UNIT-3

BOTTOM-UP parsing: shift Reduce parsing, LR and LALR parsing, Error recovery in parsing, handling ambiguous gramm-ar, YACC - automatic parser generator.

Bottom-up parsing:

A general style of bottom-up syntax Analysis, known as a shift-reduce parsing.

shift reduce parsing attempts to construct parse trees for an i/p string begining at the leaves (bottom) and working up towards the root (top).

Bottom-up parsers attempt to find the right-most derivation in reverse for the given i/p string.

Ex :- $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d.$

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

a b b c d e

a A b c d e [∵ $A \rightarrow b$]

a A d e [∵ $A \rightarrow Abc$]

a A B e [∵ $B \rightarrow d$]

S [∵ $S \rightarrow aABe$]

This is the reverse of right most derivation.

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

Handle:

Handle of string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of right most derivation.

If $A \rightarrow \beta$, then β is said to be handle, since it can be reduced to A, in the string $\alpha\beta w$.

Reducing β to A in $\alpha\beta w$ is said to be "pruning the handle"

$S \underset{rm}{\Rightarrow} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, then $A \rightarrow \beta$ in the *position following* α *is a handle of* $\alpha A w$

Ex:
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

RMD is given by
$E \rightarrow \underline{E + E}$
$\rightarrow E + \underline{E * E}$
$\rightarrow E + E * \underline{id}$
$\rightarrow E + \underline{id} * id$
$\rightarrow \underline{id} + id * id.$

Note: Underlined a handle of each right-sentinel form.

Handle pruning :-

- A RMD in reverse can be obtained by "handle pruning
2 points are to be considered when we are parsing by handle pruning:

1) to locate the substring to be reduced in a right-sentential form.
2) to determine each production to choose if there is more than one production with the substring on right side.

Ex :-

Right-sentential Form	Handle	Reducing production
id + id * id	id	E → id
E + id * id	id	E → id
E + E * id	id	E → id
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E	E	

Stack implementation of shift-Reduce parsing :.

Data structures used to implement a shift reduce parser is to use a stack to hold grammar symbols and an i/p buffer to hold the string w to be parsed.

we use $ → bottom of stack & right end of i/p.

Initially, stack is empty, and the string w is on the i/p, as follows :-

STACK	INPUT
$	w$

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

STACK	INPUT
$S	$

Ex :- id + id * id.

	STACK	INPUT	ACTION
1)	$	id + id * id $	shift
2)	$ id	+ id * id $	reduce by E → id
3)	$ E	+ id * id $	shift
4)	$ E +	id * id $	shift
5)	$ E + id	* id $	reduce by E → id
6)	$ E + E	* id $	shift
7)	$ E + E *	id $	shift
8)	$ E + E * id	$	reduce by E → id
9)	$ E + E * E	$	~~shift~~ " " E → E*E
10)	$ E + E	$	" " E → E+E
11)	$ E	$	accept.

primary operations of the parser are shift and reduce. There a 4 possible actions a shift-reduce parser can make :

1. In a shift action, the next i/p symbol is shifted onto the top of the stack.
2. In a reduce action, the parser knows the right end of the handle is at the top of stack. It must then locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.
3. In an accept action, the parser announces successful completion of parsing.
4. In an error action, the parser discovers that a syntax ~~tree~~ error has occured and calls an error recovery routine.

viable prefixes:-

The set of prefixes of right sentential form that can appear on the stack of a shift-reduce parser are called viable prefixes.

conflicts During shift reducing:-

when the shift-Reduce parser is applied to CFG, it leads to some conflicts, because shift-reduce parser cannot be used for context free grammars. The conflicts are:-

1) shift-reduce conflict:- The parser even after knowing the entire stack contents and next i/p symbol, cannot decide whether to shift or to Reduce.
2) reduce/reduce conflict:- The parser knowing the entire stack contents and cannot decide whether productions to use or which reduction to make.

Ex:- consider an dangling-else grammar.

stmt → if expr then stmt

| if expr then stmt else stmt

| other.

↳ any other statements.

STACK	INPUT
... if expr then stmt	else .. $

we cannot tell, whether "if expr then stmt" is a handle or not. This leads the parser in confusion whether to shift else or reduce the stack top element. There is a shift/reduce conflict, because, it is possible to reduce "if expr then stmt" on the stack to "stmt"; or it is also possible to shift "else" and then look for another "stmt" to complete the alternative "if expr then stmt else stmt" which can be reduced to "stmt".

LR parser:-

LR parser presents an efficient, bottom-up syntax analysis tech that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing, the

"L" —> left-to-right scanning of the i/p

R —> for constructing a right most derivation in reverse.

k —> for the number of i/p symbols of look ahead that are used in making parsing decisions.

when k is omitted, k is assumed to be 1.

LR parsing is attractive for a variety of regions:-

-> LR parsers can be constructed to recognize virtually all programming lang constructs for which context-free grammar can be written

-> 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 method 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 it is possible to do so on a left-to-right scan of i/p

Drawback:-

It is too much work to construct an LR parser by hand for a typical prog lang gram.

one needs a specialized tool - LR parser generator

There are 3 tech for constructing an LR parsing table for a grammar.

1) Simple LR (SLR):-

Easy to implement, but the least powerful of three. It may fail to produce a parsing table for certain grammars on which the other method succeeds

2) Canonical LR (CLR):

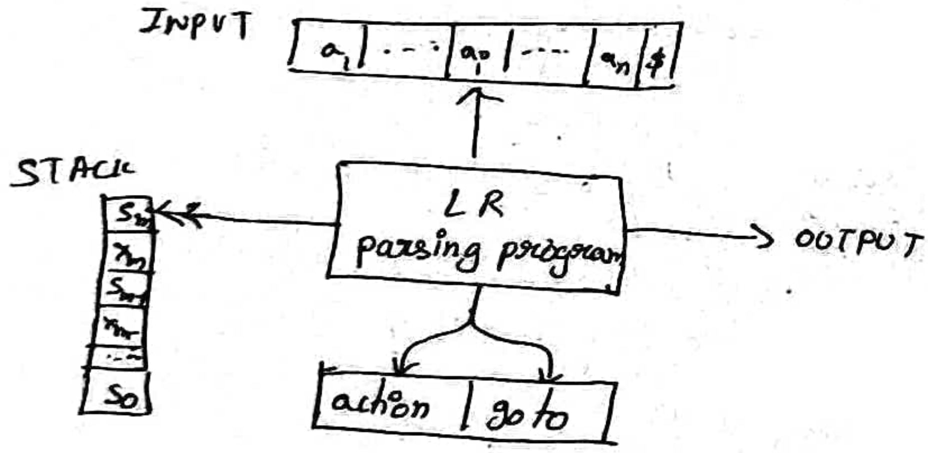
most powerful and most expensive.

3) Look-ahead LR (LALR):

Intermediate in power and cost b/w the other two. It will work on most programming-lang grammars, and, with some effort, can be implemented efficiently.

LR Parsing Algorithm:-

Model of an LR parser:



It consists of an i/p, o/p, a stack, a driver prog, and a parsing table that has two parts (action and goto).

The driver prog is same for all LR parsers, only the parsing table changes from one parser to another.

The parsing prog read characters from an i/p buffer one at a time. The prog uses a stack to store a string of the form $s_0 X_1 s_1 X_2 s_2 \dots X_m s_m$, where s_m is on top.

$X_i \rightarrow$ grammar symbol

$s_i \rightarrow$ state. which gives the information contained in the stack below it, and the comb of state symbol on top of the stack and the current i/p symbol are used to index the parsing table and determine shift-reduce parser decision.

The parsing table consists of two parts, a parsing action function action and a goto function goto. The prog driving the LR parser behaves as follows. It determines s_m, the state currently on top of the stack and a_i, the current i/p symbol. It then consults action $[s_m, a_i]$, which can have one of four values:

1) shift s, where s is the state
2) reduce by a grammar prod $A \rightarrow \beta$,
3) accept.
4) error

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

The configuration resulting after each of the four types of moves are as follows:

1. If action[s_m, a_i] = shift s, the parser executes a shift move, entering the configuration.

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m a_i s, a_{i+1} \cdots a_n \$)$$

Hence the both has shifted both the current i/p symbol a_i and the next state s, which is given in action[s_m, a_i] onto the stack, a_{i+1} becomes the i/p symbol.

2. If action [s_m, a_i] = reduce $A \to \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_{m-r} s_{m-r} A s, a_i, a_{i+1} \cdots a_n \$)$$

where s = goto [s_{m-r}, A]

$r \to$ length of β, right side production.

3) If action [s_m, a_i] = accept, parsing is completed

4) If action [s_m, a_i] = error, the parser has discovered an error and calls an error recovery routine.

Algorithm:

I/P: An i/p string w and an LR parsing table with function action and goto for a grammar G.

o/p: If w is in L(G), a bottom-up parser for w, otherwise an error indication.

method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and w\$ in the i/p buffer. The parser then executes the prog. 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 the 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
```

Ex :.

1) $E \rightarrow E + T$
2) $E \rightarrow T$
3) $T \rightarrow T * F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id.$

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			

The codes for the actions are
1) s_p means shift and stack state i,
2) r_j means reduce by production numbered j
3) acc means accept
4) blank means error.

Fig. Parsing table for expression grammar.

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

Construction of SLR parsing table:

A grammar for which an SLR parser can be constructed is said to be SLR grammar.

Augmented grammar:

For the given grammar G, we will introduce one new start symbol S', with new production S' -> S, as the grammar G'.

LR(0) items:

For each & every production we will ma of G with a dot at some position of the right side. Thus production A -> XYZ yields the

A -> ·XYZ

A -> X·YZ

A -> XY·Z

A -> XYZ.

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 added to closure(I).
2. If A -> α.Bβ is in closure(I) and B -> γ is a production then add the item B -> ·γ to I, if it is not already there. We apply this rule until no more productions new items can be added to closure(I)

Ex: consider the augmented expression grammar:

$E' \to E$

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$.

$E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$.

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

$E' \to \cdot E$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$.

Goto function:

GOTO(I, X) → I means closure (J). X means either terminal or non-terminal.

If any item $A \to \alpha \cdot X\beta$ is in closure (I) then $A \to \alpha X \cdot \beta$ is in goto (I, X)

Ex: GOTO (I, +) for $E \to E \cdot +T$

$E \to E + \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to (E)$

$F \to \cdot id$.

step 1:

$E' \to \cdot E$

$E \to \cdot E+T$

$E \to T$

$T \to T*F$, $T \to F$

$F \to (E)$

$F \to id$

I_0 : closure ($E' \to \cdot E$)

$E' \to \cdot E$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

I_1 : GoTo (I_0, E)

$E' \to E \cdot$

$E \to E \cdot + T$

I_2 : GoTo (I_0, T)

$E \to T \cdot$

$T \to T \cdot * F$

I_3 : GoTo (I_0, F)

$T \to F \cdot$

I_4 : GoTo (I_0, ( )

$F \to (\cdot E)$

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

I_5 : GoTo (I_0, id)

$F \to id \cdot$

I_6 : GoTo (I_1, +)

$E \to E + \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

I_7 : GoTo (I_2, *)

$T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

I_8 : GoTo (I_4, E)

$F \to (E \cdot)$

$E \to E \cdot + T$

I_9 : GoTo (I_6, T)

$E \to E+T \cdot$

$T \to T \cdot * F$

I_{10} : GoTo (I_7, F)

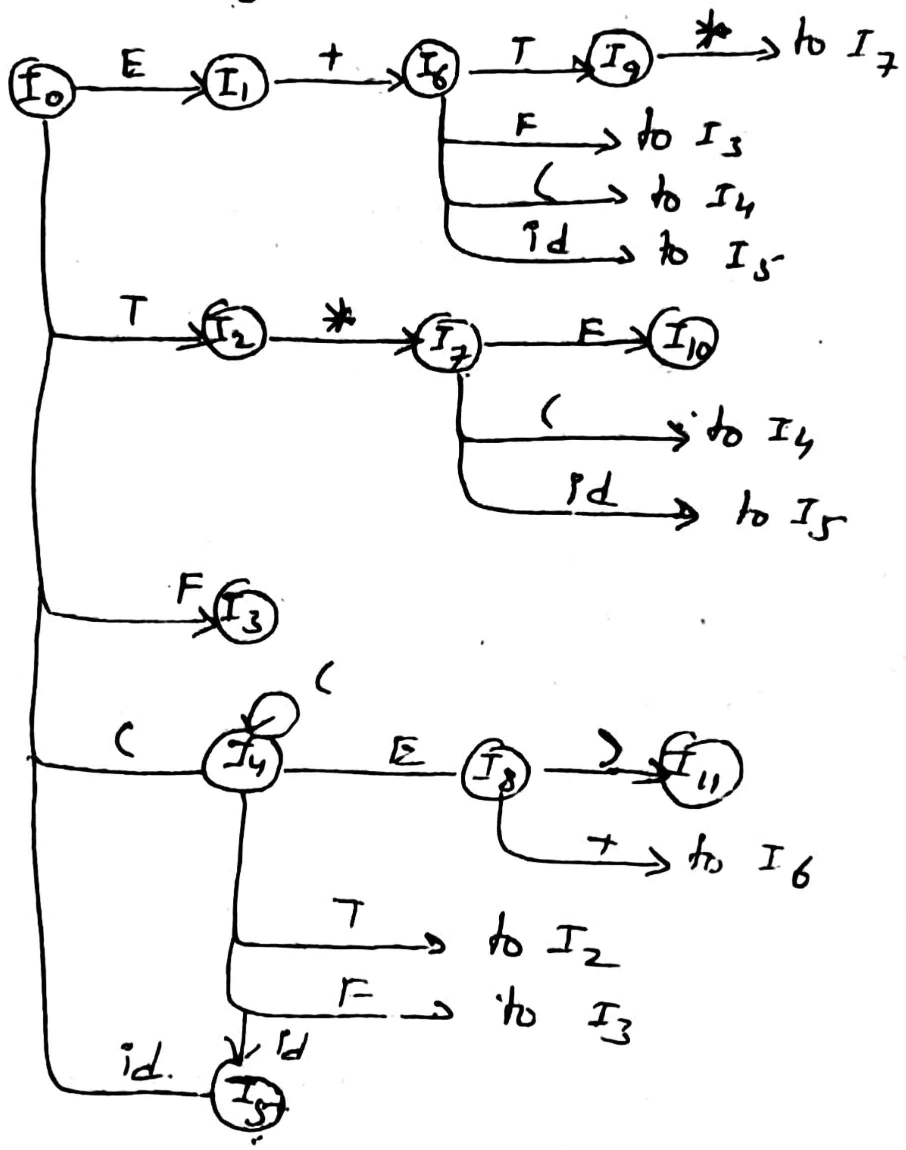
$T \to T*F \cdot$

I_{11} : GoTo (I_8,))

$F \to (E) \cdot$

$E \to E+T \cdot = r_1$
$E \to T \cdot = r_2$
$T \to T*F \cdot = r_3$
$T \to F \cdot = r_4$
$F \to (E) \cdot = r_5$
$F \to id \cdot = r_6$

Transition Diagrams for DFA



<u>Reduce:</u>

FOLLOW(E) = {+,), \$}

FOLLOW(T) = {+, *,), \$}

FOLLOW(F) = {+, *,), \$}

<u>Ex:</u> $I_2 \rightarrow E \rightarrow T \cdot$ → prod no = 2

∴ r_2 = {+,), \$} = follow (E)

I_3 = $T \rightarrow F$, prod no = 3 = FOLLOW(F)

r_3 = {+, *,), \$}.

Differences between LR and LL Parsers:

S.No	LR Parsers	LL Parsers
1.	These are bottom up parsers.	These are top down parsers.
2.	This is complex to implement.	This is simple to implement.
3.	LR Grammar is context-free-grammar that may not be free from ambiguity.	LL Grammar is context-free-grammar that is free from left-recursion and ambiguity.
4.	LR parser has 'k' lookahead symbol.	LL parser has only one lookahead symbol.
5.	LR parsers perform two actions ***shift*** and ***reduce*** to construct parsing table.	LL parsers performs two actions namely ***first()*** and ***follow()*** to construct parsing table.
6.	LR parsing is difficult to implement	LL parsers are easier to implement.
7.	These are efficient parsers.	These are less efficient parsers.
8.	It is applied to a large class of programming languages.	It is applied to small class of languages.