CS 304 Compiler Design

Text books

1. Compilers – Principles, Techniques & Tools , Aho, Ravi Sethi, D. Ullman

- Module III - Bottom-Up Parsing:
- Handle pruning, Shift Reduce parsing.
- Operator precedence parsing (Concepts only)
- LR parsing – Constructing SLR, Canonical LR parsing tables and LALR parsing tables

Bottom-Up Parsing

- Introduce a general style of bottom-up syntax analysis
  - Shift-reduce parsing
    - An easy to implement form
      - Operator-precedence parsing
    - A more general method
      - LR parsing

Shift-Reduce parsing

- Attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)
- Can be thought of as “reducing” a string w to the start symbol of the grammar
- At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production and if the substring is chosen correctly at each step, a right most derivation is traced out in reverse

Example 4.21

S → aABe

A → Abc | b

B → d

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

abbcde

aAbcde

aAde

aABe

S

- We scan abbcde looking for a substring that matches the right side of some production
- Let us choose the left most b and replace it by A
- Now substrings Abc, b and d matches the right side of some production
- Abc wins
- Obtain aAde
- Replacing d by B, we get aABe
- Now replace this entire string by S
- Thus by a sequence of four reductions we are able to reduce abbcde to S
- These reductions, trace out the following right most derivation in reverse

$$S \underset{rm}{=>} aABE \underset{rm}{=>} aAde \underset{rm}{=>} aAbcde \underset{rm}{=>} abbcde$$

Handles

- Of a string
- Is a substring that matches the RHS of a production, and whose reduction to the nonterminal on the LHS of the production represents one step along the reverse of a rightmost derivation
- In many cases the leftmost substring β that matches the RHS of some production A → β is not a handle, because a reduction by the production A → β yields a string that cannot be reduced to the start symbol
- In the eg 4.21 if we replaced b by A in the second string aAbcde we would obtain the string aAAcde that cannot be reduced to S
- So we need to give a more precise definition for handle

- Formally,
- **A handle of a right-sentential form is a production A → β and a position of where the string β may be found and replaced by A to produce the previous right sentential form in a rightmost derivation of γ**
- That is, if $S \underset{rm}{\Rightarrow}^{*} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, then A → β in the position following is a handle
- The string *w* to the right of the handle contains only terminal symbols
- We say "a handle" than "the handle" because the grammar could be ambiguous with more than one rightmost derivation of αβ*w*
- If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle

- In the example above, abbcde is a right sentential form whose handle is A →b at position 2
- Fig 4.20 portrays the handle A → β in the parse tree of a right-sentential form αβ*w*
- The handle represents the leftmost complete subtree consisting of a node and all its children

Fig 4.20

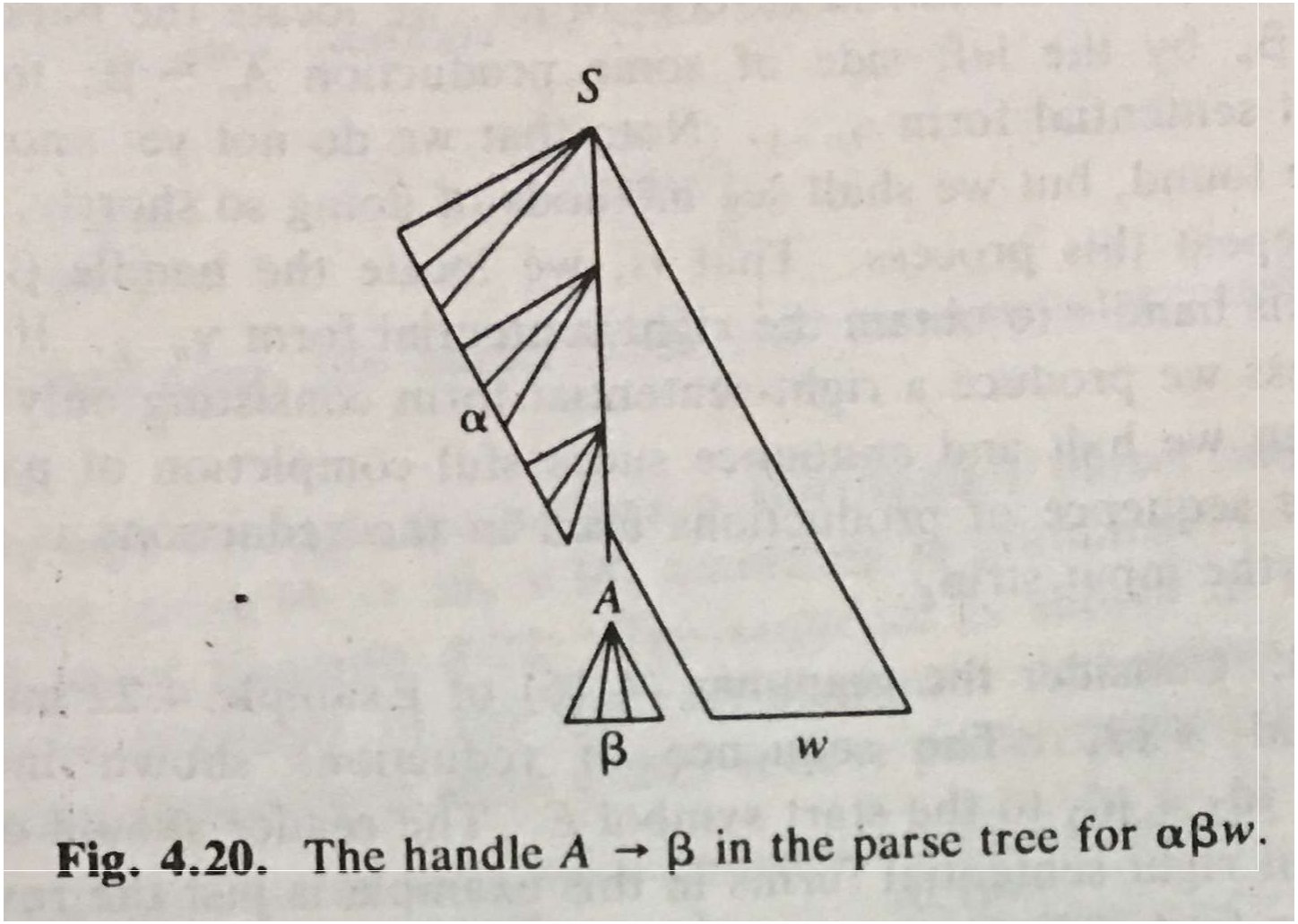
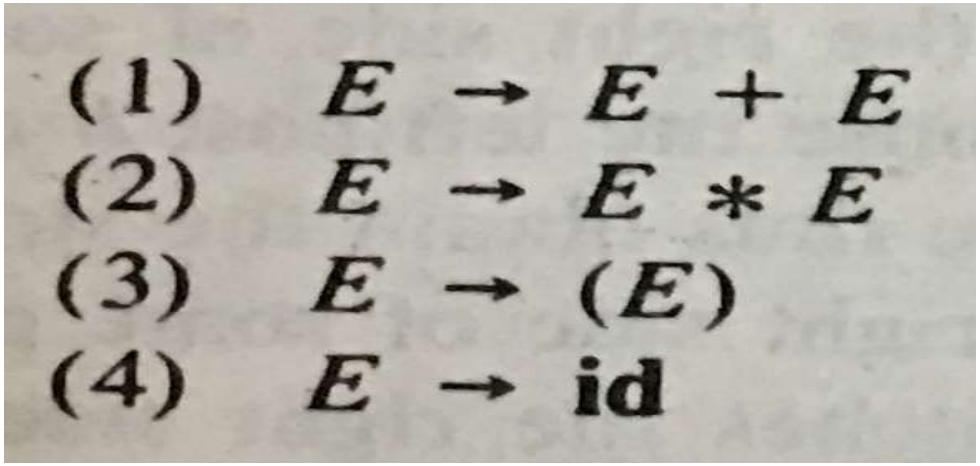


Fig. 4.20. The handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$.

- In fig 4.20,A is the bottom most leftmost interior node with all its children in the tree
- Reducing β to A in αβw can be thought of as “pruning the handle”, ie removing the children of A from the parse tree

Example 4.22

- Consider the following grammar



- Write the rightmost derivation for the string

  id + id * id

Right most derivation

$$
\begin{aligned}
E &\underset{rm}{\Rightarrow} \underline{E + E} \\
&\underset{rm}{\Rightarrow} E + \underline{E * E} \\
&\underset{rm}{\Rightarrow} E + E * \underline{\mathbf{id}_3} \\
&\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\
&\underset{rm}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{aligned}
$$

- Subscripted id's for notational convenience
- id1 is the handle of the right-sentential form id1+id2*id3 because id is the right side of the production E->id and replacing id1 by E produces the previous right-sentential form E+id2*id3
- Note that the string appearing on the RHS of a handle contains only terminal symbols
- Because the grammar is ambiguous, there is another rightmost derivation of the same string

Eg 4.23

- Consider the grammar 4.16 of example 4.22 and the input stirng id1+id2*id3
- The sequence of reductions shown in fig 4.21 reduces id1+id2*id3 to the start symbol E

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_1$	$E \rightarrow \mathbf{id}$
$E + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_2$	$E \rightarrow \mathbf{id}$
$E + E * \mathbf{id}_3$	$\mathbf{id}_3$	$E \rightarrow \mathbf{id}$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

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

Stack implementation of shift-reduce parsing

- Data structure to use in a shift-reduce parser – stack
- A convenient way to implement a shift-reduce parser is
  - To use a to hold the grammar symbols and
  - An input buffer to hold the string w to be parsed
  - Use $ to mark the bottom of the stack and the right end of the input
  - Initially, the stack is empty, and the string w is on the input as follows

STACK	INPUT
$	w$

- The parser operates by shifting zero or more input symbols onto the stack until a handle β is on the top of the stack
- The parser then reduces β to the left side of the appropriate production
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and input is empty

STACK	INPUT
$S	$

- After entering this configuration, the parser halts and announces successful completion of parsing

Example 4.24

- Parsing of the input string id1+id2*id3

	STACK	INPUT	ACTION
(1)	$	$id_1 + id_2 * id_3$	shift
(2)	id_1	$+ id_2 * id_3$	reduce by $E \rightarrow$ id
(3)	E	$+ id_2 * id_3$	shift
(4)	$E +	$id_2 * id_3$	shift
(5)	$E + id_2$	$* id_3$	reduce by $E \rightarrow$ id
(6)	$E + E$	$* id_3$	shift
(7)	$E + E *	id_3	shift
(8)	$E + E * id_3$	$	reduce by $E \rightarrow$ 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 $id_1 + id_2 * id_3$.

- While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift-reduce parser can make

1. Shift
2. Reduce
3. Accept
4. Error

- Shift
  - The next input symbol is shifted onto the top of the stack
- Reduce
  - The parser knows the right end of the handle is at the top of the stack
  - It must locate the left end within the stack and decide with what non-terminal to replace the handle

- Accept
  - The parser announces successful completion of parsing
- Error
  - The parser discovers that a syntax error has occurred and calls an error recovery routine

- Note :
  - The handle always appear on the top of the stack, never inside

Viable prefixes

- The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser

Operator precedence parsing

- The largest class of grammars for which shift-reduce parsers can be built successfully-the LR grammars
- A small but important class of grammars we can easily construct efficient shift-reduce parsers by hand
- These grammars have the property (among other essential requirements) that no production right side is ε or has two adjacent non-terminals
- A grammar with the latter property is called an operator grammar

- Example 4.27

  E -> EAE | (E) | -E | id

  A -> + | - | * | / |↑

  not an operator grammar because the RHS has two (in fact 3) consecutive non-terminals

- Substitute for A each of its alternatives to obtain the following operator grammar

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid id$$

- Technique : Operator precedence parsing
- Disadvantages
  - Hard to handle tokens like minus sign which has two different precedence(binary or unary)
  - One cannot be always sure that the parser accepts exactly the desired language
  - Only a small class of grammars can be parsed

- We define three disjoint precedence relations,

$$<\cdot \qquad \cdot> \qquad \doteq$$

and between certain pairs of terminals

- These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
$a <\cdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \cdot> b$	a "takes precedence over" b

Determining precedence relations

- Based on the traditional notions of associativity and precedence of operators
- For example, if * is to have higher precedence than +, we make + $\lessdot$ * and * $\gtrdot$ +

Using precedence relations

- The intention of the precedence relations is to delimit the handle of a right sentential form, with marking $\lessdot$ the left end, $\doteq$ appearing in the interior of the handle and $\gtrdot$ marking the right end
- Suppose that between a_i and a_{i+1} exactly one of the relations holds $\lessdot$ $\gtrdot$ or $\doteq$
- We use \$ to mark each end of the string
- Define $\$ \lessdot b$ and $b \gtrdot \$$ for all terminals b

- Precedence relations for id + id * id

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Fig. 4.23. Operator-precedence relations.

- The string with precedence relations inserted is

$ <· id ·> + <· id ·> * <· id ·> $

- The handle can be found by the following process

1. Scan the string from the left end until the first $\cdot>$ is encountered
2. Then scan backwards to the left over any $\doteq$'s until a $<\cdot$ is encountered
3. The handle contains everything to the left of the first $\cdot>$ and to the right of the $<\cdot$ encountered in step 2, including any intervening or surrounding non-terminals

- Algorithm : Operator Precedence parsing algorithm
- Input : An input string w and a table of precedence relations
- Output : If w is well-formed, a skeletal parse tree, with a placeholder non-terminal E labeling all interior nodes; otherwise an error indication
- Method :

  Initially, the stack contains $ and the input buffer the string w$. To parse, we execute the program of fig 4.24

```
(1)  set ip to point to the first symbol of w$;
(2)  repeat forever
(3)      if $ is on top of the stack and ip points to $ then
(4)          return
         else begin
(5)          let a be the topmost terminal symbol on the stack
                 and let b be the symbol pointed to by ip;
(6)          if a <· b or a ≐ b then begin
(7)              push b onto the stack;
(8)              advance ip to the next input symbol;
             end;
(9)          else if a ·> b then          /* reduce */
(10)             repeat
(11)                 pop the stack
(12)             until the top stack terminal is related by <·
                     to the terminal most recently popped
(13)         else error()
         end
```

Operator precedence relations from associativity and precedence

1. If operator θ_1 has higher precedence than operator θ_2, make $\theta_1 \cdot > \theta_2$ and $\theta_2 <\cdot \theta_1$
2. If θ_1 and θ_2 are operators of equal precedence, then make $\theta_1 \cdot > \theta_2$ and $\theta_2 \cdot > \theta_1$ if the operators are left-associative or make $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if they are right associative

3. Make $\theta \lessdot \cdot$ id, id $\cdot\gtrdot \theta$, $\theta \lessdot \cdot$ (, ($\lessdot \cdot \theta$,) $\cdot\gtrdot \theta$, $\theta \cdot\gtrdot$), $\theta \cdot\gtrdot$ \$, and \$ $\lessdot \cdot \theta$ for all operators θ. Also, let

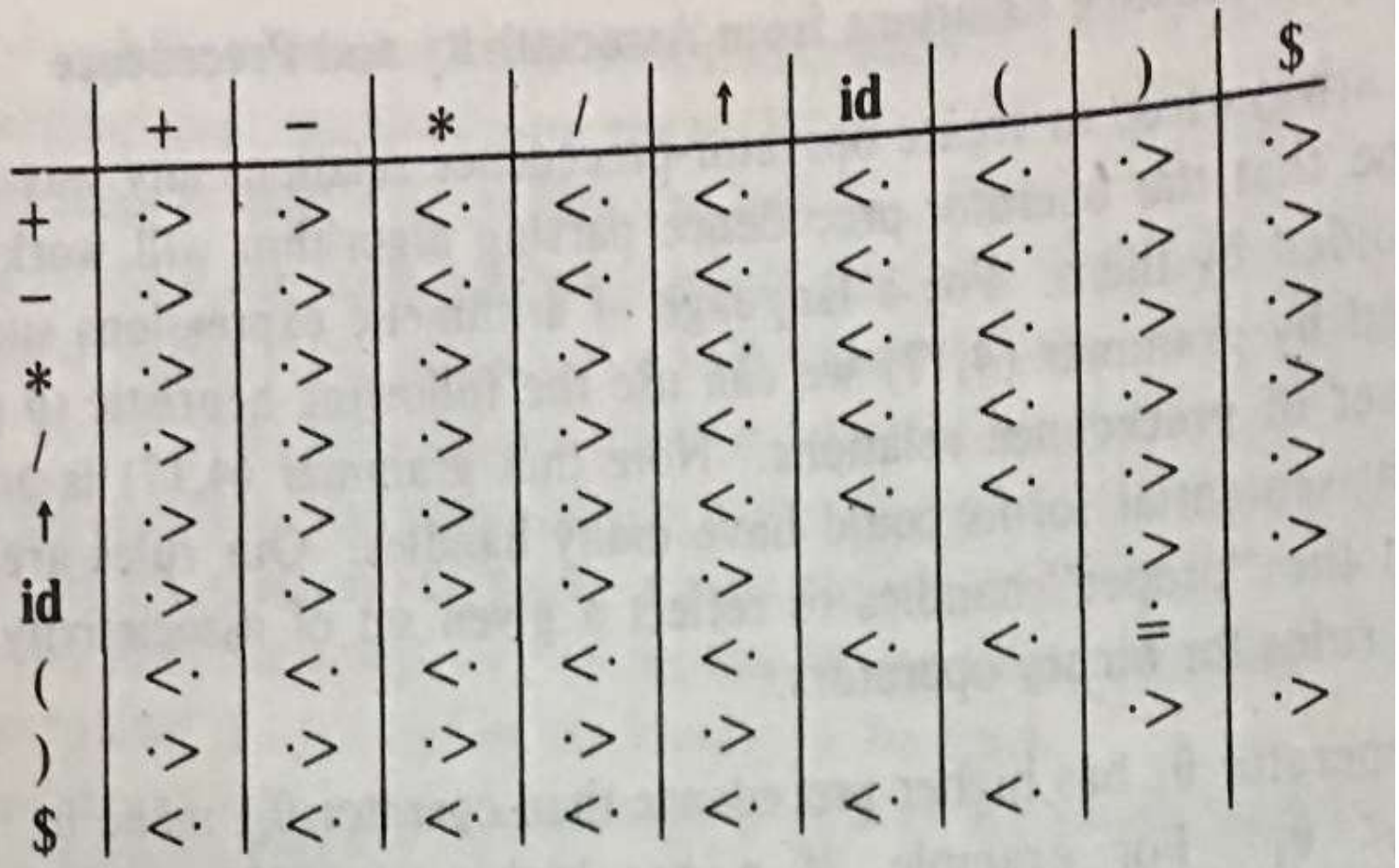
($\doteq$)	\$ $\lessdot \cdot$ (	\$ $\lessdot \cdot$ id
($\lessdot \cdot$ (	id $\cdot\gtrdot$ \$	) $\cdot\gtrdot$ \$
($\lessdot \cdot$ id	id $\cdot\gtrdot$)	) $\cdot\gtrdot$)

Example 4.28

- Figure 4.25 contains the operator precedence relations assuming

1. ↑ is of highest precedence and right-associative
2. * and / are of next highest precedence and left-associative and
3. + and – are of lowest precedence and left-associative
4. Blanks denotes error

- Input : id * (id ↑ id) – id / id



	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Fig. 4.25. Operator-precedence relations.

Handling unary operators

- Unary operator such as ¬ (logical negation)
- Suppose if ¬ is a unary prefix operator, we mαkе θ ¬

Precedence functions

- Need not store the table of precedence relations
- Can be encoded by two precedence functions f and g that map terminal symbols to integers
- We select f and g so that, for symbols a and b

1. $f(a) < g(b)$ whenever $a <\cdot b$,
2. $f(a) = g(b)$ whenever $a \doteq b$, and
3. $f(a) > g(b)$ whenever $a \cdot> b$.

	+	−	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

Algorithm 4.6 Constructing Precedence functions

Input : An operator precedence matrix

Output : Precedence functions representing the input matrix, or an indication that none exits

Method :

1. Create symbols f_a and g_a for each a that is a terminal or $
2. Partition the created symbols into as many groups as possible, in such a way that if a= b, then f_a and g_b are in the same group. Note that we may have to put symbols in the same group even if they are not related by =. For example if a=b and c=d, then fa and fc must be in the same group, since they are both in the same group as gb. If, in addition, c=d, then fa and gd are in the same group even though a=d may not hold

3. Create a directed graph whose nodes are the groups found in (2). For any a and b, if a<b, place an edge from the group of gb to the group of fa. If a>b, place an edge from the group of fa to that of gb. Note that an edge or path from fa to gb means that f(a) must exceed g(b); a path from gb to fa means that g(b) must exceed f(a)

4. If the graph constructed in (3) has a cycle, then no precedence functions exit. If there are no cycles, let f(a) be the length of the longest path beginning at the group of fa; let g(a) be the length of the longest path from the group of ga

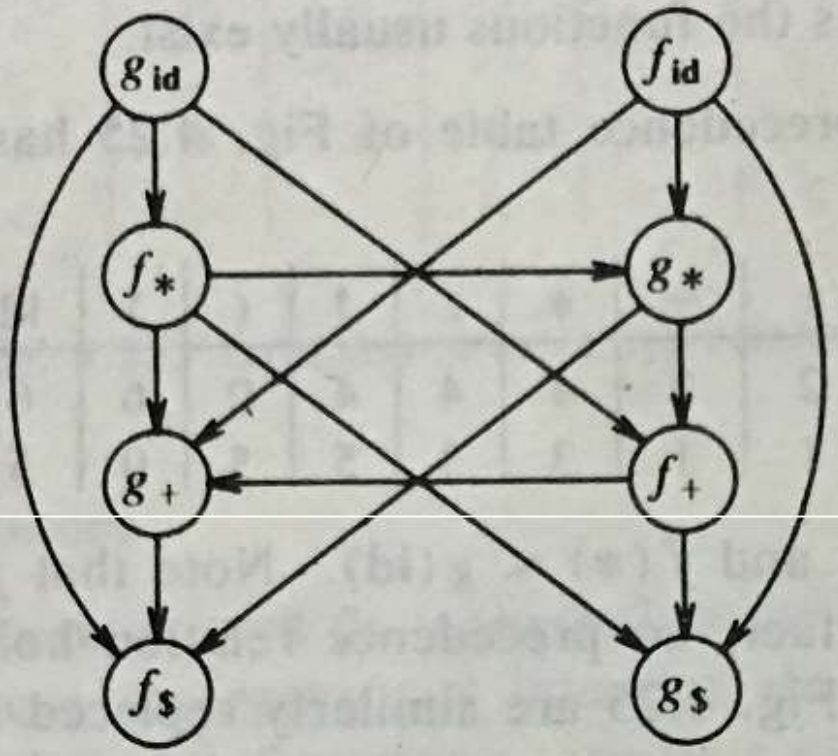


Fig. 4.26. Graph representing precedence functions.

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

LR parsing – Constructing SLR parsing tables, Canonical LR parsing tables and LALR parsing tables

LR parsers

- An efficient, bottom-up syntax analysis technique
- Used to parse a large class of CFGs
- Also called LR(k) parsing
  - L – Left-to-right scanning of the input
  - R – Constructing a right most derivation in reverse
  - k – the number of input symbols of lookahead that are used in making parsing decisions
    - When (k) is omitted, k is assumed to be 1

Advantages

- LR parsers can be constructed to recognize virtually all programming language constructs for which CFGs can be written
- The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift reduce parsers
- The class of parsers that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers
- An LR parser can detect a syntactic error as soon as possible to do so on a left-to-right scan of the input

Disadvantages

- It is too much work to construct an LR parser by hand for a typical programming language grammar
- Needs a parser generator
  - Such generators are available,YACC
  - If the grammar contains ambiguities or other constructs that are difficult to parse, then the parser generator can locate these constructs and inform the compiler designer of their presence

The LR parsing algorithm

- The schematic form of an LR parser is shown in fig 4.29
- Consists of an input, an output, a stack, a driver program and a parsing table that has two parts (action and goto)
- The driver program is the same for all LR parsers
- Only the parsing table changes from one parser to another
- The parsing program reads characters from an input buffer one at a time
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2.....X_ms_m$,where s_m is on the top of the stack

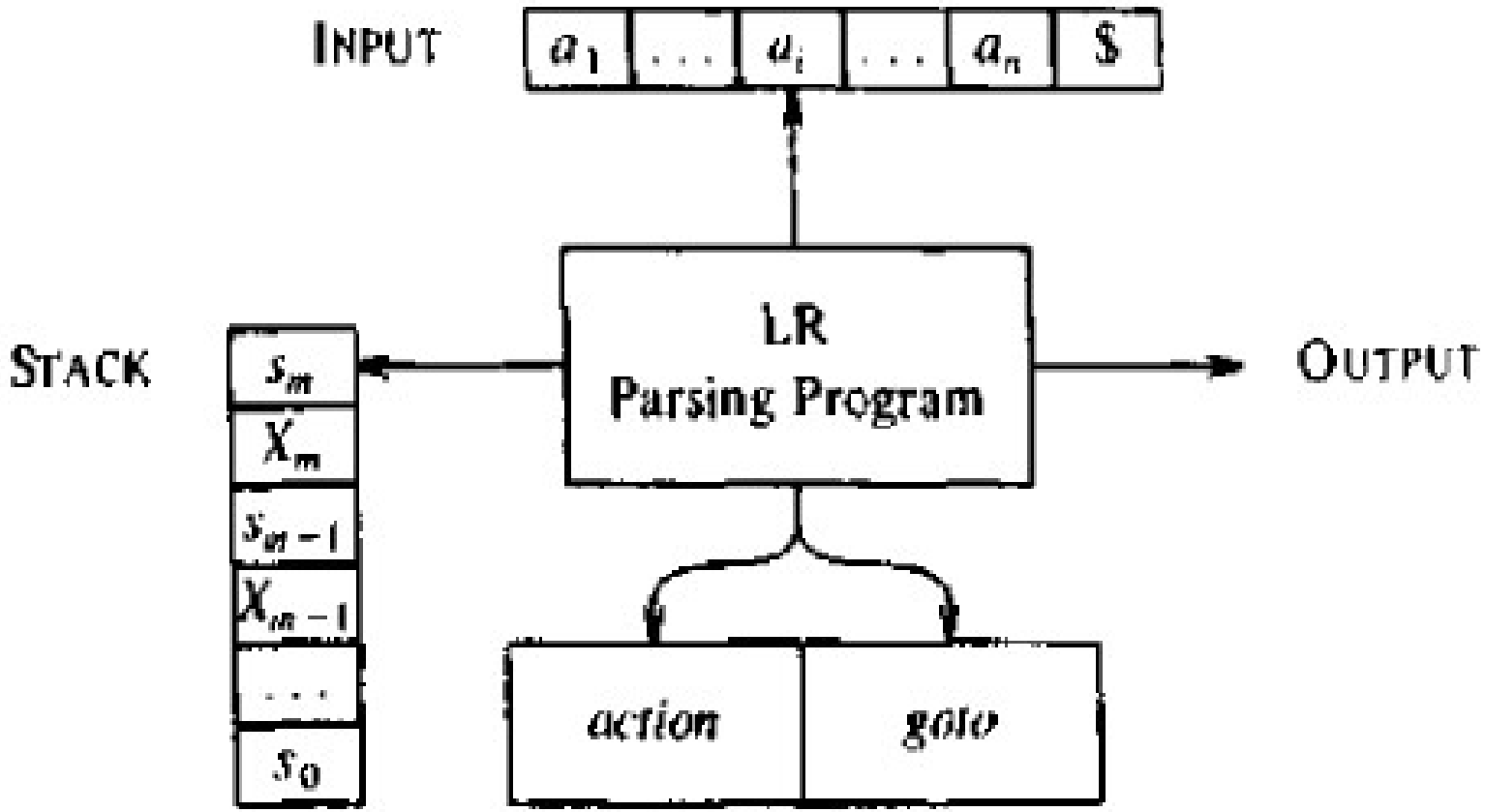


Fig. 4.29. Model of an LR parser.

- Each Xi is a grammar symbol and each si is a symbol called a state
- Each state symbol summarizes the information contained in the stack below it
- The combination of the state symbol on the top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision
- The parsing table consists of two parts , a parsing function ***action*** and a goto function ***goto***
- The program driving the parser behaves as follows

- It determines s_m, the state currently on the top of the stack and a_i, the current input symbol
- It then consults action[s_m, a_i], the parsing table entry for the state s_m and input a_i,which can have one of the four values

1. shift s, where s is a state
2. reduce by a grammar production $A \rightarrow \beta$
3. accept
4. Error

- The function goto takes a state and grammar symbol as arguments and produces a state

- A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

  $(s_0X_1s_1X_2s_2.....X_ms_m , a_ia_{i+1}.....a_n\$)$
- This configuration represents the right sentential form

  $X_1X_2.....X_m , a_ia_{i+1}.....a_n$
- The next move of the parser is determined by reading

  ai, the current input symbol and

  sm, the state on top of the stack and then consulting the parser action table entry

  action[sm,ai]
- The configurations resulting after each of the types of move are as follows:

1. If action[sm,ai] = shift s,
   - The parser executes a shift move, entering the configuration.

     $(s_0X_1s_1X_2s_2.....X_ms_ma_i\ s,\ a_{i+1}.....a_n\$)$
   - Here the parser has shifted both the current input symbol ai and the next state s, which is given in action[sm,ai] onto the stack; ai+1 becomes the current input symbol

2. If action[sm,ai] = reduce $A \rightarrow \beta$,

- Then the parser executes a reduce move, entering the configuration

$(s_0X_1s_1X_2s_2.....X_{m-r}s_{m-r}A\ s,\ a_ia_{i+1}.....a_n\$)$

- Here the parser first popped off 2r symbols off the stack(r state symbols and r grammar symbols),where r is the length of β, exposing the state s_{m-r}
- Consults goto entry , if goto[Sm-r, A] = s, then pushes A and s onto the stack
- The current input symbol is not changed

3. If action[sm,ai] = accept, parsing is completed

4. If action[sm,ai] = error, the parser has discovered an error and calls an error recovery routine

- Summarizing the algorithm

Algorithm 4.7 LR parsing algorithm

- Input : An input string w and an LR parsing table with functions action and goto for grammar G
- Output : If w is in L(G),a bottom up parse for w; otherwise, an error indication

- Method : Initially, the parser has s0 on its stack, where s0 is the initial state, and w$ in the input buffer. The parser then executes the program in fig 4.30 until an accept or error action is encountered

```
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.

Example 4.33

- Fig 4.31shows 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

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.

- The codes for the actions are
- 1. sj means shift and stack state I
- 2. rj means reduce by production numbered j
- 3. acc means accept
- 4. blank means error

- On input id * id + id, the sequence of stack and input contents is as shown in fig 4.32

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

Fig. 4.32. Moves of LR parser on **id** * **id** + **id**.

How to construct an lr parsing table for a given grammar

- A grammar for which we can construct a parsing table – an LR grammar
- An LR parser does not have to scan the entire stack to know when the handle appears on top
- Rather, the state symbol on the top of the stack contains all the information it needs

Constructing LR parsing tables

- Three methods
  - Varying in their power and ease of implementation
- Simple LR or SLR
  - Weakest of the three in terms of the number of arguments for which it succeeds, but easiest to implement
- The parsing table is called SLR parsing table
- A grammar for which an SLR parser can be constructed – SLR grammar
- The other two methods augment the SLR method with lookahead information

- An LR(0) item (item for short) of a grammar is a production of G with a dot at some position of the right side
- Thus, production $A \rightarrow XYZ$ yields the four items

  $A \rightarrow .XYZ$

  $A \rightarrow X.YZ$

  $A \rightarrow XY.Z$

  $A \rightarrow XYZ.$
- The production $A \rightarrow \varepsilon$ generates only one item

  $A \rightarrow .$
- An item can be represented by a pair of integers, the first giving the number of the production and the second the position of the dot

- An item indicates how much of a production we have seen at a given point in the parsing process
- For example, the first item indicates that we hope to see a string derivable from XYZ next on the input
- The second item indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ

- One collection of sets of LR(0) items, which we call the canonical LR(0) collection, provides the basis for constructing SLR parsers
- To construct the LR(0) collection for a grammar, we define an augmented grammar and two functions, closure and goto

- If G is a grammar with start symbol S, then G', the augmented grammar for G,is G with a new start symbol S' and production S' → S.
- The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce the acceptance of the input
- ie, acceptance occurs when and only when the parser is about to reduce by S' → S

The Closure Operation

- If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by two rules:

1. Initially, every item in I is added to closure(I)
2. If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\gamma$ to I, if it is not already there

- We apply this rule until no more new items can be added to closure(I)

- Intuitively $A \rightarrow \alpha.B\beta$ indicates that, at some point in the parsing process, we think we might see a substring derivable from $B\beta$
- If $B \rightarrow \gamma$ is a production , we also expect we might see a substring derivable from γ at this point
- Consider the augmented expression grammar

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

- Here I= {E' → .E} is an item
- The Closure(I)= Closure({E' → .E}) is given as follows:

$E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow . T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

```
function closure(I);
begin
        J := I;
        repeat
                for each item A →α.Bβ in J and each
                production B → γ of G such that
                B → .γ is not in J do

                                add B → .γ to J
        until no more items can be added to J
        return J
end
```

- All the sets of items can be divided into two classes of items

1. Kernel items
   - Which include the initial item S’ -> .S
   - And all items whose dots are not at the left end
2. Non-kernel items
   - Which have their dots at the left end

The goto operation

- Goto(I,X) where I is a set of items and X is a grammar symbol
- Goto(I,X) is defined to be the closure of the set of all items [A→αX.β] such that [A→α.Xβ] is in I

- If I is the set two items {[E' → E.] ,[E→E.+T]}, then goto(I,+) consists of

$$E \rightarrow E + .T$$
$$T \rightarrow .T * F$$
$$T \rightarrow .F$$
$$F \rightarrow .(E)$$
$$F \rightarrow .id$$

- goto(I,+) is computed by examining I for items with + immediately to the right of the dot

- E'->E. is not such an item but E ->E.+T is
- We moved the dot over the + to get {E->E+.T} and then took the closure of this set

- The algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G' – The Sets-of-Items Construction

```
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

Example 4.36

- The canonical collection of sets of LR(0) items for grammar is shown in figure 4.35
- The goto function for this set of items is shown as the transition diagram of a deterministic finite automaton D in fig 4.36

I_0: $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

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

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

I_3: $T \to F \cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_5: $F \to \mathbf{id} \cdot$

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

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

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).

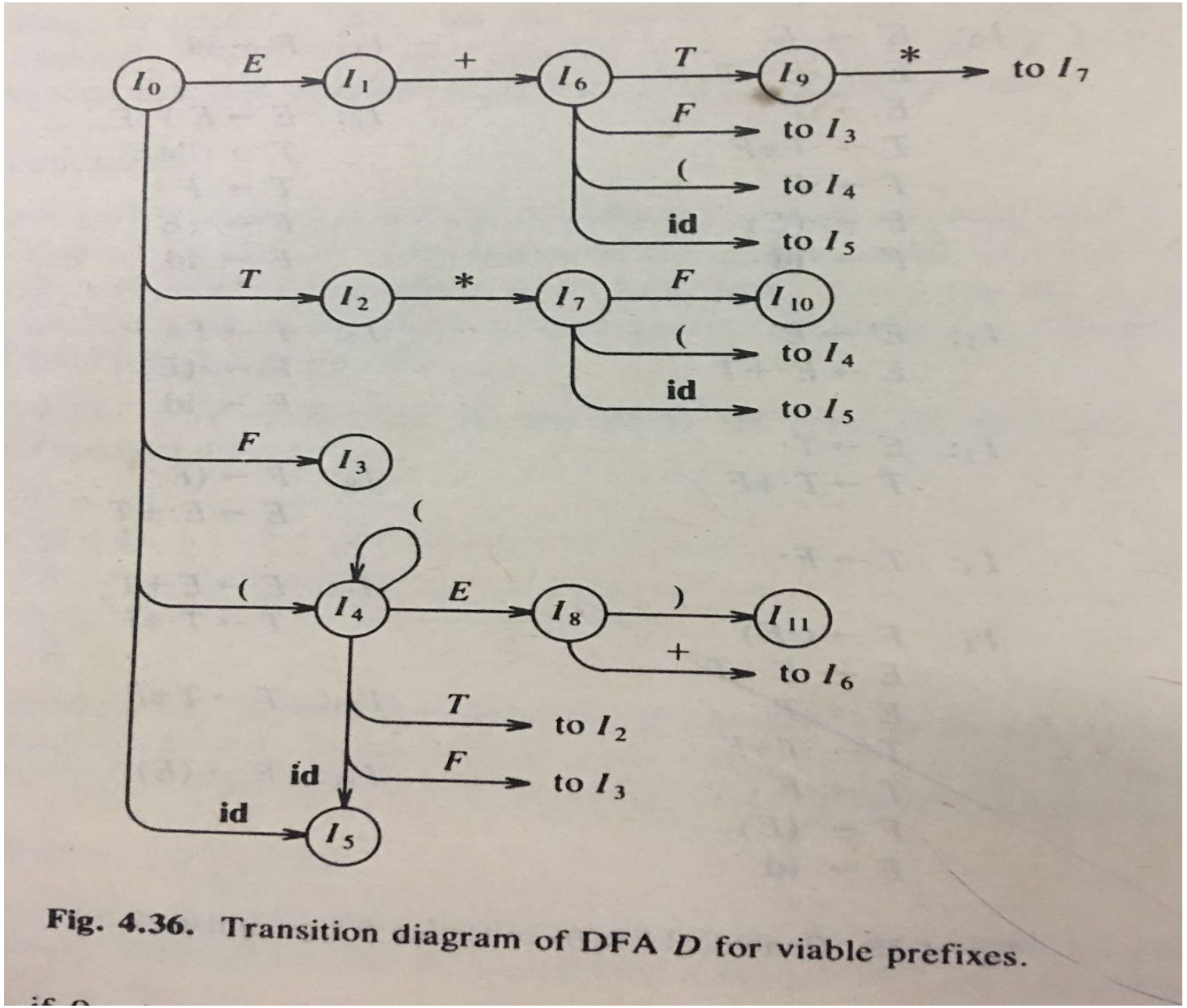
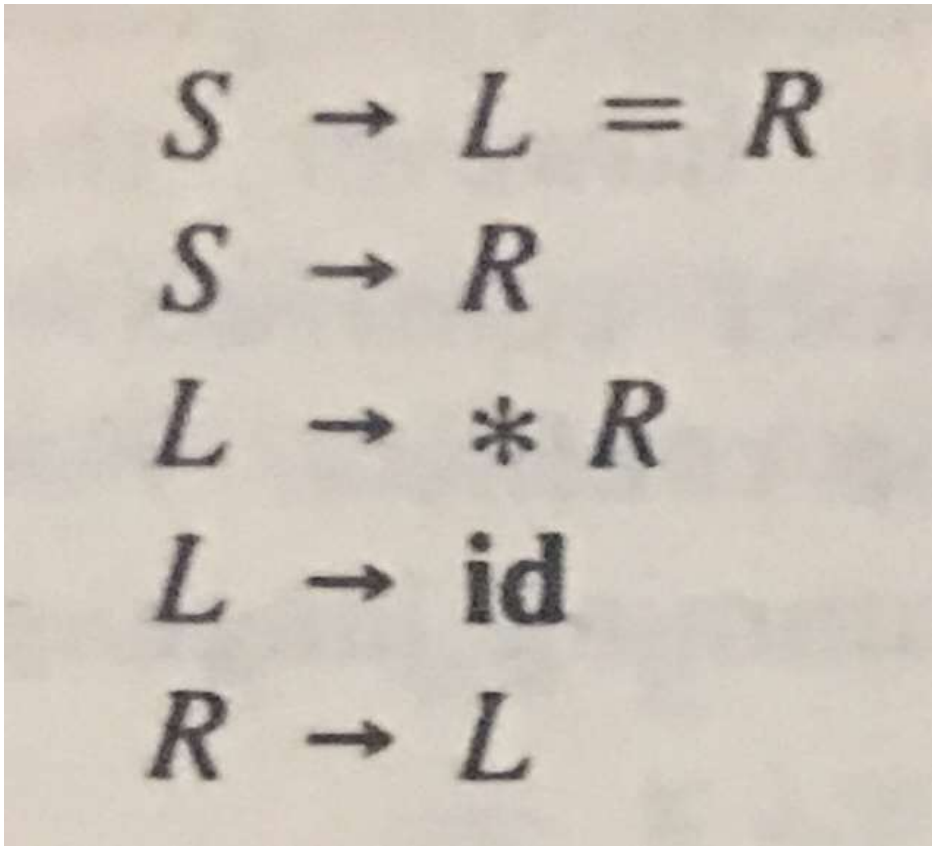


Fig. 4.36. Transition diagram of DFA D for viable prefixes.

Example 4.39

- Construct the Canonical LR(0) collection for the grammar



I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to * R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

Fig. 4.37. Canonical LR(0) collection for grammar (4.20).

Algorithm 4.8. Constructing an SLR parsing table.

Input. An augmented grammar G'.

Output. The SLR parsing table functions *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 state i are determined as follows:

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

   c) If $[S' \to S \cdot]$ 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 items containing $[S' \rightarrow \cdot S]$. □

CONSTRUCTION OF THE SETS OF LR(1) ITEMS

- Input : An augmented grammar G'
- Output : The sets of LR(1) items that are the sets of items valid for one or more viable prefixes of G'
- Method : The procedure closure and goto and the main routine items for constructing the sets of items as shown in fig 4.38

```
function closure (I);
begin
    repeat
        for each item [A → α·Bβ, a] in I,
            each production B → γ in G',
            and each terminal b in FIRST(βa)
            such that [B → ·γ, b] is not in I do
                add [B → ·γ, b] to I;
    until no more items can be added to I;
    return I
end;
```

```
function goto(I, X);
begin
    let J be the set of items [A → αX·β, a] such that
        [A → α·Xβ, a] is in I;
    return closure(J)
end;

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. 4.38. Sets of LR(1) items construction for grammar G'.

Example 4.42

- Consider the following augmented grammar

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC \mid d$

I_0:

$$S' \rightarrow \cdot S,\ \$$$
$$S \rightarrow \cdot CC,\ \$$$
$$C \rightarrow \cdot cC,\ c/d$$
$$C \rightarrow \cdot d,\ c/d$$

I_1:

$$S' \rightarrow S\cdot,\ \$$$

I_2:

$$S \rightarrow C\cdot C,\ \$$$
$$C \rightarrow \cdot cC,\ \$$$
$$C \rightarrow \cdot d,\ \$$$

I_3: $C \rightarrow c \cdot C,\ c/d$
$C \rightarrow \cdot cC,\ c/d$
$C \rightarrow \cdot d,\ c/d$

I_4: $C \rightarrow d \cdot,\ c/d$

I_5: $S \rightarrow CC \cdot,\ \$$

I_6: $C \rightarrow c \cdot C,\ \$$
$C \rightarrow \cdot cC,\ \$$
$C \rightarrow \cdot d,\ \$$

I_7: $C \rightarrow d \cdot, \$$

I_8: $C \rightarrow cC \cdot, c/d$

I_9: $C \rightarrow cC \cdot, \$$

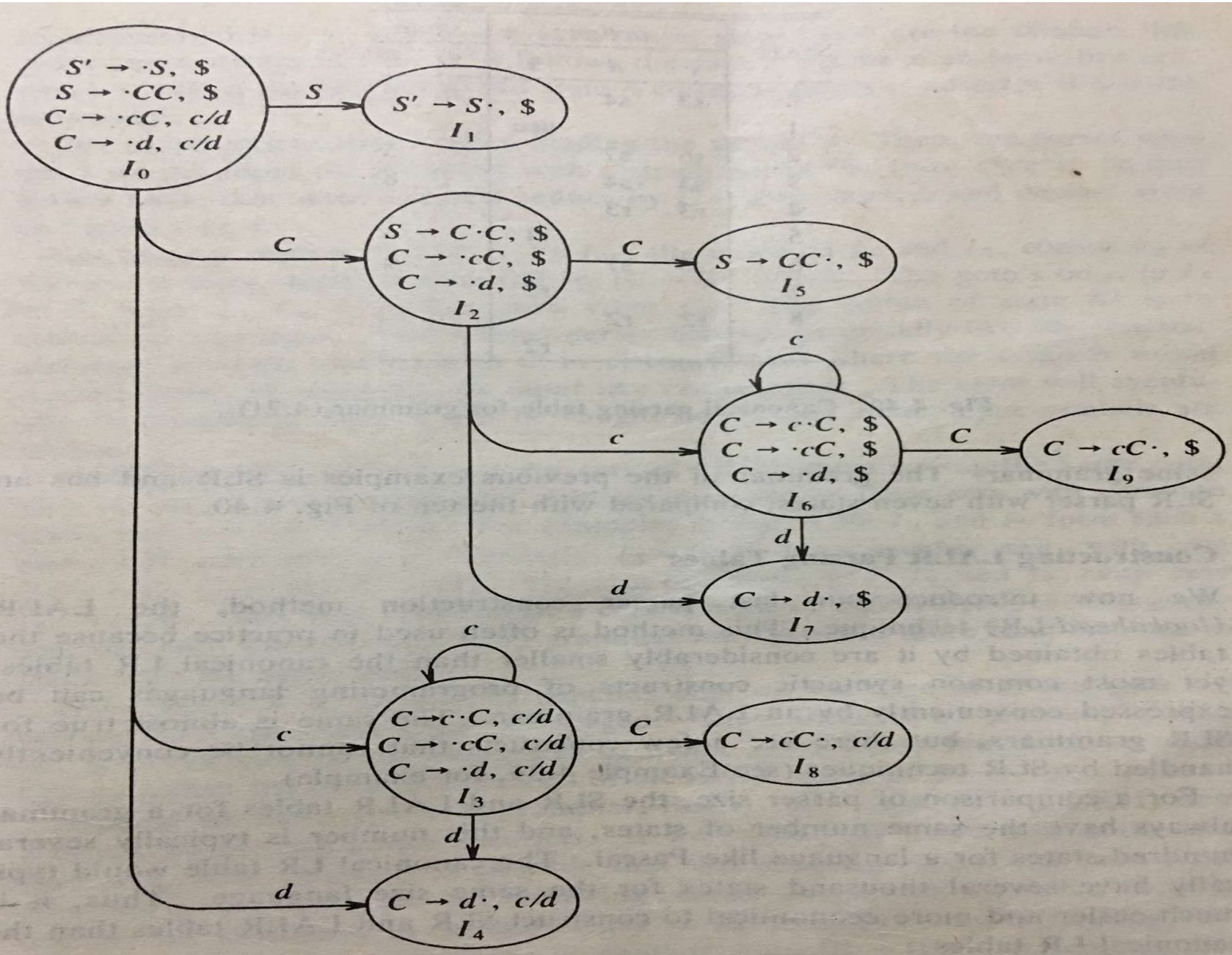


Fig. 4.39. The *goto* graph for grammar (4.21).

Algorithm 4.10. Construction of the canonical LR parsing table.

Input. An augmented grammar G'.

Output. The canonical LR parsing table functions *action* and *goto* for G'.

Method.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items for G'.

2. State i of the parser is constructed from I_i. The parsing actions for state i are determined as follows:

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

   b) If $[A \rightarrow \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set $action[i, a]$ to "reduce $A \rightarrow \alpha$."

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

If a conflict results from the above rules, the grammar is said not to be LR(1), and the algorithm is said to fail.

3. The goto transitions for state i are determined as follows: 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 containing item $[S' \rightarrow \cdot S, \$]$.

□

- The table formed from the parsing action and goto functions produced by the algorithm 4.10 is called the canonical LR(1) parsing table
- An LR parser using this table is called a canonical LR(1) parser
- If the parsing action function has no multiply-defined entries, then the given grammar is called an LR(1) grammar
- The canonical parsing table for the grammar 4.21
- Every SLR(1) grammar is an LR(1) grammar, but for an SLR(1) grammar, the canonical LR parser may have more states than the SLR parser for the same grammar

STATE	*action*			*goto*	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Fig. 4.40. Canonical parsing table for grammar (4.21).

Constructing LALR Parsing tables

- Look-ahead LR
- The general idea is to construct the sets of LR(1) items
- And if no conflict arise, merge sets with common cores
- Then construct the parsing table from the collection of merged sets of items

Algorithm 4.11. An easy, but space-consuming LALR table construction.

Input. An augmented grammar G'.

Output. The LALR parsing table functions *action* and *goto* for G'.

Method.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.10. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).
4. The *goto* table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of $goto(I_1, X)$, $goto(I_2, X)$, $\ldots$, $goto(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as $goto(I_1, X)$. Then $goto(J, X) = K$. □

I_{36}: $C \rightarrow c \cdot C,\ c/d/\$$

$C \rightarrow \cdot cC,\ c/d/\$$

$C \rightarrow \cdot d,\ c/d/\$$

I_4 and I_7 are replaced by their union:

I_{47}: $C \rightarrow d \cdot,\ c/d/\$$

and I_8 and I_9 are replaced by their union:

I_{89}: $C \rightarrow cC \cdot,\ c/d/\$$

STATE	*action*			*goto*	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Fig. 4.41. LALR parsing table for grammar (4.21).