Aula 05 – *Bottom-Up Parsing*

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

2016/2 – Compiladores
***Compiler Construction* (CC)**

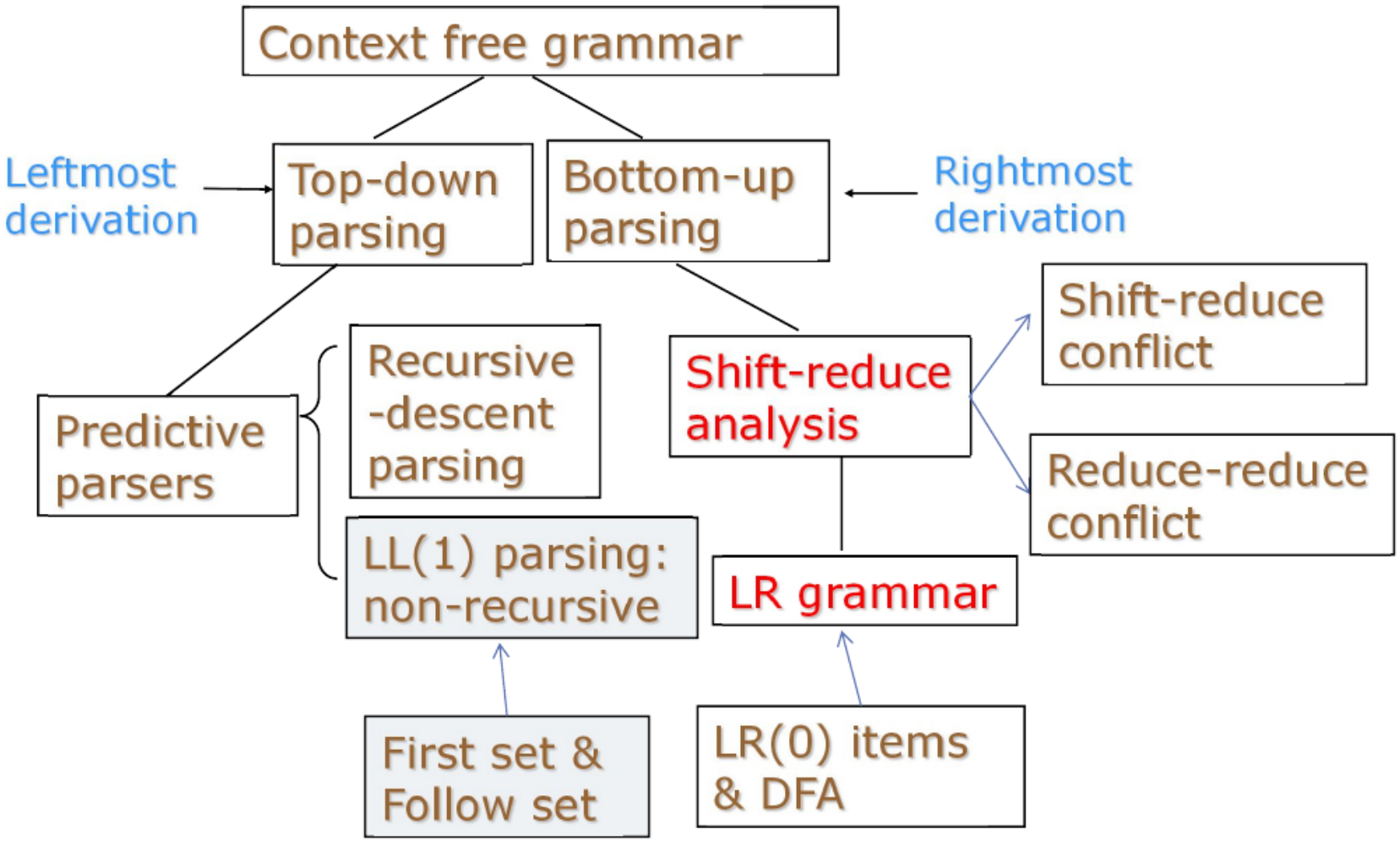
Introdução

- O segundo módulo do compilador realiza a análise sintática (*parsing*).
- *Parsing* pode ser feito em duas direções: *top-down* e *bottom-up*.
- **Aula de hoje**: discussão sobre os principais conceitos de *parsing bottom-up*.
- **Objetivos**: apresentar as teorias fundamentais para o funcionamento desse tipo de algoritmo.

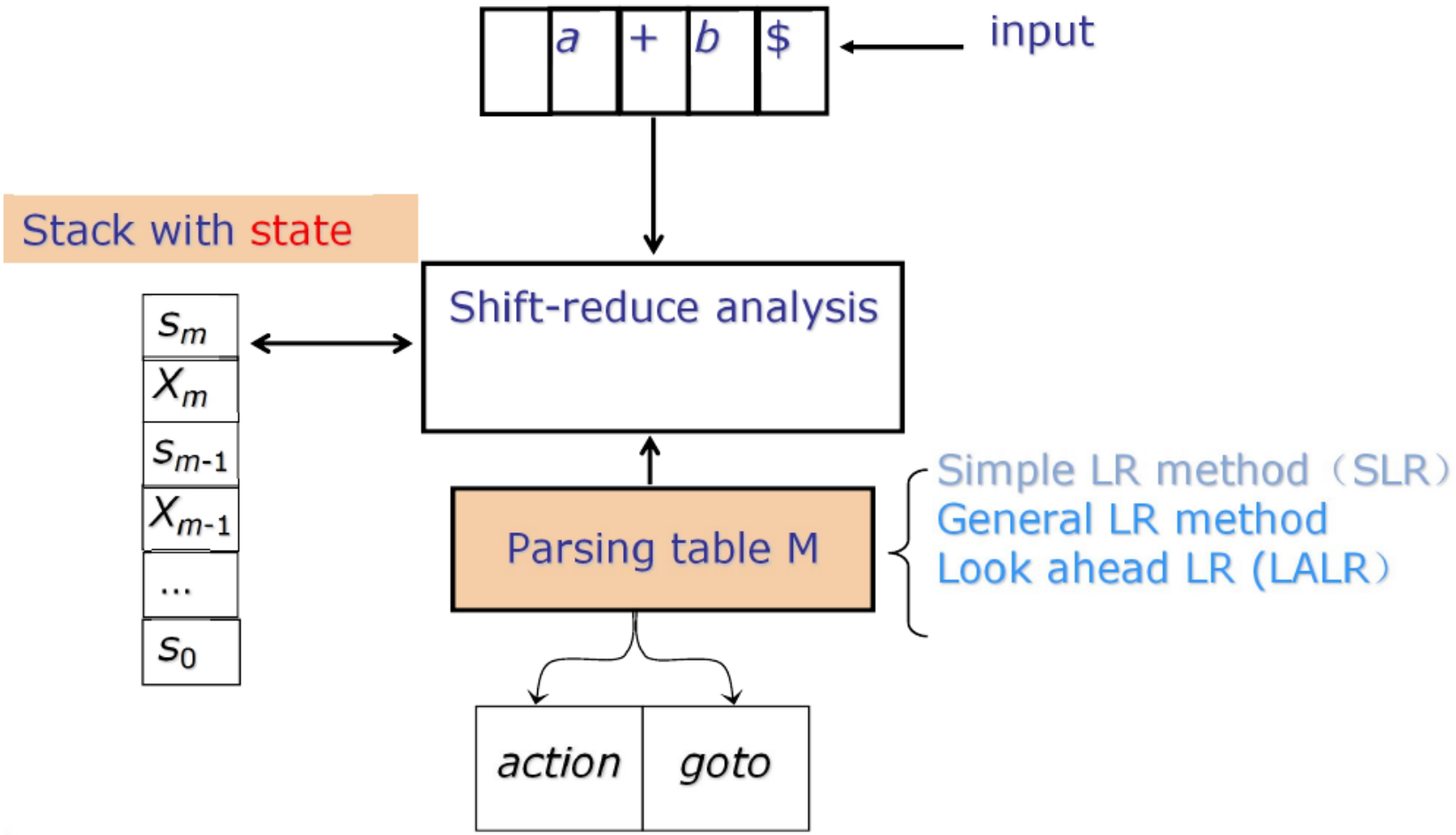
Referências

Chapter 5 – Bottom-Up Parsing
K. C. Louden
Chapter 3 – Parsers
K. D. Cooper

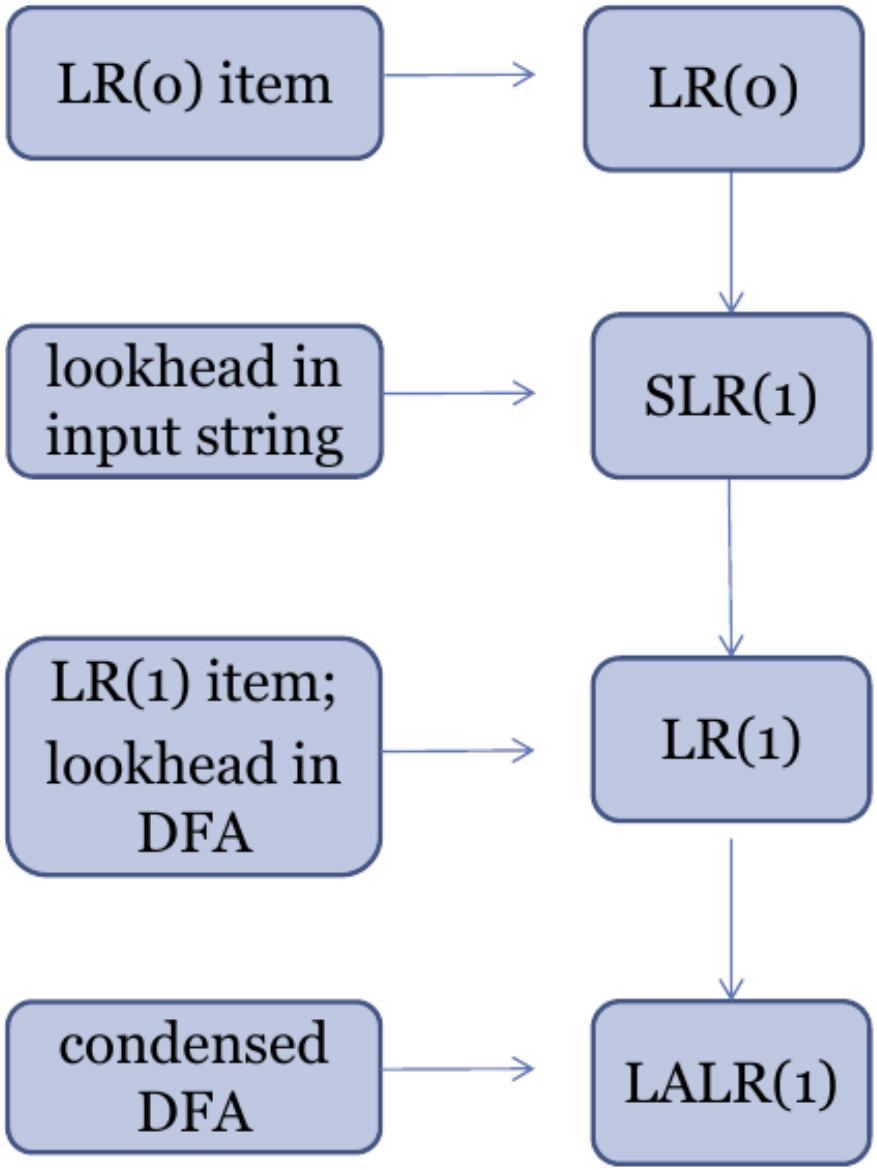
Visão Geral do Processo de *Parsing*



Visão Geral do *Parsing Bottom-Up*

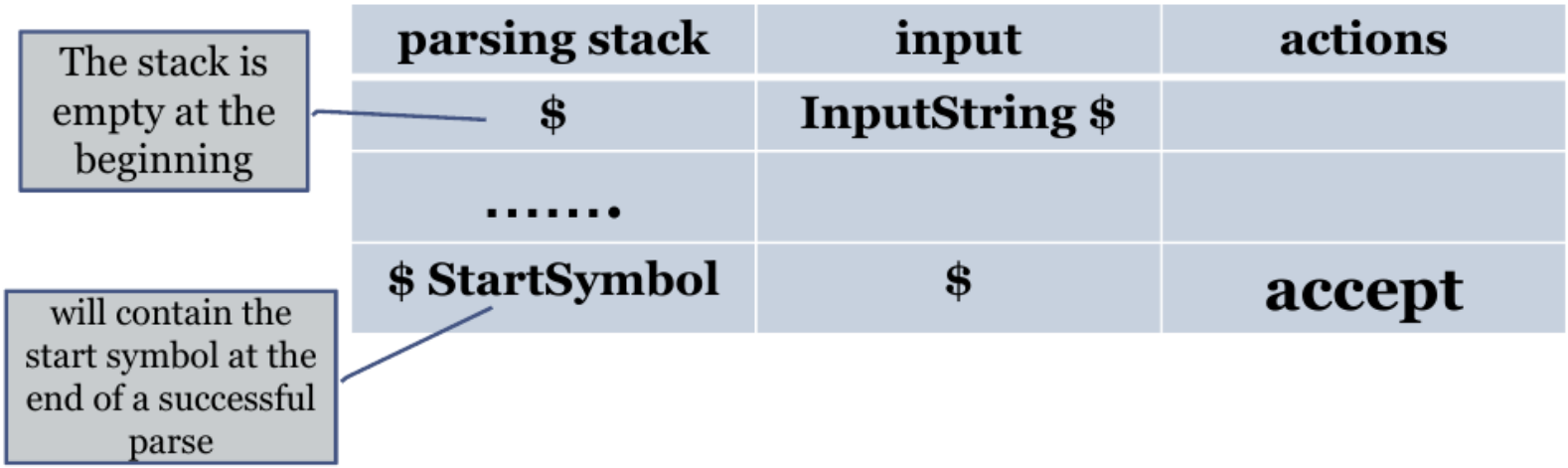


Diferentes Algoritmos de *Parsing Bottom-Up*



Visão Geral do *Parsing Bottom-Up*

- Um *parser bottom-up* usa uma pilha para realizar a análise sintática.
- A pilha de *parsing* pode conter *tokens*, não-terminais e também alguma informação extra sobre o estado do *parser*.
- A figura abaixo mostra um visão esquemática da organização do *parser*.



Visão Geral do *Parsing Bottom-Up*

- Um *parser bottom-up* pode realizar duas outras ações além de *accept*: *shift* e *reduce*.
- ***Shift***: desloca um terminal da entrada para o topo da pilha.
- ***Reduce***: reduz uma forma sentencial α no topo da pilha para um não-terminal A, segundo uma regra $A \to \alpha$ da gramática.
- Por conta das ações que ele realiza, um *parser bottom-up* também é chamado de *parser shift-reduce*.
- Para poder realizar o *parsing bottom-up*, a gramática deve ser aumentada com um novo símbolo inicial S' e uma regra $S' \to S$. (Justificativa será dada adiante.)

Visão Geral do *Parsing Bottom-Up* – Exemplo 1

Considere a gramática aumentada de parênteses balanceados.

$$S' \to S$$
$$S \to (\; S \;) \;\; S \mid \varepsilon$$

O *parse bottom-up* da entrada `()` é dado a seguir.

	Parsing stack	Input	Action
1	$	()$	shift
2	$ (	)$	reduce $S \to \varepsilon$
3	$ (S	)$	shift
4	$ (S)	$	reduce $S \to \varepsilon$
5	$ (S) S	$	reduce $S \to (\; S \;) \; S$
6	$ S	$	reduce $S' \to S$
7	$ S'	$	accept

Visão Geral do *Parsing Bottom-Up* – Exemplo 2

Considere a gramática aumentada para expressões de adição.

$$E' \to E$$
$$E \to E + \mathbf{n} \mid \mathbf{n}$$

O *parse bottom-up* da entrada `n+n` é dado a seguir.

	Parsing stack	Input	Action
1	$	n + n $	shift
2	$ n	+ n $	reduce $E \to n$
3	$ E	+ n $	shift
4	$ E +	n $	shift
5	$ E + n	$	reduce $E \to E + n$
6	$ E	$	reduce $E' \to E$
7	$ E'	$	accept

Visão Geral do *Parsing Bottom-Up* – Observações

- Um *parser shift-reduce* acompanha uma derivação mais à direita da entrada, mas os passos da derivação ocorrem na ordem inversa.
- No Exemplo 1 há quatro reduções, correspondendo a seguinte derivação mais à direita:

$$S' \Rightarrow S \Rightarrow (\ S\ )\ S \Rightarrow (\ S\ ) \Rightarrow (\ )\quad .$$

- No Exemplo 2 a derivação correspondente é

$$E' \Rightarrow E \Rightarrow E\ \texttt{+}\ \texttt{n} \Rightarrow \texttt{n}\ \texttt{+}\ \texttt{n}\quad .$$

- A cada passo da derivação, a forma sentencial está dividida entre a pilha e a entrada.

Visão Geral do *Parsing Bottom-Up* – Observações

- *Parser* realiza *shifts* até que seja possível realizar um *reduce*.
- Isso ocorre quando a sequência de símbolos a partir do topo da pilha corresponde ao corpo de uma regra da gramática.
- Um *handle* da forma sentencial é formado por:
  - Uma sequência de símbolos (*string*) na pilha.
  - A posição na forma sentencial aonde a sequência ocorre.
  - A regra que deve ser usada na redução.
- Se uma gramática é livre de ambiguidades, então existe somente uma única derivação mais à direita, e portanto os *handles* são únicos.
- Determinar o próximo *handle* é a tarefa principal de um *parser shift-reduce*.

Visão Geral do *Parsing Bottom-Up* – Observações

- A *string* de um *handle* corresponde ao corpo de uma regra e o último símbolo deste corpo está no topo da pilha.
- No entanto, para ser de fato um *handle*, não basta que a *string* no topo da pilha case com o corpo de uma regra.
- De fato, se uma produção-ϵ faz parte da gramática (como no Exemplo 1), então o seu corpo (a *string* vazia) sempre está no topo da pilha.
- Assim, temos uma restrição sobre as reduções: elas só podem ocorrem quando a *string* resultante de fato corresponde a uma forma sentencial válida na derivação.
- *Exemplo*: no passo 3 da tabela do Exemplo 1, uma redução por $S \rightarrow \epsilon$ poderia ser feita, mas a *string* resultante (S S) não é uma forma sentencial válida.
- $\Rightarrow$ Os algoritmos de *parsing bottom-up* devem computar os *handles* de forma a garantir que eles só serão reduzidos nos momentos adequados.

Itens LR(0)

- Um item LR(0) de uma CFG é uma regra com uma posição no corpo marcada por um ponto (.).
- Recebem esse nome porque não fazem referência ao *look-ahead* (em contraste com itens LR(1), que o fazem).
- Para uma regra qualquer $A \to \beta_1 \ldots \beta_n$, o marcador pode aparecer antes ou depois de qualquer β_i.
- *Exemplo*: se $A \to \alpha\beta$ é uma regra, então $A \to \alpha \,.\, \beta$ é um dos itens LR(0) da gramática.
- Itens indicam um passo intermediário no reconhecimento do corpo de uma regra.
- *Exemplo*: o item $A \to \alpha \,.\, \beta$ indica que α já foi visto (está no topo da pilha) e que é possível derivar os próximos *tokens* de entrada a partir de β.

Itens LR(0) – Exemplos

A gramática

$$S' \to S$$
$$S \to (\ S\ )\ \ S \mid \varepsilon$$

possui os seguintes oito itens LR(0)

$$S' \to .S$$
$$S' \to S.$$
$$S \to .(S)S$$
$$S \to (.S)S$$
$$S \to (S.)S$$
$$S \to (S).S$$
$$S \to (S)S.$$
$$S \to .$$

A gramática

$$E' \to E$$
$$E \to E + \mathbf{n} \mid \mathbf{n}$$

possui os seguintes oito itens LR(0)

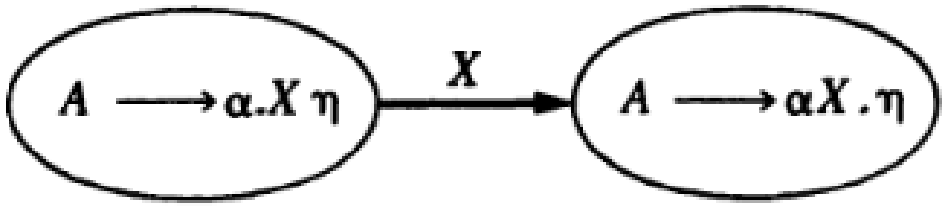
$$E' \to .E$$
$$E' \to E.$$
$$E \to .E + \mathbf{n}$$
$$E \to E.+ \mathbf{n}$$
$$E \to E +.\mathbf{n}$$
$$E \to E + \mathbf{n}.$$
$$E \to .\mathbf{n}$$
$$E \to \mathbf{n}.$$

Autômato Finito de Itens

- Itens LR(0) podem ser usados como estados de um autômato finito (FA).
- Esse FA mantém informação sobre a pilha de *parsing* e o progresso das operações de *shift-reduce*.
- Inicialmente vamos usar um autômato finito não-determinístico (NFA).
- O autômato determinístico (DFA) equivalente pode ser obtido a partir do NFA usando-se o algoritmo clássico de construção de subconjuntos.
- Transições no FA são rotuladas por um símbolo $X \in (N \cup T)$.

Autômato Finito de Itens

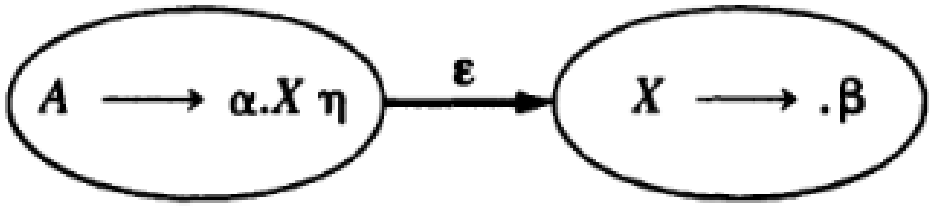
- *Exemplo*: dado um item (estado) $A \to \alpha \,.\, X\eta$, temos uma transição sobre o símbolo X para o estado $A \to \alpha X \,.\, \eta$. Graficamente:



- Se X é um *token*, a transição corresponde a um *shift* de X da entrada para o topo da pilha.
- Se X é um não-terminal, a interpretação da transição é um pouco mais complexa, já que X não aparece na entrada.

Autômato Finito de Itens

- Essa transição ainda corresponde ao empilhamento de X durante o processo de *parsing*.
- Tal empilhamento só pode acontecer com uma redução por uma regra da forma $X \rightarrow \beta$.
- Uma redução por $X \rightarrow \beta$ deve ser precedida pelo reconhecimento de β.
- O item $X \rightarrow .\beta$ representa o início desse processo.
- Assim, devemos adicionar uma transição como



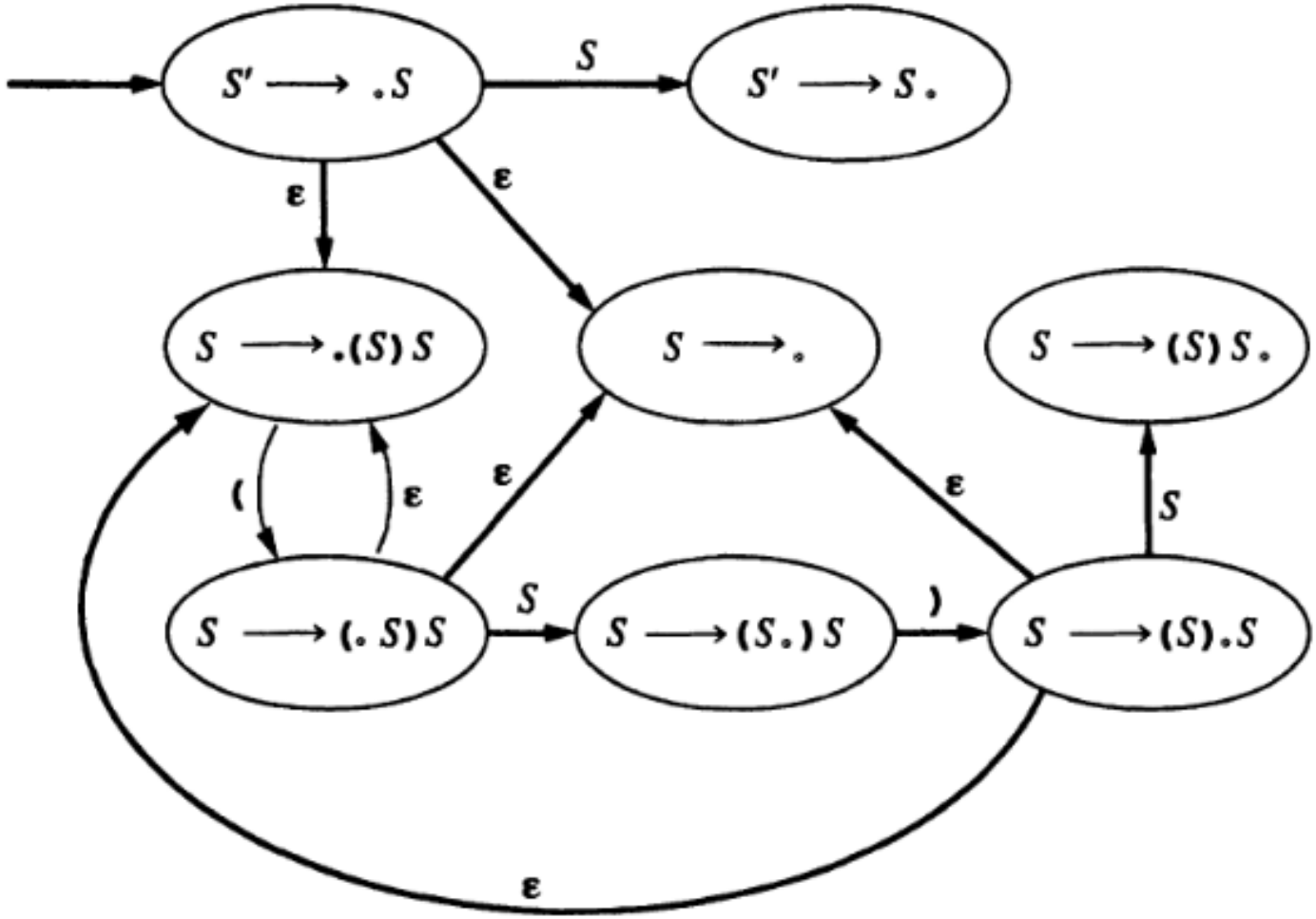
para todas as regras aonde X é a cabeça.

Autômato Finito de Itens

- O estado inicial do NFA deve corresponder ao estado inicial do *parser*: pilha vazia e pronto para reconhecer S (símbolo inicial).
- Qualquer item $S \rightarrow .\alpha$ poderia servir como estado inicial, mas a gramática pode ter várias regras aonde S é a cabeça.
- Isso justifica o uso de gramáticas aumentadas: item $S' \rightarrow .S$ garante a unicidade do estado inicial.
- O NFA não tem estados finais pois o seu propósito é acompanhar o estado do *parser* e não reconhecer *strings*.
- O próprio *parser* é que deve decidir quando aceitar a entrada.

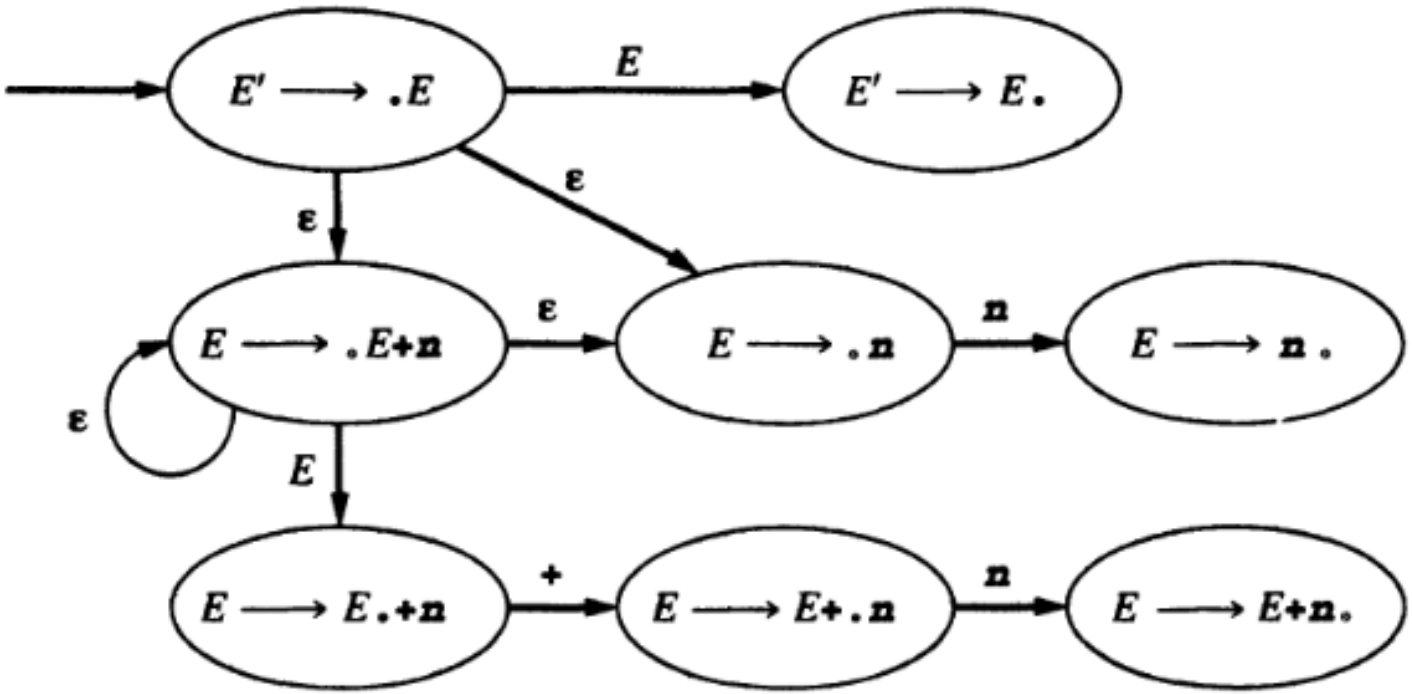
Autômato Finito de Itens – Exemplos

A gramática $S' \to S$, $S \to (\ S\ )\ \ S \mid \varepsilon$ possui o seguinte NFA:



Autômato Finito de Itens – Exemplos

A gramática $E' \to E$, $E \to E + \mathtt{n} \mid \mathtt{n}$ possui o seguinte NFA:



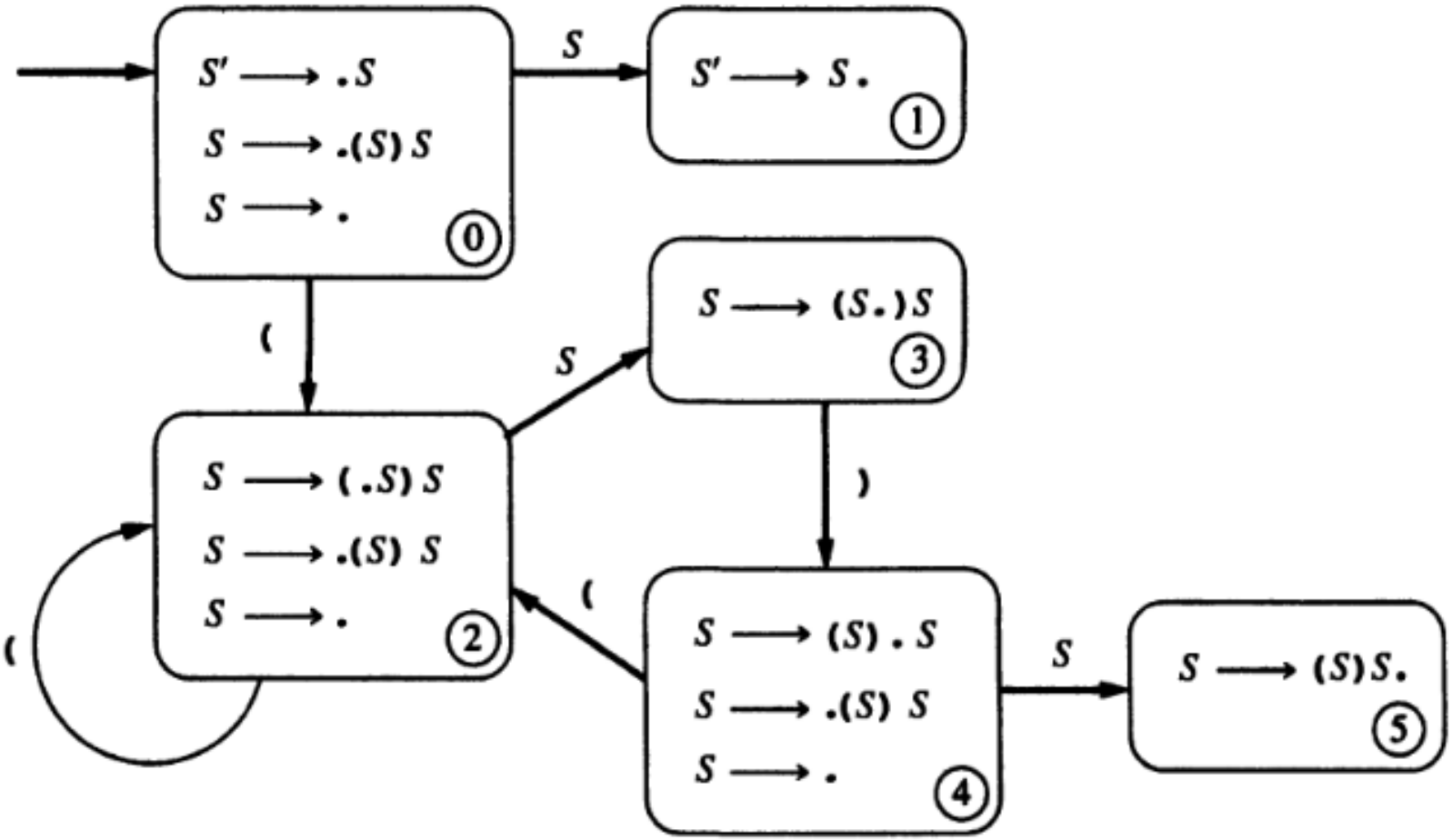
O item $E \to .E \ + \ \mathtt{n}$ tem uma transição-ϵ para ele próprio. Isso ocorre em todas as CFGs com recursão à esquerda.

Autômato Finito de Itens

- O NFA de itens LR(0) pode ser facilmente construído de forma algorítmica a partir das regras da gramática.
- No entanto, um NFA não é adequado para acompanhar os estados do *parsing* pois o *parser* deve ser um programa determinístico.
- Assim, usa-se o algoritmo clássico de construção de subconjuntos para se obter um DFA equivalente ao NFA original.
- Por questões de eficiência, nas ferramentas geradoras de *parsers bottom-up* esses dois passos são unificados em um só.
- *Exemplo*: o Bison computa diretamente o DFA a partir das regras da gramática.

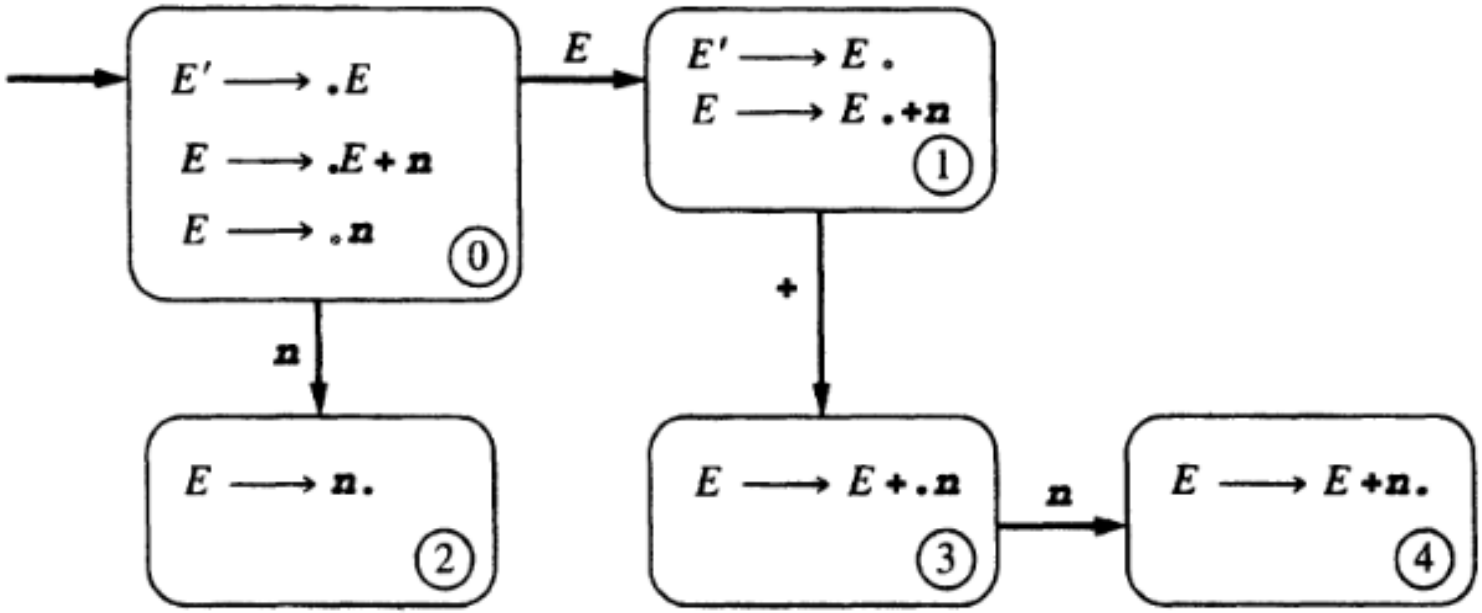
Autômato Finito de Itens – Exemplos

A gramática $S' \to S$, $S \to (\ S\ )\ \ S \mid \varepsilon$ possui o seguinte DFA:



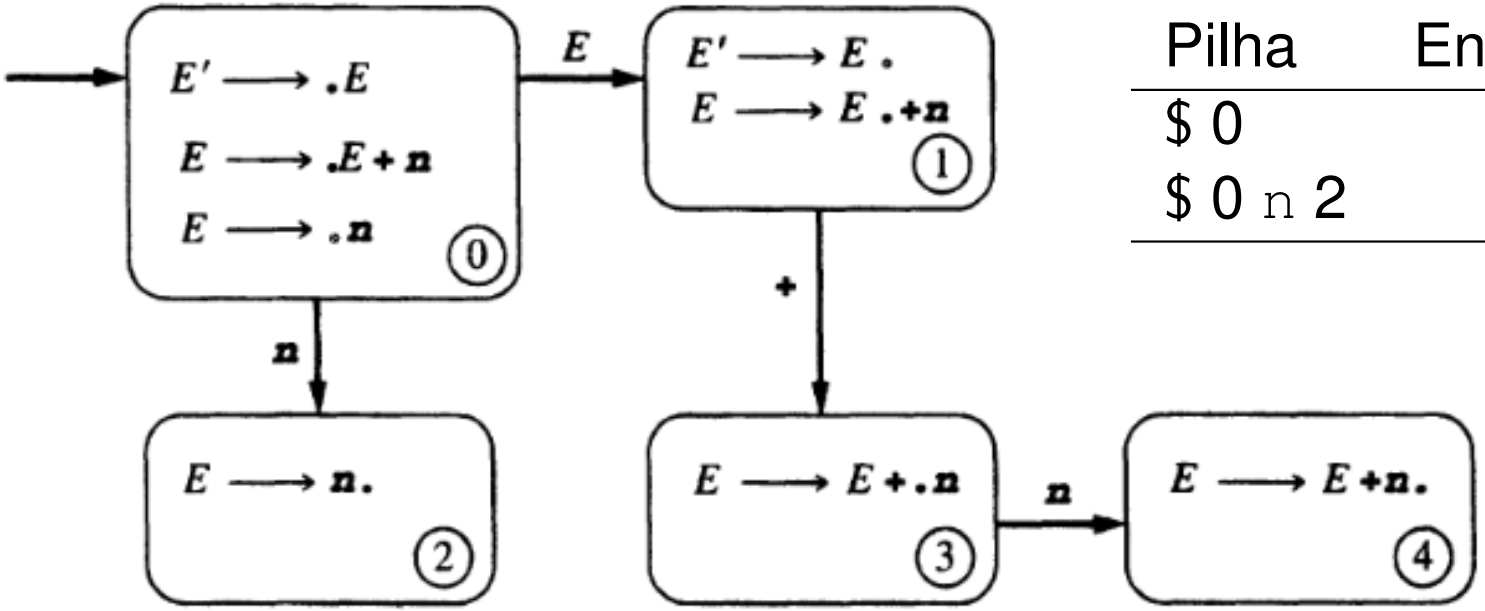
Autômato Finito de Itens – Exemplos

A gramática $E' \to E$, $E \to E + n \mid n$ possui o seguinte DFA:



Algoritmo de *Parsing* LR(0)

- O algoritmo precisa rastrear o estado atual no DFA de itens.
- Modificamos a pilha de *parsing* para armazenar os números dos estados, além dos demais símbolos.
- Empilhamos o número do novo estado após empilhar cada símbolo.



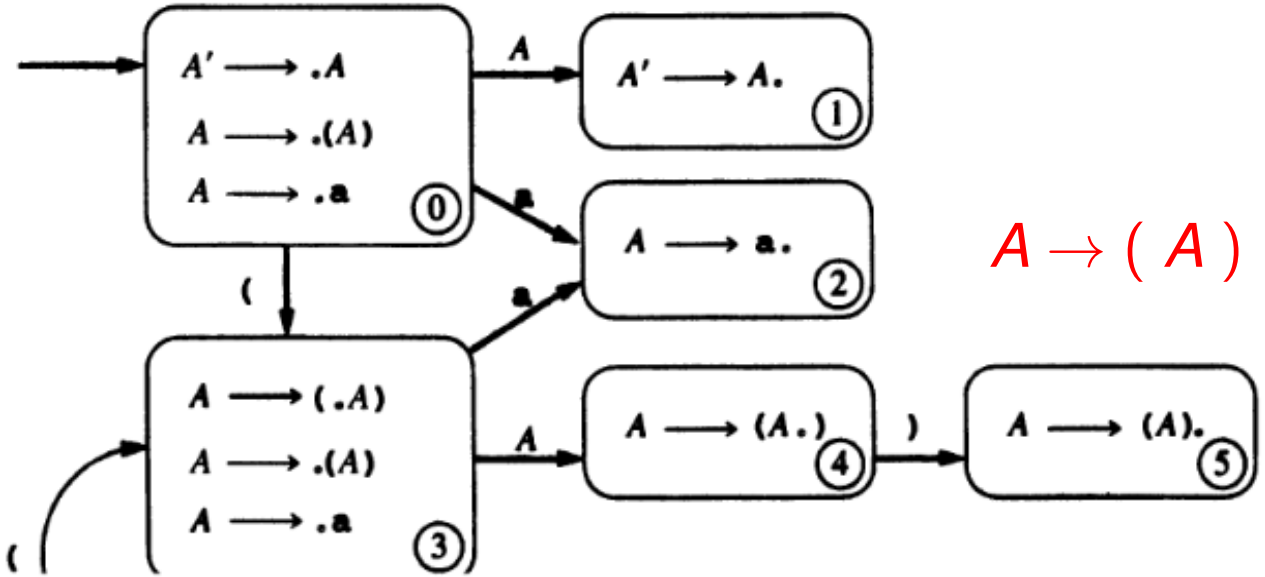
Pilha	Entrada
\$ 0	n α \$
\$ 0 n 2	α \$

Algoritmo de *Parsing* LR(0)

Seja s o estado atual no topo da pilha de *parsing*:

1. Se o estado s contém um item da forma $A \to \alpha \; . \; X\beta$, onde X é um terminal:
   1. Se o próximo *token* da entrada é X: faça *shift* de X e empilhe o novo estado que contém o item $A \to \alpha \; X \; . \; \beta$.
   2. Se o próximo *token* da entrada não é X: erro de sintaxe.
2. Se o estado s contém um item completo (da forma $A \to \gamma \; .$), faça *reduce* por essa regra e:
   1. Se S' (o símbolo inicial) foi reduzido e a entrada acabou: *accept*.
   2. Se S' foi reduzido e a entrada não acabou: erro de sintaxe.
   3. Caso contrário:
      1. Desempilhe γ, revelando o estado antigo s_γ onde a construção de γ começou.
      2. Tome a transição $s_\gamma \xrightarrow{A} s'$, empilhando A e s'.

Algoritmo de *Parsing* LR(0) – Exemplo



$A \rightarrow (\; A \;) \mid \mathrm{a}$

	Parsing stack	Input	Action
1	\$ 0	((a)) \$	shift
2	\$ 0 (3	(a)) \$	shift
3	\$ 0 (3 (3	a)) \$	shift
4	\$ 0 (3 (3 a 2	)) \$	reduce $A \rightarrow$ a
5	\$ 0 (3 (3 A 4	)) \$	shift
6	\$ 0 (3 (3 A 4) 5	) \$	reduce $A \rightarrow (A)$
7	\$ 0 (3 A 4	) \$	shift
8	\$ 0 (3 A 4) 5	\$	reduce $A \rightarrow (A)$
9	\$ 0 A 1	\$	accept

Algoritmo de *Parsing* LR(0)

- O DFA de itens LR(0) e as ações especificadas pelo algoritmo de *parsing* LR(0) podem ser combinadas em uma tabela de *parsing*.
- Isso torna o método controlado por tabela, como outros já vistos anteriormente.
- *Exemplo*:

State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	$A' \rightarrow A$				
2	reduce	$A \rightarrow$ a				
3	shift		3	2		4
4	shift				5	
5	reduce	$A \rightarrow (A)$				

Próximo estado:
terminais – colunas *Input*, não-terminais – coluna *Goto*.

Gramática LR(0)

- Suponha um estado do DFA que contém os seguintes itens, onde X é um terminal.

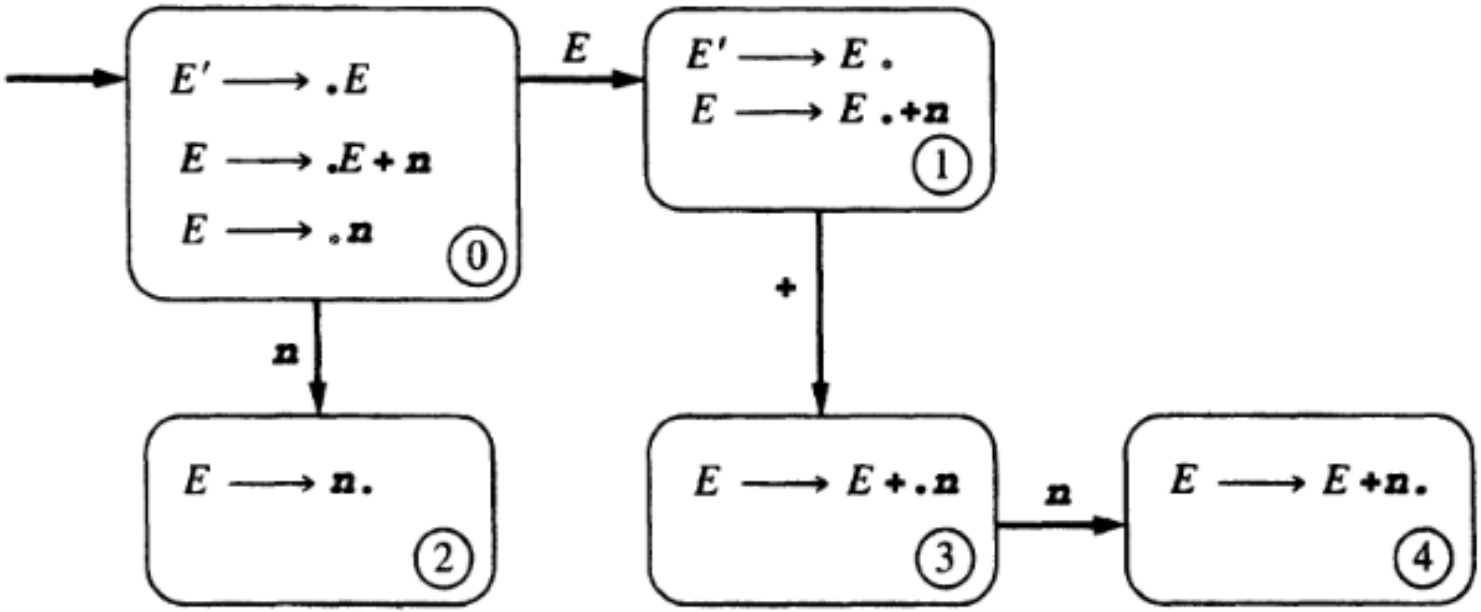
$$(1)\ A \to \alpha\ . \qquad (2)\ A \to \alpha\ .\ X\beta$$

- Seguindo o algoritmo de *parsing* LR(0), o item (2) nos diz para realizar uma ação de *shift*.
- Por outro lado, o item (1) indica uma ação de *reduce*.
- Essa indefinição sobre qual ação realizar caracteriza um conflito de *shift-reduce*.
- De forma similar, os dois itens abaixo em um mesmo estado do DFA, ilustram um conflito de *reduce-reduce*, pois não é possível decidir sobre qual regra usar.

$$(3)\ A \to \alpha\ . \qquad (4)\ B \to \alpha\ .$$

Gramática LR(0)

- Gramática LR(0): gramática livre de ambiguidades que causam conflitos de *shift-reduce* ou *reduce-reduce*.
- Na prática: um estado com um item completo não pode conter outros itens.
- A gramática de somas não é LR(0) pois o estado 1 contém um conflito de *shift-reduce*.



- Praticamente nenhuma gramática "real" é LR(0).

Parsing SLR(1)

- O *parsing* LR(1) simples, chamado SLR(1), usa o DFA de itens LR(0) como visto anteriormente.
- Além disto o algoritmo SLR(1) é significativamente mais poderoso que o LR(0) por usar um *look-ahead*.
- O *look-ahead* é combinado com o conjunto FOLLOW de um não-terminal para se decidir qual redução realizar.
- Essa modificação bastante simples no algoritmo de *parse* permite o reconhecimento de uma quantidade consideravelmente maior de linguagens.
- O algoritmo modificado é apresentado a seguir, com as alterações indicadas em **negrito**.

Algoritmo de *Parsing* SLR(1)

Seja s o estado atual no topo da pilha de *parsing*:

1. Se o estado s contém um item da forma $A \to \alpha \,.\, X\beta$, onde X é um terminal:
   1. Se o próximo *token* da entrada é X: faça *shift* de X e empilhe o novo estado que contém o item $A \to \alpha\, X \,.\, \beta$.
   2. Se o próximo *token* da entrada não é X: erro de sintaxe.
2. Se o estado s contém um item completo $A \to \gamma\,.$ **e o próximo *token* da entrada está em FOLLOW(A)**, faça *reduce* e:
   1. Se S' foi reduzido e a entrada acabou: *accept*.
   2. Se S' foi reduzido e a entrada não acabou: erro de sintaxe.
   3. Caso contrário:
      1. Desempilhe γ, revelando o estado antigo s_γ onde a construção de γ começou.
      2. Tome a transição $s_\gamma \xrightarrow{A} s'$, empilhando A e s'.
3. **Se o próximo *token* da entrada não estiver em FOLLOW(A): erro de sintaxe.**

Gramática SLR(1)

- Uma gramática é SLR(1) se, e somente se, para qualquer estado *s*, as seguintes condições forem satisfeitas:
  1. Para qualquer item $A \to \alpha \,.\, X\beta$ em *s*, onde *X* é um terminal, não existe em *s* um item completo $B \to \alpha \,.$ com *X* em FOLLOW(*B*).
  2. Para dois itens completos $A \to \alpha \,.$ e $B \to \alpha \,.$ em *s*, FOLLOW(*A*) $\cap$ FOLLOW(*B*) $= \emptyset$.
- Violação da condição 1 é um conflito de *shift-reduce*.
- Gramática pode ser "reparada" em alguns casos estabelecendo que *shift* tem prioridade sobre *reduce*. Isso resolve o problema do *else* pendente.
- Violação da condição 2 é um conflito de *reduce-reduce*.
- Conflitos desse tipo em geral indicam um erro na gramática.
- A gramática de somas apresentada anteriormente não é LR(0) mas é SLR(1).

Limitações do *Parsing* SLR(1)

- Considere a gramática de comandos em Pascal abaixo e a sua versão simplificada à direita.

$$
\begin{array}{l}
\textit{stmt} \rightarrow \textit{call-stmt} \mid \textit{assign-stmt} \\
\textit{call-stmt} \rightarrow \textbf{identifier} \\
\textit{assign-stmt} \rightarrow \textit{var} \ \texttt{:=} \ \textit{exp} \\
\textit{var} \rightarrow \textit{var} \ \texttt{[} \ \textit{exp} \ \texttt{]} \mid \textbf{identifier} \\
\textit{exp} \rightarrow \textit{var} \mid \textbf{number}
\end{array}
\qquad
\begin{array}{l}
S \rightarrow \textbf{id} \mid V \ \texttt{:=} \ E \\
V \rightarrow \textbf{id} \\
E \rightarrow V \mid \textbf{n}
\end{array}
$$

- Não é possível reconhecer a linguagem dessa gramática usando *parsing* SLR(1). (A gramática não é SLR(1).)
- O DFA de itens possui um estado contendo os itens $S \rightarrow \texttt{id} \ .$ e $V \rightarrow \texttt{id} \ .$ e temos que FOLLOW(S) $= \{\$\}$ e FOLLOW(V) $= \{\texttt{:=}, \$\}$. $\Rightarrow$ Conflito de *reduce-reduce*.
- Isso é um problema pois esse tipo de construção é comum em linguagens de programação.
- $\Rightarrow$ Precisamos de um método de *parsing* LR mais poderoso.

Parsing LR(1) Canônico

O algoritmo de *parse* SLR(1):

- Constrói um DFA de itens LR(0).
- O *look-ahead* só é utilizado durante a execução do *parser*: operações de *reduce* testam se o *look-ahead* está no conjunto FOLLOW.
- Método não pode ser aplicado para dois itens LR(0) $A \to \alpha\,.$ e $B \to \alpha\,.$ com FOLLOW(A) $\cap$ FOLLOW(B) $\neq \emptyset$.

Parsing LR(1) Canônico

O algoritmo de *parse* LR(1) geral (canônico):

- Constrói um DFA de itens LR(1).
- Um item LR(1) tem a forma $[A \rightarrow \alpha \ . \ \beta, a]$, onde $A \rightarrow \alpha \ . \ \beta$ é um item LR(0) e a é um *token* de *look-ahead*.
- Para um item completo $[A \rightarrow \alpha\beta. \ , a]$: operações de *reduce* testam se o *look-ahead* é a.
- É essencial destacar que a é o *look-ahead* esperado quando o item estiver completo.
- Os *tokens* dos itens LR(1) só servem para distinguir entre as regras que estão "habilitadas" para redução. Estes *tokens* não influenciam as operações de *shift*.
- Embutir o *look-ahead* na construção do DFA torna o método mais poderoso pois permite relaxar a restrição do algoritmo SLR(1).

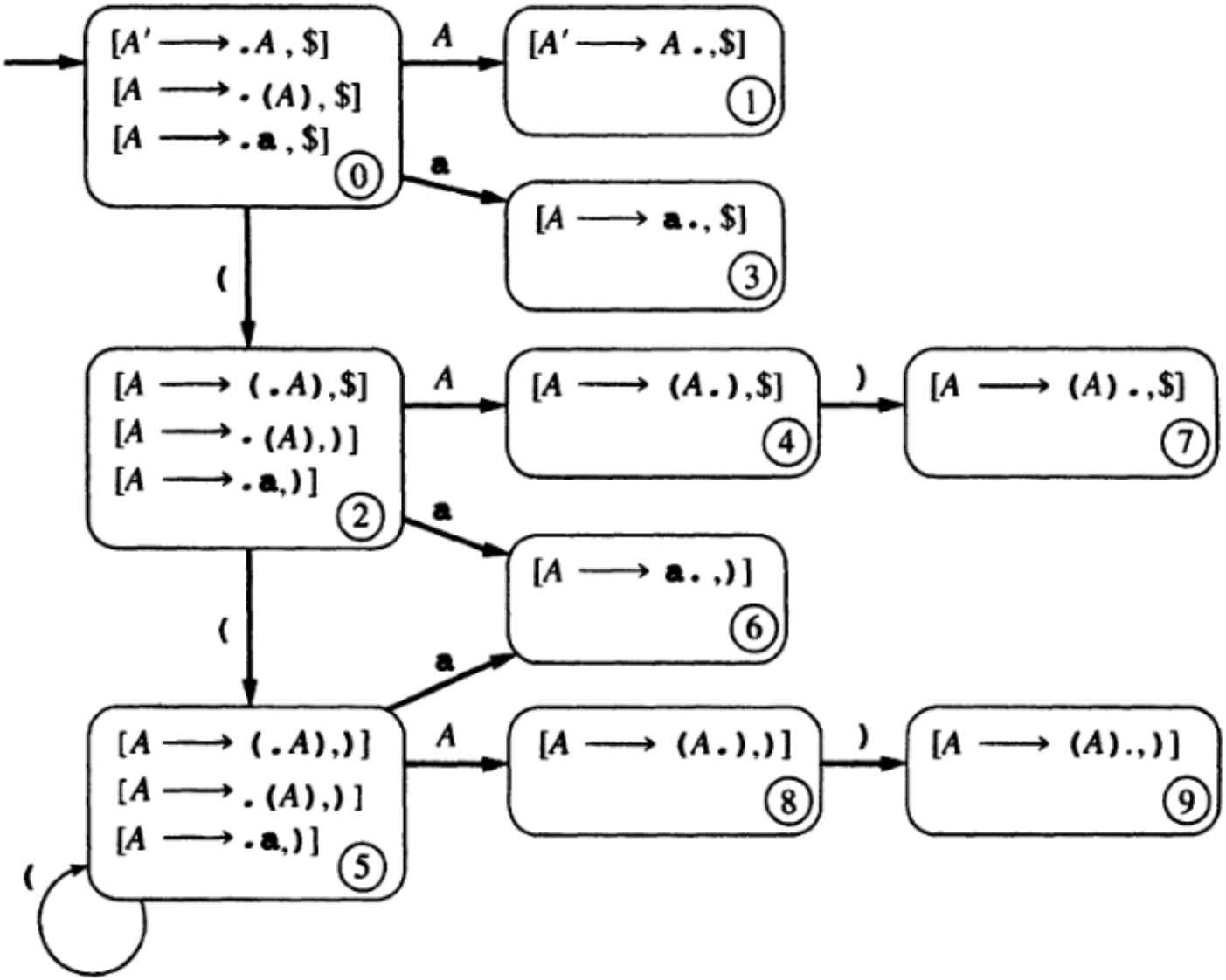
Parsing LR(1) Canônico

Construindo o NFA de itens LR(1):

- Similar à construção do NFA de itens LR(0).
- Requer uma gramática aumentada como anteriormente.
- Estados do NFA são conjuntos de itens LR(1). Estado inicial contém $[S' \to S, \$]$.
- Definição das transições:
  1. Temos que $[A \to \alpha \,.\, X\beta, a] \xrightarrow{X} [A \to \alpha X \,.\, \beta, a]$, para qualquer símbolo $X \in (N \cup T)$.
  2. Dado um item $[A \to \alpha \,.\, B\beta, a]$ onde B é um não-terminal, há transições-ϵ para itens $[B \to .\gamma, b]$, para toda regra $B \to \gamma$ e todo *token* b em FIRST(βa).
- No caso 1, se X é um terminal, temos uma operação de *shift*. É essencial notar que não é necessário que $X = a$, pois a indica qual deve ser o *look-ahead* no momento da futura operação de redução.

Parsing LR(1) Canônico

DFA de itens LR(1) para a gramática $A \to (A) \mid \texttt{a}$.



Algoritmo de *Parsing* LR(1)

Seja s o estado atual no topo da pilha de *parsing*:

1. Se o estado s contém um item da forma $[A \to \alpha \,.\, X\beta, a]$, onde X é um terminal:
   1. Se o próximo *token* da entrada é X: faça *shift* de X e empilhe o novo estado que contém o item $[A \to \alpha\, X \,.\, \beta, a]$.
   2. Se o próximo *token* da entrada não é X: erro de sintaxe.
2. Se o estado s contém um item completo $[A \to \gamma.\,, a]$ **e o próximo *token* da entrada é** a, faça *reduce* e:
   1. Se S' foi reduzido e a entrada acabou: *accept*.
   2. Se S' foi reduzido e a entrada não acabou: erro de sintaxe.
   3. Caso contrário:
      1. Desempilhe γ, revelando o estado antigo s_γ onde a construção de γ começou.
      2. Tome a transição $s_\gamma \xrightarrow{A} s'$, empilhando A e s'.
3. **Se o próximo *token* da entrada não for a: erro de sintaxe.**

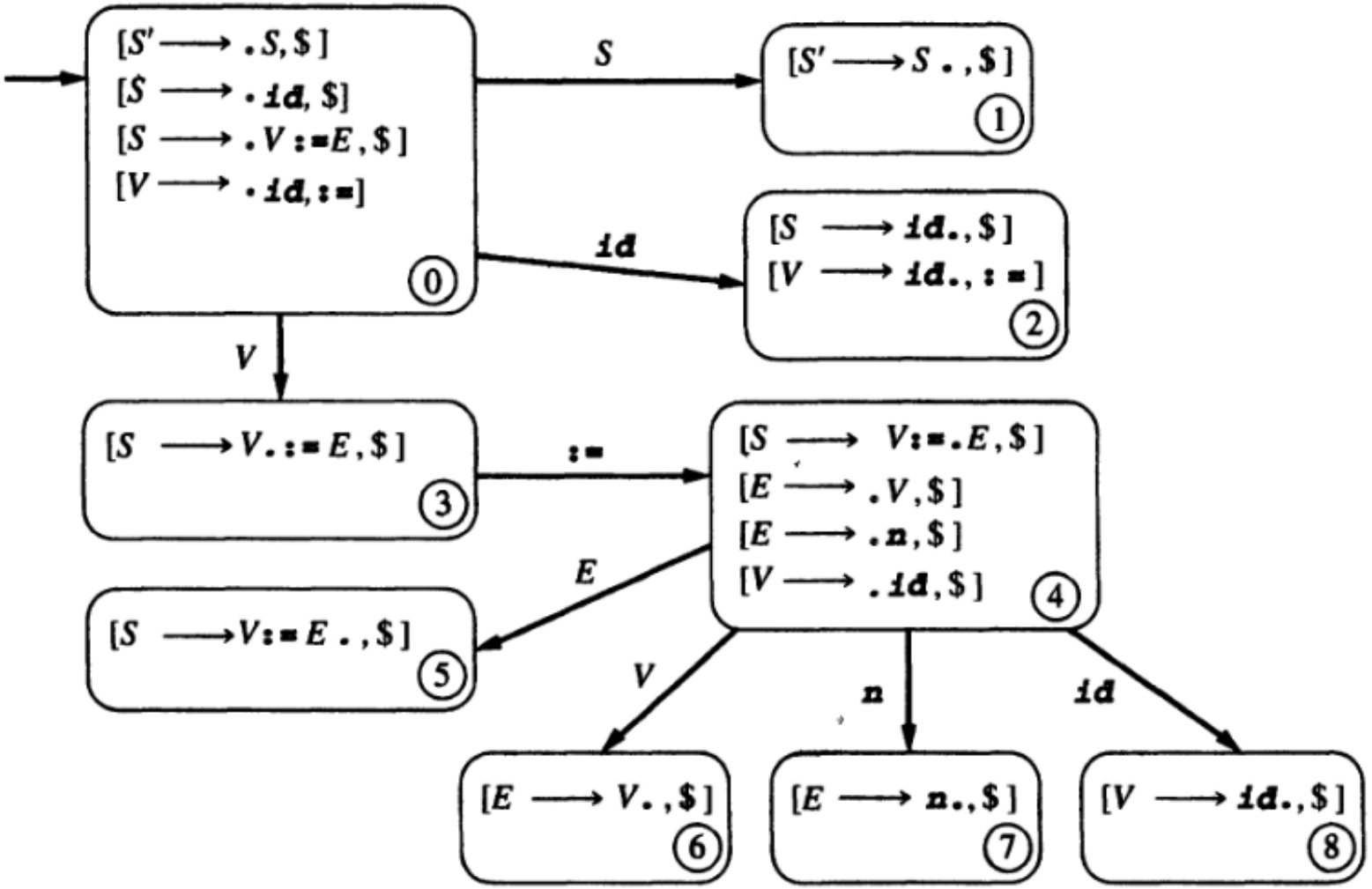
Gramática LR(1)

- Uma gramática é LR(1) se, e somente se, para qualquer estado *s*, as seguintes condições forem satisfeitas:
  1. Para qualquer item $[A \to \alpha \,.\, X\beta, a]$ em *s*, onde *X* é um terminal, não existe em *s* um item completo $[B \to \alpha \,.\,, X]$. (Caso contrário é um conflito de *shift-reduce*.)
  2. Não existem dois itens em *s* da forma $[A \to \alpha \,.\,, a]$ e $[B \to \alpha \,.\,, a]$. (Caso contrário é um conflito de *reduce-reduce*.)
- Próximo slide apresenta o DFA de itens LR(1) para a gramática que não é tratável pelo método SLR(1).
- Em geral, o DFA de itens LR(1) é bem maior (fator de 10) que o DFA de itens LR(0).
- O tamanho do DFA afeta diretamente o tamanho da tabela de *parsing*.
- Preocupações com eficiência levaram ao desenvolvimento do método LALR(1).

Parsing LR(1) Canônico

$S \rightarrow \mathbf{id} \mid V := E$
$V \rightarrow \mathbf{id}$
$E \rightarrow V \mid \mathbf{n}$

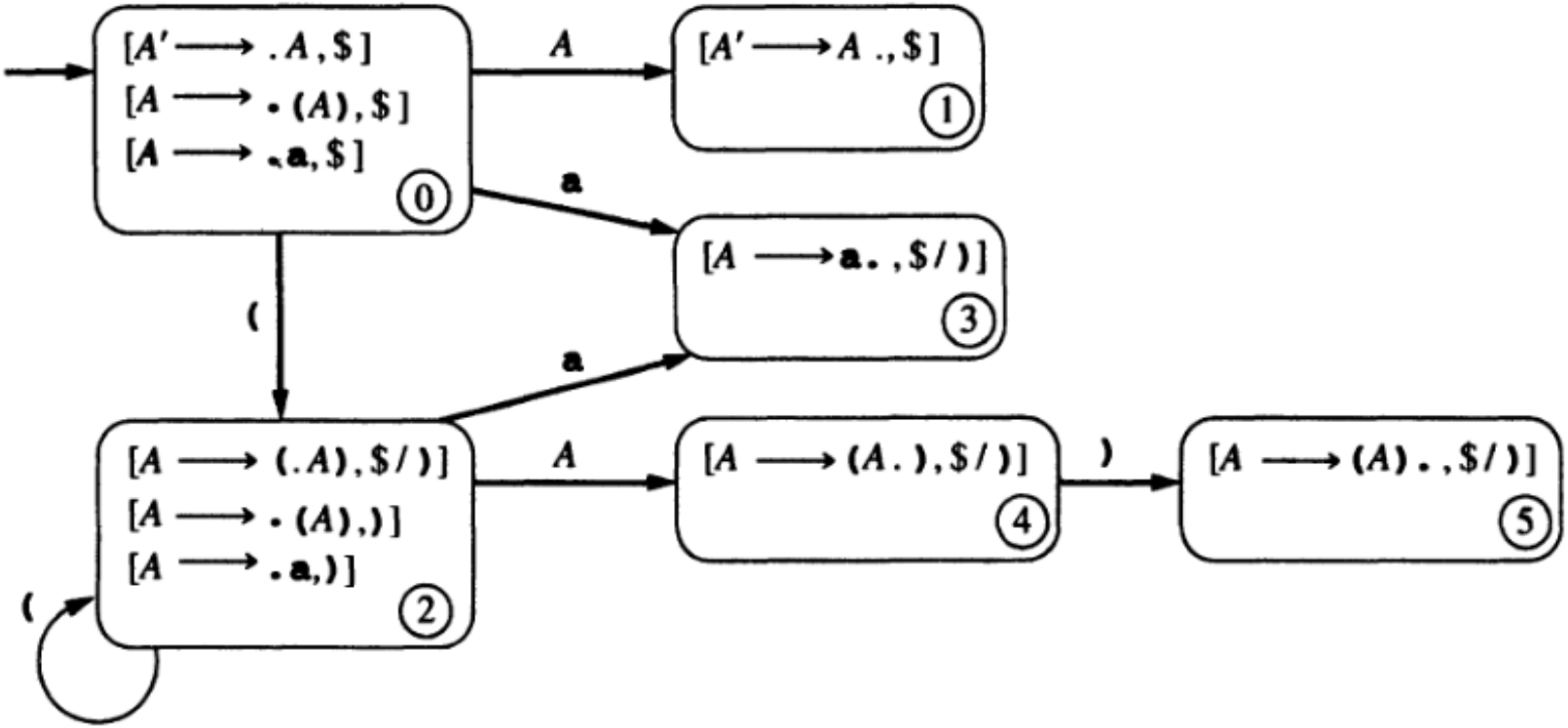
DFA de itens LR(1) para a gramática



Parsing LALR(1)

- No DFA de itens LR(1), muitos estados distintos são diferenciados apenas pelo *token* de *look-ahead*, i.e., esses estados possuem o mesmo item LR(0).
- O algoritmo de *parse* LALR(1) identifica estes estados e combina os *look-aheads*.
- Assim, o DFA contém itens da forma $[A \rightarrow \alpha \ . \ \beta, a/b/c]$.
- *Parsing* LALR(1) possui praticamente o mesmo poder de reconhecimento que o LR(1) canônico e ao mesmo tempo utiliza um DFA menor.
- Por conta disso, LALR(1) é o método preferido para a implementação de *parsers bottom-up*.
- O Bison gera *parsers* LALR(1).

DFA de itens LALR(1) para a gramática $A \to (A) \mid \texttt{a}$.



DFA de itens LR(1) tinha 10 estados, já o DFA acima tem o mesmo número de estados que o DFA de itens LR(0).

Parsing LALR(1)

- O algoritmo de *parsing* LALR(1) é o mesmo que o algoritmo de *parsing* LR(1) geral.
- A construção LALR(1) pode introduzir conflitos que não existem no método LR(1) geral, mas isso raramente acontece na prática.
- Se uma gramática é LR(1), o método LALR(1) não introduz conflitos de *shift-reduce* mas pode introduzir conflitos de *reduce-reduce*.
- Gramáticas LR(1) que não são LALR(1) geralmente não ocorrem em LPs. (Veja exemplo no exercício 5.2 do livro.)
- Vale destacar que não é necessário construir o DFA de itens LR(1) para depois condensá-lo em itens LALR(1).
- O DFA de itens LALR(1) pode ser construído diretamente a partir do DFA de itens LR(0) através de um processo chamado propagação de *look-aheads*.

Sumário

Comparativo entre os métodos *bottom-up* e *top-down*:

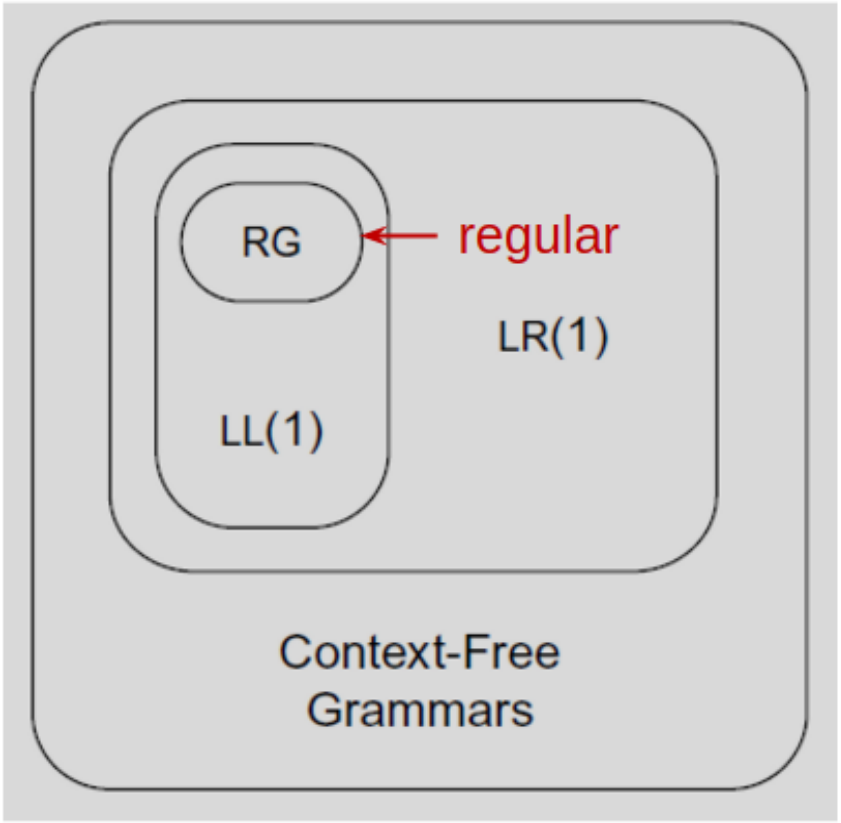
	Método LR(1)	**Método LL(1)**
Gramática	sem restrições	sem recursão à esquerda
Tabela	estados $\times$ símbolos (grande)	terminais $\times$ não-terminais (pequena)
Pilha	símbolos e estados	somente símbolos

Sumário

- Poder de reconhecimento dos algoritmos de *parsing* LR:

$$\text{LR(0)} < \text{SLR(1)} < \text{LALR(1)} < \text{LR(1)} \quad .$$

- Poder de expressão das gramáticas:



Aula 05 – *Bottom-Up Parsing*

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

2016/2 – Compiladores
***Compiler Construction* (CC)**