Chapter 6:
Bottom-Up Parsing
(Shift-Reduce)

陳奇業 成功大學資訊工程系

LALR(k) Table Construction

1	Start	$\rightarrow$	S $	
2	S	$\rightarrow$	A B	
3		|	a c	
4		|	x A c	
5	A	$\rightarrow$	a	
6	B	$\rightarrow$	b	
7		|	λ	

State 0	Goto
Start $\rightarrow$ • S $	4
S $\rightarrow$ • A B	2
S $\rightarrow$ • a c	3
S $\rightarrow$ • x A c	1
A $\rightarrow$ • a	3

State 3	Goto
S $\rightarrow$ a • c	6
A $\rightarrow$ a •	

$Follow(A) = \{c, b, \$\}$

shift/reduce conflict

LALR(k) Table Construction

1	Start	→	S $
2	S	→	A_1 B
3		|	a c
4		|	x A_2 c
5	A_1	→	a
6	A_2	→	a
7	B	→	b
8		|	λ

State 0	Goto
Start → • S $	3
S → • A_1 B	4
S → • a c	2
S → • x A_2 c	1
A_1 → • a	2

State 2	Goto
S → a • c	8
A_1 → a •	

Case 1:

$Start \rightarrow S\$ \rightarrow A_1B\$ \rightarrow A_1b\$$ $Follow(A_1) = \{b, \$\}$

$Start \rightarrow S\$ \rightarrow A_1B\$ \rightarrow A_1\$$

Case 2:

$Start \rightarrow S\$ \rightarrow xA_2c\$$ $Follow(A_2) = \{c\}$

LALR(k) Table Construction

- In this section, we consider LALR(k) (Lookahead Ahead LR with k tokens of lookahead) parsing, which offers a more specialized computation of the symbols that can follow a nonterminal.
- LALR offers superior lookahead analysis for constructing the bottom-up parsing table.
- LALR(1) parsers can be built by first constructing an LR(1) parser and then merging states

LALR(k) Table Construction

- LALR(1) parsers can be built by
  1. An LR(1) parser and then merging states (may be quite inefficient)
  2. An LR(0) parser with LALR propagation graph

LALR(k) Table Construction

```
procedure CompleteTable( Table, grammar )
    call ComputeLookahead( )
    foreach state ∈ Table do
        foreach rule ∈ Productions(grammar) do
            call TryRuleInState( state, rule )
    call AssertEntry( StartState, GoalSymbol, accept )
end
procedure AssertEntry( state, symbol, action )
    if Table[state][symbol] = error
    then   Table[state][symbol] ← action
    else
        call ReportConflict( Table[state][symbol], action )
end
```

```
procedure TryRuleInState( s, r )
    if LHS( r ) → RHS( r ) • ∈ s
    then
        foreach X ∈ Follow( LHS( r ) ) do
            call AssertEntry( s, X, reduce r )
end
```

```
procedure TryRuleInState( s, r )
    if LHS( r ) → RHS( r ) • ∈ s
    then
        foreach X ∈ Σ do
            if X ∈ ItemFollow(( s, LHS( r ) → RHS( r ) • ))
            then  call AssertEntry( s, X, reduce r )
end
```

LALR(k) Table Construction

```
procedure ComputeLookahead( )
    call BuildItemPropGraph( )
    call EvalItemPropGraph( )
end
```

LALR Propagation Graph

- We have not formally named each LR(0) item, but an item occurs at most once in any state. Thus, the pair $(s, A \rightarrow \alpha \bullet \beta)$ suffices to identify an item $A \rightarrow \alpha \bullet \beta$ that occurs in state s.
- For each valid state and item pair, we create a vertex v in the LALR propagation graph.

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex((s, item))
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor(Start) do
        ItemFollow((StartState, Start→ •RHS(p))) ← {$}
    foreach s ∈ States do
        foreach A→α•Bγ ∈ s do
            v ← Graph.FindVertex((s, A→α•Bγ))
            call Graph.AddEdge(v, (Table[s][B], A→αB•γ))
            foreach (w ← (s, B→ •δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)
                if AllDeriveEmpty(γ)
                then call Graph.AddEdge(v, w)
end
```

LALR Propagation Graph

- The ItemFollow sets are initially empty, except for the augmenting item Start → ●$S$$ in the LR(0) start-state.

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex( (s, item) )
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor( Start ) do
        ItemFollow((StartState, Start→ • RHS(p))) ← { $ }
    foreach s ∈ States do
        foreach A→α • Bγ ∈ s do
            v ← Graph.FindVertex( (s, A→α • Bγ) )
            call Graph.AddEdge( v, (Table[s][B], A→αB • γ) )
            foreach (w ← (s, B→ • δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First( γ )
                if AllDeriveEmpty( γ )
                then call Graph.AddEdge( v, w )
end
```

LALR Propagation Graph

- Edges are placed in the graph between items i and j when the symbols that follow the reducible form of item i should be included in the corresponding set of symbols for item j.

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex( (s, item) )
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor( Start ) do
        ItemFollow((StartState, Start→ • RHS(p))) ← { $ }
    foreach s ∈ States do
        foreach A→α • Bγ ∈ s do
            v ← Graph.FindVertex( (s, A→α • Bγ) )
            call Graph.AddEdge( v, (Table[s][B], A→αB • γ) )
            foreach (w ← (s, B→ • δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First( γ )
                if AllDeriveEmpty( γ )
                then call Graph.AddEdge( v, w )
end
```

LALR Propagation Graph

- For the item $A \to \alpha \bullet B\gamma$, any symbol in First($\gamma$) can follow each closure item $\mathrm{B} \to \bullet\delta$.

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex((s, item))
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor(Start) do
        ItemFollow((StartState, Start→ • RHS(p))) ← {$}
    foreach s ∈ States do
        foreach A→α • Bγ ∈ s do
            v ← Graph.FindVertex((s, A→α • Bγ))
            call Graph.AddEdge(v, (Table[s][B], A→αB • γ))
            foreach (w ← (s, B→ • δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)
                if AllDeriveEmpty(γ)
                then call Graph.AddEdge(v, w)
end
```

LALR Propagation Graph

- Consider again the item $A \rightarrow \alpha \bullet B\gamma$ and the closure items introduced when B is a nonterminal. When $\gamma \Longrightarrow^{*} \lambda$, either because γ is absent or because the string of symbols in γ can derive λ, then any symbol that can follow A can also follow B.

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex( (s, item) )
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor( Start ) do
        ItemFollow((StartState, Start→ • RHS(p))) ← { $ }
    foreach s ∈ States do
        foreach A→α • Bγ ∈ s do
            v ← Graph.FindVertex( (s, A→α • Bγ) )
            call Graph.AddEdge( v, (Table[s][B], A→αB • γ) )
            foreach (w ← (s, B→ • δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)
                if AllDeriveEmpty(γ)
                then call Graph.AddEdge( v, w )
end
```

LALR Propagation Graph

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex((s, item))
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor(Start) do
        ItemFollow((StartState, Start→ •RHS(p))) ← {$}
    foreach s ∈ States do
        foreach A→α•Bγ ∈ s do
            v ← Graph.FindVertex((s, A→α•Bγ))
            call Graph.AddEdge(v, (Table[s][B], A→αB•γ))
            foreach (w ← (s, B→ •δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)
                if AllDeriveEmpty(γ)
                then call Graph.AddEdge(v, w)
end
```

State		LR(0) Item	Goto State	Prop Edges Placed by Step 27	Prop Edges Placed by Step 29	Initialize *ItemFollow* First(γ)	Initialize *ItemFollow* 28
0	1	Start→ •S $	4	13		$	2,3,4
	2	S→ •A B	2	8	5	b	5
	3	S→ •a c	3	11			
	4	S→ •x A c	1	6			
	5	A→ •a	3	12			
1	6	S→x•A c	9	18		c	7
	7	A→ •a	10	19			
2	8	S→A•B	8	17	9,10		
	9	B→ •b	7	16			
	10	B→ •					
3	11	S→a•c	6	15			
	12	A→a•					
4	13	Start→S•$	5	14			
5	14	Start→S $•					
6	15	S→a c•					
7	16	B→b•					
8	17	S→A B•					
9	18	S→x A•c	11	20			
10	19	A→a•					
11	20	S→x A c•					

LALR Propagation Graph

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex((s, item))
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor(Start) do
        ItemFollow((StartState, Start→ •RHS(p))) ← {$}
    foreach s ∈ States do
        foreach A→α • Bγ ∈ s do
            v ← Graph.FindVertex((s, A→α • Bγ))
            call Graph.AddEdge(v, (Table[s][B], A→αB • γ))
            foreach (w ← (s, B→ •δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)
                if AllDeriveEmpty(γ)
                then call Graph.AddEdge(v, w)
end
```

State		LR(0) Item	Goto State	Prop Edges Placed by Step (27)	Prop Edges Placed by Step (29)	Initialize ItemFollow First(γ)	Initialize ItemFollow (28)
0	1	Start→ • S $	4	13		$	2,3,4
	2	S→ • A B	2	8	5	b	5
	3	S→ • a c	3	11			
	4	S→ • x A c	1	6			
	5	A→ • a	3	12			
1	6	S→x • A c	9	18		c	7
	7	A→ • a	10	19			
2	8	S→A • B	8	17	9,10		
	9	B→ • b	7	16			
	10	B→ •					
3	11	S→a • c	6	15			
	12	A→a •					
4	13	Start→S • $	5	14			
5	14	Start→S $ •					
6	15	S→a c •					
7	16	B→b •					
8	17	S→A B •					
9	18	S→x A • c	11	20			
10	19	A→a •					
11	20	S→x A c •					

1	Start	$\rightarrow$	S \$
2	S	$\rightarrow$	A B
3		|	a c
4		|	x A c
5	A	$\rightarrow$	a
6	B	$\rightarrow$	b
7		|	λ

LALR Propagation Graph

procedure BuildItemPropGraph( )
  foreach $s \in States$ **do**
    foreach $item \in state$ **do**
      $v \leftarrow Graph.\textsc{AddVertex}((s, item))$
      $ItemFollow(v) \leftarrow \emptyset$
  foreach $p \in \textsc{ProductionsFor}(\text{Start})$ **do**
    $ItemFollow((StartState, \text{Start} \rightarrow \bullet \text{RHS}(p))) \leftarrow \{\$\}$
  foreach $s \in States$ **do**
    foreach $\text{A} \rightarrow \alpha \bullet \text{B}\gamma \in s$ **do**
      $v \leftarrow Graph.\textsc{FindVertex}((s, \text{A} \rightarrow \alpha \bullet \text{B}\gamma))$
      call $Graph.\textsc{AddEdge}(v, (Table[s][B], \text{A} \rightarrow \alpha\text{B} \bullet \gamma))$
      foreach $(w \leftarrow (s, \text{B} \rightarrow \bullet \delta)) \in Graph.Vertices$ **do**
        $ItemFollow(w) \leftarrow ItemFollow(w) \cup \text{First}(\gamma)$
        if $\textsc{AllDeriveEmpty}(\gamma)$
        then **call** $Graph.\textsc{AddEdge}(v, w)$
end

State	LR(0) Item		Goto State	Prop Edges Placed by Step (27)	Prop Edges Placed by Step (29)	Initialize *ItemFollow* First(γ)	(28)
0	1	Start$\rightarrow \bullet$S \$	4	13		\$	2,3,4
	2	S$\rightarrow \bullet$A B	2	8	5	b	5
	3	S$\rightarrow \bullet$a c	3	11			
	4	S$\rightarrow \bullet$x A c	1	6			
	5	A$\rightarrow \bullet$a	3	12			
1	6	S$\rightarrow$x$\bullet$A c	9	18		c	7
	7	A$\rightarrow \bullet$a	10	19			
2	8	S$\rightarrow$A$\bullet$B	8	17	9,10		
	9	B$\rightarrow \bullet$b	7	16			
	10	B$\rightarrow \bullet$					
3	11	S$\rightarrow$a$\bullet$c	6	15			
	12	A$\rightarrow$a$\bullet$					
4	13	Start$\rightarrow$S$\bullet$\$	5	14			
5	14	Start$\rightarrow$S \$$\bullet$					
6	15	S$\rightarrow$a c$\bullet$					
7	16	B$\rightarrow$b$\bullet$					
8	17	S$\rightarrow$A B$\bullet$					
9	18	S$\rightarrow$x A$\bullet$c	11	20			
10	19	A$\rightarrow$a$\bullet$					
11	20	S$\rightarrow$x A c$\bullet$					

1	Start	$\rightarrow$	S $	
2	S	$\rightarrow$	A B	
3		|	a c	
4		|	x A c	
5	A	$\rightarrow$	a	
6	B	$\rightarrow$	b	
7		|	λ	

LALR Propagation Graph

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex((s, item))
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor(Start) do
        ItemFollow((StartState, Start→ •RHS(p))) ← { $ }
    foreach s ∈ States do
        foreach A→α • Bγ ∈ s do
            v ← Graph.FindVertex((s, A→α • Bγ))
            call Graph.AddEdge(v, (Table[s][B], A→αB • γ))
            foreach (w ← (s, B→ •δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)
                if AllDeriveEmpty(γ)
                then call Graph.AddEdge(v, w)
end
```

State	LR(0) Item	Goto State	Prop Edges Placed by Step (27)	(29)	Initialize ItemFollow First(γ)	(28)
0	1 Start→ • S $\boxed{\$}$	4	13		$	2,3,4
	2 S→ • A $\boxed{B}$	2	8	5	b	5
	3 S→ • a c	3	11			
	4 S→ • x A c	1	6			
	5 A→ • a	3	12			
1	6 S→x • A $\boxed{c}$	9	18		c	7
	7 A→ • a	10	19			
2	8 S→A • B	8	17	9,10		
	9 B→ • b	7	16			
	10 B→ •					
3	11 S→a • c	6	15			
	12 A→a •					
4	13 Start→S • $	5	14			
5	14 Start→S $ •					
6	15 S→a c •					
7	16 B→b •					
8	17 S→A B •					
9	18 S→x A • c	11	20			
10	19 A→a •					
11	20 S→x A c •					

LALR Propagation Graph

procedure EVALITEMPROPGRAPH( )
    repeat
        $changed \leftarrow$ **false**
        foreach $(v, w) \in Graph.Edges$ **do**
            $old \leftarrow ItemFollow(w)$
            $ItemFollow(w) \leftarrow ItemFollow(w) \cup ItemFollow(v)$
            if $ItemFollow(w) \neq old$
            then $changed \leftarrow$ **true**
    until not $changed$
end

LALR Propagation Graph

State	LR(0) Item	Goto State	Prop Edges Placed by Step (27)	Prop Edges Placed by Step (29)	Initialize *ItemFollow* First(γ)	Initialize *ItemFollow* (28)
0	1 Start→ •S $	4	13		$	2,3,4
	2 S→ •A B	2	8	5	b	5
	3 S→ •a c	3	11			
	4 S→ •x A c	1	6			
	5 A→ •a	3	12			
1	6 S→x•A c	9	18		c	7
	7 A→ •a	10	19			
2	8 S→A•B	8	17	9,10		
	9 B→ •b	7	16			
	10 B→ •					
3	11 S→a•c	6	15			
	12 A→a•					
4	13 Start→S•$	5	14			
5	14 Start→S $•					
6	15 S→a c•					
7	16 B→b•					
8	17 S→A B•					
9	18 S→x A•c	11	20			
10	19 A→a•					
11	20 S→x A c•					

Item	Prop To	Initial	Pass 1
1	13	$	
2	5,8	$	
3	11	$	
4	6	$	
5	12	b	$
6	18		$
7	19	c	
8	9,10,17		$
9	16		$
10			$
11	15		$
12			b $
13	14		$
14			$
15			$
16			$
17			$
18	20		$
19			c
20			$

LR(k) Table Construction

1 Start $\rightarrow$ S \$
2 S $\rightarrow$ lp M rp
3 | lb M rb
4 | lp U rb
5 | lb U rp
6 M $\rightarrow$ expr
7 U $\rightarrow$ expr

State 0	Goto
Start $\rightarrow$ • S \$	1
S $\rightarrow$ • lp M rp	2
S $\rightarrow$ • lb M rb	3
S $\rightarrow$ • lp U rb	2
S $\rightarrow$ • lb U rp	3

State 1	Goto
Start $\rightarrow$ S • \$	13

State 2	Goto
S $\rightarrow$ lp • M rp	10
S $\rightarrow$ lp • U rb	9
M $\rightarrow$ • expr	6
U $\rightarrow$ • expr	6

State 3	Goto
S $\rightarrow$ lb • M rb	5
S $\rightarrow$ lb • U rp	4
M $\rightarrow$ • expr	6
U $\rightarrow$ • expr	6

State 4	Goto
S $\rightarrow$ lb U • rp	8

State 5	Goto
S $\rightarrow$ lb M • rb	7

State 6	Goto
M $\rightarrow$ expr •	
U $\rightarrow$ expr •	

reduce /reduce conflict

State 7	Goto
S $\rightarrow$ lb M rb •	

State 8	Goto
S $\rightarrow$ lb U rp •	

State 9	Goto
S $\rightarrow$ lp U • rb	12

State 10	Goto
S $\rightarrow$ lp M • rp	11

State 11	Goto
S $\rightarrow$ lp M rp •	

State 12	Goto
S $\rightarrow$ lp U rb •	

State 13	Goto
Start $\rightarrow$ S \$ •	

Figure 6.36: LR(0) construction.

LR(k) Table Construction

State		LR(0) Item	Goto State	Prop Edges Placed by Step 27	Prop Edges Placed by Step 29	Initialize *ItemFollow* First(γ)	Initialize *ItemFollow* 28
0	1	Start→ • S \$	1	??		\$	2,3,4,5
	2	S→ • lp M rp	2	6			
	3	S→ • lb M rb	3	10			
	4	S→ • lp U rb	2	7			
	5	S→ • lb U rp	3	11			
2	6	S→lp • M rp	10	??		rp	8
	7	S→lp • U rb	9	??		rb	9
	8	M→ • expr	6	14			
	9	U→ • expr	6	15			
3	10	S→lb • M rb	5	??		rb	12
	11	S→lb • U rp	4	??		rp	13
	12	M→ • expr	6	14			
	13	U→ • expr	6	15			
6	14	M→expr •					
	15	U→expr •					

$$ItemFollow(14)$$
$$= ItemFollow(15)$$
$$= \{ rb, rp \}$$

LR(k) Table Construction

$$
\begin{array}{lllll}
1 & \text{Start} & \rightarrow & \text{S} & \$ \\
2 & \text{S} & \rightarrow & \text{lp M rp} & \\
3 & & | & \text{lb M rb} & \\
4 & & | & \text{lp U rb} & \\
5 & & | & \text{lb U rp} & \\
6 & \text{M} & \rightarrow & \text{expr} & \\
7 & \text{U} & \rightarrow & \text{expr} &
\end{array}
$$

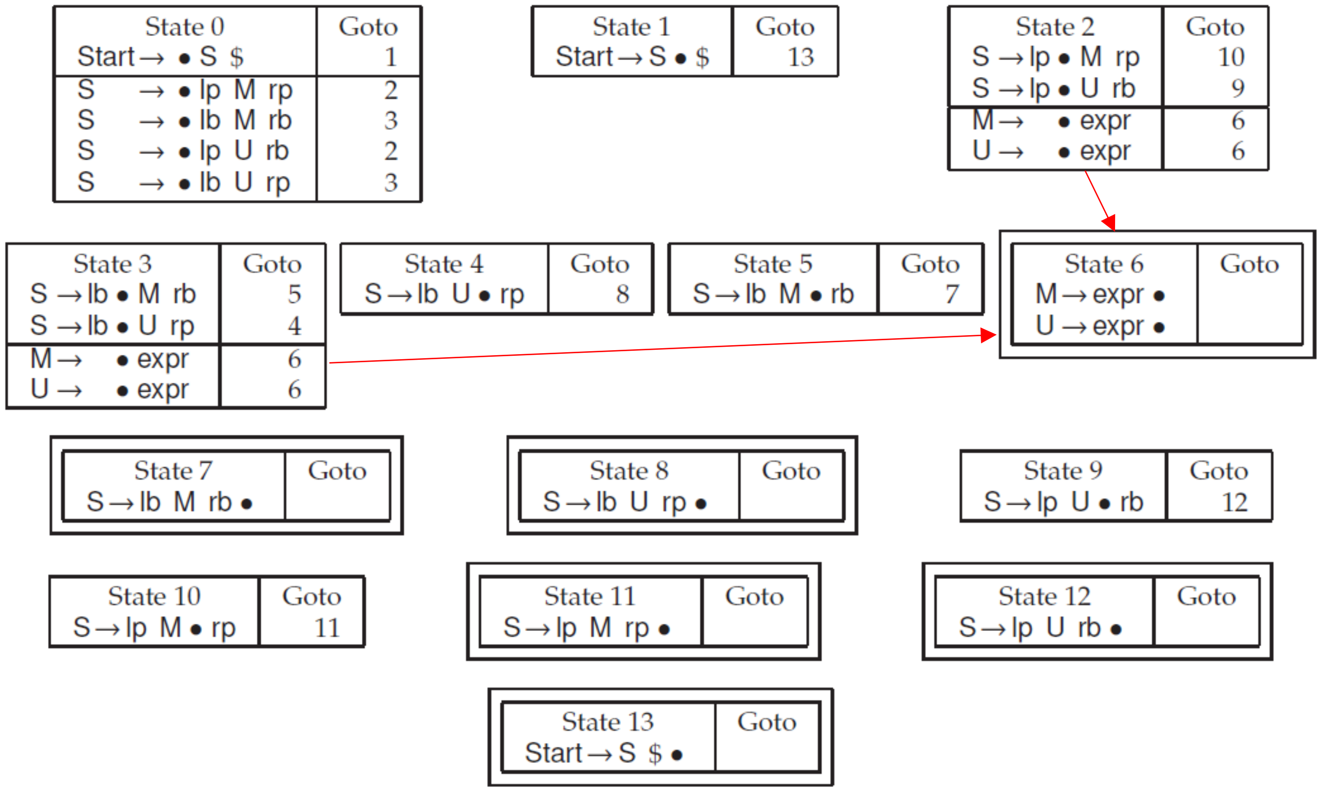


Figure 6.36: LR(0) construction.

LR(k) Table Construction

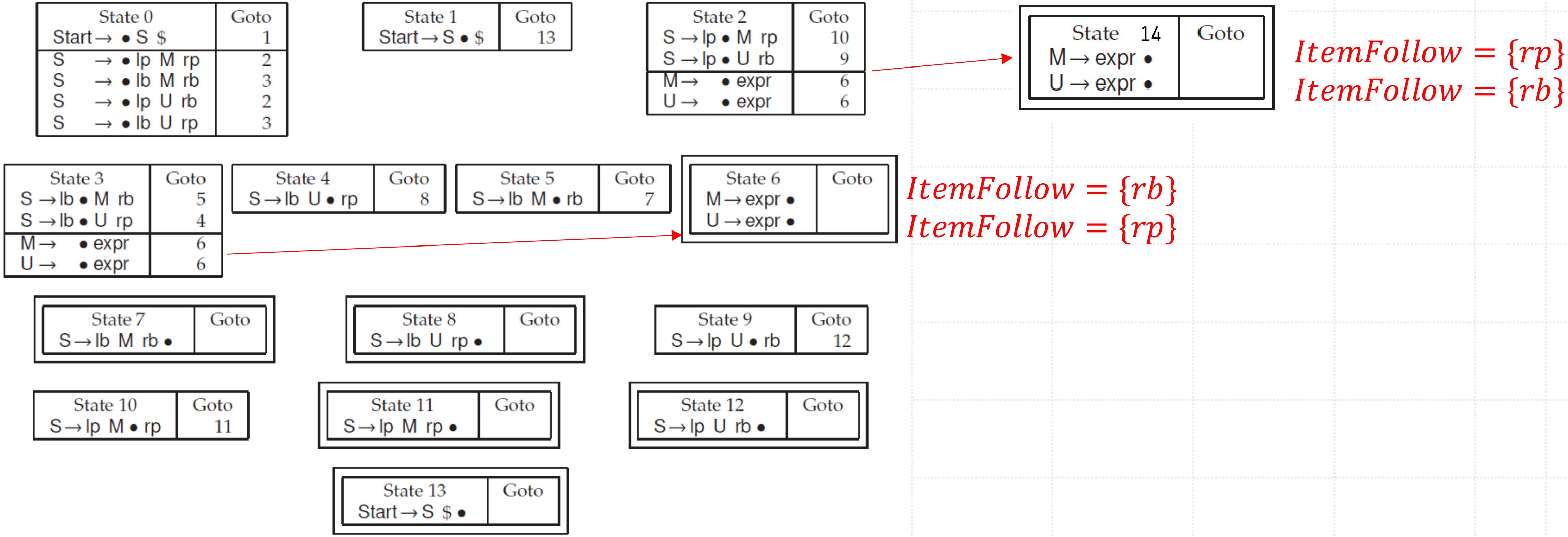


Figure 6.36: LR(0) construction.

LR(k) Table Construction

- For LR(k), we extend an item's notation from $\mathrm{A} \rightarrow \alpha \bullet \beta$ to $[\mathrm{A} \rightarrow \alpha \bullet \beta, w]$.
- For LR(1), w is a (terminal) symbol that can follow A when this item becomes reducible.
- For LR(k), $k \geq 0$, w is a k-length string that can follow A after reduction.
- If symbols x and y can both follow A when $\mathrm{A} \rightarrow \alpha \bullet \beta$ becomes reducible, then the corresponding LR(1) state contains both $[\mathrm{A} \rightarrow \alpha \bullet \beta, x]$ and $[\mathrm{A} \rightarrow \alpha \bullet \beta, y]$.
- Notice how nicely the notation for LR(k) generalizes LR(0). For LR(0), w must be a 0-length string. The only such string is λ, which provides no information at a possible point of reduction, since λ does not occur as input.

LR(k) Table Construction

State 3	Goto
[S → lb • M rb , $]	5
[S → lb • U rp , $]	4
[M → • expr , rb]	14
[U → • expr , rp]	14

$[S \rightarrow lb \bullet M\ rb, \$]$ is not ready for reduction, but indicates that $ will follow the reduction to S when the item eventually becomes reducible

State 6	Goto
[M → expr • , rp]	
[U → expr • , rb]	

The item calls for a reduction by rule $\mathrm{M} \rightarrow expr$ when rp is the next input token.

function ComputeLR0(*Grammar*) **returns** (*Set*, *State*)

  $States \leftarrow \emptyset$

  $StartItems \leftarrow \{\, \mathsf{Start} \rightarrow \bullet\, \mathrm{RHS}(p) \mid p \in \textsc{ProductionsFor}(\mathsf{Start}) \,\}$ (7)

  $StartState \leftarrow \textsc{AddState}(States, StartItems)$

  while $(s \leftarrow WorkList.\textsc{ExtractElement}()) \neq \perp$ **do** (8)

    call $\textsc{ComputeGoto}(States, s)$

  return $((States, StartState))$

end

function AddState(*States*, *items*) **returns** *State*

  if $items \notin States$ (9)

  then

    $s \leftarrow newState(items)$ (10)

    $States \leftarrow States \cup \{s\}$

    $WorkList \leftarrow WorkList \cup \{s\}$ (11)

    $Table[s][\star] \leftarrow \mathsf{error}$ (12)

  else $s \leftarrow FindState(items)$

  return (s)

end

function AdvanceDot(*state*, $\mathcal{X}$) **returns** *Set*

  return $(\{\, \mathsf{A} \rightarrow \alpha\mathcal{X} \bullet \beta \mid \mathsf{A} \rightarrow \alpha \bullet \mathcal{X}\beta \in state \,\})$ (13)

end

Marker (7): We initialize *StartItems* by including LR(1) items that have \$ as the follow symbol:

  $StartItems \leftarrow \{\, [\, \mathsf{Start} \rightarrow \bullet\, \mathrm{RHS}(p), \$ \,] \mid p \in \textsc{ProductionsFor}(\mathsf{Start}) \,\}$

Marker (13): We augment the LR(0) item so that AdvanceDot returns the appropriate LR(1) items:

  return $(\{\, [\, \mathsf{A} \rightarrow \alpha\mathcal{X} \bullet \beta, \mathsf{a} \,] \mid [\, \mathsf{A} \rightarrow \alpha \bullet \mathcal{X}\beta, \mathsf{a} \,] \in state \,\})$

Marker (15): This entire loop is replaced by the following:

  foreach $[\, \mathsf{A} \rightarrow \alpha \bullet \mathsf{B}\gamma, \mathsf{a} \,] \in ans$ **do**

    foreach $p \in \textsc{ProductionsFor}(B)$ **do**

      foreach $b \in \mathsf{First}(\gamma \mathsf{a})$ **do** (31)

        $ans \leftarrow ans \cup \{\, [\, \mathsf{B} \rightarrow \bullet\, \mathrm{RHS}(p), b \,] \,\}$

```
function Closure(state) returns Set
    ans ← state
    repeat                                                    (14)
        prev ← ans
        foreach A → α • Bγ ∈ ans do                           (15)
            foreach p ∈ ProductionsFor(B) do
                ans ← ans ∪ { B → • RHS(p) }                  (16)
    until ans = prev
    return (ans)
end
procedure ComputeGoto(States, s)
    closed ← Closure(s)                                       (17)
    foreach X ∈ (N ∪ Σ) do                                    (18)
        RelevantItems ← AdvanceDot(closed, X)                 (19)
        if RelevantItems ≠ ∅
        then
            Table[s][X] ← shift AddState(States, RelevantItems)   (20)
end
```

Marker (7): We initialize *StartItems* by including LR(1) items that have \$ as the follow symbol:

$StartItems \leftarrow \{ [\,\mathrm{Start} \rightarrow \bullet\, \mathrm{RHS}(p), \$\,] \mid p \in \textsc{ProductionsFor}(\mathrm{Start}) \}$

Marker (13): We augment the LR(0) item so that AdvanceDot returns the appropriate LR(1) items:

return $(\{ [\,\mathrm{A} \rightarrow \alpha\mathcal{X} \bullet \beta, \mathrm{a}\,] \mid [\,\mathrm{A} \rightarrow \alpha \bullet \mathcal{X}\beta, \mathrm{a}\,] \in state \})$

Marker (15): This entire loop is replaced by the following:

```
foreach [A → α • Bγ, a] ∈ ans do
    foreach p ∈ ProductionsFor(B) do
        foreach b ∈ First(γa) do                              (31)
            ans ← ans ∪ { [B → • RHS(p), b] }
```

LR(k) Table Construction

procedure COMPLETETABLE(*Table*, *grammar*)
    ~~**call** COMPUTELOOKAHEAD( )~~
    foreach *state* ∈ *Table* **do**
        foreach *rule* ∈ *Productions*(*grammar*) **do**
            call TRYRULEINSTATE(*state*, *rule*)
    call ASSERTENTRY(*StartState*, *GoalSymbol*, accept)
end

procedure ASSERTENTRY(*state*, *symbol*, *action*)
    if *Table*[*state*][*symbol*] = error
    then *Table*[*state*][*symbol*] ← *action*
    else
        call REPORTCONFLICT(*Table*[*state*][*symbol*], *action*)
end

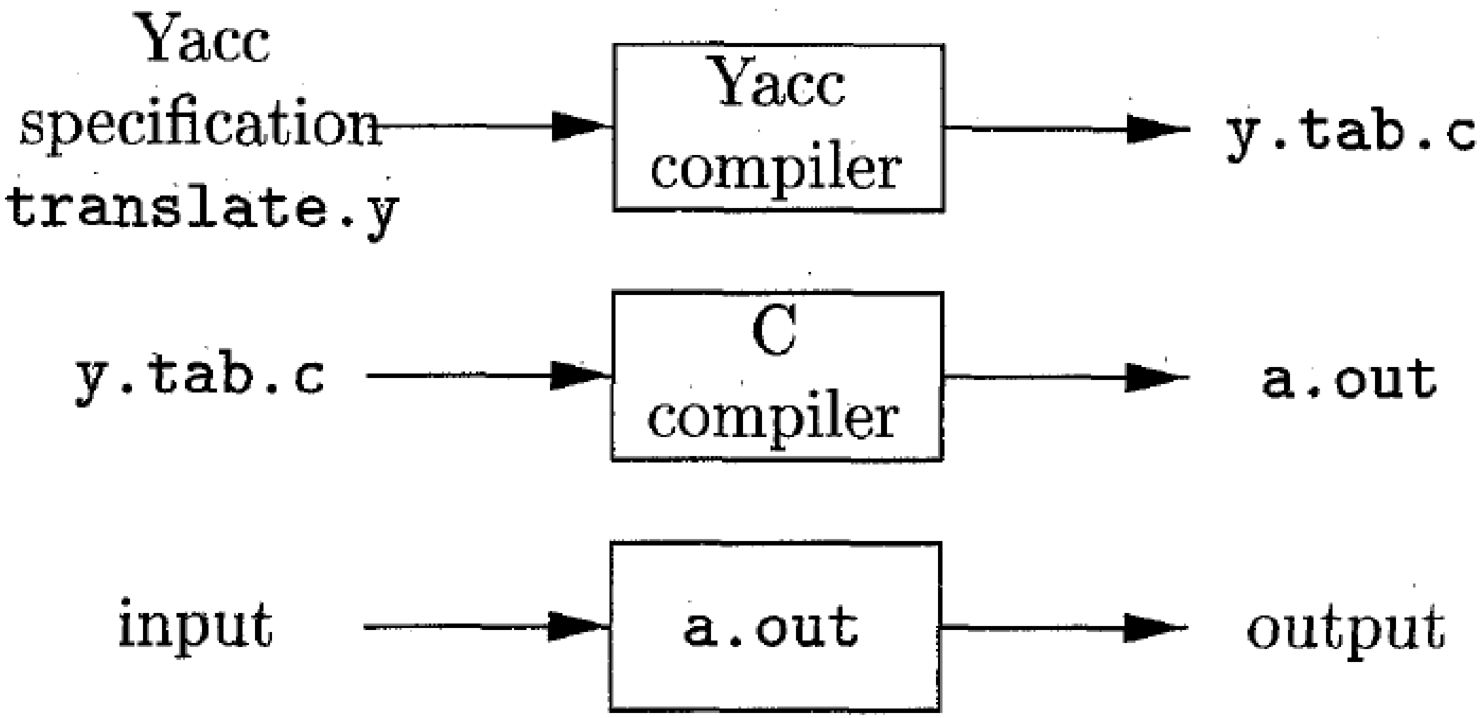
procedure TRYRULEINSTATE(s, r)
    if LHS(r) → RHS(r) • ∈ s
    then
        foreach X ∈ Follow(LHS(r)) **do**
            call ASSERTENTRY(s, X, reduce r)
end

↓

procedure TRYRULEINSTATE(s, r)
    if [LHS(r) → RHS(r) • , w] ∈ s
    then **call** ASSERTENTRY(s, w, reduce r)
end

- Exercises 27, 40, 41, 45

Yacc



Yacc specification translate.y
Yacc compiler
y.tab.c
y.tab.c
C compiler
a.out
input
a.out
output

Disambiguating Rules for Yacc (*required only when there exists a conflict)

1. In a shift/reduce conflict the default is to shift.

2. In a reduce/reduce conflict the default is to reduce by the earlier grammar rule in the input sequence.

3. Precedence and associativity (left, right, nonassoc) are recorded for each token that have them.

4. Precedence and associativity of a production rule is that (if any) of its final (rightmost) token unless a

   "%prec " overrides. Then it is the token given following %prec.

5. In a shift/reduce conflict where both the grammar rule and the input (lookahead) have precedence, resolve in favor of the rule of higher precedence. In a tie, use associativity. That is, left assoc. => reduce; right assoc. => shift; nonassoc => error.

6. Otherwise use 1 and 2.

(Please See Page 238 of the Textbook)

Yacc

declarations
%%
translation rules
%%
supporting C routines

$$
\begin{array}{llll}
\langle\text{head}\rangle & : & \langle\text{body}\rangle_1 & \{\ \langle\text{semantic action}\rangle_1\ \} \\
 & | & \langle\text{body}\rangle_2 & \{\ \langle\text{semantic action}\rangle_2\ \} \\
 & & \cdots & \\
 & | & \langle\text{body}\rangle_n & \{\ \langle\text{semantic action}\rangle_n\ \} \\
 & ; & &
\end{array}
$$

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line    : expr '\n'          { printf("%d\n", $1); }
        ;
expr    : expr '+' term      { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor    { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'       { $$ = $2; }
        | DIGIT
        ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

declared in "y.tab.h"

Yacc

- The lexical analyzer yylex() produces tokens consisting of a token name and its associated attribute value. If a token name such as DIGIT is returned, the token name must be declared in the first section of the Yacc specification. The attribute value associated with a token is communicated to the parser through a Yacc-defined variable yylval.
- Whenever the lexer returns a token to the parser, if the token has an associated value, the lexer must store the value in yylval before returning. In this first example, we explicitly declare yylval. In more complex parsers, yacc defines yylval as a union and puts the definition in y.tab.h.

Yacc

declared in "y.tab.h"

```
%{
#include "y.tab.h"
extern int yylval;
%}
```

declared by yacc

```
%%
[0-9]+      { yylval = atoi(yytext);   return NUMBER; }
[ \t] ;            /* ignore whitespace */
\n     return 0;   /* logical EOF */
.      return yytext[0];
%%
```

The Lexer

```
%token NAME NUMBER
%%
statement:   NAME '='  expression
        |    expression         { printf("= %d\n", $1); }
        ;

expression:  expression '+' NUMBER { $$ = $1 + $3; }
        |    expression '-' NUMBER { $$ = $1 - $3; }
        |    NUMBER                { $$ = $1; }
        ;
```

A Yacc Parser

Yacc

- On a UNIX system, yacc takes your grammar and creates y.tab.c, the C language parser, and y.tab.h, the include file with the token number definitions. Lex creates lex.yy.c, the C language lexer. You need only compile them together with the yacc and lex libraries. The libraries contain usable default versions of all of the supporting routines, including a main() that calls the parser yyparse() and exits.

```
% yacc -d ch3-01.y    # makes y.tab.c and "y.tab.h
% lex ch3-01.l        # makes lex.yy.c
% cc -o ch3-01 y.tab.c lex.yy.c -ly -ll     # compile and link C files
```

Yacc

```
%token NAME NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%%

statement:  NAME '=' expression
      |     expression       { printf("= %d\n", $1); }
      ;

expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                            {    if($3 == 0)
                                      yyerror("divide by zero");
                                 else
                                      $$ = $1 / $3;
                            }

      |     '-' expression %prec UMINUS { $$ = -$2; }
      |     '(' expression ')'     { $$ = $2; }
      |     NUMBER                 { $$ = $1; }
      ;
%%
```

```
%{
double vbltable[26];
%{

%union {
      double dval;
      int vblno;
}

%token <vblno> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement '\n'
      |    statement_list statement '\n'
      ;

statement: NAME '=' expression   { vbltable[$1] = $3; }
      |    expression       { printf("= %g\n", $1); }
      ;

expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                        {    if($3 == 0.0)
                                     yyerror("divide by zero")
                             else
                                     $$ = $1 / $3;
                         }
      |     '-' expression %prec UMINUS { $$ = -$2; }
      |     '(' expression ')'     { $$ = $2; }
      |     NUMBER
      |     NAME                  { $$ = vbltable[$1]; }
      ;
%%
```

Example 3-3. Lexer for calculator with variables and real values ch3-03.l

```
%{
#include "y.tab.h"
#include <math.h>
extern double vbltable[26];
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
      yylval.dval = atof(yytext); return NUMBER;
      }

[ \t] ;             /* ignore whitespace */

[a-z] { yylval.vblno = yytext[0] - 'a'; return NAME; }

"$"   { return 0; /* end of input */ }

\n    |
.     return yytext[0];
%%
```

Yacc

- The generated header file y.tab.h includes a copy of the definition so that you can use it in the lexer. Here is the y.tab.h generated from this grammar:

```
#define NAME 257
#define NUMBER 258
#define UMINUS 259
typedef union {
      double dval;
      int vblno;
} YYSTYPE;
extern YYSTYPE yylval;
```

Symbol table

Example 3-4. Header for parser with symbol table ch3hdr.h

```
#define NSYMS 20  /* maximum number of symbols */

struct symtab {
      char *name;
      double value;
} symtab[NSYMS];

struct symtab *symlook();
```

```
/* look up a symbol table entry, add if not present */
struct symtab *
symlook(s)
char *s;
{
      char *p;
      struct symtab *sp;
      for(sp = symtab;  sp < &symtab[NSYMS]; sp++) {
            /* is it already here? */
            if(sp->name && !strcmp(sp->name, s))
                  return sp;

            /* is it free */
            if(!sp->name) {
                  sp->name = strdup(s);
                  return sp;
            }
            /* otherwise continue to next */
      }
      yyerror("Too many symbols");
      exit(1);    /* cannot continue */
} /* symlook */
```

```
%{
#include "ch3hdr.h"
#include <string.h>
%}

%union {
	double dval;
	struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement '\n'
	|    statement_list statement '\n'
	;

statement: NAME '=' expression   { $1->value = $3; }
	|    expression        { printf("= %g\n", $1); }
	;

expression: expression '+' expression { $$ = $1 + $3; }
	|    expression '-' expression { $$ = $1 - $3; }
	|    expression '*' expression { $$ = $1 * $3; }
	|    expression '/' expression
			{    if($3 == 0.0)
					yyerror("divide by zero"
				else
					$$ = $1 / $3;
			}
	|    '-' expression %prec UMINUS  { $$ = -$2; }
	|    '(' expression ')'      { $$ = $2; }
	|    NUMBER
	|    NAME               { $$ = $1->value; }
	;
%%
```

Symbol table (Lex)

```
%{
#include "y.tab.h"
#include "ch3hdr.h"
#include <math.h>
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
            yylval.dval = atof(yytext);
            return NUMBER;
      }
[ \t] ;            /* ignore whitespace */

[A-Za-z][A-Za-z0-9]*    {     /* return symbol pointer */
            yylval.symp = symlook(yytext);
            return NAME;
      }

"$"   { return 0; }
\n    |
.     return yytext[0];
%%
```