Análise Sintática

Sandro Rigo
sandro@ic.unicamp.br

LR Parsing

- O ponto fraco da técnica LL(k) é precisar prever que produção usar
  - Com base nos primeiros k tokens do lado direito da produção
- LR(k) posterga a decisão até ter visto todo o lado direito de uma produção, mais os k próximos tokens da entrada
  - Left-to-right parse, rightmost-derivation, k-token lookahead

LR Parsing

- O parser tem uma pilha e a entrada.
- Os primeiros k tokens da entrada formam o *lookahead*
- Dois tipos de ações:
  - SHIFT: move o primeiro token para o topo da pilha
  - REDUCE:
    - escolhe uma produção $X \rightarrow A\ B\ C$;
    - desempilha *C, B, e A*;
    - empilha X

Exemplo

$$0.\ S' \rightarrow S\$;$$

$$1. S \rightarrow S; S$$

$$2. S \rightarrow \text{id} := E$$

$$3. S \rightarrow \text{print}\ (L)$$

$$4. E \rightarrow \text{id}$$

$$5. E \rightarrow \text{num}$$

$$6. E \rightarrow E + E$$

$$7. E \rightarrow (S, E)$$

$$8. L \rightarrow E$$

$$9. L \rightarrow L, E$$

Derivação para:

```
a : = 7;
b : = c + (d : = 5 + 6, d)
```

Exemplo

Stack	*Input*	*Action*
${}_1$	`a := 7 ; b := c + ( d := 5 + 6 , d ) $`	*shift*
${}_1\ \mathrm{id}_4$	`:= 7 ; b := c + ( d := 5 + 6 , d ) $`	*shift*
${}_1\ \mathrm{id}_4\ :=_6$	`7 ; b := c + ( d := 5 + 6 , d ) $`	*shift*
${}_1\ \mathrm{id}_4\ :=_6\ \mathrm{num}_{10}$	`; b := c + ( d := 5 + 6 , d ) $`	*reduce* $E \rightarrow \mathrm{num}$
${}_1\ \mathrm{id}_4\ :=_6\ E_{11}$	`; b := c + ( d := 5 + 6 , d ) $`	*reduce* $S \rightarrow \mathrm{id} := E$
${}_1\ S_2$	`; b := c + ( d := 5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3$	`b := c + ( d := 5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4$	`:= c + ( d := 5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6$	`c + ( d := 5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ \mathrm{id}_{20}$	`+ ( d := 5 + 6 , d ) $`	*reduce* $E \rightarrow \mathrm{id}$
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}$	`+ ( d := 5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}$	`( d := 5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8$	`d := 5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ \mathrm{id}_4$	`:= 5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ \mathrm{id}_4\ :=_6$	`5 + 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ \mathrm{id}_4\ :=_6\ \mathrm{num}_{10}$	`+ 6 , d ) $`	*reduce* $E \rightarrow \mathrm{num}$
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ \mathrm{id}_4\ :=_6\ E_{11}$	`+ 6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}$	`6 , d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ \mathrm{num}_{10}$	`, d ) $`	*reduce* $E \rightarrow \mathrm{num}$
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ E_{17}$	`, d ) $`	*reduce* $E \rightarrow E + E$
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ \mathrm{id}_4\ :=_6\ E_{11}$	`, d ) $`	*reduce* $S \rightarrow \mathrm{id} := E$
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ S_{12}$	`, d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ S_{12}\ ,_{18}$	`d ) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ S_{12}\ ,_{18}\ \mathrm{id}_{20}$	`) $`	*reduce* $E \rightarrow \mathrm{id}$
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ S_{12}\ ,_{18}\ E_{21}$	`) $`	*shift*
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ (_8\ S_{12}\ ,_{18}\ E_{21}\ )_{22}$	`$`	*reduce* $E \rightarrow (S, E)$
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ E_{17}$	`$`	*reduce* $E \rightarrow E + E$
${}_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}$	`$`	*reduce* $S \rightarrow \mathrm{id} := E$
${}_1\ S_2\ ;_3\ S_5$	`$`	*reduce* $S \rightarrow S; S$
${}_1\ S_2$	`$`	*accept*

LR Parsing Engine

- Como o parser sabe quando fazer um shift ou um reduce?
- Usando um DFA aplicado a pilha!
- As arestas são nomeadas com os símbolos que podem aparecer na pilha

Exemplo

	id	num	print	;	,	+	:=	(	)	$	*S*	*E*	*L*
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

Tabela de Transição

- 4 tipos de ações:
  - **s***n:* Shift para o estado *n*;
  - **g***n:* Vá para o estado *n*;
  - **r***k:* Reduza pela regra *k*;
  - **a:** Accept;
  - : Error (entrada em branco).
- As arestas do DFA são as ações shift e goto
- No exemplo anterior, cada número indica o estado destino

Algoritmo

- Look up top stack state, and input symbol, to get action; If action is
- Shift(*n*):Advance input one token; push *n* on stack.
- Reduce(*k*):
  - Pop stack as many times as the number of symbols on the right-hand side of rule *k*;
  - Let *X* be the left-hand-side symbol of rule *k*;
    In the state now on top of stack, look up *X* to get "goto *n*";
    Push *n* on top of stack.
- Accept:Stop parsing, report success.
- Error:Stop parsing, report failure.

Geração de Parsers LR(0)

- O exemplo anterior mostrou o uso de 1 símbolo de lookahead
- Para k, a tabela terá colunas para todas as seqüências de k tokens
- K>1 praticamente não é usado para compilação
- Maioria das linguagens de programação podem ser descritas por gramáticas LR(1)

Geração de Parsers LR(0)

- LR(0) são as gramáticas que podem ser analisadas olhando somente a pilha

- $S' \rightarrow S\$$
1. $S \rightarrow (L)$
2. $S \rightarrow x$
3. $L \rightarrow S$
4. $L \rightarrow L, S$

Estados

$$\begin{array}{l} S' \rightarrow .S\$ \\ S \rightarrow .x \\ S \rightarrow .(L) \end{array}$$

1

- *Estado Inicial*

$$S \rightarrow x.$$

2

$$\begin{array}{l} S \rightarrow (.L) \\ L \rightarrow .L , S \\ L \rightarrow .S \\ S \rightarrow .(L) \\ S \rightarrow .x \end{array}$$

3

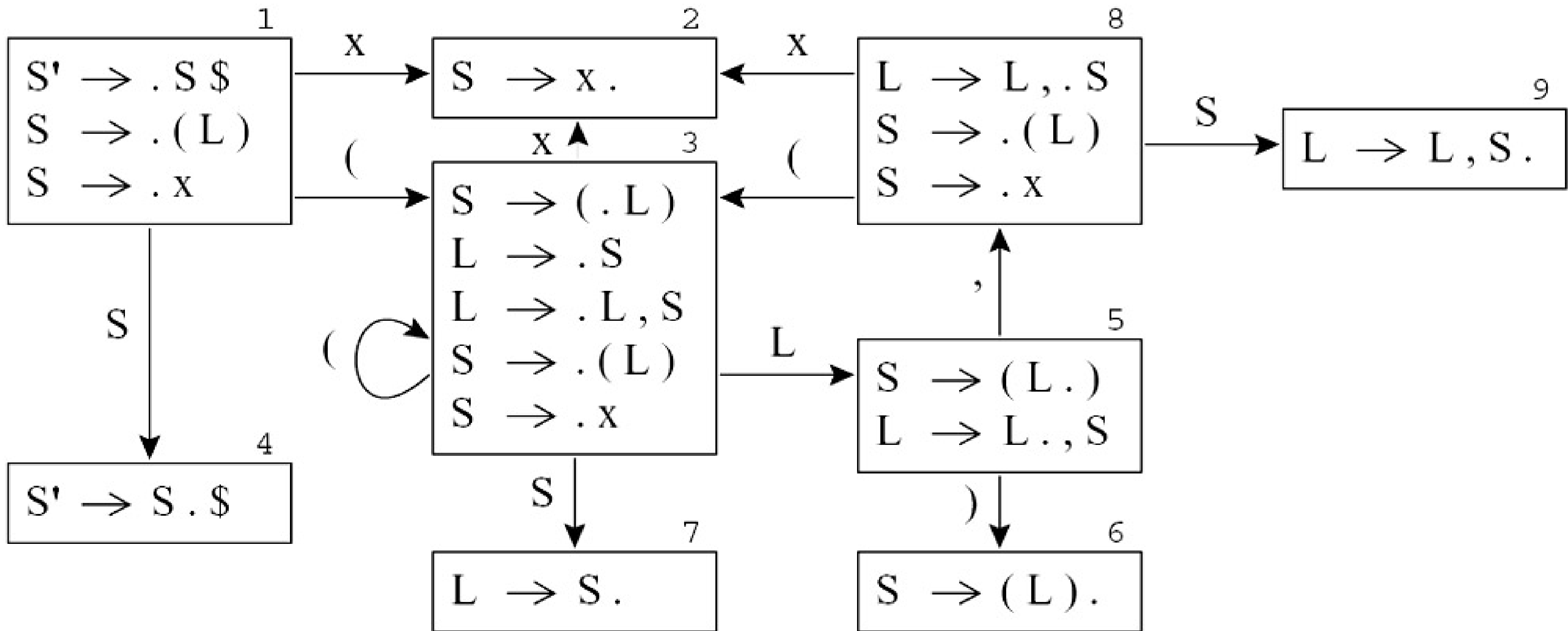
- *Shift de "x" e "("*
- Estado 2 permite reduce

Estados

- *Goto Action:*
  - Imagine um shift de x ou “(“ no estado 1 seguido de redução pela produção de S correspondente.
  - Todos os símbolos do lado direito da produção serão desempilhados e o parser vai executar um goto para S no estado 1.
  - Isso se representa movendo-se o ponto para após o S e colocando este item em um novo estado (4)

$$S' \rightarrow S.\$$$ 4

Exemplo



Algoritmos

- Closure(I): adiciona itens a um estado quando um "." precede um não terminal
- Goto(I,X): movimenta o "." para depois de X em todos os itens

```
Closure(I) =
 repeat
  for any item A → α.Xβ in I
    for any production X → γ
      I ← I ∩ {X → .γ}
 until I does not change.
 return I
```

```
Goto(I, X) =
   set J to the empty set
   for any item A → α.Xβ in I
       add A → αX.β to J
   return Closure(J)
```

Algoritmos

- Construção do parser LR(0)

```
Initialize T to {Closure({S' → .S$})}
Initialize E to empty.
repeat
 for each state I in T
   for each item A → α.Xβ in I
      let J be Goto(I, X)
      T ← T U {J}
      E ← E U {I ->X J}
until E and T did not change in this iteration
```

Exemplo

	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

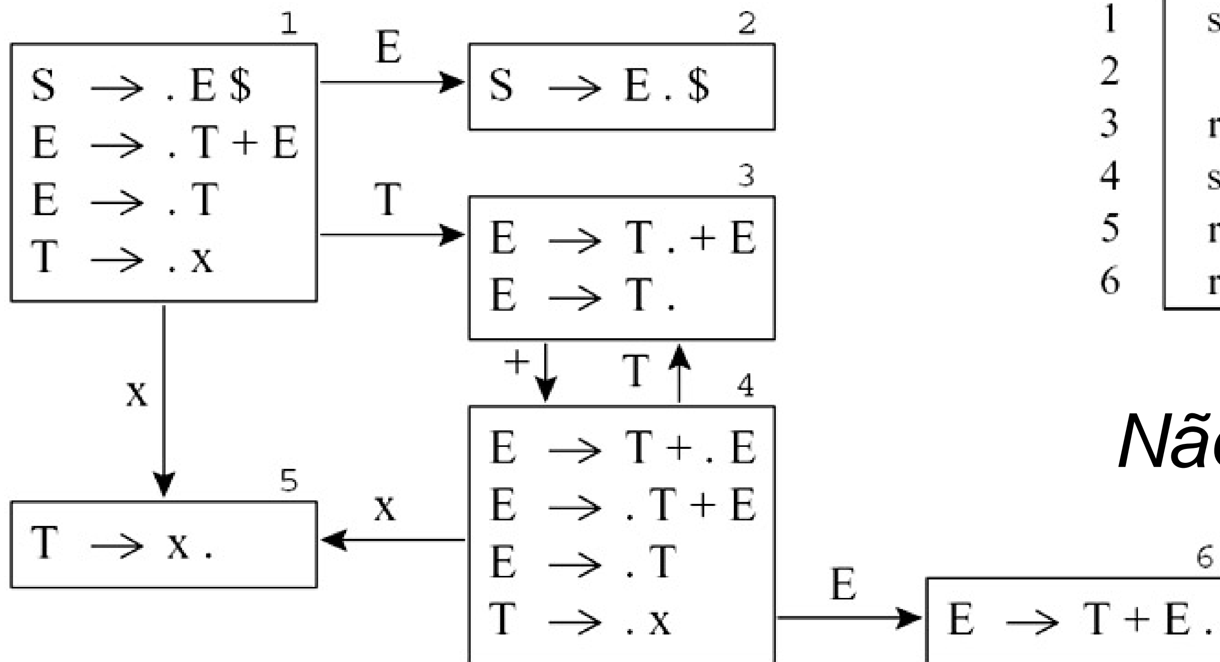
Tente construir um parser LR(0)

- $S \rightarrow E\,\$$

1. $E \rightarrow T + E$
2. $E \rightarrow T$
3. $T \rightarrow x$

SLR Parser

- $S \rightarrow E\,\$$
1. $E \rightarrow T + E$
2. $E \rightarrow T$
3. $T \rightarrow x$



	x	+	$	E	T
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

Não é LR(0)!!!

SLR Parser

- Colocar reduções somente onde indicado pelo conjunto FOLLOW
- Ex: FOLLOW(E) = {$}

	x	+	$	*E*	*T*
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

É SLR!!!

LR(1)

- Mais poderoso do que SLR
- Maioria das linguagens de programação são LR(1)

- Algoritmo similar a LR(0)
- Item: $(A \rightarrow \alpha.\beta, x)$ — **Lookahead**

Algoritmos - Closure(I) e Goto(I,X)

```
Closure(I) =
 repeat
  for any item A → (α.Xβ,z) in I
    for any production X → γ
      for any w ∈ FIRST(βz)
        I ← I ∩ {X → .γ,w}
 until I does not change.
 return I
```

```
Goto(I, X) =

set J to the empty set

for any item A → (α.Xβ,z) in I

add A → (αX.β,z) to J

return Closure(J)
```

Exemplo

- $S' \rightarrow S\ \$$
1. $S \rightarrow V = E$
2. $S \rightarrow E$
3. $E \rightarrow V$
4. $V \rightarrow x$
5. $V \rightarrow * E$

É SLR???

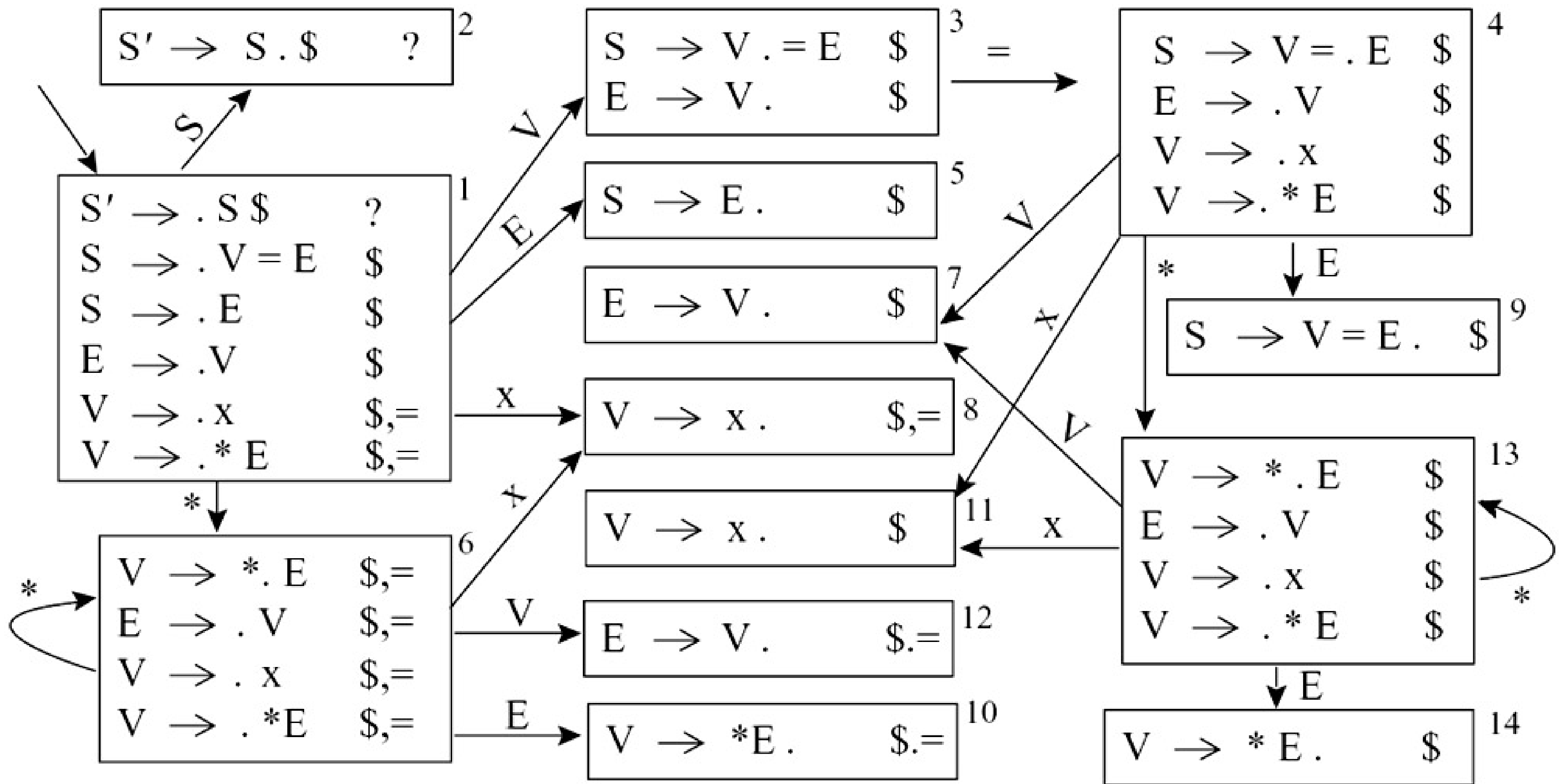
Exemplo

- $S' \rightarrow S\ \$$
1. $S \rightarrow V = E$
2. $S \rightarrow E$
3. $E \rightarrow V$
4. $V \rightarrow x$
5. $V \rightarrow * E$

S'	$\rightarrow$	$.\ S\ \$$	?
S	$\rightarrow$	$.\ V = E$	\$
S	$\rightarrow$	$.\ E$	\$
E	$\rightarrow$	$.\ V$	\$
V	$\rightarrow$	$.\ x$	\$
V	$\rightarrow$	$.\ * E$	\$
V	$\rightarrow$	$.\ x$	=
V	$\rightarrow$	$.\ * E$	=

S'	$\rightarrow$	$.\ S\ \$$	?
S	$\rightarrow$	$.\ V = E$	\$
S	$\rightarrow$	$.\ E$	\$
E	$\rightarrow$	$.\ V$	\$
V	$\rightarrow$	$.\ x$	\$, =
V	$\rightarrow$	$.\ * E$	\$, =

Exemplo



Exemplo

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

(a) LR(1)

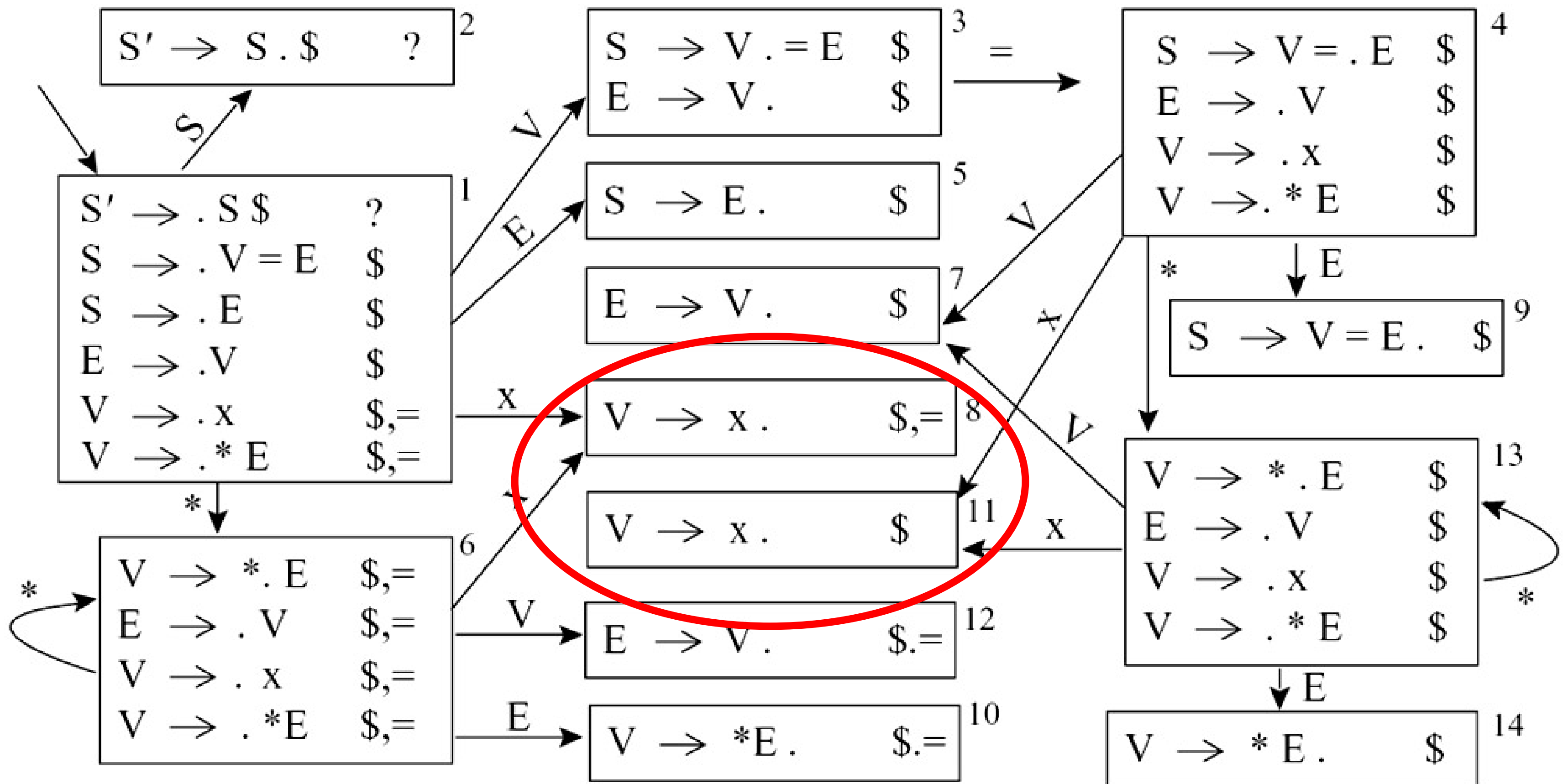
```
R ← {}
for each state I in T
 for each item (A → α., z) in I
  R ← R U{(I, z, A → α)}
```

Parser LALR(1)

- O tamanho das tabelas LR(1) pode ser muito grande
- É possível reduzir unindo estados do DFA
  - Junte os estados que possuam os itens idênticos, exceto pelo lookahead
- Vejamos o exemplo anterior

Parser LALR(1)

Mais algum???



Exemplo

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

(a) LR(1)

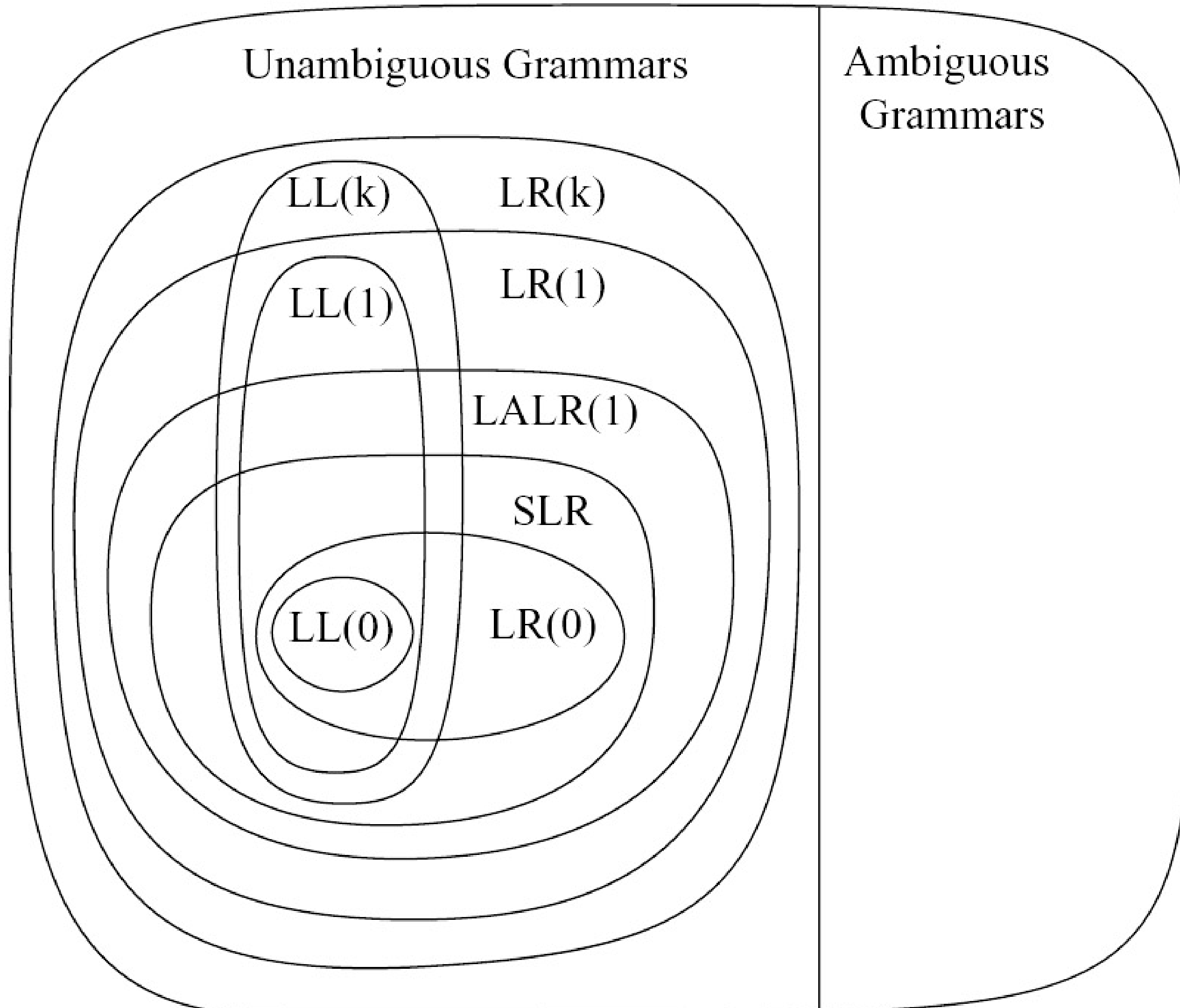
	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

(b) LALR(1)

Parser LALR(1)

- Pode gerar uma tabela com conflitos, onde a LR(1) não possuia
  - Na prática, o efeito de redução no uso de memória é bastante desejável
- A maioria das linguagens de programação é LALR(1)
- É o tipo mais usado em geradores automáticos de parser

Hierarquia das Gramáticas



Ambiguidade

- $S \rightarrow$ if E then S else S
- $S \rightarrow$ if E then S
- $S \rightarrow$ other

Como seria a parser tree para:

```
if a then if b then s1 else s2
```

Ambiguidade

- $S \rightarrow$ if E then S else S
- $S \rightarrow$ if E then S
- $S \rightarrow$ other

- *(1)* `if a then { if b then s1 else s2 }`
- *(2)* `if a then { if b then s1 } else s2`

$S \rightarrow$ if E then S .	else
$S \rightarrow$ if E then S . else S	*(any)*

Conflito shift-reduce !

Eliminando

- $S \rightarrow M$
- $S \rightarrow U$
- $M \rightarrow$ if E then M else M
- $M \rightarrow$ other
- $U \rightarrow$ if E then S
- $U \rightarrow$ if E then M else U

Como seria a parser tree para:

```
if a then if b then s1 else s2
```

Eliminando

- Pode-se usar a gramática ambígua, decidindo os conflitos sempre por shift em casos desse tipo
- Somente aconselhável em casos bem conhecidos

Diretivas de Precedência

- Nenhuma gramática ambígua é LR(k), para nenhum k
- Podemos usá-las se encontrarmos uma maneira de resolver os conflitos
- Relembrando um exemplo anterior ...

Diretivas de Precedência

- $E \rightarrow \text{id}$
- $E \rightarrow \text{num}$
- $E \rightarrow E * E$
- $E \rightarrow E / E$
- $E \rightarrow E + E$
- $E \rightarrow E - E$
- $E \rightarrow (E)$

- $E \rightarrow E + T$
- $E \rightarrow E - T$
- $E \rightarrow T$
- $T \rightarrow T * F$
- $T \rightarrow T / F$
- $T \rightarrow F$
- $F \rightarrow \text{id}$
- $F \rightarrow \text{num}$
- $F \rightarrow (E)$

Diretivas de Precedência

Tabela LR(1): muitos conflitos

	id	num	+	-	*	/	(	)	$	E
1	s2	s3					s4			g7
2			r1	r1	r1	r1		r1	r1	
3			r2	r2	r2	r2		r2	r2	
4	s2	s3					s4			g5
5								s6		
6			r7	r7	r7	r7		r7	r7	
7			s8	s10	s12	s14			a	
8	s2	s3					s4			g9
9			s8,r5	s10,r5	s12,r5	s14,r5		r5	r5	
10	s2	s3					s4			g11
11			s8,r6	s10,r6	s12,r6	s14,r6		r6	r6	
12	s2	s3					s4			g13
13			s8,r3	s10,r3	s12,r3	s14,r3		r3	r3	
14	s2	s3					s4			g15
15			s8,r4	s10,r4	s12,r4	s14,r4		r4	r4	

Diretivas de Precedência

Vejamos o estado 13:

$E \rightarrow E * E .$	$+$
$E \rightarrow E . + E$	*(any)*

Pilha:

. . . E * E (+)

Shift:

. . . E*E+

Chegando a:

. . . E*E+E

Reduzindo para: . . . E*E

Pilha:

. . . E * E (+)

Reduce:

. . . E (+)

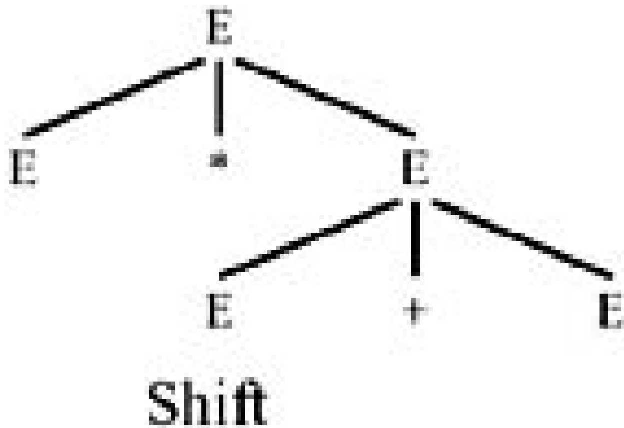
Chegando a:

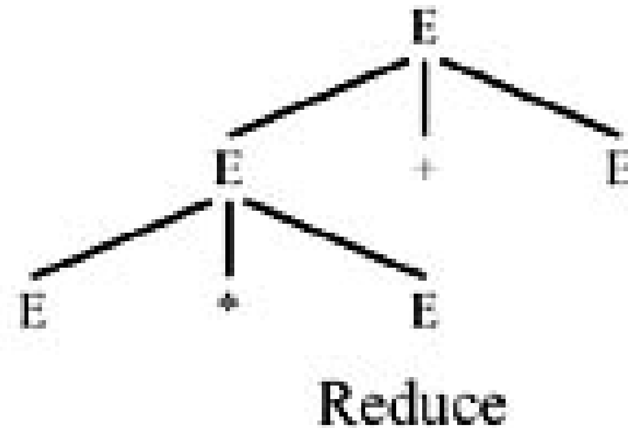
. . . E+

Diretivas de Precedência

Vejamos o estado 13:

$E \rightarrow E * E .$	$+$
$E \rightarrow E . + E$	*(any)*





Qual queremos?

Diretivas de Precedência

```
precedence nonassoc EQ, NEQ;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
precedence right EXP;
```

Indicaria que:

- + e – têm igual precedência e são associativos à esquerda
- * e / são associativos à esquerda e têm maior precedência que + e -

$E \rightarrow E * E \,.$	$+$
$E \rightarrow E \,.\, + E$	*(any)*

Como seria resolvido?

Diretivas de Precedência

- Podem ajudar
- Não devem ser abusivamente utilizadas
- Se não consegue explicar ou bolar um uso de precedências que resolva o seu problema, reescreva a gramática!

Sintaxe vs Semântica

- Imagine uma linguagem que aceita
  - Expressões aritméticas do tipo:
    - $x + y$
  - Expressões booleanas do tipo:
    - $a \& (b = c)$
- Operadores aritméticos têm maior precedência que os booleanos
- Expressões booleanas não podem ser somadas a aritméticas

Sintaxe vs Semântica

```
%token ID ASSIGN PLUS MINUS AND EQUAL
%start stm
%left OR
%left AND
%left PLUS
%%
stm : ID ASSIGN ae
   | ID ASSIGN be
be : be OR be
   | be AND be
   | ae EQUAL ae |
    ID
ae : ae PLUS ae | ID
```

Tem algum problema?

Sintaxe vs Semântica

- Conflito reduce/reduce
- O parser não tem como diferenciar variáveis booleanas de aritméticas
  - Sintaticamente são idênticas
- Esse tipo de análise deve ser deixado para a fase semântica

Sintaxe vs Semântica

- $S \rightarrow \text{id} : E$
- $E \rightarrow \text{id}$
- $E \rightarrow E \,\&\, E$
- $E \rightarrow E = E$
- $E \rightarrow E + E$

Agora:

a + 5&b

será considerada legal pelo parser. Fases seguintes do compilador devem reportar o erro.

Recuperação de Erros

- Tentativa de reportar o maior número de erros
- Mecanismos existentes variam de gerador para gerador
- Verifique a documentação do gerador em uso

Recuperação de Erros Local

- Ajusta a pilha do parser no ponto onde o erro é detectado
- O parser continua a partir deste ponto
- Exemplo: YACC
  - Usa um símbolo especial: error
  - Controla o processo de recuperação
  - Onde ele aparecer na gramática, podem existir símbolos errôneos sendo consumidos

Recuperação de Erros Local

- *exp* → *ID*
- *exp* → *exp* + *ext*
- *exp* → (*exps*)
- *exps* → *exp*
- *exps* → *exps* ; *exp*
- *exp* → (*error*)
- *exps* → *error* ; *exp*

Recuperação de Erros Local

- A idéia é definir o fecha parênteses e o ponto-e-vírgula como tokens de sincronização
- Erros no meio de uma expressão causam avanço até o próximo token de sincronização
- error é um terminal: na tabela aparecem ações de shift

Recuperação de Erros Local

- Quando um erro é encontrado:
  1. Desempilha, se necessário, até atingir um estado onde haja a ação de shift para o token error
  2. Shift para o token error
  3. Descarta símbolos da entrada até que um símbolo com ação diferente de erro para o estado corrente seja alcançado
  4. Retoma procedimento normal
- No nosso exemplo, a ação do item 3 sempre será shift
  - Cuidado com reduce
    - Não consome a entrada, podendo o mesmo símbolo continuar causando erros
    - Evitar regras do tipo exp -> error

Recuperação de Erros Local

- Cuidado com ações semânticas:

```
statements: statements exp SEMICOLON
| statements error SEMICOLON
| /* empty */
exp : increment exp decrement
|ID
increment: LPAREN {: nest=nest+1; :}
decrement: RPAREN {: nest=nest-1; :}
```