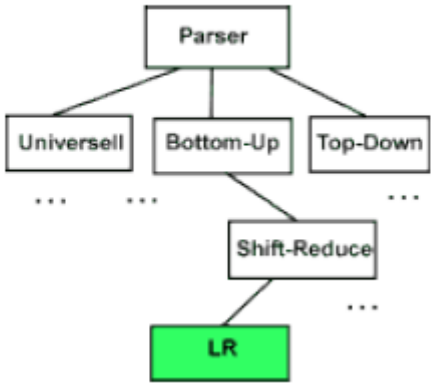
UNIT 4

LR PARSER

4.1 LR PARSING INTRODUCTION

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.



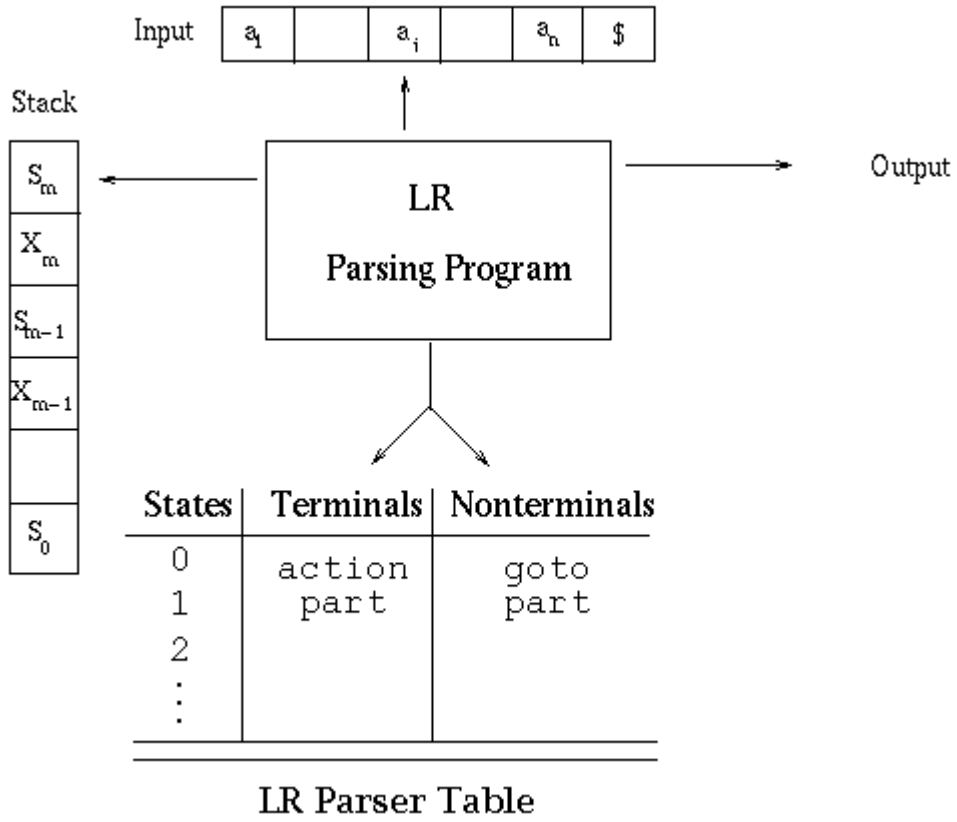
4.2 WHY LR PARSING:

- ✓ LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars 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 methods.
- ✓ The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- ✓ An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

4.3.MODELS OF LR PARSERS

The schematic form of an LR parser is shown below.



The program uses a stack to store a string of the form s0X1s1X2...Xmsm where sm is on top. Each Xi is a grammar symbol and each si is a symbol representing a state. Each state symbol summarizes the information contained in the stack below it. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shiftreduce parsing decision. The parsing table consists of two parts: a parsing action function action and a goto function goto. The program driving the LR parser behaves as follows: It determines sm the state currently on top of the stack and ai the current input symbol. It then consults action[sm,ai], which can have one of four values:

- shift s, where s is a state
- reduce by a grammar production A -> b
- accept
- error

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

For a parsing table constructed for a grammar G, the goto table is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G. Recall that the viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shiftreduce parser because they do not extend past the rightmost handle.

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

(s0 X1 s1 X2 s2... Xm sm, ai ai+1... an$)

This configuration represents the right-sentential form

X1 X1 ... Xm ai ai+1 ...an

in essentially the same way a shift-reduce parser would; only the presence of the states on the stack is new. Recall the sample parse we did (see Example 1: Sample bottom-up parse) in which we assembled the right-sentential form by concatenating the remainder of the input buffer to the top of the stack. The next move of the parser is determined by reading ai and sm, and consulting the parsing action table entry action[sm, ai]. Note that we are just looking at the state here and no symbol below it. We'll see how this actually works later.

The configurations resulting after each of the four types of move are as follows:

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

(s0 X1 s1 X2 s2... Xm sm ai s, ai+1... an$)

Here the parser has shifted both the current input symbol ai and the next symbol.

If action[sm, ai] = reduce A -> b, then the parser executes a reduce move, entering the configuration,

(s0 X1 s1 X2 s2... Xm-r sm-r A s, ai ai+1... an$)

where s = goto[sm-r, A] and r is the length of b, the right side of the production. The parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state sm-r. The parser then pushed both A, the left side of the production, and s, the entry for goto[sm-r, A], onto the stack. The current input symbol is not changed in a reduce move.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For example, we might just print out the production reduced.

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

4.4.OPERATOR PRECEDENCE PARSING

Precedence Relations

Bottom-up parsers for a large class of context-free grammars can be easily developed using operator grammars.Operator grammars have the property that no production right side is empty or has two adjacent nonterminals. This property enables the implementation of efficient operator-precedence parsers. These parser rely on the following three precedence relations:

Relation Meaning

a <· b a yields precedence to b

a =· b a has the same precedence as b

a ·> b a takes precedence over b

These operator precedence relations allow to delimit the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

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

Example: The input string:

id1 + id2 * id3

after inserting precedence relations becomes

$ <· id1 ·> + <· id2 ·> * <· id3 ·> $

Having precedence relations allows to identify handles as follows:

- scan the string from left until seeing ·>
- scan backwards the string from right to left until seeing <·
- everything between the two relations <· and ·> forms the handle

4.5 OPERATOR PRECEDENCE PARSING ALGORITHM

Initialize: Set ip to point to the first symbol of w$

Repeat: Let X be the top stack symbol, and a the symbol pointed to by ip

if $ is on the top of the stack and ip points to $ then return

else

Let a be the top terminal on the stack, and b the symbol pointed to

by ip

if a <· b or a =· b then

push b onto the stack

advance ip to the next input symbol

else if a ·> b then

repeat

pop the stack

until the top stack terminal is related by <·

to the terminal most recently popped

else error()

end

4.6 ALGORITHM FOR CONSTRUCTING PRECEDENCE FUNCTIONS

1. Create functions fa for each grammar terminal a and for the end of string symbol;

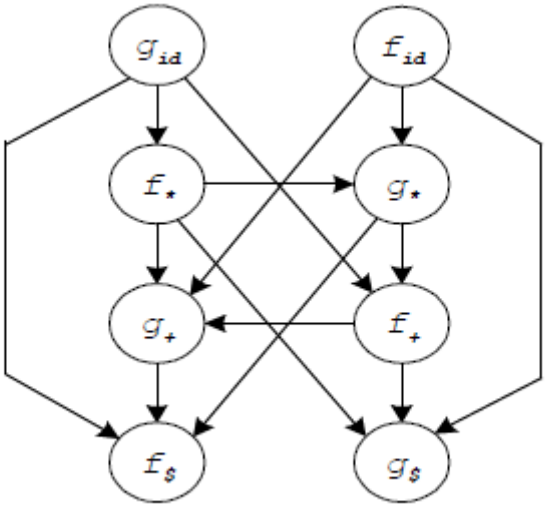
2. Partition the symbols in groups so that fa and gb are in the same group if a =· b (there can be symbols in the same group even if they are not connected by this relation)

3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of gb to the group of fa if a <· b, otherwise if a ·> b place an edge from the group of fa to that of gb;

4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of fa and gb Example:

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

Consider the above table Using the algorithm leads to the following graph:



4.7 SHIFT REDUCE PARSING

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols. During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS,replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input. The parser is nothing but a stack automaton which may be in one of several discrete states. A state is usually represented simply as an integer. In reality, the parse stack contains states, rather than

grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

The operation of the parser is controlled by a couple of tables:

4.8 ACTION TABLE

The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state s and the current lookahead terminal is t, the action taken by the parser depends on the contents of action[s][t], which can contain four different kinds of entries:

Shift s'

Shift state s' onto the parse stack.

Reduce r

Reduce by rule r. This is explained in more detail below.

Accept

Terminate the parse with success, accepting the input.

Error

Signal a parse error

4.9 GOTO TABLE

The goto table is a table with rows indexed by states and columns indexed by nonterminal symbols. When the parser is in state s immediately after reducing by rule N, then the next state to enter is given by goto[s][N].

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1. Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.

2. Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

· If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

· If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r. Then consult the goto table and

push the state given by goto[s'][N] onto the stack. The lookahead token is not changed by this step.

- If the action table entry is accept, then terminate the parse with success.
- If the action table entry is error, then signal an error.

3. Repeat step (2) until the parser terminates.

For example, consider the following simple grammar

0) $S: stmt <EOF>

1) stmt: ID ':=' expr

2) expr: expr '+' ID

3) expr: expr '-' ID

4) expr: ID

which describes assignment statements like a:= b + c - d. (Rule 0 is a special augmenting production added to the grammar).

One possible set of shift-reduce parsing tables is shown below (sn denotes shift n, rn denotes reduce n, acc denotes accept and blank entries denote error entries):

Parser Tables

Parser Tables

	Action Table					Goto Table	
	ID	**':='**	**'+'**	**'-'**	**<EOF>**	**stmt**	**expr**
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

A trace of the parser on the input a:= b + c - d is shown below:

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10
0/$S 1/a 3/:= 6/expr 8/- 10/d	<EOF>	r3
0/$S 1/a 3/:=	<EOF>	g6 on expr
0/$S 1/a 3/:= 6/expr	<EOF>	r1
0/$S	<EOF>	g2 on stmt
0/$S 2/stmt	<EOF>	s4
0/$S 2/stmt 4/<EOF>		*accept*

Each stack entry is shown as a state number followed by the symbol which caused the transition to that state.

4.10 SLR PARSER

An *LR(0) item* (or just *item*) of a grammar *G* is a production of *G* with a dot at some position of the right side indicating how much of a production we have seen up to a given point.

For example, for the production E -> E + T we would have the following items:

[E -> .E + T]

[E -> E. + T]

[E -> E +. T]

[E -> E + T.]

Stack	State	Comments
Empty	[E'-> .E]	can't go anywhere from here
	e-transition	so we follow an e-transition
Empty	[F -> .(E)]	now we can shift the (
(	[F -> (.E)]	building the handle (E); This state says: "I have (on the stack and expect the input to give me tokens that can eventually be reduced to give me the rest of the handle, E)."

4.11 CONSTRUCTING THE SLR PARSING TABLE

To construct the parser table we must convert our NFA into a DFA. The states in the LR table will be the e-closures of the states corresponding to the items SO...the process of creating the LR state table parallels the process of constructing an equivalent DFA from a machine with e-transitions. Been there, done that - this is essentially the subset construction algorithm so we are in familiar territory here.

We need two operations: closure()

and goto().

closure()

If I is a set of items for a grammar *G*, then closure(I) is the set of items constructed from I by the two rules: Initially every item in I is added to closure(I)

If *A* -> a.*B*b is in closure(I), and *B* -> g is a production, then add the initial item [*B* -> .g] to I, if it is not already there. Apply this rule until no more new items can be added to closure(I).

From our grammar above, if I is the set of one item {[E'-> .E]}, then closure(I) contains:

I0: E' -> .E

E -> .E + T

E -> .T

T -> .T * F

T -> .F

F -> .(E)

F -> .id

goto()

goto(I, *X*), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items [*A* -> a*X*.b] such that [*A* -> a.*X*b] is in I. The idea here is fairly intuitive: if I is the set of items that are valid for some viable prefix g, then goto(I, *X*) is the set of items that are valid for the viable prefix g*X*.

4.12 SETS-OF-ITEMS-CONSTRUCTION

To construct the canonical collection of sets of LR(0) items for

augmented grammar G'.

procedure items(G')

begin

C := {closure({[S' -> .S]})};

repeat

for each set of items 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;

4.13 ALGORITHM FOR CONSTRUCTING AN SLR PARSING TABLE

Input: augmented grammar G'

Output: SLR parsing table functions action and goto for G'

Method:

Construct C = {I0, I1 , ..., In} the collection of sets of LR(0) items for G'.

State i is constructed from Ii:

if [A -> a.ab] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.

if [A -> a.] is in Ii, then set action[i, a] to "reduce A -> a" for all a in FOLLOW(A). Here A may

not be S'.

if [S' -> S.] is in Ii, then set action[i, $] to "accept"

If any conflicting actions are generated by these rules, the grammar is not SLR(1) and the algorithm fails to produce a parser. The goto transitions for state i are constructed for all *nonterminals A* using the rule: If goto(Ii, A)= Ij, then goto[i, *A*] = j.

All entries not defined by rules 2 and 3 are made "error".

The inital state of the parser is the one constructed from the set of items containing [*S*' -> .S].

Let's work an example to get a feel for what is going on,

An Example

(1) E -> E * B

(2) E -> E + B

(3) E -> B

(4) B -> 0

(5) B -> 1

The Action and Goto Table The two LR(0) parsing tables for this grammar look as follows:

	action					*goto*	
state	*	+	0	1	$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		