III. Parsing

Parsing é o processo pelo qual é possível determinar se um string pode ser gerado por uma gramática.

Grafo mais à esquerda de uma gramática

É um grafo dirigido onde os nós representam formas sentenciais de uma derivação e os arcos indicam a regra aplicada. Ele contém todas as derivações mais à esquerda possíveis de serem obtidas a partir de S.

Ex: $S \rightarrow aS \mid bB \mid \lambda$
$B \rightarrow aB \mid b$

Parsing Descendente (Top–Down)

Um algoritmo de parsing descendente busca derivações mais à esquerda que produzam o string examinado, a partir de S.

Parsing Descendente Por Profundidade

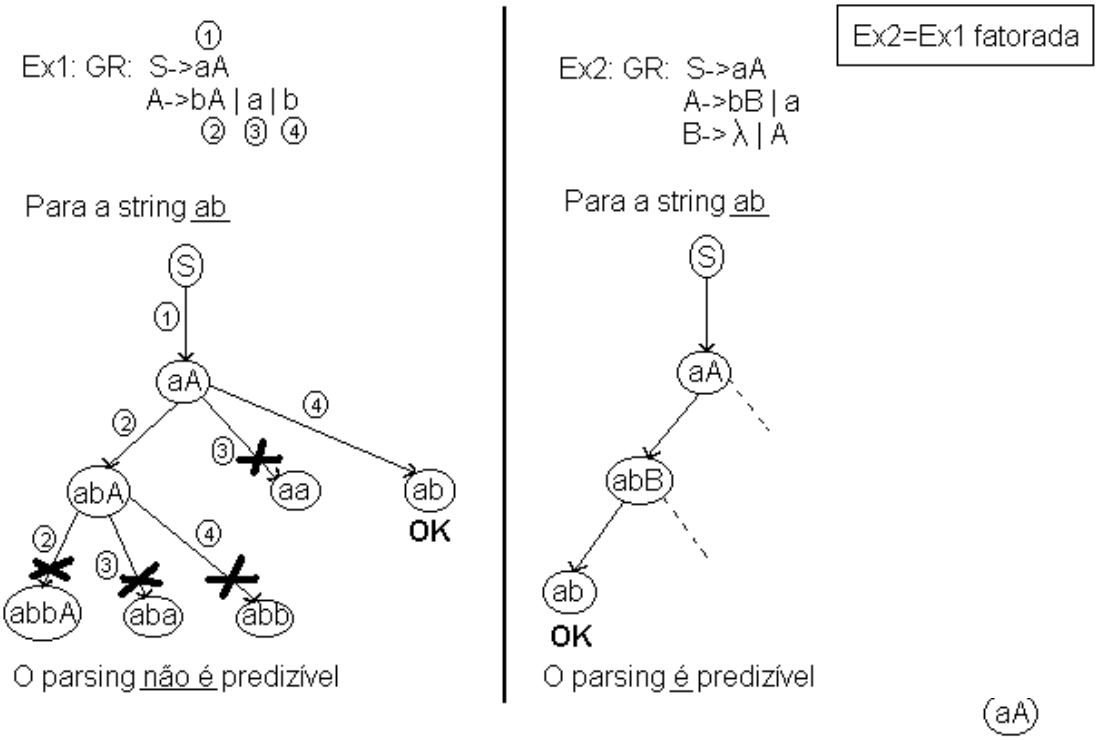
A partir da raiz S, cada nodo é expandido em profundidade, gerando um galho da árvore, até que seja encontrada uma folha correspondente ao string examinado, ou até que a expansão gere um nodo filho com prefixo diferente do string. Neste caso, o algoritmo deve retornar ao último nodo visitado que permita uma produção alternativa para a geração do string (backtracking). O processo termina quando o string é encontrado ou quando não houver nodos a serem expandidos;

Obs.: O processo pode não terminar caso a gramática possua recursão direta à esquerda.

Parsing Descendente Preditivo

Em uma forma particular do parsing descendente por profundidade, não há necessidade de retornar a vértices internos da árvore para se tentar um novo galho de derivação. Isto ocorre quando utilizamos gramáticas LL(1), que evitam a escolha de um caminho inútil. Neste caso, ou o string é derivado até o fim, ou ocorre um erro sintático. Parsers deste tipo são denominados “preditivos”.

Uma GLC é dita LL(1) se durante o processo de parsing de cada um



dos strings da linguagem, a regra de produção a ser utilizada em cada passo de derivação puder ser determinada de forma única, através da leitura do próximo token (lookahead).

Conjuntos FIRST e FOLLOW

A construção de analisadores descendentes e ascendentes é auxiliada por duas funções, FIRST e FOLLOW, associadas a uma gramática G.

Para calcular o FIRST(X) de todos os símbolos X da gramática, aplique as seguintes regras até que não haja mais terminais ou ϵ que possam ser acrescentados a algum dos conjuntos FIRST.

1. Se X é um símbolo terminal, então FIRST(X) = $\{X\}$.
2. Se X é um símbolo não-terminal e $X \rightarrow Y_1 Y_2 \cdots Y_k$ é uma produção para algum $k \geq 1$, então acrescente a a FIRST(X) se, para algum i, a estiver em FIRST(Y_i), e ϵ estiver em todos os FIRST(Y_1),..., FIRST(Y_{i-1}); ou seja, $Y_1 \cdots Y_{i-1} \Rightarrow \epsilon$. Se ϵ está em FIRST(Y_j) para todo $j = 1, 2, ..., k$, então adicione ϵ a FIRST(X). Por exemplo, tudo em FIRST(Y_1) certamente está em FIRST(X). Se Y_1 não derivar ϵ, então não acrescentamos mais nada a FIRST(X), mas se $Y_1 \Rightarrow \epsilon$, então adicionamos FIRST(Y_2), e assim por diante.
3. Se $X \rightarrow \epsilon$ é uma produção, então acrescente ϵ a FIRST(X).

Para calcular o FOLLOW(A) para todos os não-terminais A, aplique as seguintes regras até que nada mais possa ser acrescentado a nenhum dos conjuntos FOLLOW.

1. Coloque \$ em FOLLOW(S), onde S é o símbolo inicial da gramática, e \$ é o marcador de fim da entrada ou fim de arquivo.
2. Se houver uma produção $A \rightarrow \alpha B \beta$, então tudo em FIRST($\beta$) exceto ϵ está em FOLLOW(B).
3. Se houver uma produção $A \rightarrow \alpha B$, ou uma produção $A \rightarrow \alpha B \beta$, onde o FIRST($\beta$) contém ϵ, então inclua o FOLLOW(A) em FOLLOW(B).

Tabela de Parsing Preditivo

É um dispositivo prático para se determinar se uma gramática é LL(1). Se a tabela possuir mais de uma regra em alguma de suas entradas, a gramática não será LL(1).

Algorithm 4.31 : Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

Exemplo:

Mostre que a gramática abaixo é LL(1):

S → aAb | bB | λ
A → Ba | bA
B → c | λ

Exemplo:

Mostre que a gramática abaixo não é LL(1):

S → ABc | a
A → b | λ
B → a | b | cB | λ

Parsing Ascendente (Bottom-Up)

Um algoritmo de parsing ascendente busca reduções mais à direita que transformem o string examinado no símbolo S. Uma redução é a substituição de uma seqüência de símbolos w pelo não terminal A, a partir de uma regra A →w. Neste caso, w é denominado handle. Os analisadores ascendentes incluem LR(0), SLR, LR(1) e o LALR.

O analisador sintático shift-reduce

O analisador sintático shift-reduce é uma forma de análise ascendente em que uma pilha contém símbolos da gramática e um buffer de entrada contém o restante da cadeia a ser reconhecida sintaticamente. Usamos o símbolo $ para marcar o fundo da pilha e também o extremo direito da entrada.

- Inicialmente, a pilha está vazia, e a cadeia w representa a entrada.
- O analisador transfere símbolos da entrada para a pilha, até que uma cadeia de símbolos no topo da pilha possa ser reduzida para o lado esquerdo de uma produção.
- O analisador repete esse ciclo até detectar um erro ou até que a pilha contenha apenas o símbolo inicial e a entrada esteja vazia
- Durante o processo podem ocorrer conflitos quando o analisador não sabe se transfere ou reduz

Ex: string id * id

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

PILHA	ENTRADA	AÇÃO
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	transfere
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduz segundo $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduz segundo $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	transfere
$ $T *$	$\mathbf{id}_2$ $	transfere
$ $T * \mathbf{id}_2$	$	reduz segundo $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduz segundo $T \rightarrow T * F$
$ T	$	reduz segundo $E \rightarrow T$
$ E	$	aceita

FIGURA 4.28 Configurações de um analisador shift-reduce sob a entrada $\mathbf{id}_1 * \mathbf{id}_2$.

Parsers LR(0) e SLR(1)

Itens e o autômato LR(0)

Para a construção do analisador ascendente precisamos determinar um conjunto de itens a partir das regras da gramática, que indicam o ponto do processo de reconhecimento.

Uma regra da forma E → E + T produzirá 4 itens:
E → . E + T
E → E . + T
E → E + . T
E → E + T .

A gramática a ser analisada precisa ser estendida com um novo símbolo inicial:

S' → S

Definimos também 2 funções: o Fechamento de itens (closure) e a função de transição (GoTo).

Função de fechamento

Fechamento de conjuntos de itens

Se I é um conjunto de itens para uma gramática G, então CLOSURE(I) é o conjunto de itens construídos a partir de I pelas duas regras:

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

Ex:

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

Se um conjunto de itens contém o item E’ → . E, então deverão ser acrescentados os itens:

E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

A função de transição

Representada por GOTO(I,X), onde I é um conjunto de itens e X é um símbolo da gramática. GOTO(I,X) é definido como o fechamento do conjunto de todos os itens [A →aX.B] tais que [A→a. XB] está em I.

Intuitivamente, a função de transição GOTO é utilizada para definir as transições no autômato LR(0) para urna gramática. Os estados do autômato correspondem aos conjuntos de itens, e GOTO(I,X) especifica a transição do estado I para um novo estado sob a entrada X.

Ex: Se I representa o conjunto [[E' → E .],[E → E . + T]] então GOTO(I,+) contém:

E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

A coleção canônica de conjuntos de itens LR(0)

```
void itens(G') {
    C = CLOSURE({[S'→ ·S]});
    repeat
        for ( cada conjunto de itens I em C )
            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;
}
```

FIGURA 4.33 Computação da coleção canônica de conjuntos de itens LR(0).

O autômato LR(0)

O estado inicial é dado por CLOSURE([S' → . S]) . Cada conjunto de itens forma um estado e todos são finais.

As decisões entre shift-reduce podem ser feitas como a seguir: Estando no estado j, avance sob o próximo símbolo *a* da entrada se o estado possuir uma transição sob *a*. Caso contrário, escolha reduzir; os itens no estado j nos dirão qual produção utilizar.

Ex:

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

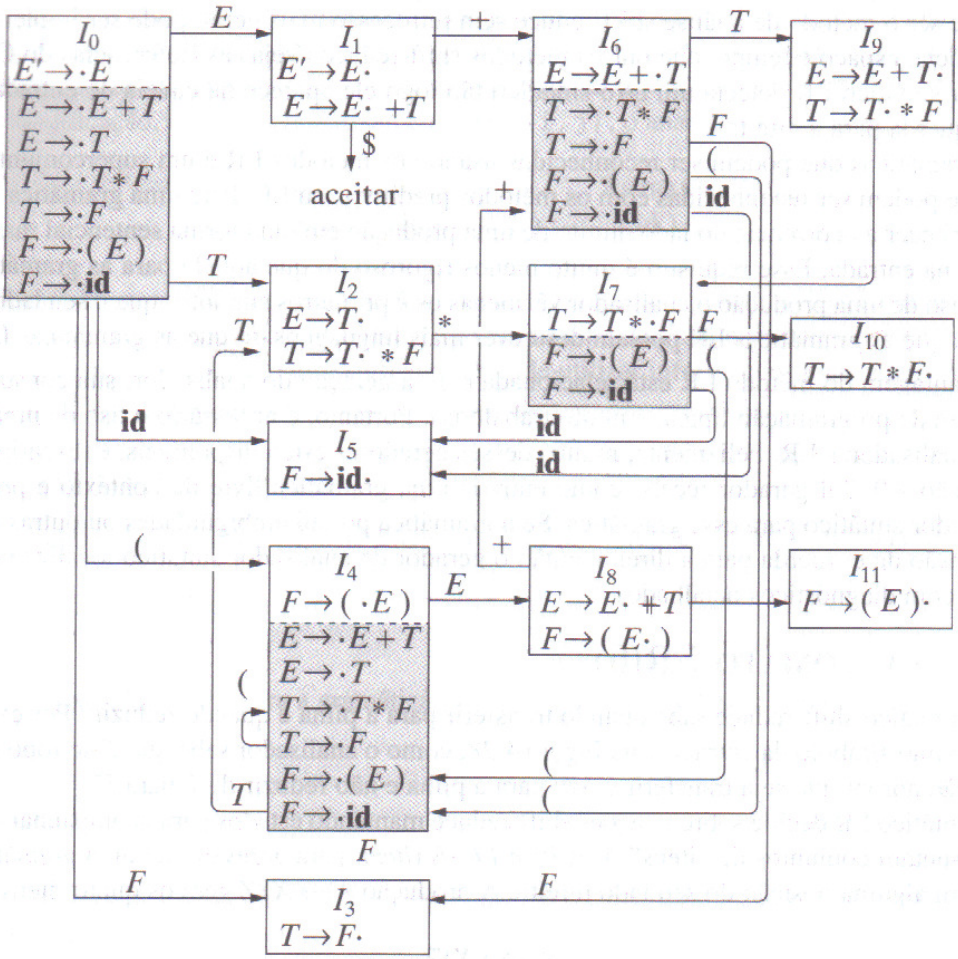


FIGURA 4.31 Autômato LR(0) para a gramática da expressão (4.1).

O Algoritmo de análise LR

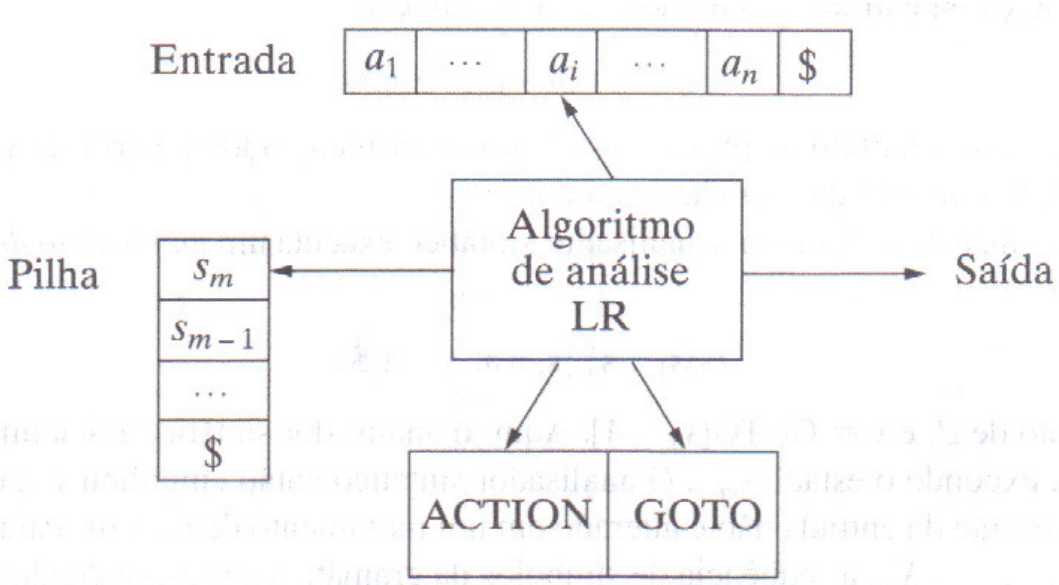


FIGURA 4.35 Modelo de um analisador sintático LR.

Estrutura da tabela de análise LR

A tabela de análise LR consiste em duas partes ACTION, uma função de ação de análise, e GOTO, uma função de transição.

1. A função ACTION recebe como argumentos um estado i e um terminal a ,ou $. O valor de ACTION[i, a] pode ser:

(a) Shift j, onde j é um estado. A ação tomada pelo analisador sintático efetivamente transfere a entrada *a* para a pilha, mas usa o estado j para representar *a*.
(b) Reduce A → w. A ação do analisador sintático efetivamente reduz w no topo da pilha para A.

(c) Accept. O analisador sintático aceita a entrada e termina a análise.
(d) Error. O analisador sintático descobre um erro em sua entrada.

2. Estendemos a função GOTO, definida sobre os conjuntos de itens, para estados: se GOTO[Ii , A] = Ij , então GOTO também mapeia um estado i e um não-terminal A para o estado j.

Comportamento do analisador LR

É essencialmente o mesmo de um analisador sintático shift-reduce; a única diferença é que, em vez de símbolos da gramática, a pilha contém estados, a partir dos quais os símbolos da gramática podem ser recuperados.

O próximo movimento do analisador, a partir da configuração dada, é determinado pela leitura de *a*, o símbolo de entrada corrente, e *s*, o estado no topo da pilha, e então consultando a entrada ACTION [s,a] na tabela de ação de análise.

As movimentações possíveis são as seguintes:

1. Se ACTION[s,a] = shift p, o analisador empilha o próximo estado p e avança na entrada.
2. Se ACTION[s,a] = reduce A —> B, o analisador executa uma redução, removendo a representação de B da pilha e empilhando GOTO(topo,A).
3. Se ACTION[s,a] = accept, a análise está concluída.
4. Se ACTION[s,a] = error, o analisador sintático descobriu um erro.

Todos os analisadores sintáticos LR se comportam dessa maneira; a única diferença entre eles diz respeito à informação contida nos campos ACTION e GOTO da tabela de análise.

ALGORITMO 4.44: Algoritmo de análise LR.

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

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

MÉTODO: Inicialmente, o analisador sintático possui $w\$$ no buffer de entrada e s_0 em sua pilha, onde s_0 representa o estado inicial. O analisador sintático, então, executa o algoritmo da Figura 4.36.

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

FIGURA 4.36 Algoritmo de análise LR.

Ex:

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

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

FIGURA 4.37 Tabela de análise para a gramática de expressão.

Os códigos para as ações são:
1. si significa avança na entrada e empilha o estado i na pilha,
2. ri significa reduce segundo a produção de número i,
3. acc significa accept,
4. entrada em branco significa error.

Para a entrada id * id + id, a seqüência de movimentos do analisador sintático mostrando o conteúdo de pilha e da entrada é dado por

	PILHA	SÍMBOLOS	ENTRADA	AÇÃO
(1)	0		**id** $*$ **id** + **id** $	empilha 5 e avança
(2)	0 5	**id**	$*$ **id** + **id** $	reduz segundo $F \rightarrow$ **id**
(3)	0 3	F	$*$ **id** + **id** $	reduz segundo $T \rightarrow F$
(4)	0 2	T	$*$ **id** + **id** $	empilha 7 e avança
(5)	0 2 7	$T*$	**Id** + **id** $	empilha 5 e avança
(6)	0 2 7 5	$T*$ **id**	+ **id** $	reduz segundo $F \rightarrow$ **id**
(7)	0 2 7 10	$T*F$	+ **id** $	reduz segundo $T \rightarrow T*F$
(8)	0 2	T	+ **id** $	reduz segundo $E \rightarrow T$
(9)	0 1	E	+ **id** $	empilha 6 e avança
(10)	0 1 6	$E+$	**id** $	empilha 5 e avança
(11)	0 1 6 5	$E+$ **id**	$	reduz segundo $F \rightarrow$ **id**
(12)	0 1 6 3	$E+F$	$	reduz segundo $T \rightarrow F$
(13)	0 1 6 9	$E+T$	$	reduz segundo $E \rightarrow E+T$
(14)	0 1	E	$	aceitar

FIGURA 4.38 Movimentos de um analisador LR para a entrada **id** $*$ **id** + **id**.

A Construção de tabelas de análise SLR(1)

ALGORITMO 4.46: Construção de uma tabela SLR.

ENTRADA: Uma gramática estendida G'.

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

MÉTODO:

1. Construa $C = \{I_0, I_1, \ldots, I_n\}$, a coleção de conjuntos de itens LR(0) para G'.
2. O estado i é construído a partir de I_i. As ações de reconhecimento sintático para o estado i são determinadas da seguinte forma:
   (a) Se o item $[A \rightarrow \alpha \cdot a\beta]$ está em I_i e GOTO$(I_i, a) = I_j$, então defina ACTION$[i, a]$ como "shift j". Aqui, a deve ser um terminal.
   (b) Se o item $[A \rightarrow \alpha \cdot]$ está em I_i, então defina ACTION$[i,a]$ como "reduce $A \rightarrow \alpha$" para todo a em FOLLOW(A); aqui, A pode não pode ser S', o símbolo inicial da gramática.
   (c) Se o item $[S' \rightarrow S \cdot]$ estiver em I_i, então defina ACTION$[i, \$]$ como "accept".

   Se quaisquer ações de conflito resultar das regras anteriores, dizemos que a gramática não é SLR(1). O algoritmo deixa de produzir um analisador sintático nesse caso.
3. As transições **goto** para o estado i são construídas para todos os não-terminais A usando a regra: Se GOTO$(I_i, A) = I_j$, então GOTO$[i, A] = j$.
4. Todas as entradas não definidas pelas regras (2) e (3) caracterizam "error".
5. O estado inicial do analisador é aquele construído a partir do conjunto de itens contendo $[S' \rightarrow \cdot S]$.

Se para algum estado, a tabela apresentar mais de uma opção de shift e/ou reduce para o mesmo símbolo (entrada múltipla), a gramática não será SLR(1).

A Construção de tabelas de análise LR(0)

Segue o algoritmo usado para a geração de tabelas SLR, exceto no item 2(b). Sempre que houver um item da forma A→w. serão colocadas opções de redução em todas as colunas ACTION para aquele estado e não apenas para os símbolos em FOLLOW(A). A gramática não será LR(0) se houver mais de uma opção de shift e/ou reduce para o mesmo símbolo (entrada múltipla). Um parser LR(0) não necessita ler símbolos a frente para decidir entre ações de shift e reduce, pois o símbolo no topo da pilha é suficiente para isso.

Ex:

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

É SLR, mas não LR(0).

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

FIGURA 4.37 Tabela de análise para a gramática de expressão.

Ex: Mostre que a gramática abaixo não é SLR:

S´ → S
S → aSa | Sa | b

Parsers LR(1) e LALR

O algoritmo do analisador sintático LR(1) - LR(1) canônico - é mais poderoso que o do analisador SLR. A maioria das linguagens de programação pode ser descrita por uma gramática livre de contexto e analisada por um analisador LR(1).

Para a regra A → α é possível indicar exatamente quais símbolos de entrada podem se seguir após uma redução para A. A informação é incorporada ao estado pela redefinição dos itens, de forma que incluam símbolos terminais como um segundo componente.

A regra do símbolo inicial da gramática estendida produz o item:

S´→ .S, $

Um item LR(1) formado pela regra de produção com um ponto e um símbolo da entrada (lookahead). A forma geral é A → α.β, *a*, onde A → αβ é uma regra de produção e *a* é um terminal ou $. Um item da forma [A → α., *a*] determina a redução A → α somente se o próximo símbolo de entrada for *a*.

Para determinar os símbolos de entrada que farão parte do segundo componente, a função de fechamento utiliza o FIRST da forma sentencial seguinte ao ponto. Por exemplo, para [A → α.Bβ, *a*], inserimos as regras de produção de B, pois o ponto está antes do não-terminal B. Para definir os terminais que fazem parte do segundo componente das regras de produção de B utilizamos o FIRST(β*a*).

A construção do autômato de itens LR(1) é semelhante à do LR(0), mas as funções de closure e goto são definidas como:

```
SetOfItems CLOSURE(I) {
        repeat
                for ( each item [A → α·Bβ, a] in I )
                        for ( each production B → γ in G' )
                                for ( each terminal b in FIRST(βa) )
                                        add [B → ·γ, b] to set I;
        until no more items are added to I;
        return I;
}

SetOfItems GOTO(I, X) {
        initialize J to be the empty set;
        for ( each item [A → α·Xβ, a] in I )
                add item [A → αX·β, a] to set J;
        return CLOSURE(J);
}
```

A tabela LR(1) é construída da mesma forma que a LR(0) exceto na maneira de inserir as reduções. Quando há um item [A → α., *a*] em um estado i, então deve-se inserir na linha i coluna *a* a redução da regra A → α.

Ex:

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

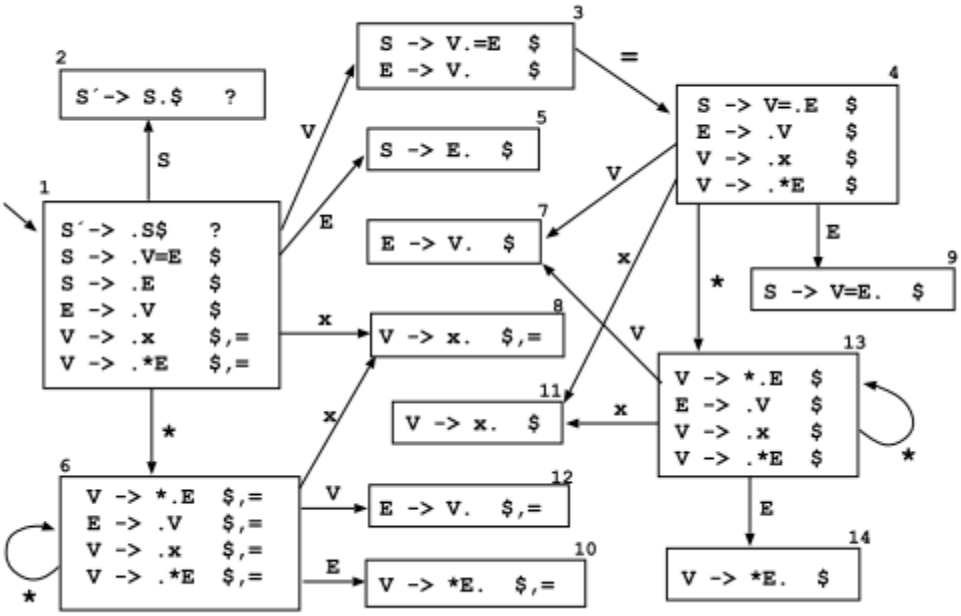


Tabela LR(1)

	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			

O parser LALR se baseia no LR(1), mas o autômato é simplificado através da união de estados que contenham os mesmos conjuntos de itens, exceto pelo lookahead. Isso reduz a memória exigida para o armazenamento da tabela, mas pode gerar conflitos que não existiam na tabela LR(1).

LALR

- diminui o número de estados do LR(1)
- junta os estados idênticos em relação aos itens

	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			

Ex: Construa a tabela LR(1) e LALR para as seguintes gramáticas:

a) S´ → S$
S → CC
C → cC | d

b) S´ → S$
S → L = R | R
L → □R | id
R → L

Os parsers LALR são mais fracos que os LR(1), mas capazes de tratar a maioria das linguagens de programação. Desta forma, são considerados uma opção de compromisso para a escrita de compiladores.