = R (**OK**)
↑
L
$S' \overset{*}{\not\Longrightarrow}$ R = R

Então o estado 2 que corresponde somente ao prefixo viável L, não deve efetuar a redução deste L para R.
• **PROBLEMA:** O reconhecedor SLR nem sempre se lembra do contexto à esquerda do *handle*.

Exemplo de Gramática SLR(1)

Definição: Uma gramática G é dita ser SLR(1) se cada entrada da tabela ACTION contiver no máximo um valor.

	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r4	r4	r4		
5			r2		
6	r3	r3	r3		

(1) $S' \to$ S
(2) S $\to$ CC — FOLLOW(S') = {$}
(3) C $\to$ cC — FOLLOW(S) = {$}
(4) C $\to$ d — FOLLOW(C) = {c,d,$}

I_0: $S' \to \bullet$S $\xrightarrow{S} I_1$
S $\to \bullet$CC $\xrightarrow{C} I_2$
C $\to \bullet$cC $\xrightarrow{c} I_3$
C $\to \bullet$d $\xrightarrow{d} I_4$

I_1: $S' \to$ S$\bullet$

I_2: S $\to$ C$\bullet$C $\xrightarrow{C} I_5$
C $\to \bullet$cC $\xrightarrow{c} I_3$
C $\to \bullet$d $\xrightarrow{d} I_4$

I_3: C $\to$ c$\bullet$C $\xrightarrow{C} I_6$
C $\to \bullet$cC $\xrightarrow{c} I_3$
C $\to \bullet$d $\xrightarrow{d} I_4$

I_4: C $\to$ d$\bullet$

I_5: S $\to$ CC$\bullet$

I_6: C $\to$ cC$\bullet$

Outro Exemplo de Gramática ¬ SLR(K)

1 S $\to$ "("X
2 S $\to$ E "]"
3 S $\to$ F ")"
4 X $\to$ E ")"
5 X $\to$ F "]"
6 E $\to$ A
7 F $\to$ A
8 A $\to$ *empty*

{ { I0: S' $\to \bullet$ S $\xrightarrow{S}$ I1
S $\to \bullet$ (X $\xrightarrow{(}$ I2
S $\to \bullet$ E] $\xrightarrow{E}$ I3
S $\to \bullet$ F) $\xrightarrow{F}$ I4
E $\to \bullet$ A $\xrightarrow{A}$ I5
F $\to \bullet$ A $\xrightarrow{A}$ I5
A $\to \bullet$ }

{ I1: S' $\to$ S $\bullet$ }

{ I2: S $\to$ ($\bullet$ X $\xrightarrow{X}$ I6
X $\to \bullet$ E) $\xrightarrow{E}$ I7
X $\to \bullet$ F] $\xrightarrow{F}$ I8
E $\to \bullet$ A $\xrightarrow{A}$ I5
F $\to \bullet$ A $\xrightarrow{A}$ I5
A $\to \bullet$ }

{ I3: S $\to$ E $\bullet$] $\xrightarrow{]}$ I9 }

{ I4: S $\to$ F $\bullet$) $\xrightarrow{)}$ I10 }

{ I5: E $\to$ A $\bullet$
F $\to$ A $\bullet$ }

{ I6: S $\to$ (X $\bullet$ }

{ I7: X $\to$ E $\bullet$) $\xrightarrow{)}$ I11 }

{ I8: X $\to$ F $\bullet$] $\xrightarrow{]}$ I12 }

{ I9: S $\to$ E] $\bullet$ }

{ I10: S $\to$ F) $\bullet$ }

{ I11: X $\to$ E) $\bullet$ }

{ I12: X $\to$ F] $\bullet$ } }

··· Outro Exemplo de Gramática ¬ SLR(K)

1 S $\to$ "("X
2 S $\to$ E "]"
3 S $\to$ F ")"
4 X $\to$ E ")"
5 X $\to$ F "]"
6 E $\to$ A
7 F $\to$ A
8 A $\to$ *empty*

First(S) = {(,],) } | **Follow(S) = { $ }**
First(X) = {),] } | **Follow(X) = { $ }**
First(E) = { *empty* } | **Follow(E) = {],) }**
First(F) = { *empty* } | **Follow(F) = {),] }**
First(A) = { *empty* } | **Follow(A) = {),] }**

	(	)	]	$	S	X	F	E	A
0	**s2**	**r8**	**r8**		**1**		**4**	**3**	**5**
1				**acc**					
2		**r8**	**r8**			**6**	**8**	**7**	**5**
3			**s9**						
4		**s10**							
5		**r6/r7**	**r7/r6**						
6				**r1**					
7		**s11**							
8			**s12**						
9				**r2**					
10				**r3**					
11				**r4**					
12				**r5**					

Prefixos Viáveis

Prefixos viáveis são os conjuntos de prefixos de uma forma sentencial a direita que podem aparecer na pilha de um reconhecedor "shift-reduce".

··· Prefixos Viáveis

Itens Válidos

- **Prefixos viáveis são os conjuntos de prefixos de uma forma sentencial a direita que podem aparecer na pilha de um reconhecedor "shift-reduce".**
- **Dizemos que o item $A \to \beta_1 \bullet \beta_2$ é válido para o *prefixo viável* $\alpha\beta_1$, se existir uma derivação $S' \stackrel{*}{\Longrightarrow}_{rm} \alpha A w \stackrel{*}{\Longrightarrow}_{rm} \alpha\beta_1\beta_2 w$.**
- **Em geral, um item será válido para muitos prefixos viáveis. O fato de A $\to \beta_1 \bullet \beta_2$ ser válido para $\alpha\beta_1$ nos diz muito a respeito de empilhar ou reduzir ao encontrarmos $\alpha\beta_1$ na pilha do analisador sintático.**

··· Prefixos Viáveis

··· Itens Válidos

- **Se $\beta_2 \neq \mathcal{E}$ isto sugere que ainda não empilhamos o *handle* e o próximo movimento agora é empilhar.**
- **Se $\beta_2 = \mathcal{E}$, então tudo indica que $A \to \beta_1$ é o *handle* e podemos reduzir segundo esta produção.**
- **Dois itens válidos podem nos dizer duas diferentes coisas sobre o mesmo prefixo viável, gerando um conflito, que pode ser ser resolvido examinando-se o próximo símbolo.**

··· Prefixos Viáveis

• Claramente a cadeia **E + T *** é um prefixo viável da gramática de expressão analisada.

O autômato estará no estado I após ter lido **E + T ***.

O estado I_7 contém os itens: $T \to T * \bullet F$
$F \to \bullet(E)$
$F \to \bullet id$

que são precisamente itens válidos para **E + T ***.

Para ver isto, considere as seguintes derivações mais à direita:

E' ⇒ E	E' ⇒ E	E' ⇒ E
⇒ E + T	⇒ E + T	⇒ E + T
⇒ E + T * F	⇒ E + T * F	⇒ E + T * F
	⇒ E + T * (E)	⇒ E + T * **id**

• A primeira mostra a validade de T → T * • F.
• A segunda mostra a validade de F → T * • (E).
• A terceira mostra a validade de F → T * • **id** para o prefixo viável **E + T ***.

4.7 ANALISADORES SINTÁTICOS LR MAIS PODEROSOS

Analisadores Sintáticos LR mais Poderosos

As técnicas de análise LR anteriores são estendidas incorporando nos itens o primeiro símbolo da entrada, ainda não lido, símbolo de *lookahead*.

Existem dois métodos LR com esta característica:

1. O método "LR canônico", ou apenas "LR", faz uso do(s) símbolo(s) *lookahead*. Esse método usa uma tabela construída a partir do conjunto de itens denominados itens $LR(1)$.

2. O método "Look Ahead LR", ou "LALR", cuja tabela é construída a partir dos conjuntos de itens $LR(0)$, e possui muito menos estados que os analisadores sintáticos típicos, baseados no conjunto canônico de itens $LR(1)$.

Itens LR(1) Canônicos

Definição. Um item LR(1) é um par cujo primeiro componente é uma produção com "•" no seu lado direito e cujo segundo componente é um terminal da gramática ou $.

$$[A \rightarrow \alpha \bullet \beta \ , \ a]$$

Definição Formal. Um item LR(1) $[A \rightarrow \alpha \bullet \beta \ , \ a]$ é ***válido*** para um prefixo viável γ se existe uma derivação: $S \stackrel{*}{\Longrightarrow}_{rm} \delta Aw \Longrightarrow_{rm} \delta\alpha\beta w$, onde:

(1) – $\gamma = \delta\alpha$, e
(2) – "a" é o primeiro símbolo de "w" ou "w" é $\mathcal{E}$
e "a" é $.

··· Itens LR(1) Canônicos

Considere a gramática: S → BB
B → aB | b

$S \Longrightarrow_{rm} BB \Longrightarrow_{rm} BaB \Longrightarrow_{rm} Bab \Longrightarrow_{rm} aBab \Longrightarrow_{rm} aaBab \Longrightarrow_{rm} aaaBab$

- Exemplo 1:
  - item [B → a.B, a] é válido para o prefixo viável γ = aaa fazendo:
    δ = aa,
    A = B,
    w = ab,
    α = a,
    β = B de acordo com a definição mostrada.

- Exemplo 2: $S \stackrel{*}{\Longrightarrow}_{rm} BaB \Longrightarrow_{rm} BaaB$.

  O item [B → a.B, $] é válido para o prefixo viável Baa.

Mais Conceitos sobre o Reconhecedor LR(1)

- O 1 de LR(1) refere-se ao comprimento do segundo componente de um item LR(1).
- O segundo componente é chamado de *símbolo de lookahead* e o primeiro é chamado de *núcleo* do item.
- Se $\beta \neq \mathcal{E}$ então o "a" de [A $\rightarrow \alpha \bullet \beta$, a] não tem efeito algum.
- Itens da forma [A $\rightarrow \alpha\bullet$, a] indicam que a redução A $\rightarrow \alpha$ só poderá ser efetuada se o próximo símbolo da entrada for "a".
- O "a" de [A $\rightarrow \alpha\bullet$, a] pertence ao FOLLOW(A), ou seja, "a" sempre será um subconjunto do FOLLOW(A).
- A idéia básica é separar estados do reconhecedor de modo a fazer com que reduções A $\rightarrow \alpha$ não sejam permitidas em certas situações, mesmo que o próximo símbolo da entrada pertença ao FOLLOW(A).

Construção de Conjuntos de Itens LR(1)

Algoritmo 4.53: Construção dos conjuntos de itens LR(1).

ENTRADA: Uma gramática estendida G'.

SAÍDA: Os conjuntos de itens $LR(1)$ que são o conjunto de itens válidos para um ou mais prefixos viáveis de G'.

MÉTODO: Execute as funções $CLOSURE$ e $GOTO$ e a *rotina principal itens* mostradas a seguir para construir os conjuntos de itens.

··· Algoritmo 4.53: Construção de Conjuntos de Itens LR(1)

Algoritmo para Cálculo do $CLOSURE$(I)

```
SetOfItems CLOSURE(I) {
      repeat
            for (cada item [A → α•Bβ , a] em I)
                  for ( cada produção B → γ em G' )
                        for (cada terminal b em FIRST(βa))
                              tais que [B → •γ , b] ∉ I
                              inclua [B → •γ, b] no conjunto I;
      until não conseguir incluir mais itens em I.
      return I
}
```


··· Algoritmo 4.53: Construção de Conjuntos de Itens LR(1)

Algoritmo para cálculo de $GOTO$(I,X)

```
SetOfItems GOTO(I,X) {
      inicializa J para ser o conjunto vazio;
      for ( cada item [A → α·Xβ, a] em I)
            adicione o item [A → αX·β, a] ao conjunto de J;
      return CLOSURE(J);
}
```

Algoritmo 4.53: Construção de Conjuntos de Itens LR(1)

```
void itens(G') {
    C := { CLOSURE({[S' → •S, $]})}
    repeat
        for (cada conjunto I ∈ C)
            for (cada símbolo X ∈ V_N ∪ V_T)
                if ( GOTO(I,X) não é vazio e não está em C)
                    inclua GOTO(I,X) em C
    until que não haja mais conjuntos de itens para
        serem incluídos em C.
}
```


Como Construir os Conjuntos de Itens LR(1)

Seja $S' \rightarrow$ S; S $\rightarrow$ CC; C $\rightarrow$ cC; C $\rightarrow$ d

- Inicialmente computamos o CLOSURE de $\{[S' \rightarrow \bullet S, \$]\}$.
- Casamos o item $[S' \rightarrow \bullet S, \$]$ com o item $[A \rightarrow \alpha \bullet B \beta, a]$ da função *CLOSURE*: A = S', $\alpha = \mathcal{E}$, B = S, $\beta = \mathcal{E}$e a = \$.
- Função *CLOSURE* manda acrescentar $[B \rightarrow \bullet\gamma, b]$ para cada produção $B \rightarrow \gamma$ e terminal b em FIRST(βa).

Neste exemplo, $B \rightarrow \gamma$ = S $\rightarrow$ CC e como $\beta = \mathcal{E}$ e a = \$, b *só pode ser* \$.

Portanto acrescentamos [S $\rightarrow$ •CC, \$] .

Como Construir os Conjuntos de Itens LR(1)

- Continuamos computando o *CLOSURE* acrescentando todos os itens

[C $\rightarrow$ •γ, b] para *b* no FIRST(C\$), ou seja, casando [S $\rightarrow$ •CC, \$] com [A $\rightarrow \alpha \bullet B \beta$, a] temos: A = S, $\alpha = \mathcal{E}$, B = C, β = C e a = \$.

Como C não deriva ***string*** vazio, FIRST(C\$) = FIRST(C).

Como C contém terminais "c" e "d" acrescentamos os itens [C $\rightarrow$ •cC, c], [C $\rightarrow$ •cC, d], [C $\rightarrow$ •d, c] e C $\rightarrow$ •d, d].

Como nenhum item novo contém um não-terminal imediatamente à direita de •, completamos o primeiro conjunto de LR(1) itens.

Exemplo: Reconhecedor LR(1)

(1) $S' \rightarrow$ S
(2) S $\rightarrow$ CC
(3) C $\rightarrow$ cC
(4) C $\rightarrow$ d

FOLLOW(S') = {\$}
FOLLOW(S) = {\$}
FOLLOW(C) = {c,d,\$}

FIRST(C)= {c,d}
FIRST(S)= {c,d}
FIRST(S')= {c,d}

I_0: $S' \rightarrow \bullet$S,\$ $\xrightarrow{S}$ I_1
S $\rightarrow \bullet$CC,\$ $\xrightarrow{C}$ I_2
C $\rightarrow \bullet$cC,c/d $\xrightarrow{c}$ I_3
C $\rightarrow \bullet$d,c/d $\xrightarrow{d}$ I_4

I_1: $S' \rightarrow$ S•,\$

I_2: S $\rightarrow$ C•C,\$ $\xrightarrow{C}$ I_5
C $\rightarrow \bullet$cC,\$ $\xrightarrow{c}$ I_6
C $\rightarrow \bullet$d, \$ $\xrightarrow{d}$ I_7

I_3: C $\rightarrow$ c•C,c/d $\xrightarrow{C}$ I_8 ⇐
C $\rightarrow \bullet$cC,c/d $\xrightarrow{c}$ I_3
C $\rightarrow \bullet$d,c/d $\xrightarrow{d}$ I_4

I_4: C $\rightarrow$ d•,c/d

I_5: S $\rightarrow$ CC•,\$

I_6: C $\rightarrow$ c•C,\$ $\xrightarrow{C}$ I_9 ⇐
C $\rightarrow \bullet$cC,\$ $\xrightarrow{c}$ I_6
C $\rightarrow \bullet$d,\$ $\xrightarrow{d}$ I_7

I_7: C $\rightarrow$ d•,\$

I_8: C $\rightarrow$ cC•,c/d

I_9: C $\rightarrow$ cC•\$

Exemplo do Grafo GOTO para a Gramática:

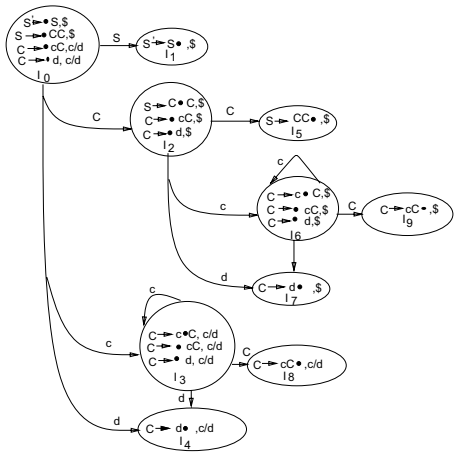
(1) $S' \rightarrow$ **S**

(2) **S** $\rightarrow$ **CC**

(3) **C** $\rightarrow$ **cC**

(4) **C** $\rightarrow$ **d**

FOLLOW(S') = {**$**}
FOLLOW(**S**) = {**$**}
FOLLOW(**C**) = {**c,d,$**}



Exemplo da Tabela Gerada para a Gramática:

(1) **S** $\rightarrow$ **CC**
(2) **C** $\rightarrow$ **cC**
(3) **C** $\rightarrow$ **d**

Estados	ACTIONS			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Outro Exemplo:

Coleção de Conjuntos LR(1) Canônico

(0) S' → S
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → **id**
(5) R → L

FOLLOW(S) = { $ }
FOLLOW(R) = { $, = }
FOLLOW(L) = { $, = }

I_0: $S' \rightarrow$ •S,$ $\xrightarrow{S} I_1$
S → •L=R,$ $\xrightarrow{L} I_2$
S → •R,$ $\xrightarrow{R} I_3$
L → •*R,=/$ $\xrightarrow{*} I_4$
L → •id,=/$ $\xrightarrow{id} I_5$
R → •L,$ $\xrightarrow{L} I_2$

I_1: $S' \rightarrow$ S•,$

I_2: S → L•=R,$ $\xrightarrow{=} I_6$
R → L•,$

I_3: S → R•,$

I_4 L → *•R,=/$ $\xrightarrow{R} I_7$
R → •L,=/$ $\xrightarrow{L} I_8$
L → •*R,=/$ $\xrightarrow{*} I_4$
L → •id,=/$ $\xrightarrow{id} I_5$

I_5: L → id•,=/$

I_6: S → L=•R,$ $\xrightarrow{R} I_9$
R → •L,$ $\xrightarrow{L} I_{10}$
L → •*R,$ $\xrightarrow{*} 11$
L → •id,$ $\xrightarrow{id} 12$

I_7: L → *R•, =/$

I_8: R → L•,=/$

I_9: S → L=R•,$

I_{10}: R → L•,$

I_{11}: L → *•R,$ $\xrightarrow{R} I_{13}$
R → •L,$ $\xrightarrow{L} I_{10}$
L → •*R,$ $\xrightarrow{*} I_{11}$
L → •id,$ $\xrightarrow{id} I_{12}$

I_{12}: L → id•,$
I_{13}: L → *R•, $

Construção da Tabela do Reconhecedor LR(1)

Algoritmo 4.56: **Construção das tabelas LR canônicas de análise.**

ENTRADA: **Uma gramática estendida G'.**

SAÍDA: **As funções $ACTION$ e $GOTO$ da tabela LR canônica de análise para G'.**

MÉTODO:

1. **construa C = { I_0, I_1, $\cdots$, I_n } conjunto de conjuntos de itens LR(1)**

2. **Associe um estado i a cada um dos conjuntos I_i e suponha a,b $\in V_T$**

··· Construção da Tabela do Reconhecedor LR(1) - Algoritmo 4.56:

3. As ações do reconhecedor para o estado i são determinadas da seguinte forma:

   3.1. se [A $\rightarrow \alpha\bullet a\beta$,b] $\in I_i$ e GOTO(I_i,a) = Ij então faça ACTION[i,a] = "shift j". "a" deve ser terminal.

   3.2. se [A $\rightarrow \alpha\bullet$, a] $\in I_i$ A $\neq S'$ então faça ACTION[i,a] = "reduce A $\rightarrow \alpha$".

   3.3. se [S' $\rightarrow$ S•, $] $\in I_i$ então faça ACTION[i,$] = "accept".

   3.4. Se conflitos aparecerem devido as regras acima, a gramática não é LR(1) e o algoritmo falha.

··· Construção da Tabela do Reconhecedor LR(1) - Algoritmo 4.56:

4. As transições GOTO para o estado i são determinadas da seguinte forma:

   4.1. se GOTO[I_i,A] = I_j então GOTO[i,A] = j.

   4.2. O estado inicial corresponde ao conjunto de itens LR(1) que contém o item [S' $\rightarrow$ •S, $] .

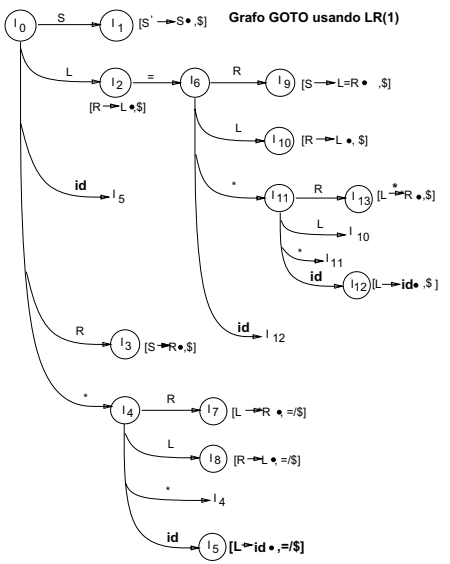
   4.3. Posições vazias em ACTION e GOTO denotam erros.

Tabela do Reconhecedor LR(1)

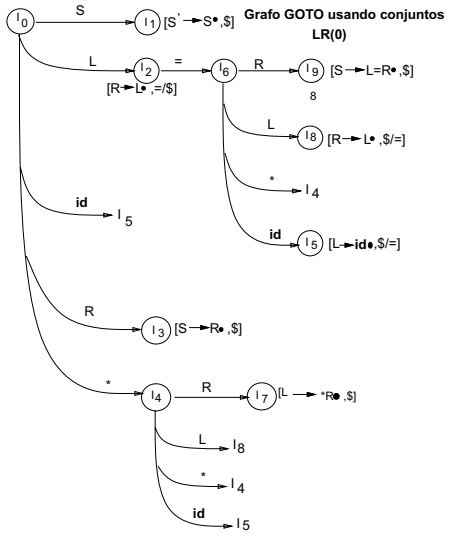
(0) S' $\rightarrow$ S
(1) S $\rightarrow$ L = R
(2) S $\rightarrow$ R
(3) L $\rightarrow$ *R
(4) L $\rightarrow$ **id**
(5) R $\rightarrow$ L

Estado	Ação				Goto		
	id	*	=	$	S	L	R
0	S5	S4			1	2	3
1				acc			
2			S6	r5			
3				r2			
4	S5	S4				8	7
5			r4	r4			
6	S12	S11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	S12	S11				10	13
12				r4			
13				r3			

Autômato de Estados Finitos Determinístico



Autômato de Estados Finitos Determinístico



Outro Exemplo de Gramática LR(1)

1 S → "("X
2 S → E "]"
3 S → F ")"
4 X → E ")"
5 X → F "]"
6 E → A
7 F → A
8 A → *empty*

{ { I0: S' → • S, $ $\Rightarrow^{S}$ I1
S → • (X, $ $\Rightarrow^{(}$ I2
S → • E], $ $\Rightarrow^{E}$ I3
S → • F), $ $\Rightarrow^{F}$ I4
E → • A,] $\Rightarrow^{A}$ I5
F → • A,) $\Rightarrow^{A}$ I5
A → • }
{ I1: S' → S •, $ }
{ I2: S → (• X, $ $\Rightarrow^{X}$ I6
X → • E),$ $\Rightarrow^{E}$ I7
X → • F], $ $\Rightarrow^{F}$ I8
E → • A,) $\Rightarrow^{A}$ I13
F → • A,] $\Rightarrow^{A}$ I13
A → • }
{ I3: S → E •], $ $\Rightarrow^{]}$ I9 }

{ I4: S → F •), $ $\Rightarrow^{)}$ I10 }
{ I5: E → A •,]
F → A •,) }
{ I6: S → (X •, $ }
{ I7: X → E •), $ $\Rightarrow^{)}$ I11 }
{ I8: X → F •], $ $\Rightarrow^{]}$ I12 }
{ I9: S → E] •, $ }
{ I10: S → F) •, $ }
{ I11: X → E) •, $ }
{ I12: X → F] •, $ }
{ I13: E → A •,)
F → A •,] } }

· · · Outro Exemplo de Gramática LR(1)

1 S → "("X
2 S → E "]"
3 S → F ")"
4 X → E ")"
5 X → F "]"
6 E → A
7 F → A
8 A → *empty*

First(S) = {(,],) } | Follow(S) = { $ }
First(X) = {),] } | Follow(X) = { $ }
First(E) = { *empty* } | Follow(E) = {],) }
First(F) = { *empty* } | Follow(F) = {),] }
First(A) = { *empty* } | Follow(A) = {),] }

	(	)	]	$	S	X	F	E	A
0	s2	r8	r8		1		4	3	5
1				acc					
2		r8	r8			6	8	7	13
3			s9						
4		s10							
5		r7	r6						
6				r1					
7		s11							
8			s12						
9				r2					
10				r3					
11				r4					
12				r5					
13		r6	r7						

Considerações sobre Gramáticas

- Uma gramática G é LR(1) se cada entrada da tabela ACTION contiver no máximo um valor.
- Toda gramática SLR(1) é LR(1).
- Existem gramáticas LR(1) que não são SLR(1).
- Em geral, o número de estados do reconhecedor LR(1) é muito maior que o do reconhecedor SLR(1) correspondente.

Algol 60: SLR(1): centenas de estados.
LR(1): milhares de estados.

RECONHECEDORES LALR(1)

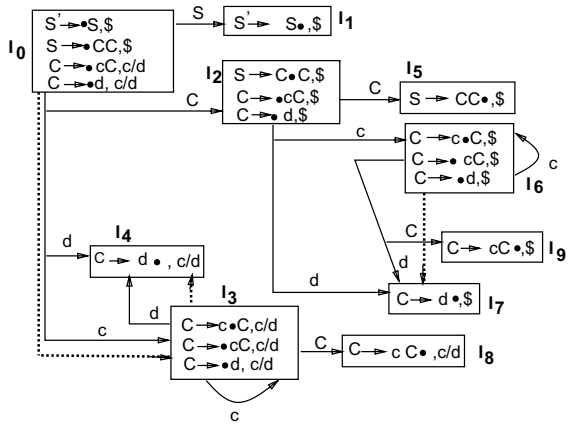
Reconhecedores LALR(1)

Considere a gramática:

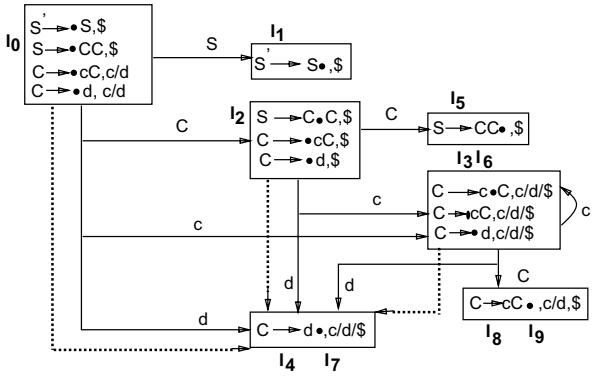
$$S' \rightarrow \mathbf{S}$$
$$\mathbf{C} \rightarrow \mathbf{cC}$$
$$\mathbf{S} \rightarrow \mathbf{CC}$$
$$\mathbf{C} \rightarrow \mathbf{d}.$$

Entrada: c*dc*d$

Conjuntos LR(1) e GOTOs:



Unindo Conjuntos com Mesmo Núcleo



Estado	Ação			Goto	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Um Método Simples para a Construção da Tabela LALR - Algoritmo 4.59

ENTRADA: Uma gramática estendida G'.

SAÍDA: Funções $ACTION$ e $GOTO$ da tabela $LALR$ de análise para G'.

MÉTODO:

1. Construa $C = \{I_0, I_1, \ldots, I_n\}$, a coleção de conjuntos de itens LR(1).

2. Para todos os núcleos presentes em conjuntos de itens $LR(1)$ determine aqueles conjuntos que tenham o mesmo núcleo, e os substitua pela sua união.

... Um Método Simples para a Construção da Tabela LALR - Algoritmo 4.59

3. **Considere que $C' = \{J_0, J_1, \ldots, J_m\}$ seja o conjunto de itens $LR(1)$ resultante.**

   As ações de análise para o estado i são construídas a partir de J_i, da mesma maneira que no Algoritmo 4.56.

   Se houver um conflito de ação de análise, o algoritmo deixa de produzir um reconhecedor sintático, e a gramática é considerada como não sendo LALR(1).

... Um Método Simples para a Construção da Tabela LALR - Algoritmo 4.59

4. **A tabela $GOTO$ é construída da forma a seguir.**

   Se J é a união de um ou mais conjuntos de itens LR(1), ou seja, $J = I_1 \cup I_2 \cup \cdots \cup I_k$,
   então os núcleos de GOTO(I_1 , X), GOTO(I_2 , X) , ... , GOTO(I_k , X) são os mesmos, desde que $I_1, I_2, \ldots, I_k$ possuam todos o mesmo núcleo.

   Considere que K seja a união de todos os conjuntos de itens tendo o mesmo núcleo que GOTO(I_1 , X).
   Então, GOTO(J , X)=K.

Detecção de Erros (LR e LALR)

- **LR – detecta erro logo**
- **LALR – prossegue efetuando algumas reduções antes de detectar o erro.**

- **Exemplo: entrada $\underline{ccd\$}$ (não está na linguagem)**

4	←
d	
3	
c	
3	
c	
0	

pilha LR

Estado 4 descobre o erro porque $ é o próximo símbolo da entrada e este estado possui uma ação de erro sobre $.

Detecção de Erros (LALR)

47	←
d	
36	
c	
36	
c	
0	Estado 47 com entrada $ ≡ *reduce* C → d, então
89	←
C	
36	
c	
36	
c	
0	Estado 89 com entrada $ ≡ *reduce* C → cC, então
89	←
C	
36	
c	
0	Após redução similar, a pilha passa a ter a configuração:
2	←
C	
0	Finalmente estado 2 possui uma ação de erro com a entrada $.

Tabela LALR(1) – Método Simples

Dada a gramática: (0) $S' \to S$
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → **id**
(5) R → L

Note que : o conflito *shift-reduce* existente no SLR desapareceu do estado 2 presente no LALR. Isto porque somente *lookahead* $ está associado a R → L•, portanto não há conflito com a ação *shift* sobre "=" gerada pelo item S → L•=R no estado 2.

I_0:	$S' \to \bullet$S,$	$\xrightarrow{S} I_1$
	S → •L=R,$	$\xrightarrow{L} I_2$
	S → •R,$	$\xrightarrow{R} I_3$
	L → •*R,=/$	$\xrightarrow{*} I_4 I_{11}$
	L → •**id**,=/$	$\xrightarrow{\text{id}} I_5 I_{12}$
	R → •L,$	$\xrightarrow{L} I_2$
I_1:	$S' \to$ S•,$	
I_2:	S → L•=R,$	$\xrightarrow{=} I_6$
	R → L•,$	
I_3:	S → R•,$	
I_4:I_{11}:	L → *•R,=/$	$\xrightarrow{R} I_7 I_{13}$
	R → •L,=/$	$\xrightarrow{L} I_8 I_{10}$
	L → •*R,=/$	$\xrightarrow{*} I_4 I_{11}$
	L → •**id**,=/$	$\xrightarrow{\text{id}} I_5 I_{12}$
I_5:I_{12}:	L → **id**•,=/$	
I_8:I_{10}:	R → L•,=/$	
I_9:	S → L=R•, $	
I_7:I_{13}:	L → *R•, =/$	
I_6:	S → L=•R,$	$\xrightarrow{R} I_9$
	R → •L,$	$\xrightarrow{L} I_8 I_{10}$
	L → •*R,$	$\xrightarrow{*} I_4$
	L → •**id**,$	$\xrightarrow{\text{id}} I_5 I_{12}$

Mais sobre Reconhecedores LALR

- **A fusão de estados (LR → LALR) pode criar conflitos** *reduce-reduce*.
  I_i = { [A → **c**•,**d**], [B → **c**•,**e**]}
  I_j = { [A → **c**•,**e**], [B → **c**•,**d**]}
  ⇓
  I_{ij} = { [A → **c**•,**d/e**], [B → **c**•,**d/e**]}

- **A fusão não pode criar conflitos** *shift-reduce*:

  união: {[A → α•,a], [B → β•aδ,b], ···,}

  então algum membro contém:

  {[A → α•,a], [B → β•aδ,c], ···,}
  ↑ ↑

  Portanto o conflito já existia. A união de estados com o mesmo núcleo nunca pode gerar conflitos *shift-reduce* que não estavam presentes em um dos estados originais porque ações *shift* dependem somente do núcleo, não do *lookahead*.

Exemplo

Seja a gramática: $S' \to$ **S**
S → aAd | bBd | aBe | bAe
A → c
B → c

- **cadeias geradas: acd, ace, bcd e bce**

Após gerar o conjunto LR(1) temos:

- **Conjunto de itens válidos para o prefixo** *ac*: **{[A → c•, d], [B → c•, e]}**

- **Conjunto de itens válidos para o prefixo** *bc*: **{[A → c•, e], [B → c•, d]}**

Note que: Nenhum destes conjuntos geram conflitos e seus núcleos são iguais. Contudo após a união: [A → c•, d/e]
[B → c•, d/e]
conflitos *reduce-reduce* são gerados porque ambas as reduções para A → c e B → c são chamadas com a entrada "d" e "e".

Exemplo: Coleção de Conjuntos LR(1) Canônico

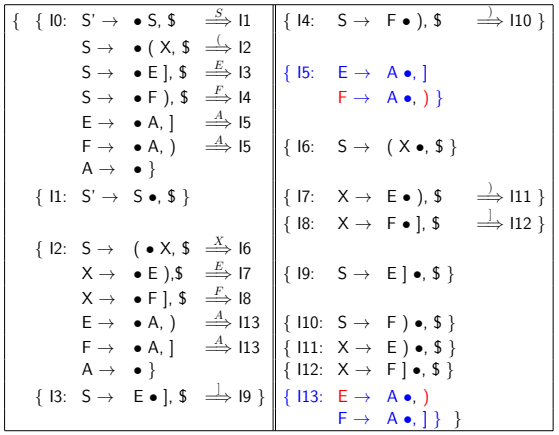
(0) S' → S
(1) S → aAd
(2) S → bBd
(3) S → aBe
(4) S → bAe
(5) A → c
(6) B → c

FOLLOW(S) = { $ }
FOLLOW(A) = { d, e }
FOLLOW(B) = { d, e }

{ I_0:	$S' \to \bullet$S,$	$\xrightarrow{S} I_1$
	S → •aAd,$	$\xrightarrow{a} I_2$
	S → •bBd,$	$\xrightarrow{b} I_3$
	S → •aBe,$	$\xrightarrow{a} I_2$
	S → •bAe,$	$\xrightarrow{b} I_3$
I_1:	$S' \to$ S•,$	
I_2:	S → a•Ad,$	$\xrightarrow{A} I_4$
	S → a•Be,$	$\xrightarrow{B} I_5$
	A → •c,d	$\xrightarrow{c} I_6$
	B → •c,e	$\xrightarrow{c} I_6$
I_3	S → b•Bd,$	$\xrightarrow{B} I_7$
	S → b•Ae,$	$\xrightarrow{A} I_8$
	B → •c,d	$\xrightarrow{c} I_9$
	A → •c,e	$\xrightarrow{c} I_9$
I_4:	S → aA•d,$	$\xrightarrow{d} I_{10}$
I_5:	S → aB•e,$	$\xrightarrow{e} I_{11}$
⋆ I_6:	A → c•,d	
	B → c•,e	
I_7:	S → bB•d,$	$\xrightarrow{d} I_{12}$
I_8:	S → bA•e,$	$\xrightarrow{e} I_{13}$
⋆ I_9:	B → c•,d	
	A → c•,e	
I_{10}	S → aAe•,$	
I_{11}:	S → aBe•,$	
I_{12}:	S → bBd•,$	
I_{13}:	S → bAe•, $ }	

Outro Exemplo de Gramática LR(1)

1 S → "(" X
2 S → E "]"
3 S → F ")"
4 X → E ")"
5 X → F "]"
6 E → A
7 F → A
8 A → *empty*



··· Outro Exemplo de Gramática LR(1)

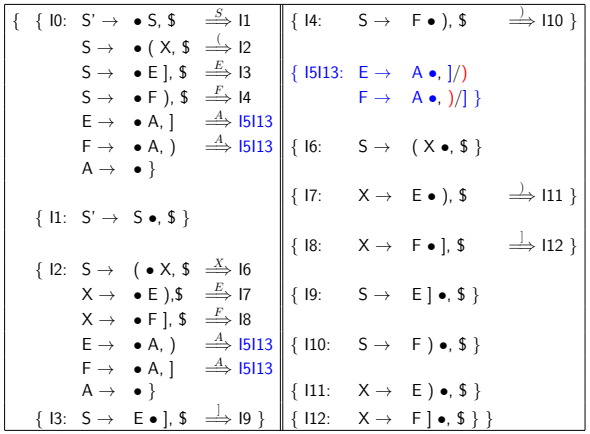
1 S → "(" X
2 S → E "]"
3 S → F ")"
4 X → E ")"
5 X → F "]"
6 E → A
7 F → A
8 A → *empty*

First(S) = {(,],) }	**Follow(S) = { $ }**
First(X) = {),] }	**Follow(X) = { $ }**
First(E) = { *empty* }	**Follow(E) = {],) }**
First(F) = { *empty* }	**Follow(F) = {),] }**
First(A) = { *empty* }	**Follow(A) = {),] }**

	(	)	]	$	S	X	F	E	A
0	**s2**	**r8**	**r8**		**1**		**4**	**3**	**5**
1				**acc**					
2		**r8**	**r8**			**6**	**8**	**7**	**13**
3			**s9**						
4		**s10**							
5		**r7**	**r6**						
6				**r1**					
7		**s11**							
8			**s12**						
9				**r2**					
10				**r3**					
11				**r4**					
12				**r5**					
13		**r6**	**r7**						

Outro Exemplo de Gramática (¬LALR(1))

1 S → "(" X
2 S → E "]"
3 S → F ")"
4 X → E ")"
5 X → F "]"
6 E → A
7 F → A
8 A → *empty*



··· Outro Exemplo de Gramática (¬LALR(1))

1 S → "(" X
2 S → E "]"
3 S → F ")"
4 X → E ")"
5 X → F "]"
6 E → A
7 F → A
8 A → *empty*

First(S) = {(,],) }	**Follow(S) = { $ }**
First(X) = {),] }	**Follow(X) = { $ }**
First(E) = { *empty* }	**Follow(E) = {],) }**
First(F) = { *empty* }	**Follow(F) = {),] }**
First(A) = { *empty* }	**Follow(A) = {),] }**

	(	)	]	$	S	X	F	E	A
0	**s2**	**r8**	**r8**		**1**		**4**	**3**	**513**
1				**acc**					
2		**r8**	**r8**			**6**	**8**	**7**	**513**
3			**s9**						
4		**s10**							
513		**r7/r6**	**r6/r7**						
6				**r1**					
7		**s11**							
8			**s12**						
9				**r2**					
10				**r3**					
11				**r4**					
12				**r513**					

Método Eficiente para Construção da Tabela LALR(1)

1. **Calcule o conjunto de conjuntos de itens LR(0).**

2. **Calcule os conjuntos *lookaheads* só para os itens [A → α•].**

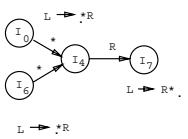
Note bem:

Na verdade, somente quando existirem conflitos no SLR(1) é que necessitamos calcular *lookaheads* LR(1). Nos estados em que não há conflitos, o FOLLOW(A) do SLR(1) pode ser usado.

Tabela LR(0)

Dada a gramática:

(0) $S' \rightarrow$ S
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → id
(5) R → L



Calculo I

I_0: $S' \rightarrow$ •S $\xrightarrow{S} I_1$		I_7: L → *R•
S → •L=R $\xrightarrow{L} I_2$		
S → •R $\xrightarrow{R} I_3$		I_8: R → L•
L → •*R $\xrightarrow{*} I_4$		
L → •id $\xrightarrow{id} I_5$		I_9: S → L=R•
R → •L $\xrightarrow{L} I_2$		
		I_5: L → id•
I_1: $S' \rightarrow$ S•		
		I_6: S → L=•R $\xrightarrow{R} I_9$
I_2: S → L•=R $\xrightarrow{=} I_6$		R → •L $\xrightarrow{L} I_8$
R → L•		
		L → •*R $\xrightarrow{*} I_4$
I_3: S → R•		L → •id $\xrightarrow{id} I_5$
I_4: L → *•R $\xrightarrow{R} I_7$		
R → •L $\xrightarrow{L} I_8$		
L → •*R $\xrightarrow{*} I_4$		
L → •id $\xrightarrow{id} I_5$		

Tabela LR(0) com Lookaheads somente em Itens A→α•

Dada a gramática:
(0) $S' \rightarrow$ S
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → id
(5) R → L

I_0: $S' \rightarrow$ •S $\xrightarrow{S} I_1$		I_7: L → *R•, =/$
S → •L=R $\xrightarrow{L} I_2$		
S → •R $\xrightarrow{R} I_3$		I_8: R → L•,=/$
L → •*R $\xrightarrow{*} I_4$		
L → •id $\xrightarrow{id} I_5$		I_9: S → L=R•, $
R → •L $\xrightarrow{L} I_2$		
		I_5: L → id•,=/$
I_1: $S' \rightarrow$ S•,$		
		I_6: S → L=•R $\xrightarrow{R} I_9$
I_2: S → L•=R $\xrightarrow{=} I_6$		R → •L $\xrightarrow{L} I_8$
R → L•,$		
		L → •*R $\xrightarrow{*} I_4$
I_3: S → R•,$		
I_4: L → *•R $\xrightarrow{R} I_7$		
R → •L $\xrightarrow{L} I_8$		
L → •*R $\xrightarrow{*} I_4$		
L → •id $\xrightarrow{id} I_5$		

Exemplo: Coleção de Conjuntos LR(1) Canônico

(0) S' → S :
(1) S → L = R
(2) S → R
(3) L → *R
(4) L → id
(5) R → L

FOLLOW(S) = { $ }
FOLLOW(R) = { $, = }
FOLLOW(L) = { $, = }

I_0: $S' \rightarrow$ •S,$ $\xrightarrow{S} I_1$		I_6: S → L=•R,$ $\xrightarrow{R} I_9$
S → •L=R,$ $\xrightarrow{L} I_2$		R → •L,$ $\xrightarrow{L} I_{10}$
S → •R,$ $\xrightarrow{R} I_3$		L → •*R,$ $\xrightarrow{*} 11$
L → •*R,=/$ $\xrightarrow{*} I_4$		L → •id,$ $\xrightarrow{id} 12$
L → •id,=/$ $\xrightarrow{id} I_5$		
R → •L,$ $\xrightarrow{L} I_2$		I_7: L → *R•, =/$
I_1: $S' \rightarrow$ S•,$		I_8: R → L•,=/$
I_2: S → L•=R,$ $\xrightarrow{=} I_6$		I_9: S → L=R•,$
R → L•,$		
		I_{10}: R → L•,$
I_3: S → R•,$		
		I_{11}: L → *•R,$ $\xrightarrow{R} I_{13}$
I_4 L → *•R,=/$ $\xrightarrow{R} I_7$		R → •L,$ $\xrightarrow{L} I_{10}$
R → •L,=/$ $\xrightarrow{L} I_8$		L → •*R,$ $\xrightarrow{*} I_{11}$
L → •*R,=/$ $\xrightarrow{*} I_4$		L → •id,$ $\xrightarrow{id} I_{12}$
L → •id,=/$ $\xrightarrow{id} I_5$		
		I_{12}: L → id•,$
I_5: L → id•,=/$		I_{13}: L → *R•, $

Construção eficiente de tabelas de análise LALR

Modificações podem ser feitas no Algoritmo 4.59 para evitar a construção da coleção completa dos conjuntos de itens LR(1) para a criação de uma tabela de análise LALR(1).

- Representar qualquer conjunto de itens I LR(0) ou LR(1) por sua base (kernel), ou seja, por aqueles itens que são o item inicial — $[S' \to \cdot S]$ ou $[S' \to \cdot S, \$]$ — ou todos os itens cujos pontos não estão mais à esquerda nos corpos das produções.
- Construir as bases dos itens LALR(1) a partir das bases de itens LR(0) pelo processo de propagação e geração espontânea de lookaheads.
- Dada as bases LALR(1), é possível gerar a tabela de análise LALR(1) fazendo o fechamento de cada base via a função CLOSURE, e depois calculando as entradas da tabela via o Algoritmo 4.56, como se os conjuntos de itens LALR(1) fossem conjuntos de itens LR(1) canônicos.

Construção eficiente de tabelas de análise LALR- Exemplo

$$
\begin{array}{rcl}
S' & \to & S \\
S & \to & L = R \mid R \\
L & \to & *R \mid \mathbf{id} \\
R & \to & L
\end{array}
$$

Bases dos conjuntos de itens LR(0) para a gramática dada

I_0: $S' \to \cdot S$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$, $R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$

Construção eficiente de tabelas de análise LALR- Exemplo

Junta-se os lookaheads apropriados às bases dos itens LR(0), a fim de criar as bases dos conjuntos de itens LALR(1).

Existem duas formas para associar um lookahead b a um item LR(0) $B \to \gamma\cdot\delta$ em algum conjunto de itens LALR(1) J:

1. Existe um conjunto de itens I, com um item base $A \to \alpha\cdot\beta, a$, e $J = GOTO(I, X)$, e a construção de
$$GOTO(CLOSURE(\{[A \to \alpha\cdot\beta, a]\}), X)$$
contém $[B \to \gamma\cdot\delta, b]$, independentemente de a.

   Diz-se que esse lookahead b é gerado espontaneamente por $B \to \gamma\cdot\delta$.

   Caso especial: o lookahead $\$$ é gerado espontaneamente para o item $S' \to \cdot S$ no conjunto inicial de itens.

Construção eficiente de tabelas de análise LALR- Exemplo

2. Tudo é como em (1), mas $a = b$,
   e $GOTO(CLOSURE(\{[A \to \alpha\cdot\beta, b]\}), X)$ contém $[B \to \gamma\cdot\delta, b]$ somente porque $A \to \alpha\cdot\beta$ tem b como um de seus lookaheads associados.

   Nesse caso, diz-se que os lookaheads se propagam a partir de $A \to \alpha\cdot\beta$ na base de I para $B \to \gamma\cdot\delta$ na base de J.

   Observação: propagação não depende de um símbolo lookahead particular; ou todos os lookaheads se propagam de um item para outro, ou nenhum se propaga.

Construção eficiente de tabelas de análise LALR- Algoritmo

Algoritmo 4.62: Determina lookaheads.

ENTRADA: A base K de um conjunto I de itens LR(0) e um símbolo da gramática X.

SAÍDA: Os lookaheads gerados espontaneamente pelos itens de I para os itens base em $GOTO(I, X)$ e os itens de I a partir dos quais os lookaheads são propagados para os itens base em $GOTO(I, X)$.

Construção eficiente de tabelas de análise LALR- Algoritmo

Algoritmo 4.62: Determina lookaheads.

MÉTODO: descobre lookaheads propagados e espontâneos.

for (cada item $A \to \alpha\cdot\beta$ em K) {
  J := CLOSURE($\{[A \to \alpha\cdot\beta, \#]\}$);
  if ($[B \to \gamma\cdot X\delta, a]$ está em J, e a não é #)
    conclui que lookahead a é gerado espontaneamente para o item $B \to \gamma X\cdot\delta$ in $GOTO(I, X)$;
  if ($[B \to \gamma\cdot X\delta, \#]$ is in J)
    conclui que lookahead a é gerado espontaneamente para o item $A \to \alpha\cdot\beta$ em I para $B \to \gamma X\cdot\delta$ em $GOTO(I, X)$;
}

É possível agora associar lookaheads às bases dos conjuntos de itens LR(0) para formar os conjuntos de itens LALR(1).

Construção eficiente de tabelas de análise LALR- Algoritmo

Algoritmo 4.63: Computação eficiente das bases dos conjuntos de itens da coleção LALR(1).

ENTRADA: Uma gramática estendida G'.

SAÍDA: As bases dos conjuntos de itens da coleção LALR(1) para G'.

MÉTODO:

1. Construa as bases dos conjuntos de itens LR(0) para G. Se espaço não for problema, o modo mais simples é construir os conjuntos de itens LR(0), e depois remover os itens que não são base.

2. Aplique o Algoritmo 4.62 a base de cada conjunto de itens LR(0) e símbolo X da gramática para determinar quais lookaheads são gerados espontaneamente para os itens base em $GOTO(I, X)$, e de quais itens em I os lookaheads são propagados para os itens base em $GOTO(I, X)$.

Construção eficiente de tabelas de análise LALR- Algoritmo

Algoritmo 4.63: Computação eficiente das bases dos conjuntos de itens da coleção LALR(1).

3. Inicie uma tabela que dê, para cada item base em cada conjunto de itens, os lookaheads associados.

   Inicialmente, cada item tem associado a ele apenas os lookaheads que determinamos no Passo (2) como sendo gerados espontaneamente.

4. Faça passadas repetidas sobre os itens base em todos os conjuntos até que não seja mais possível propagar novos lookaheads.

   Quando visitamos um item i, pesquisamos os itens base aos quais i propaga seus lookaheads, usando as informações tabuladas no Passo (2). O conjunto corrente de lookaheads para i é acrescentado aos que já estão associados a cada um dos itens aos quais i propaga seus lookaheads.

Construção eficiente de tabelas de análise LALR- Exemplo

$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

Bases dos conjuntos de itens LR(0) para a gramática dada

I_0: $S' \rightarrow \cdot S$

I_1: $S' \rightarrow S\cdot$

I_2: $S \rightarrow L\cdot = R$, $R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4: $L \rightarrow *\cdot R$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6: $S \rightarrow L = \cdot R$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

Construção das bases dos itens LALR(1) dada as bases dos itens LR(0). Ao aplicar o Algoritmo 4.62 à base do conjunto de itens I_0, primeiro é calculado o $CLOSURE(\{[S' \rightarrow \cdot S, \#]\})$, que é

$$
\begin{array}{ll}
S' \rightarrow \cdot S,\ \# & L \rightarrow \cdot * R,\ \#/= \\
S \rightarrow \cdot L = R,\ \# & L \rightarrow \cdot \mathbf{id},\ \#/= \\
S \rightarrow \cdot R,\ \# & R \rightarrow \cdot L,\ \#
\end{array}
$$

Entre os itens do fechamento, há dois onde o lookahead $=$ foi gerado espontaneamente.

(1) $L \rightarrow \cdot * R$**. Esse item, com $*$ à direita do ponto, faz surgir $[L \rightarrow *\cdot R, =]$, (I_4).**

(2) $[L \rightarrow \cdot \mathbf{id}, =]$**, (I_5).**

Construção eficiente de tabelas de análise LALR- Exemplo

$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

Como # é um lookahead para todos os seis itens do fechamento, determinamos que o item $S' \rightarrow \cdot S$ em I_0 propaga os lookaheads para os seis itens a seguir:

$S' \rightarrow S\cdot$ **em** I_1 — $L \rightarrow *\cdot R$ **em** I_4

$S \rightarrow L\cdot = R$ **em** I_2 — $L \rightarrow \mathbf{id}\cdot$ **em** I_5

$S \rightarrow R\cdot$ **em** I_3 — $R \rightarrow L\cdot$ **em** I_2

Propagação de lookaheads.

De	Para
I_0: $S' \rightarrow \cdot S$	I_1: $S' \rightarrow S\cdot$
	I_2: $S \rightarrow L\cdot = R$
	I_2: $R \rightarrow L\cdot$
	I_3: $S \rightarrow R\cdot$
	I_4: $L \rightarrow *\cdot R$
	I_5: $L \rightarrow \mathbf{id}\cdot$
I_2: $S \rightarrow L\cdot = R$	I_6: $S \rightarrow L = \cdot R$
I_4: $L \rightarrow *\cdot R$	I_4: $L \rightarrow *\cdot R$
	I_5: $L \rightarrow \mathbf{id}\cdot$
	I_7: $L \rightarrow *R\cdot$
	I_8: $R \rightarrow L\cdot$
I_6: $S \rightarrow L = \cdot R$	I_4: $L \rightarrow *\cdot R$
	I_5: $L \rightarrow \mathbf{id}\cdot$
	I_8: $R \rightarrow L\cdot$
	I_9: $S \rightarrow L = R\cdot$

Construção eficiente de tabelas de análise LALR- Exemplo

Passos do Algoritmo 4.63.

A coluna rotulada com $Inicio$ mostra os lookaheads gerados espontaneamente para cada item base.

Estas são apenas as duas ocorrências de $=$ discutidas, e o lookahead espontâneo $ para o item inicial $S' \rightarrow \cdot S$.

Passo 1: o lookahead $ se propaga de $S' \rightarrow \cdot S$ in I_0 para os seis itens:

$S' \rightarrow S\cdot$ **em** I_1 — $L \rightarrow *\cdot R$ **em** I_4

$S \rightarrow L\cdot = R$ **em** I_2 — $L \rightarrow \mathbf{id}\cdot$ **em** I_5

$S \rightarrow R\cdot$ **em** I_3 — $R \rightarrow L\cdot$ **em** I_2

O lookahead $=$ se propaga de $L \rightarrow *\cdot R$ em I_4 para os itens $L \rightarrow *R\cdot$ em I_7 e $R \rightarrow L\cdot$ em I_8.

Também se propaga para si mesmo e para $L \rightarrow \mathbf{id}\cdot$ in I_5, mas esses lookaheads já estão presentes.

Cálculo de lookaheads

Item do Conjunto	Lookaheads	
	Inicio	Passo 1
I_0: $S' \rightarrow \cdot S$	$	$
I_1: $S' \rightarrow S\cdot$		$
I_2: $S \rightarrow L\cdot = R$		$
$R \rightarrow L\cdot$		$
I_3: $S \rightarrow R\cdot$		$
I_4: $L \rightarrow *\cdot R$	=	=/$
I_5: $L \rightarrow \mathbf{id}\cdot$	=	=/$
I_6: $S \rightarrow L = \cdot R$		
I_7: $L \rightarrow *R\cdot$		=
I_8: $R \rightarrow L\cdot$		=
I_9: $S \rightarrow L = R\cdot$		

Construção eficiente de tabelas de análise LALR- Exemplo

Passos do Algoritmo 4.63.

No segundo e terceiro passos, o único lookahead novo propagado é $, descoberto para os sucessores de I_2 e I_4 no Passo 2 e para o sucessor de I_6 no Passo 3.

Nenhum lookahead novo é propagado no Passo 4, de modo que o conjunto final de lookaheads aparece na coluna mais à direita, Passo 3.

Cálculo de lookaheads

Item do Conjunto	Lookaheads			
	Inicio	Passo 1	Passo 2	Passo 3
I_0: $S' \rightarrow \cdot S$	$	$	$	$
I_1: $S' \rightarrow S\cdot$		$	$	$
I_2: $S \rightarrow L\cdot = R$		$	$	$
$R \rightarrow L\cdot$		$	$	$
I_3: $S \rightarrow R\cdot$		$	$	$
I_4: $L \rightarrow *\cdot R$	=	=/$	=/$	=/$
I_5: $L \rightarrow \mathbf{id}\cdot$	=	=/$	=/$	=/$
I_6: $S \rightarrow L = \cdot R$			$	$
I_7: $L \rightarrow *R\cdot$		=	=/$	=/$
I_8: $R \rightarrow L\cdot$		=	=/$	=/$
I_9: $S \rightarrow L = R\cdot$				$

Cálculo de Lookaheads de [A → α•]

Kristensen's paper (ACM TOPLAS Vol 1, January 1981, pp 60-82)

Sejam S, T estados da máquina LR(0)

$\mathbf{X} \in (V_N \cup V_T)$
$\alpha, \beta, \varphi, \psi \in V^*$
$\mathbf{A}, \mathbf{B} \in V_N$

$$PRED(T,\alpha) = \begin{cases} \{T\}, \text{ se } \alpha = \mathcal{E} \\ \cup\{PRED(S,\alpha') | GOTO(S,X) = T\}, \text{ se } \alpha = \alpha' X \end{cases}$$

··· Cálculo de Lookaheads de [A → α•]

$LALR_1$ **([A → α • β], T) =**
$\quad \cup \{LALR_1$ **([A →** $\bullet\alpha\beta$**], S) | S ∈ PRED(T,**α**)**$\}$

$$\Downarrow$$

$LALR_1$ **([A →** $\bullet\alpha$**], S) =**
$\quad \cup$ **{FIRST(**ψ**) | [B →** $\varphi\bullet$**A**ψ**] ∈ S}** $-$ $\{\mathcal{E}\} \cup$
$\quad \cup \{LALR_1$ **([B →** $\varphi\bullet$**A**ψ**], S) | [B →** $\varphi\bullet$**A**ψ**] ∈ S** $\wedge \psi \stackrel{*}{\Rightarrow} \mathcal{E}\}$

··· Cálculo de Lookaheads de [A → α•]

Seja:
TRANS(T) = {a | [B → $\varphi\bullet$**a**ψ**] ∈ T}** $\cup$
$\quad \cup$ **{TRANS(GOTO(T,A)) | [B →** $\varphi\bullet$**A**ψ**] ∈ T** $\wedge$ **A** $\stackrel{*}{\Rightarrow} \mathcal{E}\}$

então:

FIRST(ψ**) t.q. [B →** $\varphi\bullet$**A**ψ**] ∈ S) = TRANS(GOTO(S, A))**

e então:

$LALR_1$ **([A →** $\alpha \bullet \beta$**], T) =** $\cup$ **{L(S,A) | S ∈ PRED(T,**α**)** $\}$

L(S,A) = TRANS(GOTO(S,A)) $\cup$
$\quad \cup \{LALR_1$ **([B →** $\varphi\bullet$**A**ψ**], S) | [B →** $\varphi\bullet$**A**ψ**] ∈ S** $\wedge \psi \stackrel{*}{\Rightarrow} \mathcal{E}\}$

Coleção LR(0) com Lookaheads Somente nos Itens: A → α•

I_0:	$S' \to \bullet$S	$\xrightarrow{S} I_1$	I_6:	S → bA•c	$\xrightarrow{c} I_9$
	S → •Aa	$\xrightarrow{A} I_2$			
	S → •bAc	$\xrightarrow{b} I_3$	I_7:	S → bd•a	$\xrightarrow{a} I_{10}$
	S → •dc	$\xrightarrow{d} I_4$		A → d•,c	
	S → •bda	$\xrightarrow{b} I_3$			
	A → •d	$\xrightarrow{d} I_4$	I_8:	S → dc•,$	
I_1:	$S' \to$ S•,$		I_9:	S → bAc•,$	
I_2:	A → A•a	$\xrightarrow{a} I_5$	I_{10}:	S → bda•, $	
I_3:	S → b•Ac	$\xrightarrow{A} I_6$			
	S → b•da	$\xrightarrow{d} I_7$			
	A → •d	$\xrightarrow{d} I_7$			
I_4	S → d•c	$\xrightarrow{c} I_8$			
	A → d•,a				
I_5:	S → Aa•,$				

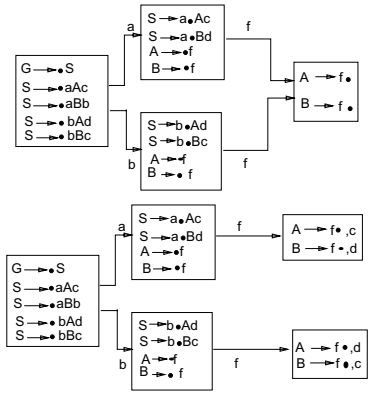
Método Eficiente para Construção da Tabela LR(1)

1. **Crie a máquina LR(0).**
2. **Cada estado inadequado é substituído por n cópias, uma para cada transição que o atinge.**
3. **O lookahead de cada estado (cópia) é calculado.**
4. **Une-se novamente as cópias que não resultam em conflitos.**

David Spector
SIGPLAN NOTICES, Ago.1981

Método Eficiente: Construção da Tabela LR(1)

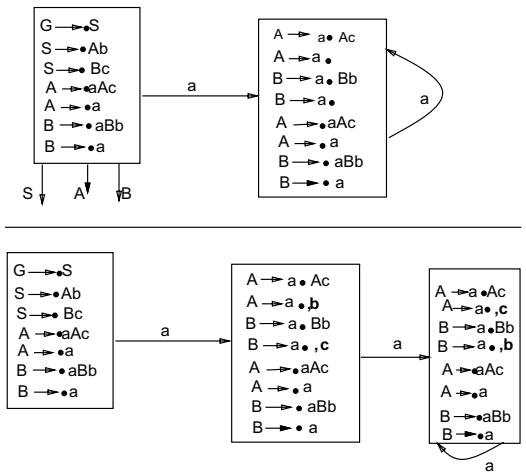
Exemplo 1: dada a gramática: G → S
S → aAc | aBd | bAd | bBc
A → f
B → f



Método Eficiente: Construção da Tabela LR(1) - Exemplo 2:

Dada a gramática:

G → S
S → Ab | Bc
A → aAc | a
B → aBb | a



Analisador Sintático LR(1), LALR(1), SLR(1)

```
begin parsing = true; scan(tipo, valor);
    S := estado inicial; topo := 1; sin[topo] := S;
    while parsing do
        case ACTION[S, tipo]
            shift k: topo := topo + 1; S := k
                sin[topo] := k ; scan(tipo, valor)
            reduce p: A := LE da produção p
                n := tamanho LD de p
                topo := topo - n
                S := GOTO[sin[topo],A]
                topo := topo + 1; sin[topo] := S
            accept: parsing := false
            error: parsing := erro
        end end
```

Analisador Sintático LR(1), LALR(1), SLR(1)

```
var parsing : lógico;
      tipo : terminal;
      valor : value;
      ACTION : array[estado, terminal] of ação;
      GOTO : array[estado, não-terminal] of estado;
      SIN : array[0..MAX] of estado;
      topo : 0..MAX;
      k, S: estado;
      A : não-terminal;
      p : número-de-produção;
```


Corpo do Analisador Sintático LR(1), LALR(1), SLR(1)

```
begin parsing = true; S := estado inicial; topo := 0; SIN[topo] := S;
    ... preenche tabelas ACTION e GOTO ...
    Scan(tipo, valor);
    while parsing do
        case ACTION[S, tipo]
            shift k: topo := topo + 1; S := k
                SIN[topo] := k; Scan(tipo, valor)
            reduce p: A := LE da produção p; n := tamanho LD de p
                topo := topo - n
                S := GOTO[SIN[topo],A]
                topo := topo + 1; SIN[topo] := S
            accept: parsing := false
            error: parsing := erro
        end end
```


4.8 – Reconhecedores LRs e Gramáticas Ambíguas

Por que usar gramáticas ambíguas? Como resolver conflitos?

Exemplo:

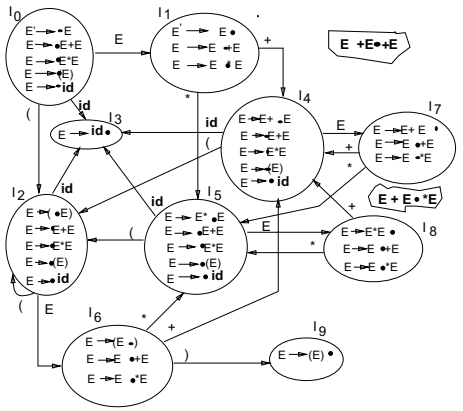
(0) $E' \rightarrow$ E	(0) E $\rightarrow$ E + T
(1) E $\rightarrow$ E + E	(1) E $\rightarrow$ T
(2) E $\rightarrow$ E * E	(2) T $\rightarrow$ T *F
(3) E $\rightarrow$ (E)	(3) T $\rightarrow$ F
(4) E $\rightarrow$ id	(4) F $\rightarrow$ (E) | id

FOLLOW(E') = { $ }
FOLLOW(E) = { $, +, *,) }

- Razões para usar gramática ambigua para expressão:

1. O nível da associatividade e precedência dos operadores "+" e "*" pode ser facilmente substituido sem alterar as produções e sem alterar o número de estados no reconhecedor gerado.
2. O reconhecedor gasta muito tempo efetuando as reduções do tipo E $\rightarrow$ T e T $\rightarrow$ F, cujo objetivo é garantir a associatividade e precedência. Usando a gramática ambigua, o reconhecedor não perde tempo com estas reduções.

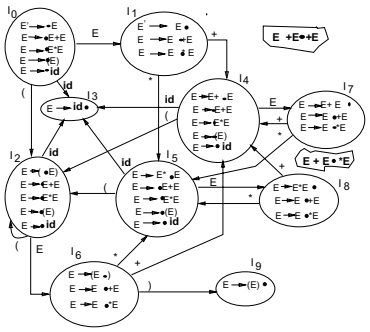
Usando Gramáticas Ambíguas



- **Conflitos:** I_7 **reduce E $\rightarrow$ E + E - shift "+" e "*"**
  I_8 **reduce E $\rightarrow$ E * E - shift "+" e "*" - FOLLOW(E) = { $, +, ⋆,) }**
- **Solução: usar informações sobre a precedência e associatividade de "*" e "+".**

· · · Usando Gramáticas Ambíguas

(0) $E' \rightarrow$ E
(1) E $\rightarrow$ E + E
(2) E $\rightarrow$ E * E
(3) E $\rightarrow$ (E)
(4) E $\rightarrow$ id



Estado					Ação		Goto
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			acc	
2	S3			S2			6
3		r4	r4		r4	r4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5		S9		
7		r1/S4	r1/S5		r1	r1	
8		r2/S4	r2/S5		r2	r2	
9		r3	r3		r3	r3	

Precedência e Associatividade para Solucionar os Conflitos Gerados

Situação 1: id + id * id

Se "*" tem precedência sobre "+", o reconhecedor efetua um *shift* de "*" para a pilha.

Se "+" tem precedência sobre "*", o reconhecedor irá reduzir E + E para E.

pilha	entrada
0 E 1 + 4 E 7	* id $

· · · Precedência e Associatividade para Solucionar os Conflitos Gerados

Situação 2: id + id + id

Se "+" se associa à esquerda, a ação correta é reduzir E $\rightarrow$ E + E.

pilha	entrada
0 E 1 + 4 E 7	+ id $

· · · Precedência e Associatividade para Solucionar os Conflitos Gerados

- Assumindo que "+" se associa à esquerda, a ação do Estado 7 à entrada "+" deve ser reduzir.
- Assumindo que "$\star$" tem precedência sobre "+", a ação do Estado 7 à entrada "$\star$" deve ser empilhar.
- Assumindo que "$\star$" seja associado à esquerda e tenha precedência sobre "+", o Estado 8, que pode aparecer no topo da pilha somente quando "E$\star$E" forem os 3 símbolos gramaticais de topo, deveria ter a ação de reduzir E $\rightarrow$ E$\star$E às entradas "$\star$" e "+".

Para a entrada "+", o "$\star$" tem precedência sobre "+".
Para "$\star$", argumentamos que "$\star$" se associa à esquerda.

Precedência e Associatividade para Solucionar os Conflitos Gerados

(0) $E' \rightarrow$ E
(1) E $\rightarrow$ E + E
(2) E $\rightarrow$ E * E
(3) E $\rightarrow$ (E)
(4) E $\rightarrow$ id

Nova Tabela

Estado					Ação		Goto
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			acc	
2	S3			S2			6
3		r4	r4		r4	r4	
4	S3			S2			8
5	S3			S2			8
6		S4	S5		S9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

A Ambiguidade do ELSE VAZIO (*Dangling-else*)

stmt $\rightarrow$ **if expr then stmt else stmt**
| **if expr then stmt**
| **other**

$S' \rightarrow$ **S**
S $\rightarrow$ **iSeS** | **iS** | **a**

Conflito shift-reduce : estado I_4.

S $\rightarrow$ iS•eS – shift de "e"
S $\rightarrow$ iS• – reduce por S $\rightarrow$ iS com a entrada "e".

I_0: $S' \rightarrow$ •S S $\rightarrow$ •iSeS S $\rightarrow$ •iS S $\rightarrow$ •a	I_3: S $\rightarrow$ a•
I_1: $S' \rightarrow$ S•	I_4: S $\rightarrow$ iS•eS S $\rightarrow$ iS•
I_2: S $\rightarrow$ i•SeS S $\rightarrow$ i•S S $\rightarrow$ •iSeS S $\rightarrow$ •iS S $\rightarrow$ •a	I_5: S $\rightarrow$ iSe•S S $\rightarrow$ •iSeS S $\rightarrow$ •iS S $\rightarrow$ •a
	I_6: S $\rightarrow$ iSeS•

I4: s/r porque follow(S) = {e,$}

Solução para a Ambiguidade do Dangling-else

• **shift** else.

Tabela SLR após esta solução:

Estado	action				*GOTO*
	i	*e*	*a*	$	*S*
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Ações sintáticas para: iiaea

	Pilha	Símbolos	Entrada	Ação
(1)	0		$iiaea\$$	empilha 2 e avança
(2)	0 2	i	$iaea\$$	empilha 2 e avança
(3)	0 2 2	ii	$aea\$$	empilha 3 e avança
(4)	0 2 2 3	iia	$ea\$$	reduz segundo $S \rightarrow a$ ⟸
(5)	0 2 2 4	iiS	$ea\$$	empilha 4 e avança ⟸
(6)	0 2 2 4 5	$iiSe$	$a\$$	empilha 3 e avança
(7)	0 2 2 4 5 3	$iiSea$	$\$$	reduz segundo $S \rightarrow a$
(8)	0 2 2 4 5 6	$iiSeS$	$\$$	reduz segundo $S \rightarrow iSeS$
(9)	0 2 4	iS	$\$$	reduz segundo $S \rightarrow iS$ ⟸
(10)	0 1	S	$\$$	aceitar

Resolução de Conflitos em Gramáticas Ambíguas

Exemplo: Precedência: "+" = 10, "<" = 5
"−" = 10, ">" = 5
"*" = 20
"/" = 20

Associatividade: "+" = esquerda
"−" = esquerda
"*" = esquerda
"<" = nenhuma
">" = nenhuma

exp $\rightarrow$ exp "+" exp
| exp "−" exp
| exp "<" exp
| exp ">" exp [5]
| exp "*" exp
| exp "/" exp [20]

Resolução de Conflitos em Gramáticas Ambíguas

- **Associar precedências e associatividades aos terminais.**
- **Associar precedências às produções.**

Note bem:

Caso uma produção não tenha uma precedência associada explicitamente, a precedência da produção é a precedência do terminal mais à direita no seu lado direito.

Resolução de Conflitos em Gramáticas Ambíguas

Em caso de conflito:

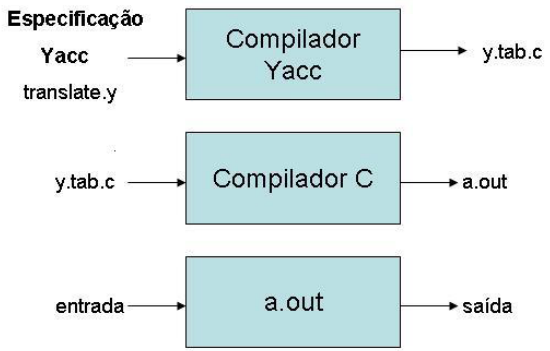
- shift-reduce

  se **precedência do terminal** $<$ **precedência da produção**

  então reduce
  senão se **precedência do terminal** $>$ **precedência da produção**
  então **shift**
  senão se **terminal associa à** esquerda
  então reduce
  senão shift

Geradores de Analisadores Sintáticos

Gerador de analisadores LALR $Yacc$ **("yet another compiler-compiler") criado por S. C. Johnson.**



O Gerador de Analisadores Sintáticos YACC

Um tradutor pode ser construído usando o $Yacc$ **da seguinte forma:**

- **Prepara-se um arquivo,** `translate.y`**, contendo uma especificação** $Yacc$ **do tradutor.**
- **O comando do sistema UNIX** `yacc translate.y` **transforma o arquivo** `translate.y` **em um programa C chamado** `y.tab.c`**, usando o método LALR esboçado no Algoritmo 4.63.**
- **O programa** `y.tab.c` **é uma representação de um analisador LALR escrito em C, junto com outras rotinas C que o usuário pode ter preparado.**

  A tabela LALR é compactada conforme método de Aho et.al.

··· O Gerador de Analisadores Sintáticos YACC

··· Um tradutor pode ser construído usando o $Yacc$ da seguinte forma:

- Compila $y.tab.c$ junto com a biblioteca ly que contém o programa de análise LR usando o comando $ccy.tab.c - ly$ e obtem-se o programa objeto desejado $a.out$, que realiza a tradução especificada pelo programa $Yacc$ original.

  Se outros procedimentos forem necessários, eles podem ser compilados ou carregados com `y.tab.c`, assim como qualquer outro programa em C.

··· O Gerador de Analisadores Sintáticos YACC

O gerador $Yacc$ possui três partes:

declarações
%%
regras de tradução
%%
rotinas C

··· O Gerador de Analisadores Sintáticos YACC

Dada a gramática para as expressões aritméticas:

$$
\begin{array}{rcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \text{digit}
\end{array}
$$

Um programa de calculadora de mesa $Yacc$ é mostrado:

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line   : expr '\n'        { printf("%d\n", $1); }
       ;
expr   : expr '+' term    { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor  { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'     { $$ = $2; }
       | DIGIT
       ;
%%
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c; }
```


Declarações

Existem duas seções na parte de declarações de um programa $Yacc$, ambas são opcionais.

Primeira seção: declarações C comuns, delimitadas por %{ e %} No exemplo, esta seção contém apenas o comando

```
#include <ctype.h>
```

que faz com que o pré-processador C inclua o arquivo de cabeçalho padrão `<ctype.h>` que contém o predicado `isdigit`.

· · · Declarações

Segunda seção: estão as declarações dos tokens da gramática.

No exemplo, o comando

```
%token DIGIT
```

declara `DIGIT` como sendo um token.

Os tokens declarados nesta seção podem, então, ser usados na segunda e terceira partes da especificação $Yacc$.

Se o Lex é usado para criar o analisador léxico que passa o token ao analisador sintático $Yacc$, então essas declarações de token também se tornam disponíveis ao analisador gerado pelo Lex (Seção 3.5.2 do livro texto).

Regras de Tradução

As regras de tradução são colocadas após o primeiro par %%, na especificação $Yacc$.

Cada regra consiste em uma produção da gramática e a ação semântica associada. Um conjunto de produções do tipo:

$$\langle \text{head} \rangle \rightarrow \langle \text{body} \rangle_1 \mid \langle \text{body} \rangle_2 \mid \cdots \mid \langle \text{body} \rangle_n$$

seria escrito no $Yacc$ como

$$\begin{aligned} \langle \text{head} \rangle \ : \ & \langle \text{body} \rangle_1 \ \texttt{\{} \ \langle \text{semantic action} \rangle_1 \ \texttt{\}} \\ \mid \ & \langle \text{body} \rangle_2 \ \texttt{\{} \ \langle \text{semantic action} \rangle_2 \ \texttt{\}} \\ & \cdots \\ \mid \ & \langle \text{body} \rangle_n \ \texttt{\{} \ \langle \text{semantic action} \rangle_n \ \texttt{\}} \\ ; & \end{aligned}$$

· · · Regras de Tradução

Em uma produção do $Yacc$, cadeias de letras e dígitos sem aspas, não declaradas como tokens, são consideradas não-terminais.

Um único caractere entre apóstrofos, por exemplo, `'c'`, é considerado o símbolo terminal c, assim como o código inteiro para o token representado por esse caractere.

Lados direitos alternativos podem ser separados por uma barra vertical, e um ponto-e-vírgula vem após cada lado esquerdo com suas alternativas e suas ações semânticas.

Na primeira produção, $head$ é considerada o símbolo inicial.

· · · Regras de Tradução

Uma ação semântica do $Yacc$ é uma seqüência de instruções C.

Em uma ação semântica, o símbolo $$ refere-se ao valor do atributo associado ao não-terminal $head$.

O símbolo $i se refere ao valor do atributo associado ao i-ésimo símbolo da gramática do corpo.

A ação semântica é efetuada sempre que reduzimos pela produção associada, de modo que normalmente a ação semântica calcula um valor para $$ em termos dos $is.

· · · Regras de Tradução

Na especificação $i, escrevemos as duas produções-E

$$E \rightarrow E + T \mid T$$

e suas ações semânticas associadas como:

```
expr : expr '+' term    { $$ = $1 + $3; }
     | term
     ;
```

Observe que o não-terminal `term` na primeira produção é o terceiro símbolo da gramática no corpo, enquanto "+" é o segundo.

· · · Regras de Tradução

```
expr : expr '+' term    { $$ = $1 + $3; }
     | term
     ;
```

Omitimos completamente a ação semântica para o segunda produção, pois a cópia do valor é a ação "default" para produções com um único símbolo da gramática no corpo.

Em geral, `{ $$ = $1; }` é a ação semântica "default".

Observe que incluímos uma nova produção inicial à especificação $Yacc$.

```
line : expr '\n'    { printf("%d\n", $1); }
```

Essa produção diz que uma entrada para a calculadora de mesa deve ser uma expressão seguida por um caractere de quebra de linha.

Rotinas de Suporte em C

A terceira parte de uma especificação $Yacc$ consiste em rotinas de suporte em C.

Um analisador léxico com o nome `yylex()` precisa ser fornecido. O uso do Lex para produzir o `yylex()` é uma opção comum.

Outros procedimentos, como rotinas de recuperação de erro, podem ser acrescentados conforme a necessidade.

Usando $Yacc$ com Gramáticas Ambíguas

```
lines : lines expr '\n'    { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```

Em $Yacc$, uma alternativa vazia, como na terceira linha, denota ϵ.

· · · Usando $Yacc$ com Gramáticas Ambíguas

$E \rightarrow E + E \mid E - E$
$E \rightarrow E * E \mid E / E$
$E \rightarrow (E) \mid \mathbf{number} \mid -E$

A especificação $Yacc$ resultante aparece em

```
%{
#include <ctype.h>
#include <stdio.h>
/* double type
#define YYSTYPE double
for Yacc stack */
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
```

```
%%
lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c; }
```


· · · Usando $Yacc$ com Gramáticas Ambíguas

A menos que instruído ao contrário, o $Yacc$ resolverá todos os conflitos de ação de análise usando as duas regras a seguir:

1. **Um conflito reduce/reduce é resolvido escolhendo-se a produção em conflito listada primeiro na especificação $Yacc$.**

2. **Um conflito shift/reduce é resolvido em favor da transferência (shift).**

   Esta regra resolve corretamente o conflito shift/reduce que surge da ambigüidade do else vazio.

· · · Usando $Yacc$ com Gramáticas Ambíguas

O $Yacc$ oferece um mecanismo geral para resolver conflitos de shift/reduce.

Na parte de declarações, podemos atribuir precedências e associatividades aos terminais.

```
%left '+' '-' %right '^' %nonassoc '<' %right UMINUS
```

O $Yacc$ resolve conflitos do tipo shift/reduce conectando uma precedência e associatividade a cada produção envolvida em um conflito, bem como a cada terminal envolvido em um conflito.

· · · Usando $Yacc$ com Gramáticas Ambíguas

Naquelas situações em que o terminal mais à direita não fornece a precedência apropriada a uma produção, podemos forçar uma precedência anexando a uma produção a tag

`%prec` ⟨**terminal**⟩

A precedência e a associatividade da produção, então, serão iguais às do terminal, presumivelmente definido na seção de declaração.

O $Yacc$ não informa conflitos shift/reduce que são resolvidos usando esse mecanismo de precedência e associatividade.

··· Usando $Yacc$ com Gramáticas Ambíguas

Esse "terminal" pode ser um marcador de lugar, como `UMINUS` no exemplo; esse terminal não é retornado pelo analisador léxico, mas é declarado unicamente para definir uma precedência para uma produção.

No exemplo, a declaração

```
%right UMINUS
```

atribui ao token `UMINUS` uma precedência que é mais alta do que a de $*$ e $/$. Na parte de regras de tradução, a tag:

```
%prec UMINUS
```

no fim da produção

```
expr  : '-' expr
```

faz com que o operador menos unário nessa produção tenha uma precedência mais alta do que qualquer outro operador.

Comparação dos Reconhecedores

Descendente Recursivo

- Fácil de implementar.
- Eficiência depende do call recursivo.
- Geração de código espalhada.
- Impõe grandes restrições à gramática.

Ascendente LR(K), LALR(K), SLR(K)

- Difícil gerar tabela sem auxílio do computador.
- Eficiente (LALR, SLR).
- Pouca programação.
- Geração de código localizada.
- Funciona para linguagens determinísticas (LR(k)).
- Pouca visibilidade de contexto.

FIM