Análise Sintática

Universidade Federal do Ceará - Campus de Quixadá

Lucas Ismaily
ismailybf@ufc.br

Semestre 2022.1

Compiladores
Baseado nos slides do Prof. Sandro Rigo (IC-Unicamp)

Seção 1

Revisão

Construindo um Predictive Parser

- Cada função relativa a um não-terminal precisa conter uma cláusula para cada produção.
- Precisa saber escolher, baseado no próximo token, qual a produção apropriada.
- Isto é feito através da tabela do predictive parsing.

Construindo um Predictive Parser

- Dada uma produção X $\longrightarrow \gamma$.
- Para cada T $\in$ FIRST(γ)
  - Coloque a produção X $\longrightarrow \gamma$ na linha X, coluna T.
- Se γ é nullable
  - Coloque a produção na linha X, coluna T para cada T $\in$ FOLLOW[X].

Exemplo

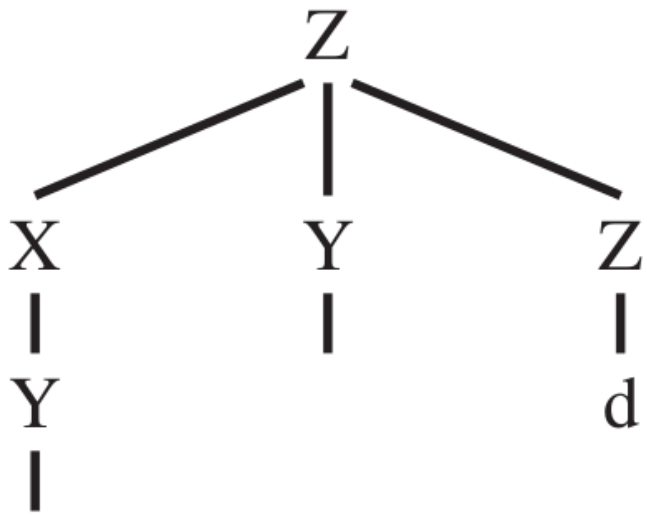
$$
\begin{aligned}
Z &\longrightarrow d \\
Z &\longrightarrow XYZ \\
Y &\longrightarrow \\
Y &\longrightarrow c \\
X &\longrightarrow Y \\
X &\longrightarrow a
\end{aligned}
$$

	nullable	FIRST	FOLLOW
X	true	a,c	a,c,d
Y	true	c	a,c,d
Z	false	a,c,d	-

	a	c	d
X	$X \longrightarrow a$ $X \longrightarrow Y$	$X \longrightarrow Y$	$X \longrightarrow Y$
Y	$Y \longrightarrow$	$Y \longrightarrow c$ $Y \longrightarrow$	$Y \longrightarrow$
Z	$Z \longrightarrow XYZ$	$Z \longrightarrow XYZ$	$Z \longrightarrow XYZ$ $Z \longrightarrow d$

Construindo um Predictive Parser

- Não é possível fazer um parser predictive para essa gramática porque ela é ambígua.
  - Note que algumas células da tabela do predictive parser têm mais de uma entrada!
  - Isso sempre acontece com gramáticas ambíguas (Mas pode acontece também em gramáticas não ambíguas.)



Construindo um Predictive Parser

- Linguagens cujas tabelas não possuam entradas duplicadas são denominadas de LL(1).
  - *left to right parsing*, *leftmost derivation*, *1-symbol lookahead*.
- A definição de conjuntos FIRST pode ser generalizada para os primeiros k tokens de uma string.
  - Gera uma tabela onde as linhas são os não-terminais e as colunas são todas as sequências possíveis de k terminais.

Construindo um Predictive Parser

- Isso raramente é feito devido ao tamanho explosivo das tabelas geradas (deve ter um simbolo para cada combinação, não-terminal-lookahead)
- Gramáricas analisáveis com tabelas LL(K) são chamadas LL(K).
- Nenhuma gramática ambígua é LL(K) para nenhum k!

Recursão à Esquerda

0. $S \longrightarrow E\ \$$
1. $E \longrightarrow E + T$
2. $E \longrightarrow E - T$
3. $E \longrightarrow T$
4. $T \longrightarrow T * F$
5. $T \longrightarrow T/F$
6. $T \longrightarrow F$
7. $F \longrightarrow id$
8. $F \longrightarrow num$
9. $F \longrightarrow (E)$

Consigo gerar um parser LL(1) para essa gramática?

Recursão à esquerda

- Problema:
  - A função que implementa E precisa chamar a si mesma caso escolha E+T.
  - Porém, é a primeira ação dela, antes de avançar na cadeia de entrada.
  - Laço infinito!
  - Acontece devido à recursão à esquerda.
- Como Resolver??? (Fatoração - recursão à direita).

$$
\begin{aligned}
E &\longrightarrow E+T \\
E &\longrightarrow E\text{-}T \\
T &\longrightarrow T*F
\end{aligned}
$$

$$
\begin{aligned}
E &\longrightarrow TE' \\
E' &\longrightarrow +TE' \\
E' &\longrightarrow
\end{aligned}
$$

Recursão à esquerda

- Generalizando:
  - Tendo X $\longrightarrow$ Xγ e X $\longrightarrow \alpha$, onde α não começa com X.
- Derivamos strings da forma $\alpha\gamma^*$
  - α seguindo de zero ou mais γ.
- Podemos reescrever:

$$\begin{pmatrix} X \rightarrow X\ \gamma_1 \\ X \rightarrow X\ \gamma_2 \\ X \rightarrow \alpha_1 \\ X \rightarrow \alpha_2 \end{pmatrix} \Longrightarrow \begin{pmatrix} X \rightarrow \alpha_1\ X' \\ X \rightarrow \alpha_2\ X' \\ X' \rightarrow \gamma_1\ X' \\ X' \rightarrow \gamma_2\ X' \\ X' \rightarrow \end{pmatrix}$$

Eliminando Recursão à Esquerda

0. S $\longrightarrow$ E $
1. E $\longrightarrow$ TE'
2. E $\longrightarrow$ +TE'
3. E' $\longrightarrow$ -TE'
4. E' $\longrightarrow$

5. T $\longrightarrow$ FT'
6. T' $\longrightarrow$ *FT'
7. T' $\longrightarrow$ /FT'
8. T' $\longrightarrow$

9. F $\longrightarrow$ id
10. F $\longrightarrow$ num
11. F $\longrightarrow$ (E)

	nullable	FIRST	FOLLOW
S	false	(id num	
E	false	(id num	)$
E'	True	+ -	)$
T	false	(id num	)$ +-
T'	True	* /	)$ +-
F	false	(id num	)*/+-$

Eliminando Recursão à Esquerda

0. S $\longrightarrow$ E $
1. E $\longrightarrow$ TE'
2. E $\longrightarrow$ +TE'
3. E' $\longrightarrow$ -TE'
4. E' $\longrightarrow$
5. T $\longrightarrow$ FT'
6. T' $\longrightarrow$ *FT'
7. T' $\longrightarrow$ /FT'
8. T' $\longrightarrow$
9. F $\longrightarrow$ id
10. F $\longrightarrow$ num
11. F $\longrightarrow$ (E)

	+	*	id	(	)	$
S			S $\longrightarrow$ E$	S $\longrightarrow$ E$		
E			E $\longrightarrow$ TE'	E $\longrightarrow$ TE'		
E'	E' $\longrightarrow$ +TE'				E' $\longrightarrow$	E' $\longrightarrow$
T			T $\longrightarrow$ FT'	T $\longrightarrow$ FT'		
T'	T' $\longrightarrow$	T' $\longrightarrow$ *FT'			T' $\longrightarrow$	T' $\longrightarrow$
F			F $\longrightarrow$ id	F $\longrightarrow$ (E)		

Fatoração à esquerda

- Um outro problema para predictive parsing ocorre em situação do tipo:

$$\begin{aligned} S &\longrightarrow \texttt{if E then S else S} \\ S &\longrightarrow \texttt{if E then S} \end{aligned}$$

- Regras do mesmo não terminal começam com os mesmos símbolos.

Fatoração à esquerda

- Criar um novo não-terminal para os finais permitidos:

$$
\begin{aligned}
S &\longrightarrow \texttt{if E then S X} \\
X &\longrightarrow \\
X &\longrightarrow \texttt{else S}
\end{aligned}
$$

- Gramática ainda é ambígua, mas o conflito pode ser resolvido escolhendo sempre a segunda regra.

Recuperação de erros

- Uma entrada em branco na tabela indica um caractere não esperado.
- Parar o processo no primeiro erro encontrado não é desejável.
- Duas alternativas:
  - Inserir símbolo:
    - Assume que encontrou o que esperava.
  - Deletar símbolo(s):
    - Pula tokens até que um elemento do FOLLOW seja atingido.

Recuperação de Erros

```
void T() {switch (tok) {
        case ID:
        case NUM:
        case LPAREN: F(); Tprime(); break;
        default:  print("expected id, num, or left-paren");
      }}
```

Recuperação de Erros

```
int Tprime_follow [] = {PLUS, RPAREN, EOF};

void Tprime() { switch (tok) {
        case PLUS:   break;
        case TIMES:  eat(TIMES); F(); Tprime(); break;
        case RPAREN: break;
        case EOF:    break;
        default:   print("expected +, *, right-paren,
                          or end-of-file");
                   skipto(Tprime_follow);
      }}
```

Seção 2

LR Parsing

LR Parsing

- O ponto fraco da técnica LL(k) é precisar prever que produção usar

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 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.

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.

LR Parsing

- O parser tem uma pilha e a entrada.
- Os primeiros k tokens da entrada forma o lookahead.

LR Parsing

- O parser tem uma pilha e a entrada.
- Os primeiros k tokens da entrada forma o lookahead.
- Dois tipos de ações:

LR Parsing

- O parser tem uma pilha e a entrada.
- Os primeiros k tokens da entrada forma o lookahead.
- Dois tipos de ações:
  - SHIFT: move o primeiro token para o topo da pila.

LR Parsing

- O parser tem uma pilha e a entrada.
- Os primeiros k tokens da entrada forma o lookahead.
- Dois tipos de ações:
  - SHIFT: move o primeiro token para o topo da pila.
  - REDUCE:

LR Parsing

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

LR Parsing

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

LR Parsing

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

LR Parsing

0. S' $\longrightarrow$ S$;
1. S $\longrightarrow$ S;S
2. S $\longrightarrow$ id := E
3. S $\longrightarrow$ print(L)
4. E $\longrightarrow$ id
5. E $\longrightarrow$ num
6. E $\longrightarrow$ E + E
7. E $\longrightarrow$ (S,E)
8. L $\longrightarrow$ E
9. L $\longrightarrow$ L, E

LR Parsing

0. S' $\longrightarrow$ S$;
1. S $\longrightarrow$ S;S
2. S $\longrightarrow$ id := E
3. S $\longrightarrow$ print(L)
4. E $\longrightarrow$ id
5. E $\longrightarrow$ num
6. E $\longrightarrow$ E + E
7. E $\longrightarrow$ (S,E)
8. L $\longrightarrow$ E
9. L $\longrightarrow$ 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 id_4	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=$_6$ num_{10}	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
1 id_4 :=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ 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 ;$_3$ id_4	:= c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$	c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ id_{20}	+ (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ id
1 S_2 ;$_3$ id_4 :=$_6$ E_{11}	+ (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$	(d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$	d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ id_4	:= 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ id_4 :=$_6$	5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ id_4 :=$_6$ num_{10}	+ 6 , d) $	*reduce* $E \rightarrow$ num
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ id_4 :=$_6$ E_{11}	+ 6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ id_4 :=$_6$ E_{11} +$_{16}$	6 , d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ id_4 :=$_6$ E_{11} +$_{16}$ num_{10}	, d) $	*reduce* $E \rightarrow$ num
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ id_4 :=$_6$ E_{11} +$_{16}$ E_{17}	, d) $	*reduce* $E \rightarrow E + E$
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ id_4 :=$_6$ E_{11}	, d) $	*reduce* $S \rightarrow$ id :=E
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ S_{12}	, d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ S_{12} ,$_{18}$	d) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ S_{12} ,$_{18}$ id_{20}	) $	*reduce* $E \rightarrow$ id
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ S_{12} ,$_{18}$ E_{21}	) $	*shift*
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (${}_8$ S_{12} ,$_{18}$ E_{21})$_{22}$	$	*reduce* $E \rightarrow (S, E)$
1 S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ E_{17}	$	*reduce* $E \rightarrow E + E$
1 S_2 ;$_3$ id_4 :=$_6$ E_{11}	$	*reduce* $S \rightarrow$ 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?

LR Parsing Engine

- Como o parser sabe quando fazer um SHIFT ou um REDUCE?
- Usando um DFA aplicado a pilha!!

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:

Tabela de transição

- 4 tipos de ações:
  - sn: SHIFT para o estado n;

Tabela de transição

- 4 tipos de ações:
  - sn: SHIFT para o estado n;
  - gn: vá para o estado n;

Tabela de transição

- 4 tipos de ações:
  - sn: SHIFT para o estado n;
  - gn: vá para o estado n;
  - rk: REDUCE pela regra k;

Tabela de transição

- 4 tipos de ações:
  - sn: SHIFT para o estado n;
  - gn: vá para o estado n;
  - rk: REDUCE pela regra k;
  - a: Accept;

Tabela de transição

- 4 tipos de ações:
  - sn: SHIFT para o estado n;
  - gn: vá para o estado n;
  - rk: REDUCE pela regra k;
  - a: Accept;
  - : Error (entrada em branco)

Tabela de transição

- 4 tipos de ações:
  - sn: SHIFT para o estado n;
  - gn: vá para o estado n;
  - rk: REDUCE pela regra k;
  - a: Accept;
  - : Error (entrada em branco)
- As arestas do DFA são as ações SHIFT e goto.

Tabela de transição

- 4 tipos de ações:
  - sn: SHIFT para o estado n;
  - gn: vá para o estado n;
  - rk: REDUCE 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

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):
  - A entrada avança um símbolo e é feito um *push* de n na pilha.

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):
  - A entrada avança um símbolo e é feito um *push* de n na pilha.
- Se a ação for REDUCE(k):

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):
  - A entrada avança um símbolo e é feito um *push* de n na pilha.
- Se a ação for REDUCE(k):
  - Desempilhe r símbolos da pilha (*pop*), onde r é o número de símbolos no lado direito da regra k.

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):
  - A entrada avança um símbolo e é feito um *push* de n na pilha.
- Se a ação for REDUCE(k):
  - Desempilhe r símbolos da pilha (*pop*), onde r é o número de símbolos no lado direito da regra k.
  - Seja X o símbolo do lado esquerdo da regra k;

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):
  - A entrada avança um símbolo e é feito um *push* de n na pilha.
- Se a ação for REDUCE(k):
  - Desempilhe r símbolos da pilha (*pop*), onde r é o número de símbolos no lado direito da regra k.
  - Seja X o símbolo do lado esquerdo da regra k;
    - No estado (atual) do topo da pilha, procure X para obter um "goto n".

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):
  - A entrada avança um símbolo e é feito um *push* de n na pilha.
- Se a ação for REDUCE(k):
  - Desempilhe r símbolos da pilha (*pop*), onde r é o número de símbolos no lado direito da regra k.
  - Seja X o símbolo do lado esquerdo da regra k;
    - No estado (atual) do topo da pilha, procure X para obter um "goto n".
    - Adicione X ao topo da pilha.

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):
  - A entrada avança um símbolo e é feito um *push* de n na pilha.
- Se a ação for REDUCE(k):
  - Desempilhe r símbolos da pilha (*pop*), onde r é o número de símbolos no lado direito da regra k.
  - Seja X o símbolo do lado esquerdo da regra k;
    - No estado (atual) do topo da pilha, procure X para obter um "goto n".
    - Adicione X ao topo da pilha.
- Accept: para o parsing e reporta sucesso.

Algoritmo

- Olhe para o estado no topo da pilha e para o símbolo de entrada para determinar a ação.
- Se a ação for SHIFT(n):
  - A entrada avança um símbolo e é feito um *push* de n na pilha.
- Se a ação for REDUCE(k):
  - Desempilhe r símbolos da pilha (*pop*), onde r é o número de símbolos no lado direito da regra k.
  - Seja X o símbolo do lado esquerdo da regra k;
    - No estado (atual) do topo da pilha, procure X para obter um "goto n".
    - Adicione X ao topo da pilha.
- Accept: para o parsing e reporta sucesso.
- Error: para o parsing e reporta falha.

Tabela de transição

- O exemplo anterior mostrou o uso de 1 símbolo de lookahead.

Tabela de transição

- O exemplo anterior mostrou o uso de 1 símbolo de lookahead.
- Para k, a tabela terá colunas para todas as sequências de k tokens.

Tabela de transição

- O exemplo anterior mostrou o uso de 1 símbolo de lookahead.
- Para k, a tabela terá colunas para todas as sequências de k tokens.
- k>1 praticamente não é usado para compilação.

Tabela de transição

- O exemplo anterior mostrou o uso de 1 símbolo de lookahead.
- Para k, a tabela terá colunas para todas as sequências 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)

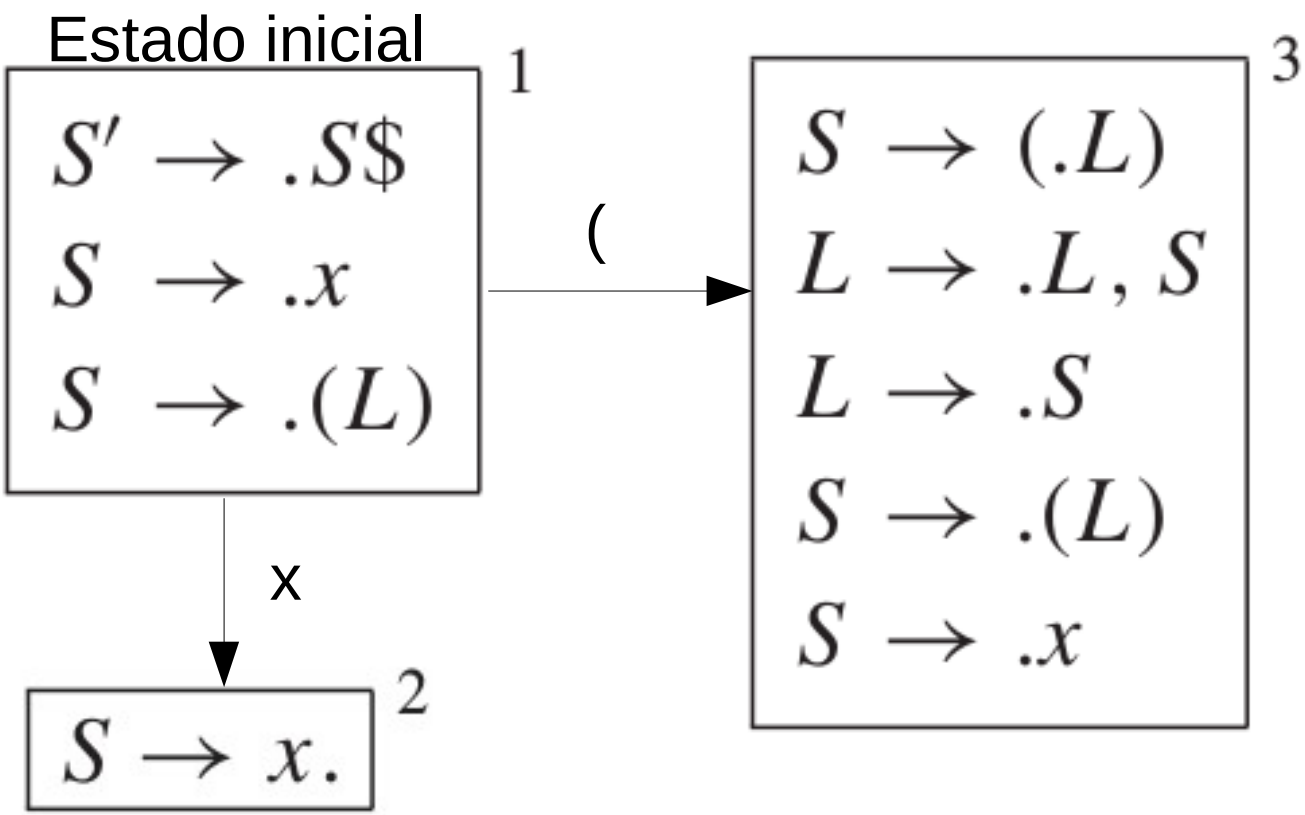
LR(0) são as gramáticas que podem ser analisadas olhando somente a pilha.

Geração de parsers LR(0)

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

0. S' $\longrightarrow$ S$;
1. S $\longrightarrow$ (L)
2. S $\longrightarrow$ x
3. L $\longrightarrow$ S
4. L $\longrightarrow$ L,S

Estados



SHIFT de "x" e "("

Estado 2 permite reduce

Estados

- Ação de Goto:

Estados

- Ação de Goto:
  - Imagine um SHIFT de “x” ou “(” no estado 1 seguido de redução pela produção de S correspondente.

Estados

- Ação de Goto:
  - 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.

Estados

- Ação de Goto:
  - 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).

Estados

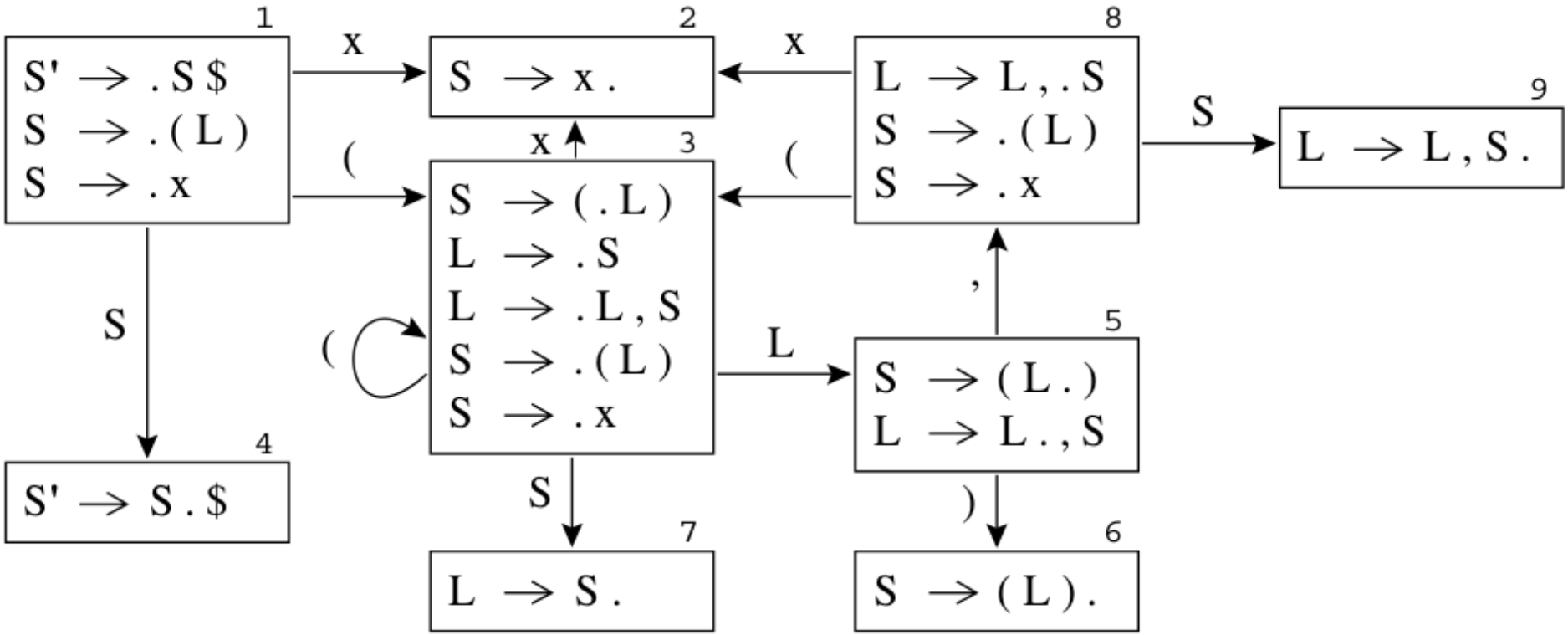
- Ação de Goto:
  - Imagine um SHIFT de “x” ou “(” no estado 1 seguido de redução pela produção de S correspondente.
  - Todos os símbolos do lado direito da produção serão desempilhados e o parser vai executar um goto para S no estado 1.
  - Isso se representa movendo-se o ponto para após o S e colocando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

Estados

- Ação de Goto:
  - 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).

$$\boxed{S' \rightarrow S.\$}^{4}$$

Exemplo


1
S' → . S $
S → . (L)
S → . x
2
S → x .
3
S → (. L)
L → . S
L → . L , S
S → . (L)
S → . x
4
S' → S . $
5
S → (L .)
L → L . , S
6
S → (L) .
7
L → S .
8
L → L , . S
S → . (L)
S → . x
9
L → L , S .
x
(
S
L
,
)

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 $A \to \alpha.X\beta$ in I
      for any production $X \to \gamma$
        $I \leftarrow I \cup \{X \to .\gamma\}$
  until I does not change.
  return I

Goto(I, X) =
  set J to the empty set
  for any item $A \to \alpha.X\beta$ in I
    add $A \to \alpha X.\beta$ to J
  return Closure(J)

Algoritmos

- Construção do parser LR(0)

Initialize T to $\{\textbf{Closure}(\{S' \rightarrow .S\$\})\}$
Initialize E to empty.
repeat
  for each state I in T
    for each item $A \rightarrow \alpha.X\beta$ in I
      let J be **Goto**(I, X)
      $T \leftarrow T \cup \{J\}$
      $E \leftarrow E \cup \{I \xrightarrow{X} J\}$
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		

Seção 3

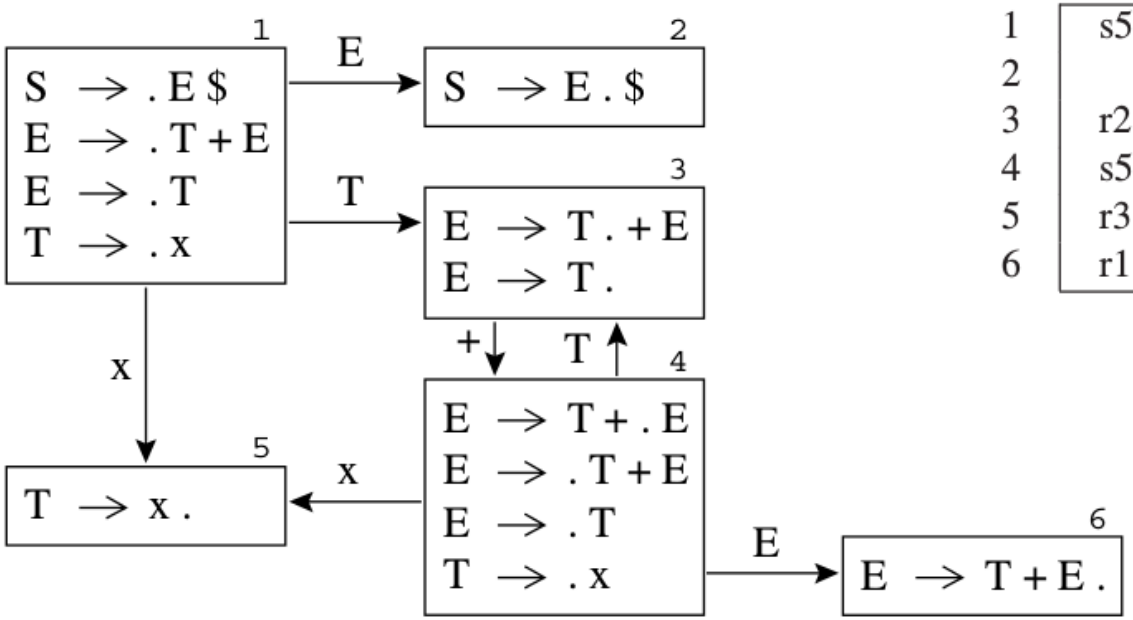
SLR

SLR Parser

0. $S \longrightarrow E\$;$
1. $E \longrightarrow T + E$
2. $E \longrightarrow T$
3. $T \longrightarrow x$

SLR Parser

0. $S \longrightarrow E\$;$
1. $E \longrightarrow T + E$
2. $E \longrightarrow T$
3. $T \longrightarrow x$



	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		

SLR Parser

- Colocar reduções somente onde indicado pelo conjunto FOLLOW.

SLR Parser

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

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		

LR(1)

- Mais poderoso do que SLR.

LR(1)

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

LR(1)

- Mais poderoso do que SLR.
- Maioria das linguagens de programação são LR(1).
  - Exceção notável: C++!!

LR(1)

- Mais poderoso do que SLR.
- Maioria das linguagens de programação são LR(1).
  - Exceção notável: C++!!
- Algoritmo similar ao LR(0).

LR(1)

- Mais poderoso do que SLR.
- Maioria das linguagens de programação são LR(1).
  - Exceção notável: C++!!
- Algoritmo similar ao LR(0).
- Item: (A $\longrightarrow \alpha.\beta$, x).

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

Closure(I) =
  repeat
    for any item $(A \to \alpha.X\beta, z)$ in I
      for any production $X \to \gamma$
        for any $w \in \text{FIRST}(\beta z)$
          $I \leftarrow I \cup \{(X \to .\gamma,\ w)\}$
  until I does not change
  return I

Goto(I, X) =
  $J \leftarrow \{\}$
  for any item $(A \to \alpha.X\beta,\ z)$ in I
    add $(A \to \alpha X.\beta,\ z)$ to J
  return Closure(J).

Exemplo

0. $S \longrightarrow S\$;$
1. $S \longrightarrow V = E$
2. $S \longrightarrow E$
3. $E \longrightarrow V$
4. $V \longrightarrow x$
5. $V \longrightarrow * E$

É SLR???

Exemplo

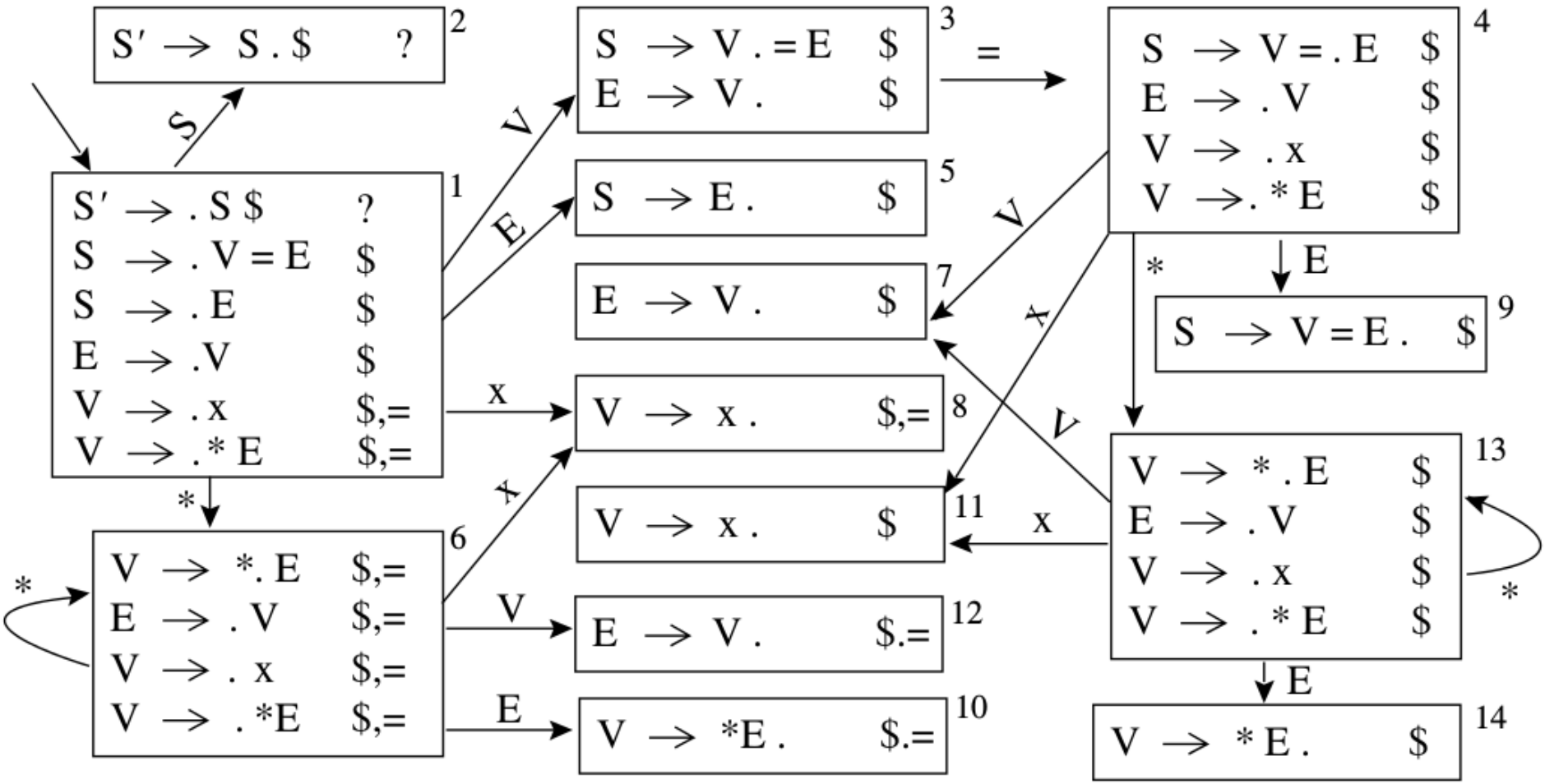
0. S $\longrightarrow$ S\$;
1. S $\longrightarrow$ V = E
2. S $\longrightarrow$ E

3.E $\longrightarrow$ V
4.V $\longrightarrow$ x
5.V $\longrightarrow$ * E

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

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

Exemplo


S′ → S . $? 2
S → V . = E $
E → V . $ 3
=
S → V = . E $
E → . V $
V → . x $
V → . * E $ 4
S′ → . S $? 1
S → . V = E $
S → . E $
E → . V $
V → . x $,=
V → . * E $,=
S → E . $ 5
E → V . $ 7
V → x . $,= 8
S → V = E . $ 9
V → x . $ 11
V → * . E $,=
E → . V $,=
V → . x $,=
V → . * E $,= 6
E → V . $.= 12
V → * E . $.= 10
V → * . E $
E → . V $
V → . x $
V → . * E $ 13
V → * E . $ 14
S
V
E
x
*
=

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			

$R \leftarrow \{\}$
for each state I in T
  for each item $(A \rightarrow \alpha. \ , \ z)$ in I
    $R \leftarrow R \cup \{(I, z, A \rightarrow \alpha)\}$

Parser LALR(1)

- O tamanho das tabelas LR(1) pode ser muito grande.

Parser LALR(1)

- O tamanho das tabelas LR(1) pode ser muito grande.
- É possível reduzir o tamanho unindo estados do DFA.

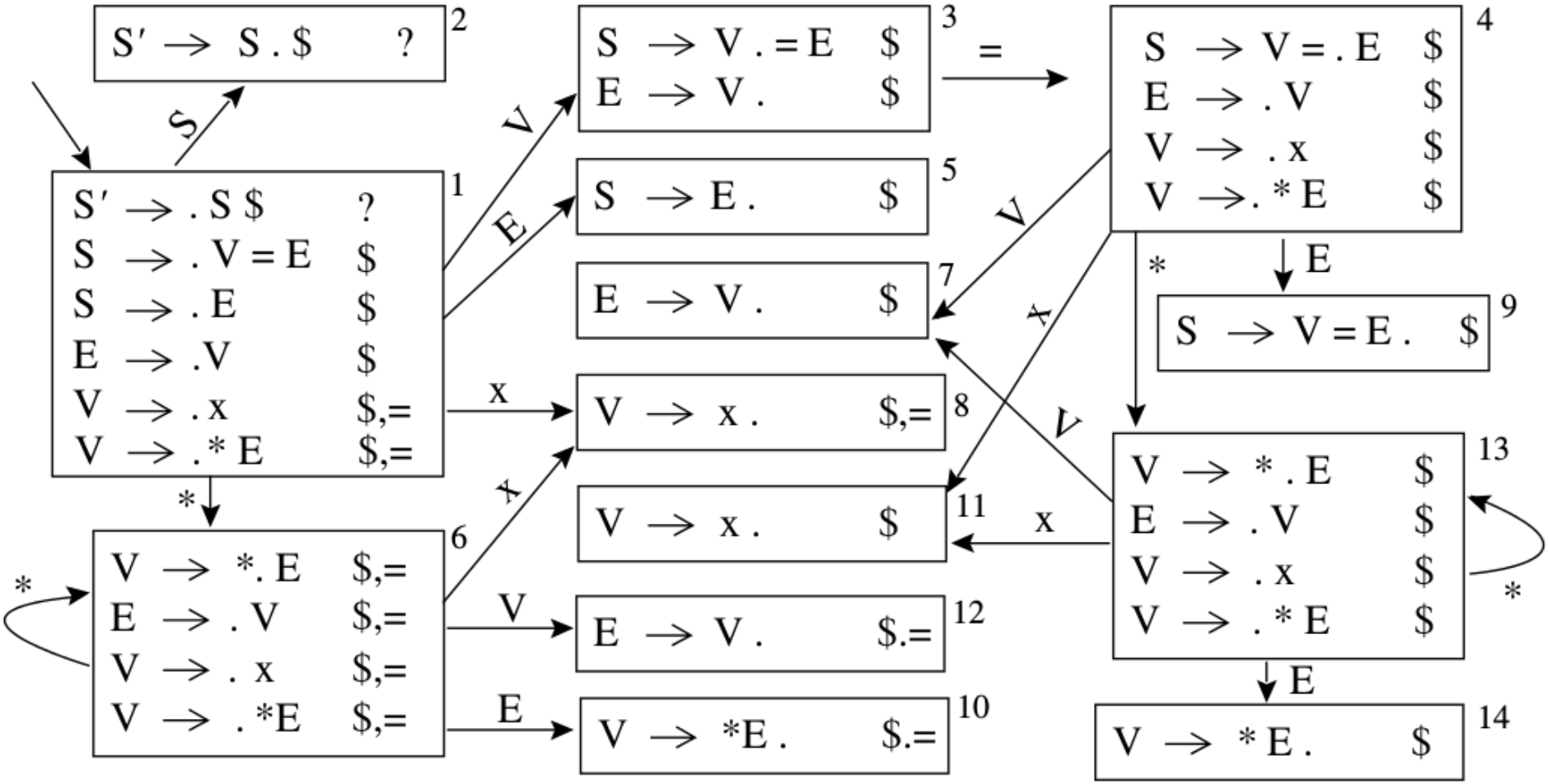
Parser LALR(1)

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

Parser LALR(1)

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

Parser LALR(1)



Parser LALR(1)

	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 LR(1) não possuía.

Parser LALR(1)

- Pode gerar uma tabela com conflitos, onde LR(1) não possuía.
  - Na prática, o efeito de redução no uso de memória é bastante desejável.

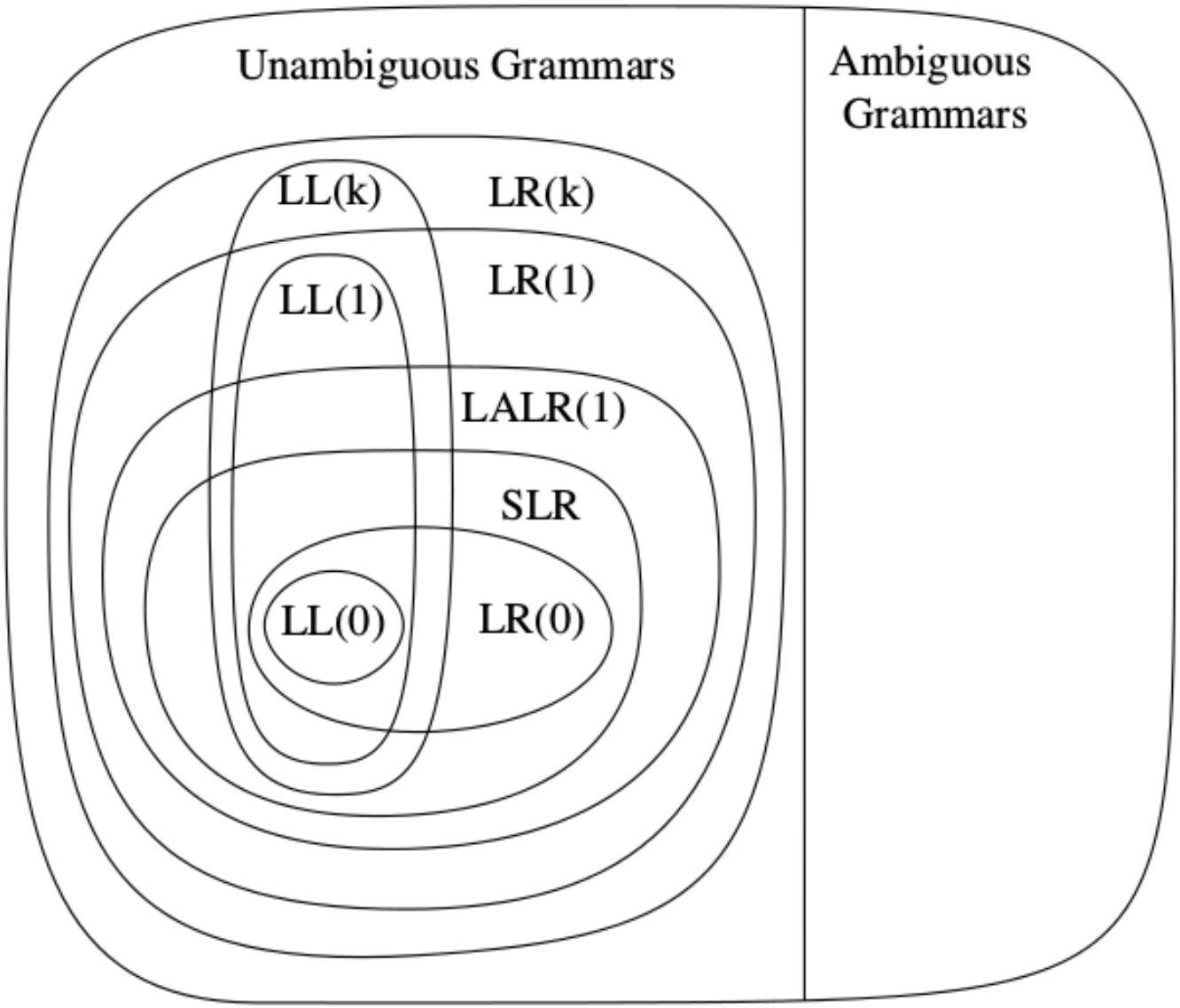
Parser LALR(1)

- Pode gerar uma tabela com conflitos, onde LR(1) não possuía.
  - Na prática, o efeito de redução no uso de memória é bastante desejável.
- A maioria das linguagens de programação é LALR(1).

Parser LALR(1)

- Pode gerar uma tabela com conflitos, onde LR(1) não possuía.
  - 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 parsers.

Hierarquia das gramáticas



Ambiguidade

$$
\begin{aligned}
S &\longrightarrow \texttt{if E then S else S} \\
S &\longrightarrow \texttt{if E then S} \\
S &\longrightarrow \texttt{other}
\end{aligned}
$$

Como seria a parser tree para:

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

Ambiguidade

$$
\begin{aligned}
S &\longrightarrow \texttt{if E then S else S} \\
S &\longrightarrow \texttt{if E then S} \\
S &\longrightarrow \texttt{other}
\end{aligned}
$$

Ambiguidade

$$
\begin{aligned}
S &\longrightarrow \texttt{if E then S else S} \\
S &\longrightarrow \texttt{if E then S} \\
S &\longrightarrow \texttt{other}
\end{aligned}
$$

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

Ambiguidade

$$
\begin{aligned}
S &\longrightarrow \texttt{if E then S else S} \\
S &\longrightarrow \texttt{if E then S} \\
S &\longrightarrow \texttt{other}
\end{aligned}
$$

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

Conflito SHIFT-REDUCE!

$S \rightarrow \text{if } E \text{ then } S \,.$	else
$S \rightarrow \text{if } E \text{ then } S \,.\, \text{else } S$	(*any*)

Eliminando

$$
\begin{array}{lcl}
S & \longrightarrow & \texttt{M} \\
S & \longrightarrow & \texttt{U} \\
M & \longrightarrow & \texttt{if E then M else M} \\
M & \longrightarrow & \texttt{other} \\
U & \longrightarrow & \texttt{if E then S} \\
U & \longrightarrow & \texttt{if E then M else U}
\end{array}
$$

Como seria a parser tree para:

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

Eliminando

- Pode-se usar a gramática ambígua, decidindo os conflitos sempre por SHIFT em casos desse tipo.

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.

Diretivas de precedência

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

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

1. E $\longrightarrow$ id
2. E $\longrightarrow$ num
3. E $\longrightarrow$ E * E
4. E $\longrightarrow$ E/E

5. E $\longrightarrow$ E + E
6. E $\longrightarrow$ E - E
7. E $\longrightarrow$ (E)

1. E $\longrightarrow$ E + T
2. E $\longrightarrow$ E - T
3. E $\longrightarrow$ T

4. T $\longrightarrow$ T * F
5. T $\longrightarrow$ T/F
6. T $\longrightarrow$ F

7. F $\longrightarrow$ id
8. F $\longrightarrow$ num
9. F $\longrightarrow$ (E)

Diretivas de precedência

	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)

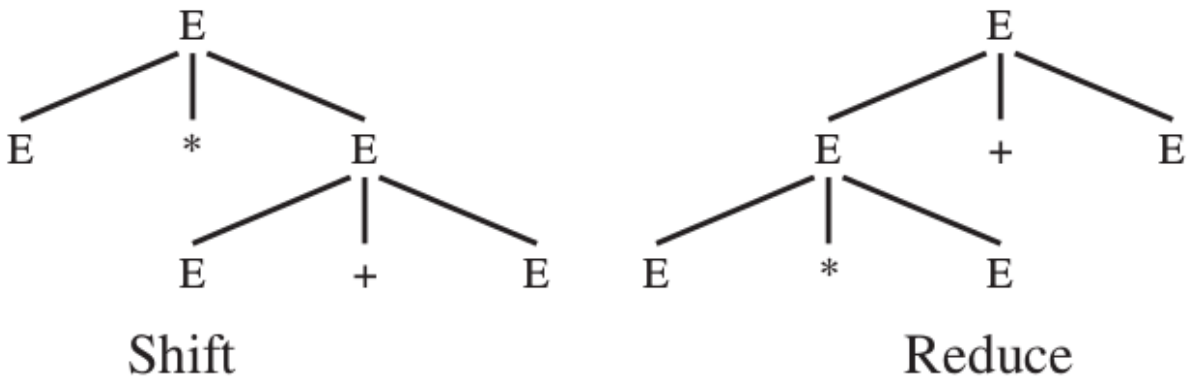
- Pilha:
  - ... E * E (+)
- SHIFT:
  - ... E*E+
- Chegando a:
  - ...E*E+E
- Reduzindo para: ...E*E

- 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;
```

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.

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 -.

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 -.

Diretivas de precedência

- Podem ajudar.

Diretivas de precedência

- Podem ajudar.
- Não devem ser abusivamente utilizadas.

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.

Exercício

3.11 Construct the LR(0) states for this grammar, and then determine whether it is an SLR grammar.

0 $S \rightarrow B\ \$$

1 $B \rightarrow \text{id}\ P$
2 $B \rightarrow \text{id}\ (\ E\ ]$

3 $P \rightarrow$
4 $P \rightarrow (\ E\ )$

5 $E \rightarrow B$
6 $E \rightarrow B\ ,\ E$

3.12 a. Build the LR(0) DFA for this grammar:

0 $S \rightarrow E\ \$$

1 $E \rightarrow \text{id}$
2 $E \rightarrow \text{id}\ (\ E\ )$
3 $E \rightarrow E\ +\ \text{id}$

b. Is this an LR(0) grammar? Give evidence.

c. Is this an SLR grammar? Give evidence.

d. Is this an LR(1) grammar? Give evidence.

3.13 Show that this grammar is LALR(1) but not SLR:

0 $S \rightarrow X\ \$$
1 $X \rightarrow M\ a$
2 $X \rightarrow b\ M\ c$

3 $X \rightarrow d\ c$
4 $X \rightarrow b\ d\ a$
5 $M \rightarrow d$

Análise Sintática

Universidade Federal do Ceará - Campus de Quixadá

Lucas Ismaily
ismailybf@ufc.br

Semestre 2022.1

Compiladores
Baseado nos slides do Prof. Sandro Rigo (IC-Unicamp)