Bottom-UP-All

<<Bottom-UP-All.ppt>>

Example 4.21. Consider the grammar

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

The sentence *abbcde* can be reduced to *S* by the following steps:

abbcde
aAbcde
aAde
aABe
S

$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$$

Example 4.22. Consider the following grammar

$$
\begin{aligned}
&(1)\quad E \rightarrow E + E \\
&(2)\quad E \rightarrow E * E \\
&(3)\quad E \rightarrow (E) \\
&(4)\quad E \rightarrow \mathbf{id}
\end{aligned}
$$

and the rightmost derivation

$$
\begin{aligned}
E &\underset{rm}{\Rightarrow} \underline{E + E} \\
&\underset{rm}{\Rightarrow} E + \underline{E * E} \\
&\underset{rm}{\Rightarrow} E + E * \underline{\mathbf{id}}_{\mathbf{3}} \\
&\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}}_{\mathbf{2}} * \mathbf{id}_{\mathbf{3}} \\
&\underset{rm}{\Rightarrow} \underline{\mathbf{id}}_{\mathbf{1}} + \mathbf{id}_{\mathbf{2}} * \mathbf{id}_{\mathbf{3}}
\end{aligned}
$$

Example 4.23. Consider the grammar (4.16) of Example 4.22 and the input string **id+ id* id.** The sequence of reductions shown in Fig. 4.21 reduces **id+ id* id** to the start symbol *E*. The reader should observe that the sequence of right-sentential forms in this example is just the reverse of the sequence in the first rightmost derivation in Example 4.22.

==

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}+\mathbf{id}*\mathbf{id}$	$\mathbf{id}$	$E \to \mathbf{id}$
$E+\mathbf{id}*\mathbf{id}$	$\mathbf{id}$	$E \to \mathbf{id}$
$E+E*\mathbf{id}$	$\mathbf{id}$	$E \to \mathbf{id}$
$E+E*E$	$E*E$	$E \to E*E$
$E+E$	$E+E$	$E \to E+E$
E		

Fig. 4.21. Reductions made by shift-reduce parser.

	STACK	INPUT	ACTION
(1)	$\$$	$\mathbf{id_1 + id_2 * id_3\$}$	shift
(2)	$\$\ \mathbf{id_1}$	$\mathbf{+ id_2 * id_3\$}$	reduce by $E \rightarrow \mathbf{id}$
(3)	$\$E$	$\mathbf{+ id_2 * id_3\$}$	shift
(4)	$\$E +$	$\mathbf{id_2 * id_3\$}$	shift
(5)	$\$E + \mathbf{id_2}$	$\mathbf{* id_3\$}$	reduce by $E \rightarrow \mathbf{id}$
(6)	$\$E + E$	$\mathbf{* id_3\$}$	shift
(7)	$\$E + E *$	$\mathbf{id_3\$}$	shift
(8)	$\$E + E * \mathbf{id_3}$	$\$$	reduce by $E \rightarrow \mathbf{id}$
(9)	$\$E + E * E$	$\$$	reduce by $E \rightarrow E * E$
(10)	$\$E + E$	$\$$	reduce by $E \rightarrow E + E$
(11)	$\$E$	$\$$	accept

Fig. 4.22. Configurations of shift-reduce parser on input **id+ id* id**.

Example 4.25. An ambiguous grammar can never be LR. For example, conder the dangling-else grammar (4.7) of Section 4.3:

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

we have a shift-reduce parser in configuration

STACK	INPUT
···**if** *expr* **then** *stmt*	**else**···**$**

Example 4.33. Figure 4.31 shows the parsing action and goto functions of an LR parsing table for the following grammar for arithmetic expressions with binary operators + and $*$:

(1) E -> E + T
(2) E -> T
(3) T -> T $*$ F
(4) T -> F
(5) F -> (E)
(6) F -> **id**

```
set ip to point to the first symbol of w$;
repeat forever begin
let s be the state on top of the stack and a the symbol pointed to by ip;
 if action[s, a] = shift s' then begin
    push a then s' on top of the stack;
    advance ip to the next input symbol
 end
 else if action[s, a] = reduce A ->β then begin
    pop 2∗|β| symbols off the stack;
    let s' be the state now on top of the stack;
    push A then goto[s', A] on top of the stack;
    output the production A ->β
 end
 else if action[s, a] = accept then
    return
 else error()
 End
```

Fig. 4.30 LR parsing program.

STATE	*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			

Fig. 4.31. Parsing table for expression grammar.

STACK	INPUT	ACTION
(1) 0	**id** $*$ **id** + **id** \$	shift
(2) 0 **id** 5	$*$ **id** + **id** \$	reduce by F -> **id**
(3) 0F3	$*$ **id** + **id** \$	reduce by T -> F
(4) 0T2	$*$ **id** + **id** \$	shift
(5) 0T2 $*$ 7	**id** + **id** \$	shift
(6) 0T2 $*$ 7 **id** 5	+ **id** \$	reduce by F -> **id**
(7) 0T2 $*$ 7 F 10	+ **id** \$	reduce by T -> T $*$ F
(8) 0T2	+ **id** \$	reduce by E -> T
(9) 0E1	+ **id** \$	shift
(10) 0E1 + 6	**id** \$	shift
(11) 0E1 + 6 **id** 5	\$	reduce by F -> **id**
(12) 0E1 + 6 F 3	\$	reduce by T -> F
(13) 0E1 + 6 T 9	\$	E -> E + T
(14) 0E1	\$	accept

Figure 4.32 Moves of LR parser on **id** $*$ **id** + **id**.

Example 4.34. Consider the augmented expression grammar:

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
\qquad (4.19)
$$

IF ***I*** is the set of one item {[$E' \rightarrow \cdot E$]}, then *closure(I)* contains the items

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

function *closure(I)*;
begin
    J := I;
    repeat
    for each item $A \rightarrow \alpha \cdot B\beta$ in J and each production
        $B \rightarrow \gamma$ of G such that $B \rightarrow \cdot \gamma$ is not in J **do**
            add $B \rightarrow \cdot \gamma$ to *J*
    until no more items can be added to J;
    return ***J***
end

Fig. 4.33. Computation of *closure.*

Example 4.35. If I is the set of two items {[$E' \to E\ \cdot$], [$E \to E\ \cdot +T$]}, then goto (I, +) consists of

$$E \to E + \cdot\ T$$
$$T \to\ \cdot\ T * F$$
$$T \to\ \cdot\ F$$
$$F \to\ \cdot\ (\ E\ )$$
$$F \to\ \cdot\ \boldsymbol{id}$$

procedure *items(G');*
begin
C := { closure ({ [S' → ·S] }) };
repeat
for each set of items ***I*** in ***C*** and each grammar symbol ***X***
such that *goto* **(I, X**) is not empty and not in **C do**
add *goto* ***(I, X)*** to C
until no more sets of items can be added to *C*
End

Fig. 4.34. The sets-of-items construction

I_0 : $E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

I_1 : $E' \rightarrow E \cdot$
$E \rightarrow E \cdot + T$

I_2 : $E \rightarrow T \cdot$
$T \rightarrow T \cdot * F$

I_3 : $T \rightarrow F \cdot$

I_4 : $F \rightarrow (\cdot E)$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

Fig. 4.35. Canonical LR(0) collection for grammar (4.19).

I_5 : $F \to id\cdot$

I_6 : $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot id$

I_7 : $T \to T * \cdot F$
$F \to \cdot(E)$
$F \to \cdot id$

I_8 : $F \to (E \cdot)$
$E \to E \cdot + T$

I_9 : $E \to E + T\cdot$
$T \to T \cdot * F$

I_{10} : $T \to T * F\cdot$

I_{11} : $F \to (E)\cdot$

Fig. 4.35. Canonical LR(0) collection for grammar (4.19). (cont.)

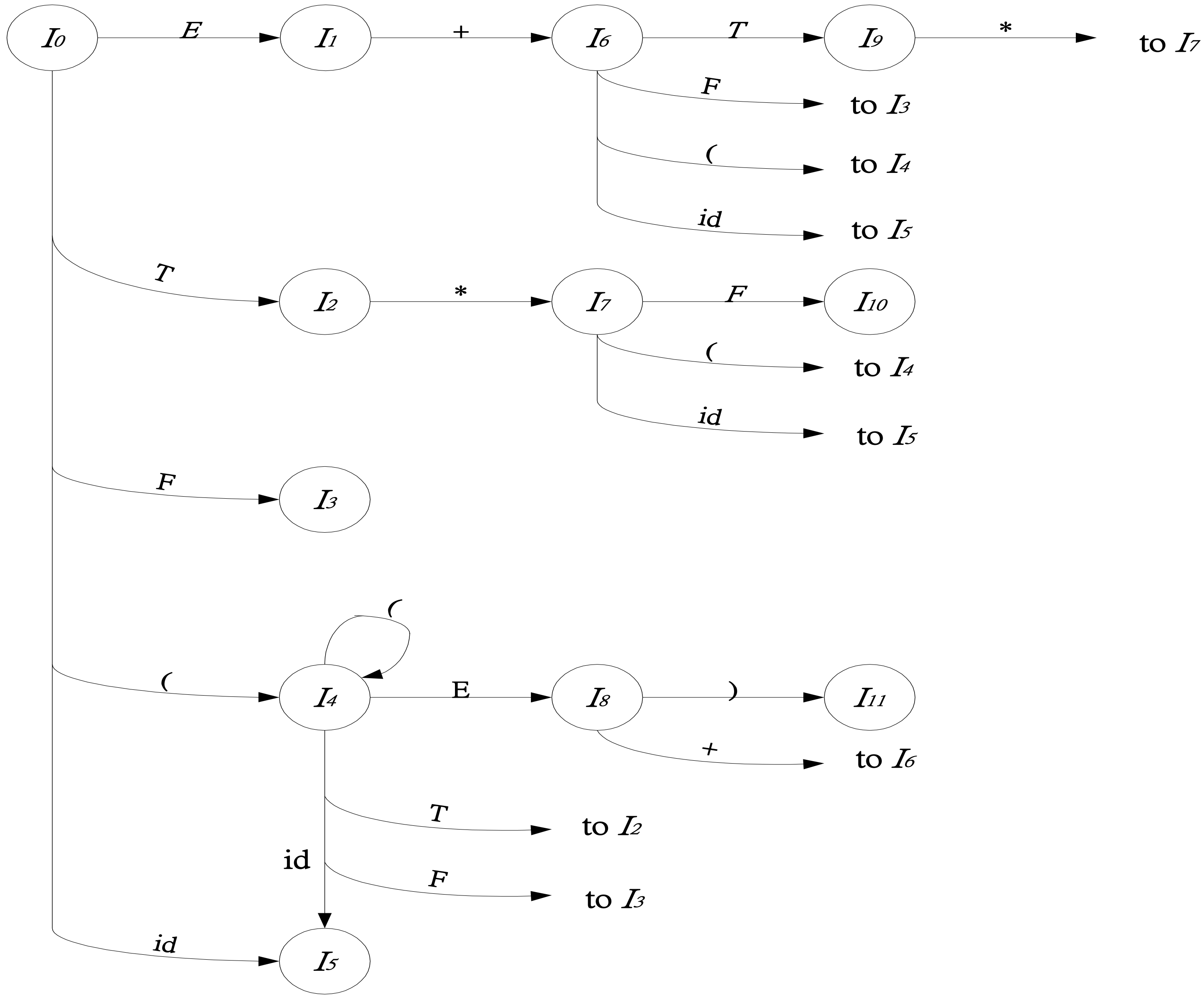

Fig. 4.36. Transition diagram of DFA D for viable prefixes.

Algorithm 4.8. Constructing an SLR parsing table.
Input. An augmented grammar G' .
Output. The SLR parsing table function *action* and *goto* for G' .
Method.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G' .
2. State i is constructed from I_i .The parsing actions for stats i are determined as follows:

    a) If [A $\rightarrow \alpha \cdot a\beta$] is in I_i and goto(I_i, a) = I_j , then set action[i , a] to "shift j." Here a must be a terminal.

    b) If [A $\rightarrow \alpha \cdot$] is in I_i , then set action[i, a] to "reduce A" for all a in FOLLOW(A);here A may not be S'.

    c) If [$S' \rightarrow S \cdot$] is in I_i ,then set action[i , $] to "accept."

If any conflicting actions are generated by the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If goto(I_i , A) = I_j , then goto[i, A] = j.

4. All entries not defined by rules (2) and (3) are made “error.”

5. The initial state of the parser is the one constructed from the set of item containing $[S' \rightarrow \cdot S]$.