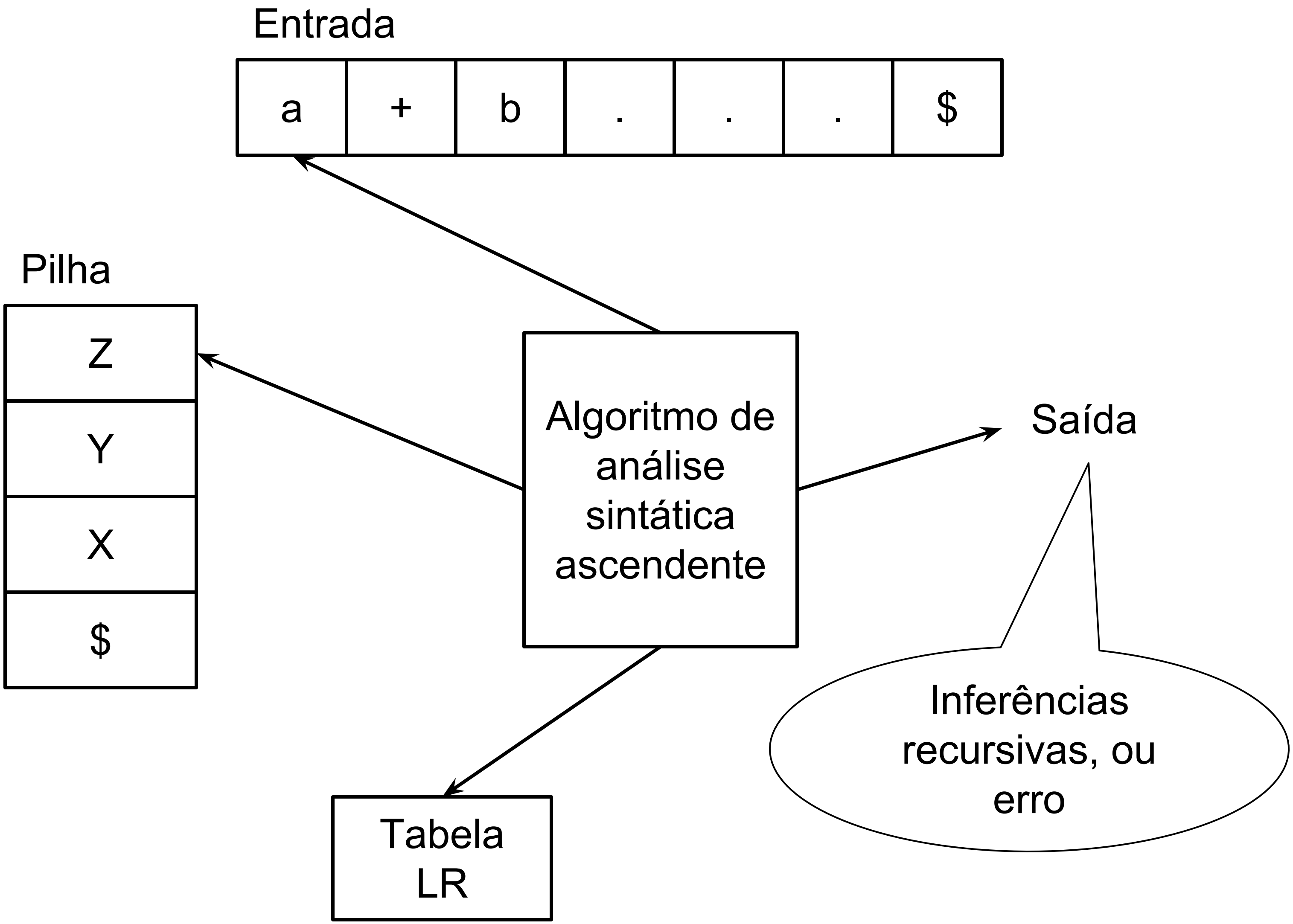
Construção de compiladores

Prof. Daniel Lucrédio
Departamento de Computação - UFSCar
1º semestre / 2015
Aula 6

Análise sintática ascendente LR

ASA



Entrada
a
+
b
.
.
.
$
Pilha
Z
Y
X
$
Algoritmo de análise sintática ascendente
Saída
Inferências recursivas, ou erro
Tabela LR

ASA LR

- Analisador LR (k)
  - **L**eft to right with **R**ightmost derivation
  - Lê a sentença em análise da esquerda para a direita
  - Produz uma derivação mais à direita ao reverso
    - Inferência recursiva
  - Considerando-se k símbolos na cadeia de entrada

Tabela de análise LR

- Existem diferentes tipos de tabelas LR
  - Cada uma com vantagens/desvantagens (veremos depois)
- A tabela LR é dividida em duas
  - Ação
  - Transição
- A tabela é construída diretamente a partir da gramática
- Estados = armazenam a situação atual de leitura
  - Permitem detectar o aparecimento de um “gancho”

Exemplo de tabela de análise LR

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

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				OK			
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			

Tabela de análise LR

- Os códigos para as ações são:
  - si = *shift* i
    - “avança na entrada e empilha o estado i na pilha”
  - rj = *reduce* j
    - “reduz segundo a produção de número j”
  - OK
    - “aceita a entrada”
  - Entrada em branco
    - Erro sintático

Algoritmo de análise LR

- ENTRADA: uma cadeia de entrada w e uma tabela de análise LR com as ações e transições definidas para uma gramática G
- SAÍDA: se w está em L(G), os passos de inferência recursiva (análise ascendente para w). Caso contrário, uma indicação de erro
- CONDIÇÕES INICIAIS:
  - w$ no buffer de entrada
  - s0 na pilha (estado inicial)

Algoritmo de análise LR

```
a := primeiro símbolo de w$;
while(1) { /* repita indefinidamente */
   s := estado no topo da pilha;
   if(ACAO[s,a] = "shift t") {
      empilha t;
      a := próximo símbolo da entrada;
   } else if (ACAO[s,a] = "reduce A --> β") {
      desempilha |β| símbolos;
      t := topo da pilha;
      empilha TRANSICAO[t,A];
      imprima "A --> β"
   } else if (ACAO[s,a] = "OK") pare; /* fim */
   else erro;
}
```

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

Exemplo

Entrada = **id * id + id**

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				OK			
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			

Pilha	Símbolos	Entrada	Ação

Exemplo

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

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				OK			
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			

Entrada = **id * id + id**

Pilha	Símbolos	Entrada	Ação
0		id*id+id$	s5
0 5	id	*id+id$	r6
0 3	F	*id+id$	r4
0 2	T	*id+id$	s7
0 2 7	T *	id+id$	s5
0 2 7 5	T * id	+id$	r6
0 2 7 10	T * F	+id$	r3
0 2	T	+id$	r2
0 1	E	+id$	s6
0 1 6	E +	id$	s5
0 1 6 5	E + id	$	r6
0 1 6 3	E + F	$	r4
0 1 6 9	E + T	$	r1
0 1	E	$	OK

Exercício

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

Entrada = **id * (id + id)**

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				OK			
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			

Pilha	Símbolos	Entrada	Ação

Exercício

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

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				OK			
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			

Entrada = **id * (id + id)**

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

Exercício

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

Entrada = **id * (id**

Pilha	Símbolos	Entrada	Ação

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				OK			
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			

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

Exercício

Entrada = **id * (id**

Pilha	Símbolos	Entrada	Ação
0		id*(id$	s5
0 5	id	*(id$	r6
0 3	F	*(id$	r4
0 2	T	*(id$	s7
0 2 7	T *	(id$	s4
0 2 7 4	T * (	id$	s5
0 2 7 4 5	T * (id	$	r6
0 2 7 4 3	T * (F	$	r4
0 2 7 4 2	T * (T	$	r2
0 2 7 4 8	T * (E	$	erro

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				OK			
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			

Análise LR

- 3 tipos de tabela (o algoritmo é o mesmo):
  - Simple LR (SLR)
    - Fácil de implementar
    - Aplicável a uma classe mais restrita de gramáticas
  - Look Ahead LR (LALR)
    - Nível intermediário e implementação eficiente
    - Funciona para a maioria das linguagens de programação
  - LR Canônico
    - Mais poderoso e complexo
    - Pode ser aplicado a um grande número de gramáticas

Tabela SLR

- Construindo a tabela sintática SLR
  - A construção da tabela SLR se baseia na coleção canônica de conjuntos de itens LR(0)
    - LR(0): 0 porque não se olha nenhum símbolo a frente
- Um item LR(0) para uma gramática G é uma produção com alguma indicação (.) de até onde essa produção já foi analisada no processo de reconhecimento
- Exemplo – a produção `A→XYZ` dá origem a 4 itens LR(0):
  - `A→.XYZ`
  - `A→X.YZ`
  - `A→XY.Z`
  - `A→XYZ.`

- Produções do tipo `A→ε` geram somente um item `A→.`

Tabela SLR

- Construindo a tabela sintática SLR
  - A construção da tabela SLR se baseia na coleção canônica de conjuntos de itens LR(0)
    - LR(0): 0 porque não se olha nenhum símbolo a frente
- Um item LR(0) para uma gramática G é uma produção com alguma indicação (.) de até onde essa produção já foi analisada no processo de reconhecimento
- Exemplo – a produção A→XYZ dá o ... itens LR(0):
  - A→.XYZ
  - A→X.YZ
  - A→XY.Z
  - A→XYZ.

Item inicial

Item completo

- Produções do tip ... ente um item A→.

Itens LR(0)

- Exercício: encontre os itens LR(0) para as seguintes gramáticas
  - G1:

    $S' \rightarrow S$

    $S \rightarrow (\; S \;) \; S \mid \varepsilon$

  - G2:

    $E' \rightarrow E$

    $E \rightarrow E + n \mid n$

Itens LR(0)

- Resposta:

G1:

```
S' → .S
S' → S.
S  → .(S)S
S  → (.S)S
S  → (S.)S
S  → (S).S
S  → (S)S.
S  → .
```

G2:

```
E' → .E
E' → E.
E  → .E+n
E  → E.+n
E  → E+.n
E  → E+n.
E  → .n
E  → n.
```

Fechamento de conjuntos de itens

- Função CLOSURE(I)
  - I é um conjunto de itens para uma gramática G
- Regras:

1. Inicialmente, acrescente todo item de I no CLOSURE(I)
2. Se $A \rightarrow \alpha.B\beta$ está em CLOSURE(I) e $B \rightarrow \gamma$ é uma produção, então adicione o item $B \rightarrow .\gamma$ em CLOSURE(I), se ainda não estiver lá.
   - Aplique essa regra até que nenhum outro item possa ser incluído no CLOSURE(I)

Fechamento de conjuntos de itens

- Algoritmo:

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

Fechamento de conjuntos de itens

- Exemplo:

```
E’→ E
E → E+T | T
T → T*F | F
F → (E) | id
```

- `CLOSURE({[E’→.E]}) =`

```
{ [E→.E+T],
  [E→.T],
  [T→.T*F],
  [T→.F],
  [F→.(E)],
  [F→.id]
}
```

Fechamento de conjuntos de itens

- Exercício:

  E' → E

  E → E+T | T

  T → T*F | F

  F → (E) | id

- CLOSURE({[E→.E+T]}) =

  { [E→.E+T],

  [E→.T],

  [T→.T*F],

  [T→.F],

  [F→.(E)],

  [F→.id]

  }

Fechamento de conjuntos de itens

- Exercício :

$$E' \to E$$
$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id$$

- CLOSURE({[T→T.*F]}) =

{ [T→T.*F] }

Fechamento de conjuntos de itens

- Exercício :

  $E' \rightarrow E$
  $E \rightarrow E+T \mid T$
  $T \rightarrow T*F \mid F$
  $F \rightarrow (E) \mid id$

- CLOSURE({[T→T*.F]}) =

  { [T→T*.F],
  [F→.(E)],
  [F→.id]
  }

Função de transição

- Função GOTO(I,X)
  - I é um conjunto de itens
  - X é um símbolo da gramática
- Regra
  - GOTO(I,X) é o fechamento do conjunto de todos os itens $[A \rightarrow \alpha X.\beta]$ tais que $[A \rightarrow \alpha.X\beta]$ está em I
- Em outras palavras:
  1. Examinamos os itens em I com ponto imediatamente à esquerda de X
  2. Para cada item encontrado no passo 1)
     a. Movemos o ponto para imediatamente à direita de X
     b. Calculamos o fechamento desse conjunto

Função de transição

- Exemplo:

$$E' \to E$$
$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id$$

  - I = { [E'→E.], [E→E.+T]}
  - GOTO(I,+) =

{ [E→E+.T],
[T→.T*F],
[T→.F],
[F→.(E),
[F→.id]
}

Função de transição

- Exercício:

$$E' \to E$$
$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id$$

- I = {[F→.(E)], [F→(E.)], [F→.id]}
- GOTO(I,)) = { [F→(E).] }
- GOTO(I,id) = { [F→id.] }
- GOTO(I,() = { [F→(.E)],[E→.E+T],[E→.T],
  [T→.T*F],[T→.F],[F→.(E)],
  [F→.id] }

Coleção canônica de conjuntos de itens LR(0)

- Veremos agora um algoritmo para:
  - Dada uma gramática G
  - Obtermos itens LR(0) para G
  - Agruparmos os itens em conjuntos
  - Agrupar os conjuntos em uma coleção
- Essa coleção será utilizada para a construção de um autômato
  - Chamado autômato LR(0)
- Esse autômato é a base do analisador sintático SLR
  - A tabela SLR é montada a partir desse autômato

Coleção canônica de conjuntos de itens LR(0)

- Passo 1:
  - Dada uma gramática G
  - Criar uma gramática aumentada G’
    - G’ consiste de G com uma produção adicional inicial S’→S
    - onde S é o símbolo inicial de G
    - e S’ não ocorre em G
- Exemplo:

E′ → E

```
E → E+T | T
T → T*F | F
F → (E) | id
```

E’ é o novo símbolo inicial

Coleção canônica de conjuntos de itens LR(0)

- Passo 1:
  - Dada uma gramática G
  - Criar uma gramática a
    - G' consiste de G com un S
    - onde S é o símbolo inic
    - e S' não ocorre em
- Exemplo:

$\underline{\mathbf{E' \rightarrow E}}$

```
E → E+T | T
T → T*F | F
F → (E) | id
```

Na gramática aumentada tem-se a certeza de que o símbolo inicial tem somente uma produção associada

Dessa forma é mais fácil detectar o fim da análise sintática. Mais precisamente, o item `S'→S.` marca o fim da análise

Coleção canônica de conjuntos de itens LR(0)

- Passo 2 (algoritmo)

```
void itens(G′) {
  C := CLOSURE({[S′→.S]});
  repeat
    (a) for(cada conjunto de itens I em C)
      (b) for(cada símbolo de gramática X)
            if(GOTO(I,X) não é vazio e não
                              está em C)
              adicione GOTO(I,X) em C;
  until nenhum novo conjunto de itens
        seja adicionado em C em uma rodada;
}
```

Coleção canônica de conjuntos de itens LR(0)

- Exemplo:

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E+T \mid T \\
T &\rightarrow T*F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

$$
C = CLOSURE(\{[E' \rightarrow .E]\}) = \{ [E' \rightarrow .E], [E \rightarrow .E+T], [E \rightarrow .T], [T \rightarrow .T*F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .id] \} = I_0
$$

Coleção canônica de conjuntos de itens LR(0)

$C = \{ I_0 \}$

a: $I_0 = \{ [E' \to .E], [E \to .E+T], [E \to .T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .id] \}$

b: $GOTO(I_0, E') = \emptyset$

$GOTO(I_0, E) = \{ [E' \to E.], [E \to E.+T] \} = I_1$

$GOTO(I_0, T) = \{ [E \to T.], [T \to T.*F] \} = I_2$

$GOTO(I_0, F) = \{ [T \to F.] \} = I_3$

$GOTO(I_0, +) = \emptyset$

$GOTO(I_0, *) = \emptyset$

$GOTO(I_0, () = \{ [F \to (.E)], [E \to .E+T], [E \to .T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .id] \} = I_4$

$GOTO(I_0,)) = \emptyset$

$GOTO(I_0, id) = \{ [F \to id.] \} = I_5$

Coleção canônica de conjuntos de itens LR(0)

$C = \{ I_0, I_1, I_2, I_3, I_4, I_5 \}$

a: $I_1 = \{$
[E' → E.],
[E → E.+T]
$\}$

b: GOTO (I_1, E') = ∅
GOTO (I_1, E) = ∅
GOTO (I_1, T) = ∅
GOTO (I_1, F) = ∅
GOTO (I_1, +) = { [E → E+.T], [T → .T*F], [T → .F], [F → .(E)], [F → .id] } = I_6
GOTO (I_1, *) = ∅
GOTO (I_1, ( ) = ∅
GOTO (I_1,)) = ∅
GOTO (I_1, id) = ∅

Coleção canônica de conjuntos de itens LR(0)

a: $I_2 = \{$
$[E \to T.]$,
$[T \to T.*F]$
$\}$

b: $GOTO(I_2, *) = \{ [T \to T*.F],$
$[F \to .(E)],$
$[F \to .id] \} = I_7$

a: $I_3 = \{ [T \to F.] \}$
b: todos GOTO são vazios

Coleção canônica de conjuntos de itens LR(0)

a: $I_4 = \{$
$[F \to (.E)]$,
$[E \to .E+T]$,
$[E \to .T]$,
$[T \to .T*F]$,
$[T \to .F]$,
$[F \to .(E)]$,
$[F \to .id]$
$\}$

b: $GOTO(I_4, E) = \{ [F \to (E.)], [E \to E.+T] \} = I_8$
$GOTO(I_4, T) = \{ [E \to T.], [T \to T.*F] \} = I_2$
$GOTO(I_4, F) = \{ [T \to F.] \} = I_3$
$GOTO(I_4, ( ) = \{ [F \to (.E)], [E \to .E+T], [E \to .T],$
$[T \to .T*F], [T \to .F], [F \to .(E)],$
$[F \to .id] \} = I_4$
$GOTO(I_4, id) = \{ [F \to id.] \} = I_5$

Coleção canônica de conjuntos de itens LR(0)

a: $I_5 = \{ [F \to id.] \}$

b: todos GOTO são vazios

Agora, $C = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8 \}$

Coleção canônica de conjuntos de itens LR(0)

a: $I_6 = \{ [E \to E + .T],$
$[T \to .T * F],$
$[T \to .F],$
$[F \to .(E)],$
$[F \to .id] \}$

b: $GOTO(I_6, T) = \{ [E \to E + T.], [T \to T. * F] \} = I_9$
$GOTO(I_6, F) = \{ [T \to F.] \} = I_3$
$GOTO(I_6, () = \{ [F \to (.E)], \ldots \} = I_4$
$GOTO(I_6, id) = \{ [F \to id.] \} = I_5$

Coleção canônica de conjuntos de itens LR(0)

a: $I_7 = \{ [T \to T * . F], [F \to .(E)], [F \to .id] \}$

b: $GOTO(I_7, F) = \{ [T \to T * F.] \} = I_{10}$

$GOTO(I_7, () = \{ [F \to (.E)], \ldots \} = I_4$

$GOTO(I_7, id) = \{ [F \to id.] \} = I_5$

Coleção canônica de conjuntos de itens LR(0)

a: $I_8 = \{ [F \to (E.)], [E \to E. + T] \}$

b: $GOTO(I_8,)) = \{ [F \to (E).] \} = I_{11}$

$GOTO(I_8, +) = \{ [E \to E + .T], \dots \} = I_6$

Agora, $C = \{ I_0, \dots, I_8, I_9, I_{10}, I_{11} \}$

Coleção canônica de conjuntos de itens LR(0)

a: $I_9 = \{[E \to E+T.],$
$[T \to T.*F]\}$

b: $GOTO(I_9, *) = \{[T \to T*.F], \ldots\} = I_7$

a: $I_{10} = \{[T \to T*F.]\}$
b: todos GOTO são vazios

a: $I_{11} = \{[F \to (E).]\}$
b: todos GOTO são vazios

Coleção canônica de conjuntos de itens LR(0)

- Neste momento, nenhum novo conjunto de itens é adicionado
- Fim do algoritmo
- Podemos agora montar o autômato LR(0)
  - Esse autômato captura as possíveis transições entre estados de reconhecimento parcial
- Cada conjunto de itens LR(0) é um estado
- As transições são dadas pela função GOTO

Autômato LR(0)

- Coleção canônica

$I_0 = \{ [E' \to .E], [E \to .E+T], [E \to .T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .id] \}$

$I_1 = \{ [E' \to E.], [E \to E.+T] \}$

$I_2 = \{ [E \to T.], [T \to T.*F] \}$

$I_3 = \{ [T \to F.] \}$

$I_4 = \{ [F \to (.E)], [E \to .E+T], [E \to .T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .id] \}$

$I_5 = \{ [F \to id.] \}$

$I_6 = \{ [E \to E+.T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .id] \}$

$I_7 = \{ [T \to T*.F], [F \to .(E)], [F \to .id] \}$

$I_8 = \{ [F \to (E.)], [E \to E.+T] \}$

$I_9 = \{ [E \to E+T.], [T \to T.*F] \}$

$I_{10} = \{ [T \to T*F.] \}$

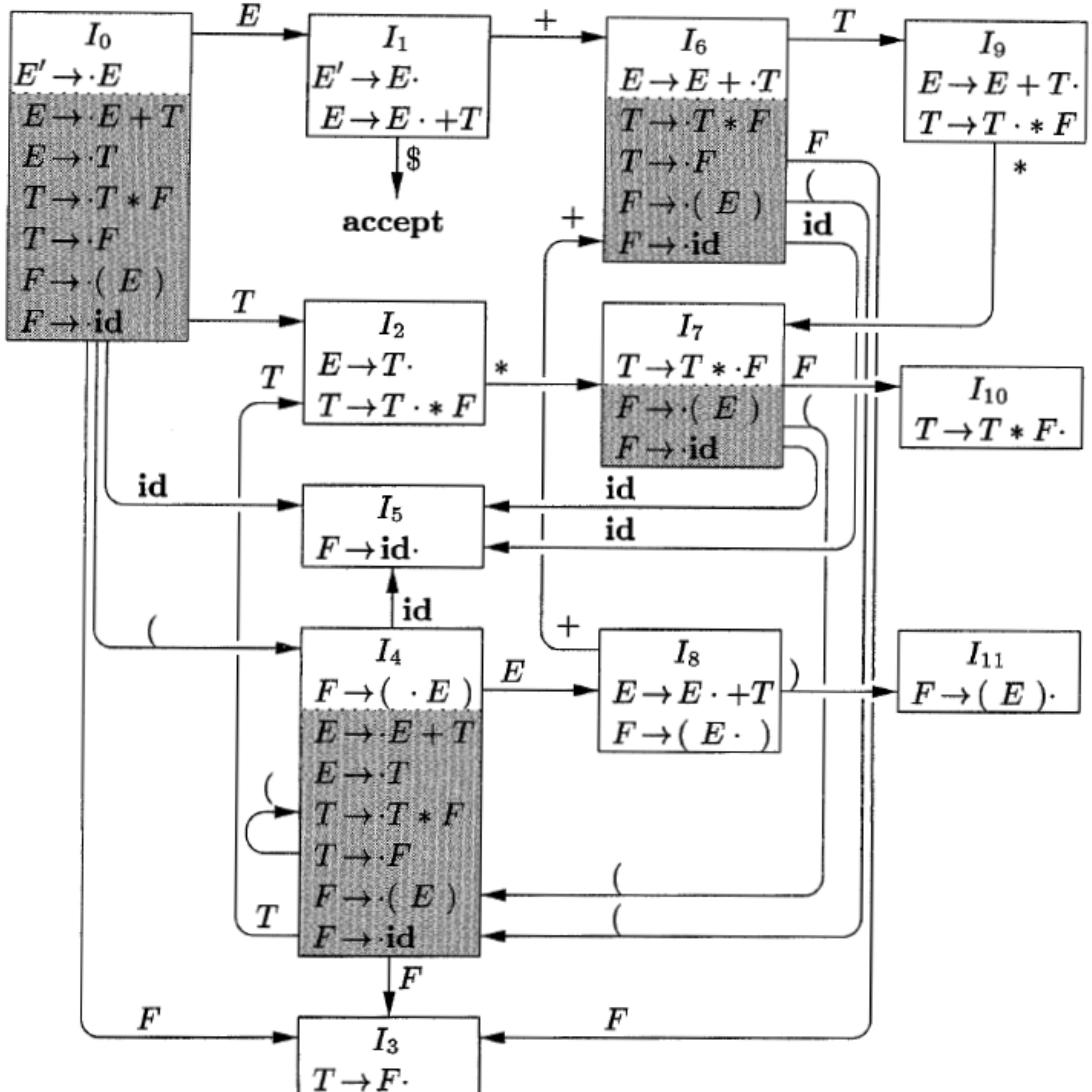
$I_{11} = \{ [F \to (E).] \}$

Autômato LR(0)

- Função GOTO

	id	+	*	(	)	E	T	F
I_0	I_5			I_4		I_1	I_2	I_3
I_1		I_6						
I_2			I_7					
I_3								
I_4	I_5			I_4		I_8	I_2	I_3
I_5								
I_6	I_5			I_4			I_9	I_3
I_7	I_5			I_4				I_{10}
I_8		I_6			I_{11}			
I_9			I_7					
I_{10}								
I_{11}								

Autômato LR(0)



Do autômato LR(0) para a tabela SLR

- O autômato LR(0) mostra o que acontece quando algum símbolo da entrada é consumido
  - Em outras palavras, as transições representam todos os movimentos do tipo “shift”
- Para o analisador SLR, é necessário definir também os movimentos do tipo “reduce”
- Existe um outro algoritmo para isso
  - Mas é preciso conhecer o conjunto SEGUIDORES (A) para cada não-terminal A de uma gramática

Do autômato LR(0) para a tabela SLR

- Exemplo (relembrando):

$$E' \to E$$
$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id$$

```
primeiros(E')={(,id}            seguidores(E')={$}
primeiros(E)={(,id}             seguidores(E)={+,),$}
primeiros(T)={(,id}             seguidores(T)={+,),*,$}
primeiros(F)={(,id}             seguidores(F)={+,),*,$}
```

Do autômato LR(0) para a tabela SLR

- Em seguida, vamos atribuir um índice para cada regra de produção (com exceção da regra aumentada)

	E' → E
(1)	E → E+T
(2)	E → T
(3)	T → T*F
(4)	T → F
(5)	F → (E)
(6)	F → id

Do autômato LR(0) para a tabela SLR

- Agora podemos ver o algoritmo completo

- ENTRADA: uma gramática aumentada G’
- SAÍDA: as funções AÇÃO e TRANSIÇÃO da tabela de análise SLR para G’

Do autômato LR(0) para a tabela SLR

- Algoritmo:

1. Construa C = $\{I_0, I_1, ..., I_n\}$, a coleção de conjuntos de itens LR(0) para G'
2. O estado i é construído a partir de I_i. As ações para o estado são determinadas da seguinte forma:
   (a) Se o item [A→α.aβ] está em I_i e GOTO(I_i,a)=I_j
   então **ACAO[i,a] := shift j** *// a deve ser um terminal*
   (b) Se o item [A→α.] está em I_i
   então **ACAO[i,a] := reduce A→α** para todo **a** em **seguidores(A)** *// A não pode S'*
   (c) Se o item [S'→S.] está em I_i
   então **ACAO[i,$] := OK** *// aceita a cadeia*
3. Se GOTO(I_i,A) = I_j
   entao **TRANSICAO[i,A] := j**
4. As entradas não definidas nas regras 2 e 3 caracterizam **erro**
5. O estado inicial é aquele construído a partir do conjunto de itens contendo [S'→.S]

Do autômato LR(0) para a tabela SLR

- Algoritmo:

1. Construa C = {I_0,I_1,...,I_n}, a cole
   de it
2. O estado i é construído a partir
   para o estado são determ
   seguinte forma:
   (a) Se o item [A→α.aβ] está em I_i e GOTO(I_i,a)=I_j
   então **ACAO[i,a] := shift j** *// a deve ser um terminal*
   (b) Se o item [A→α.] está em I_i
   então **ACAO[i,a] := reduce A→α** para todo **a**
   em **seguidores(A)** *// A não pode S'*
   (c) Se o item [S'→S.] está em I_i
   então **ACAO[i,$] := OK** *// aceita a cadeia*
3. Se GOTO(I_i,A) = I_j
   entao **TRANSICAO[i,A] := j**
4. As entradas não definidas nas regras 2 e 3 caracterizam **erro**
5. O estado inicial é aquele construído a partir do conjunto de
   itens contendo [S'→.S]

Basta copiar da tabela do autômato LR(0), escolhendo a ação "shift"

Tabela SLR

	id	+	*	(	)	E	T	F
I0	I5			I4		I1	I2	I3
I1		I6						
I2			I7					
I3								
I4	I5			I4		I8	I2	I3
I5								
I6	I5			I4			I9	I3
I7	I5			I4				I10
I8		I6			I11			
I9			I7					
I10								
I11								

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2			s7						
3									
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Do autômato LR(0) para a tabela SLR

- Alg...

O passo 2b é feito linha a linha:

$I_1, ..., I_n\}$, a c...

d...

2. O ... é construído a par...

para o estado são dete...

seguinte forma:

(a) Se o item [A→α.aβ] está em I_i ... GOTO(I_i,a)=I_j

então **ACAO[i,a] := shift j** *// a deve ser um terminal*

(b) Se o item [A→α.] está em I_i

então **ACAO[i,a] := reduce A→α** para todo **a**

em **seguidores(A)** *// A não pode S'*

(c) Se o item [S'→S.] está em I_i

... **ACAO[...] := OK** *// aceita a cadeia*

3. ...

... **= j**

... das na ...

5. ... aquele cons...

itens contendo [S'→.S]

Procura todos os itens completos (terminando em ponto) que estão no estado daquela linha

Procura as colunas com terminais que estão em seguidores(A)

E coloca rn nas células correspondentes (onde n é o índice da produção)

Tabela SLR

```
Linha 0:
I0={ [E'→.E],
     [E→.E+T],
     [E→.T],
     [T→.T*F],
     [T→.F],
     [T→.(E)],
     [F→.id]
   }
```

```
Nenhum item
completo!
```

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2			s7						
3									
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Tabela SLR

```
Linha 1:
I1={ [E'→E.],
     [E→E.+T]
   }
```

```
Um item
completo, mas é
com S', então
nada é
adicionado
```

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2			s7						
3									
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Tabela SLR

```
Linha 2:
I2={ [E→T.],
     [T→T.*F]
   }

Item completo:
[E→T.]
seg(E)={+,),$}
Índice E→T = 2
Ação = r2
```

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3									
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Tabela SLR

Linha 3:
I3={ [T→F.] }

Item completo:
[T→F.]
seg(T)={*,+,),$}
Índice T→F = 4
Ação = r4

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Tabela SLR

```
Linha 4:
I4={ [F→(.E)],
     [E→.E+T],
     [E→.T],
     [T→.F*T],
     [T→.F],
     [F→.(E)],
     [F→.id]
 }

Nenhum item
completo!
```

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Tabela SLR

```
Linha 5:
I5={ [F→id.]
 }

Item completo:
[F→id.]
seg(F)={*,+,),$}
Índice F→id = 6
Ação = r6
```

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5		r6	r6		r6	r6			
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Tabela SLR

```
Linhas 6,7,8:
I6={  [E→E+.T],
      [T→.T*F],
      [T→.F],
      [F→.(E)],
      [F→id]
   }
I7={  [T→T*.F],
      [F→.(E)],
      [F→.id]
   }
I8={  [F→(E.)],
      [E→E.+T]
   }
Nenhum item
completo!
```

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5		r6	r6		r6	r6			
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Tabela SLR

```
Linha 9:
I9={ [E→E+T.],
     [T→T.*F]
   }

Item completo:
[E→E+T.]
seg(E)={+,),$}
Índice E→E+T = 1
Ação = r1
```

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5		r6	r6		r6	r6			
6	s5			s4					
7	s5			s4					
8		s6			s11				
9		r1	s7		r1	r1			
10									
11									

Tabela SLR

```
Linha 10:
I10={ [T→T*F.] }

Item completo:
[T→T*F.]
seg(T)={*,+,),$}
Índice T→T*F = 3
Ação = r3
```

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5		r6	r6		r6	r6			
6	s5			s4					
7	s5			s4					
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11									

Tabela SLR

Linha 11:
I11={ [F→(E).] }

Item completo:
[F→(E).]
seg(F)={*,+,),$}
Índice F→(E) = 5
Ação = r5

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

Do autômato LR(0) para a tabela SLR

- Algoritmo:

1. Construa C = {I_0,I_1,...,I_n}, a coleção de conjuntos de itens LR(0) para G'
2. O estado i é construído a partir de I_i. As ações para o estado são deter… seguinte forma:
   (a) Se o item [A→α.aβ] está em …
   então **ACAO[i,a] := shift** … *…erminal*
   (b) Se o item [A→α.] está em I_i
   então **ACAO[i,a] := reduce** … para todo **a** em **seguidores(A)** *// A não pode S'*
   (c) Se o item [S'→S.] está em I_i
   então **ACAO[i,$] := OK** … *…a cadeia*
3. Se GOTO(I_i,A) = I_j
   entao **TRANSICAO[i,A] := j**
4. As entradas não definidas nas regr… **…rro**
5. O estado inicial é aquele construído … …nto de itens contendo [S'→.S]

Equivalente à regra b, mas com o símbolo inicial, escolhendo ação “OK”

O “truque” é achar a linha com um item completo envolvendo S'

Tabela SLR

```
Linha 1:
I1={ [E'→E.],
     [E→E.+T]
   }

Item completo:
[E'→E.]
Ação = OK
```

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

Do autômato LR(0) para a tabela SLR

- Algoritmo:

1. Construa C = {I_0,I_1,...,I_n}, a coleção de conjuntos de itens LR(0) para G'
2. O estado i é construído a partir de I_i. As ações para o estado são determinadas da seguinte forma:
   (a) Se o item [A→α.aβ] está em I_i e GOTO(I_i,a)=I_j
   então **ACAO[i,a] := shift j** *// a deve ser um terminal*
   (b) Se o item [A→α.] está em I_i
   então **ACAO[i,a] := reduce**
   em **seg**… *…ode S'*
   (c) Se o item [S'→S.] está em I_i
   então **ACAO[i,$] := OK** *…a a cadeia*
3. Se GOTO(I_i,A) = I_j
   entao **TRANSICAO[i,A] := j**
4. As entradas não definidas nas regras 2 e 3 caracterizam **erro**
5. O estado inicial é aquele construído a partir do conjunto de itens contendo [S'→.S]

Basta copiar da tabela do autômato LR(0)

Tabela SLR

	id	+	*	(	)	E	T	F
I_0	I_5			I_4		I_1	I_2	I_3
I_1		I_6						
I_2			I_7					
I_3								
I_4	I_5			I_4		I_8	I_2	I_3
I_5								
I_6	I_5			I_4			I_9	I_3
I_7	I_5			I_4				I_{10}
I_8		I_6			I_{11}			
I_9			I_7					
I_{10}								
I_{11}								

Estados	Ações						Transições		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				OK			
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			

Tabela SLR

- Exercício
  - Construa a tabela SLR para a gramática a seguir

```
S → a | ( L )
L → L ; S | S
```

Tabela SLR

- Resposta:
- Passo 1 – aumentar a gramática

```
S’ → S
S → a | ( L )
L → L ; S | S
```

Tabela SLR

- Resposta:
- Passo 2 – calcular a coleção canônica de conjuntos de itens LR(0)

$$C = \text{CLOSURE}(S' \rightarrow .S) = \{ [S' \rightarrow .S], [S \rightarrow .a], [S \rightarrow .(L)] \} = I_0$$

a: $I_0 = \{ [S' \rightarrow .S],$
$[S \rightarrow .a],$
$[S \rightarrow .(L)]$
$\}$

b: $\text{GOTO}(I_0, S) = \{ [S' \rightarrow S.] \} = I_1$
$\text{GOTO}(I_0, a) = \{ [S \rightarrow a.] = I_2$
$\text{GOTO}(I_0, () = \{ [S \rightarrow (.L)], [L \rightarrow .L;S],$
$[L \rightarrow .S], [S \rightarrow .a],$
$[S \rightarrow .(L)] \} = I_3$

Tabela SLR

- Resposta:
- Passo 2 – calcular a coleção canônica de conjuntos de itens LR(0)

a: $I_1 = \{[S' \to S.]\}$

b: todos GOTO são vazios

a: $I_2 = \{[S \to a.]\}$

b: todos GOTO são vazios

Tabela SLR

- Resposta:
- Passo 2 – calcular a coleção canônica de conjuntos de itens LR(0)

a: $I_3 = \{ [S \to (.L)], [L \to .L;S], [L \to .S], [S \to .a], [S \to .(L)] \}$

b: $GOTO(I_3, L) = \{ [S \to (L.)], [L \to L.;S] \} = I_4$

$GOTO(I_3, S) = \{ [L \to S.] \} = I_5$

$GOTO(I_3, a) = \{ [S \to a.] \} = I_2$

$GOTO(I_3, () = \{ [S \to (.L)], [L \to .L;S], [L \to .S], [S \to .a], [S \to .(L)] \} = I_3$

Tabela SLR

- Resposta:
- Passo 2 – calcular a coleção canônica de conjuntos de itens LR(0)

a: $I_4: \{ [S \rightarrow (L.)], \; [L \rightarrow L.;S] \}$

b: $GOTO(I_4,)) = \{ [S \rightarrow (L).] \} = I_6$

$GOTO(I_4, ;) = \{ [L \rightarrow L;.S], \; [S \rightarrow .a], \; [S \rightarrow .(L)] \} = I_7$

Tabela SLR

- Resposta:
- Passo 2 – calcular a coleção canônica de conjuntos de itens LR(0)

a: I_5: $\{ [L \to S.] \}$
b: todos GOTO são vazios

a: $I_6 = \{ [S \to (L).] \}$
b: todos GOTO são vazios

Tabela SLR

- Resposta:
- Passo 2 – calcular a coleção canônica de conjuntos de itens LR(0)

a: $I_7 = \{ [L \rightarrow L;.S], [S \rightarrow .a], [S \rightarrow .(L)] \}$

b: $GOTO(I_7, S) = \{ [L \rightarrow L;S.] \} = I_8$

$GOTO(I_7, a) = \{ [S \rightarrow a.] \} = I_2$

$GOTO(I_7, () = \{ [S \rightarrow (.L)], \ldots \} = I_3$

Tabela SLR

- Resposta:
- Passo 2 – calcular a coleção canônica de conjuntos de itens LR(0)

a: $I_8 = \{ [L \to L;S.] \}$

b: todos GOTO são vazios

Tabela SLR

- Resposta
- Conjunto canônico:

I0={[S'→.S],[S→.a],[S→.(L)]}

I1={[S'→S.]}

I2={[S→a.]}

I3={[S→(.L)],[L→.L;S],[L→.S],
[S→.a],[S→.(L)]}

I4={[S→(L.)],[L→L.;S]}

I5={[L→S.]}

I6={[S→(L).]}

I7={[L→L;.S],[S→.a],[S→.(L)]}

I8={[L→L;S.]}

Tabela SLR

- Resposta
- Autômato LR(0)
  - Função GOTO

GOTO	a	;	(	)	S	L
I_0	I_2		I_3		I_1	
I_1						
I_2						
I_3	I_2		I_3		I_5	I_4
I_4		I_7		I_6		
I_5						
I_6						
I_7	I_2		I_3		I_8	
I_8						

Tabela SLR

- Calculando conjuntos seguidores

$S' \rightarrow S$

$S \rightarrow a \mid (L)$

$L \rightarrow L ; S \mid S$

```
primeiros(E')={a,(}
primeiros(S)={a,(}
Primeiros(L)={a,(}
```

```
seguidores(S')={$}
seguidores(S)={$,),;}
seguidores(L)={),;}
```

Tabela SLR

- Indexando a gramática

```
      S' → S
(1)   S → a
(2)   S → ( L )
(3)   L → L ; S
(4)   L → S
```

Tabela SLR

- Movimentos “shift”

GOTO	a	$;$	$($	$)$	S	L
I_0	I_2		I_3		I_1	
I_1						
I_2						
I_3	I_2		I_3		I_5	I_4
I_4		I_7		I_6		
I_5						
I_6						
I_7	I_2		I_3		I_8	
I_8						

Estados	Ações					Transições	
	a	**;**	**(**	**)**	**$**	**S**	**L**
0	s2		s3				
1							
2							
3	s2		s3				
4		s7		s6			
5							
6							
7	s2		s3				
8							

Tabela SLR

- Movimentos "reduce"

```
Linha 0:
I0={[S'→.S],
    [S→.a],
    [S→.(L)]
   }

Nenhum item
completo!
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2							
3	s2		s3				
4		s7		s6			
5							
6							
7	s2		s3				
8							

Tabela SLR

- Movimentos “reduce”

```
Linha 1:
I1={[S’→S.]}

Item completo:
[S’→S.]
Mas envolve S’,
portanto ignora
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2							
3	s2		s3				
4		s7		s6			
5							
6							
7	s2		s3				
8							

Tabela SLR

- Movimentos “reduce”

```
Linha 2:
I2={[S→a.]}

Item completo:
[S→a.]
seg(S)={$,),;}
Índice S→a = 1
Ação = r1
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2		r1		r1	r1		
3	s2		s3				
4		s7		s6			
5							
6							
7	s2		s3				
8							

Tabela SLR

- Movimentos “reduce”

```
Linha 3:
I3={[S→(.L)],
    [L→.L;S],
    [L→.S],
    [S→.a],
    [S→.(L)]
   }

Nenhum item
completo!
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2		r1		r1	r1		
3	s2		s3				
4		s7		s6			
5							
6							
7	s2		s3				
8							

Tabela SLR

- Movimentos “reduce”

```
Linha 4:
I4={[S→(L.)],
    [L→L.;S]}

Nenhum item
completo!
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2		r1		r1	r1		
3	s2		s3				
4		s7		s6			
5							
6							
7	s2		s3				
8							

Tabela SLR

- Movimentos “reduce”

```
Linha 5:
I5={[L→S.]}

Item completo:
[L→S.]
seg(L)={),;}
Índice L→S = 4
Ação = r4
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2		r1		r1	r1		
3	s2		s3				
4		s7		s6			
5		r4		r4			
6							
7	s2		s3				
8							

Tabela SLR

- Movimentos “reduce”

```
Linha 6:
I6={[S→(L).]}

Item completo:
[S→(L).]
seg(S)={$,),;}
Índice S→(L) = 2
Ação = r2
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2		r1		r1	r1		
3	s2		s3				
4		s7		s6			
5		r4		r4			
6		r2		r2	r2		
7	s2		s3				
8							

Tabela SLR

- Movimentos “reduce”

```
Linha 7:
I7={[L→L;.S],
    [S→.a],
    [S→.(L)]
  }

Nenhum item
completo!
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2		r1		r1	r1		
3	s2		s3				
4		s7		s6			
5		r4		r4			
6		r2		r2	r2		
7	s2		s3				
8							

Tabela SLR

- Movimentos “reduce”

```
Linha 8:
I8={[L→L;S.]}

Item completo:
[L→L;S.]
seg(L)={),;}
Índice L→L;S = 3
Ação = r3
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1							
2		r1		r1	r1		
3	s2		s3				
4		s7		s6			
5		r4		r4			
6		r2		r2	r2		
7	s2		s3				
8		r3		r3			

Tabela SLR

- Movimentos “OK”

```
Linha 1:
I1={[S’→S.]}

Item completo:
[S’→S.]
Ação = OK
```

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3				
1					OK		
2		r1		r1	r1		
3	s2		s3				
4		s7		s6			
5		r4		r4			
6		r2		r2	r2		
7	s2		s3				
8		r3		r3			

Tabela SLR

- Transições

GOTO	a	;	(	)	S	L
I_0	I_2		I_3		I_1	
I_1						
I_2						
I_3	I_2		I_3		I_5	I_4
I_4		I_7		I_6		
I_5						
I_6						
I_7	I_2		I_3		I_8	
I_8						

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3			1	
1					OK		
2		r1		r1	r1		
3	s2		s3			5	4
4		s7		s6			
5		r4		r4			
6		r2		r2	r2		
7	s2		s3			8	
8		r3		r3			

Tabela SLR

- Resposta!

Estados	Ações					Transições	
	a	;	(	)	$	S	L
0	s2		s3			1	
1					OK		
2		r1		r1	r1		
3	s2		s3			5	4
4		s7		s6			
5		r4		r4			
6		r2		r2	r2		
7	s2		s3			8	
8		r3		r3			

Conflitos

- Gramática SLR(1)
  - Uma gramática é SLR(1) sse, para qualquer estado s, as duas condições a seguir são satisfeitas:
    - Para qualquer item $[A \rightarrow \alpha . X \beta]$ em s em que X é um terminal, não existe um item completo $[B \rightarrow \gamma .]$ em s com X em seguidores(B)
    - Para quaisquer dois itens completos $[A \rightarrow \alpha .]$ e $[B \rightarrow \beta .]$ em s, seguidores(A) ∩ seguidores(B) é vazio
- Violações nessas condições geram conflitos, respectivamente:
  - Empilha-reduz *(shift-reduce)*
  - Reduz-reduz *(reduce-reduce)*
- Na tabela, isso aparece na forma de duas ou mais ações em uma mesma célula

Conflitos

- Exemplo
- Conflito empilha-reduz

```
decl → decl-if | 'outra'
decl-if → 'if' '(' exp ')' decl
        | 'if' '(' exp ')' decl 'else' decl
exp → '0' | '1'
```

- Na gramática acima, haverá um conflito na transição

```
[decl-if → 'if' '(' exp ')' decl.]
[decl-if → 'if' '(' exp ')' decl. 'else' decl]
```

- Trata-se de um conflito empilha-reduz, já que
  - O item completo indica que uma redução deve ocorrer
  - Enquanto o outro item que o ‘else’ deve ser empilhado

Conflitos

- Exemplo
- Conflito reduz-reduz

```
decl → ativa-decl | atrib-decl
ativa-decl → ID
atrib-decl → var ':=' exp
var → ID
exp → var | NUM
```

- Na gramática acima, há um estado no qual estão presentes os seguintes itens

```
[ativa-decl→ID.]
[var→ID.]
```

- Ocorre um conflito reduz-reduz, já que
  - `seguidores(ativa-decl) = {$}`
  - `seguidores(var) = {':=', $}`
- Nesse caso a redução deve ocorrer:
  - com símbolo **$** para `ativa-decl→ID`
  - com símbolo **':='** para `var→ID`

Conflitos

- Resolução de conflitos
  - Empilha-reduz
    - Opção default (sempre empilha)
    - Resolve a ambiguidade do “else sobrando”, pois um else deve sempre ser agrupado com o if mais próximo
  - Ex:

```
if (1) then if (2) then outra else outra
```

Decisão errada:

Pilha	**Entrada**	**Ação**
if (exp) then if(exp) then outra	else outra $	reduce
if (exp) then decl-if	else outra $	shift
if (exp) then decl-if else	outra $	shift
if (exp) then decl-if else outra	$	reduce
decl-if	$	

Conflitos

- Resolução de conflitos
  - Empilha-reduz
    - Opção default (sempre empilha)
    - Resolve a ambiguidade do “else sobrando”, pois um else deve sempre ser agrupado com o if mais próximo
  - Ex:

```
if (1) then if (2) then outra else outra
```

Decisão certa:

Pilha	**Entrada**	**Ação**
if (exp) then if(exp) then outra	else outra $	shift
if (exp) then if(exp) then outra else	outra $	shift
if (exp) then if(exp) then outra else outra	$	reduce
if (exp) then decl-if	$	reduce
decl-if	$	

Conflitos

- Resolução de conflitos
  - Empilha-reduz
    - Associatividade/precedência
    - Resolve ambiguidades de expressões
- Consiste em definir explicitamente a relação de precedência/associatividade entre os terminais
  - Ex: * > +, + > +, etc
- No momento da dúvida:
  - a = o terminal mais à direita na pilha
  - b = a entrada atual
  - Se a>b, reduz
  - Se a<b, empilha

Conflitos

- Ex:

  `id+id*id`

Momento de dúvida, pois existe um estado com os seguintes itens:

`[E→E+E.]`

`[E→E.*E]`

e * está em seguidores(E)

Pilha	Entrada	Ação
E + E	* id $	shift
E + E *	id $	shift
E + E * id	$	reduce
E + E * E	$	reduce
E + E	$	reduce
E	$	...

Neste exemplo

a = +
b = *

+ < *, portanto empilha

Conflitos

- Ex:

  `id+id+id`

Momento de dúvida, pois existe um estado com os seguintes itens:

`[E→E+E.]`

`[E→E.+E]`

e + está em seguidores(E)

Pilha	Entrada	Ação
E + E	+ id $	reduce
E	+ id $	shift
E +	id $	shift
E + id	$	reduce
E + E	$	reduce
E	$	...

Neste exemplo

a = +
b = +

+ > +, portanto reduz

Conflitos

- Resolução de conflitos
  - Empilha-reduz
    - Modificar a gramática
- Ex:
  - De:

$$E \rightarrow E+E \mid E*E \mid id$$

  - Para:

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

$$T \rightarrow T*F \mid F$$

$$F \rightarrow id$$

Conflitos

- Resolução de conflitos
  - Reduz-reduz
    - Normalmente indica ambiguidade ou outro problema no projeto da gramática
    - Precisa re-escrever a gramática
    - Não existe regra para isso
    - É uma das desvantagens da análise LR
- Mas na análise SLR existem muitos conflitos que surgem porque o método não é suficientemente poderoso
  - Existem outros métodos de análise LR melhores
  - Análise LR(1) canônica
  - Análise LALR

Análise LR(1) canônica

- Similar à SLR, mas considerando um conjunto maior de itens
  - Itens LR(1)
- Consiste em estender a coleção de itens LR(0), para incluir um segundo componente
  - Um símbolo lookahead
  - LR(1) = itens com 1 símbolo lookahead
- Formato dos itens:
  - `[A→α.β,a]`
  - onde A→αβ é uma produção, e a é um terminal ou $

Análise LR(1) canônica

- Os itens LR(1) servem para restringir a decisão pela redução
  - Ou seja, um item `[A→α.,a]` indica que a redução A→α só ocorre se o próximo símbolo de entrada for a
- A construção da coleção de itens LR(1) é feita da mesma forma
  - Só mudam as funções GLOSURE e GOTO

Fechamento de conjuntos de itens LR(1)

- Algoritmo:

```
SetOfItems CLOSURE(I) {
   J := I;
   repeat
      for(cada item "[A→α.Bβ,a]" em J)
         for(cada produção "B→γ" de G)
            for(cada terminal b em primeiros(βa))
               if("B→.γ" não está em J)
                  adicione "[B→.γ,b]" em J;
   until nenhum item seja adicionado a
         J em um passo do loop;
   return J;
}
```

Mudanças

Função GOTO para itens LR(1)

- Nada muda, apenas é necessário analisar os itens considerando também o componente lookahead
- As transições serão exatamente as mesmas, exceto que só é possível fazer a transição entre itens com o mesmo componente lookahead
- Ex:
  - GOTO({[E.+E,$]}) = {[E+.E,$],...}

Tabela de análise LR(1) canônica

- Algoritmo:

1. Construa C' = {I_0,I_1,...,I_n}, a coleção de conjuntos de itens LR(1) para G'
2. O estado i é construído a partir de I_i. As ações para o estado são determinadas da seguinte forma:
   (a) Se o item [A→α.aβ,b] está em I_i e GOTO(I_i,a)=I_j
   então **ACAO[i,a] := shift j** *// a deve ser um terminal*
   (b) Se o item [A→α.,a] está em I_i
   então **ACAO[i,a] := reduce A→α** ~~para todo **a** em **seguidores(A)**~~
   (c) Se o item [S'→S.,$] está em I_i
   então **ACAO[i,$] := OK** *// aceita a cadeia*
3. Se GOTO(I_i,A) = I_j
   entao **TRANSICAO[i,A] := j**
4. As entradas não definidas nas regras 2 e 3 caracterizam **erro**
5. O estado inicial é aquele construído a partir do conjunto de itens contendo [S'→.S,$]

Mudanças

Análise LR(1) canônica

- Exemplo
  - Conflito reduz-reduz

```
decl → ativa-decl | atrib-decl
ativa-decl → ID
atrib-decl → var ':=' exp
var → ID
exp → var | NUM
```

- Na gramática acima, para o conflito anterior teríamos

```
[ativa-decl→ID.,$]
[var→ID.,':=']
```

- O conflito é solucionado já que
  - Os itens LR(1) diferenciam as reduções com base nas suas verificações à frente:
    - Reduzir usando a primeira opção se o símbolo à frente for ‘**$**’
    - Reduzir usando a segunda se for ‘**:=**’

Análise LR(1) canônica

- Essencialmente, a tabela LR(1) canônica inclui todas as possibilidades de movimentos considerando-se um símbolo à frente
  - Ou seja, os itens incluem toda combinação de movimentos empilha/reduz para TODO símbolo terminal
  - Isso resulta em tabelas muito grandes

Análise LALR (*Lookahead* LR)

- Frequentemente usado na prática
  - Tabelas consideravelmente menores do que as LR canônicas
  - Gramáticas LALR são capazes de expressar a maioria das construções sintáticas comuns das linguagens de programação
  - Não computa o conjunto completo de itens LR(1) na prática
- Vantagens
  - Preserva alguns dos benefícios da análise sintática SLR (1)
  - Também preserva o menor tamanho do autômato de itens LR(0)
  - Porém utiliza a capacidade lookahead de (alguns) itens LR(1)

Análise LALR

- Outra característica interessante:
- Analisadores LR canônicos e LALR se comportam da mesma forma se a entrada for correta
  - Significa que farão a mesma sequência de shifts e reduces
- Quando houver uma entrada errada:
  - O analisador LALR pode tentar fazer umas reduções a mais antes de detectar o erro
    - Ou seja, o analisador LR detecta o erro antes
  - Mas o LALR vai eventualmente encontrar o erro
    - Na verdade, antes de fazer o próximo shift
- Em resumo: LALR é fortemente indicado!
  - Por isso é o mais usado na prática para análise bottom-up

Recuperação de erros na análise LR

- Erros são detectados ao ler a tabela AÇÃO
  - Nunca ao ler a tabela TRANSIÇÃO
- Modo pânico:
  - Procure na pilha (de cima para baixo) um estado que tenha um GOTO para um não-terminal A
    - Ou seja, ache o momento em que A começou a ser analisado
  - Joga fora todos os estados da pilha, incluindo esse encontrado
  - Então basta ignorar os símbolos de entrada, até achar um que possa vir legitimamente após A
  - Essa estratégia tenta eliminar a frase contendo o erro sintático. O analisador volta até o começo, “finge” que reconheceu A corretamente, e continua a análise

Recuperação de erros na análise LR

- Recuperação em nível de frase
  - Examina-se cada entrada de erro na tabela e decide-se, com base na linguagem, que comportamento errado do programador poderia gerar esse erro
    - Pode-se projetar uma rotina de recuperação
    - Cada entrada vazia da tabela pode ser um ponteiro para uma rotina
    - As ações da rotina podem incluir inserção/remoção de símbolos da pilha/entrada
- Estratégia é normalmente ad hoc

ASA vs ASD

- Características
  - Os métodos de análise sintática ascendente são mais poderosos (capazes de lidar com um número maior de gramáticas)
  - As verificações à frente na pilha são mais simples do que as verificações à frente na entrada
- LL(k) vs. LR(k)
  - LL(k)
    - A produção é selecionada olhando-se apenas os k primeiros símbolos que seu lado direito pode derivar
  - LR(k)
    - O lado direito de uma produção é reconhecido conhecendo-se tudo que foi derivado a partir desse lado direito (na pilha) mais o esquadrinhamento antecipado (lookahead) de k símbolos (da cadeia de entrada)
- LR é mais poderoso do que LL

ASA vs ASD

- Análise sintática ascendente LR
- Desvantagens
  - Exige manipulação complexa da tabela sintática
  - Construção trabalhosa
  - (manual) de um analisador sintático LR para uma gramática típica de uma linguagem de programação
  - Conflitos reduce/reduce (pedra no sapato)
  - Resolução de conflitos é mais trabalhosa

ASA vs ASD

- ASA
  - Particularmente boa para expressões
  - Mais flexibilidade nas regras
  - Pior para depurar e resolver conflitos
  - Mais “declarativa”
- ASD (em particular o analisador recursivo)
  - Exige mais trabalho para definir a gramática
    - Não tolera recursão à esquerda, precisa fatorar às vezes
    - Problema reduzido com EBNF e técnica LL(*)
  - Mas depois é mais fácil depurar e resolver não-determinismos
  - Mais “imperativa”

ASA vs ASD

- Em resumo
  - Vai do gosto do freguês
  - Apenas a prática permite decidir
- ASD
  - ANTLR
  - Um dos mais utilizados atualmente (segundo o google)
- ASA
  - Yacc,Bison,muitos outros
  - Sempre foi mais utilizada (mas estão perdendo terreno para o ANTLR)

Fim