Bottom up passing:-

** Bottom-up passers are those which starts from the input string ie, from the leaves of the parse tree and move towards the Root. i.e., tries to get back the start symbol of the grammar.

** if the start symbol of the Grammar can be obtained, from the input string then the string is said to be accepted by the language.

** the input string is Reduced by the given productions of the grammar so that the start symbol is obtained

** Bottom-up passers attempt to find the Rightmost derivation in reverse for the given input string.

Ex:- Consider the Grammar

$S \rightarrow aABe$

$A \rightarrow ABc/b$

$B \rightarrow d$

Given the input string abbcde, which has to be checked for its acceptance by the language. using bottom-up passing the acceptance is as shown below

abbcde [∵ $A \rightarrow b$]

aAbcde [∵ $A \rightarrow Abc$]

aAde [∵ $B \rightarrow d$]

aABe [∵ $S \rightarrow aABe$]

S

Thus the start symbol 'S' is obtained, which shows the acceptance of the string abbcde by the Grammar.

** This is the Reverse of the Rightmost derivation, where the Right-most derivation for the same is shown below.

$S \Rightarrow aABe$
$\Rightarrow aAde$ [∵ $B \to d$]
$\Rightarrow aAbcde$ [∵ $A \to Abc$]
$\Rightarrow abbcde$ [∵ $A \to b$]

Handles: "Handle" is a substring of a string that matches the Right side of a production, which when Reduced to a non-terminal on the left side of the production Represents one step of the Reverse Rightmost derivation.

If There is a production $A \to \beta$, 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".

Ex:- Consider the Grammar.

$E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$

the Rightmost derivation is given by

$E \Rightarrow \underline{E+E}$

$\Rightarrow E+\underline{E*E}$

$\Rightarrow E+E*\underline{id_3}$

$\Rightarrow E+\underline{id_1}*id_3$

$\Rightarrow id_1+id_2*id_3$.

The id's are subscripted for our convenience.

$E \rightarrow E+E \mid E*E \mid E-E \mid id$ (3)

(String $id+id*id$)

Handle pruning:

** Handle pruning is nothing but Reducing the handle, by the non-terminal, which is towards the left of the production.

** A Rightmost derivation in Reverse can be obtained by "handle pruning".

Two points are to be concerned when we are passing by handle pruning They are,

1, To locate the substring to be Reduced in a Right-Sentential form;

2, To determine what production to choose if there is more than one production with the substring on the R.H.S.

Right Sentential form	Handle	production
$id+id+id$	id	$E \rightarrow id$
$E+id+id$	id	$E \rightarrow id$
$E+E+id$	id	$E \rightarrow id$
$E+E+E$	$E+E$	$E \rightarrow E+E$
$E+E$	$E+E$	$E \rightarrow E+E$
E		

Handle pruning: - to find the substring of R.H.S that could be Reduced by appropriate non-terminal. such a substring is called 'handle'.

In otherwords 'handle' is a string of substring that matches the Right side of production and we can Reduce such string by a non-terminal on left hand side production. such Reduction Represents one step along the Reverse of R.m.d.

* Thus bottom parser is essentially a process of detecting handles and using them in Reduction. This process is called handle pruning.

Shift Reduce parsing:-

** Shift-reduce parser is a bottom-up parser, which attempts to construct a parse tree for an input string beginning at the leaves and working up towards the Root.

** The process is to Reduce the string 'w' to the start symbol of the grammar.

** At each step of Reduction, a particular substring matching the Right side of a production is Replaced by the symbol on the left of that production.

The Reductions made by the shift-Reduce parser is shown below

Consider the grammar.

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

Input string
$w = id_1+id_2*id_3$

Reductions made by shift-Reduce parser:

Right-sentential form	Handle	Reducing Production
$id_1+id_2*id_3$	id_1	$E \rightarrow id$
$E+id_2*id_3$	id_2	$E \rightarrow id$
$E+E*id_3$	id_3	$E \rightarrow id$
$E+E*E$	$E*E$	$E \rightarrow E*E$
$E+E$	$E+E$	$E \rightarrow E+E$
E		

- there are 4 possible actions a shift-Reduce parser can make.

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

Conflicts during shift-Reduce parsing:

when the S.R.P is applied to some CFG it leads to some conflicts because shift-Reduce Parser can not be used for CFG. The conflicts are.

- **Shift/Reduce conflict:** the parser even after knowing the entire stack contents and the next i/p symbol, can not decide whether to shift (or) Reduce.
- **Reduce/Reduce conflict:** the parser knowing the entire stack contents and next input symbol, can not decide which productions to use (or) which of several reductions to make.

Ex:- consider an dangling-else grammar.

```
Stmt → If expr then stmt
     | if expr then stmt else stmt
     | other
```

where other stands for any other statements.

when shift-Reduce parser is applied to this grammer, we Reach the configuration.

STACK	Input
... if exp$ then stmt	else ... $

under this configuration, we can not tell whether "if exp$ then stmt" is the 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, bcz, it is possible to Reduce "if exp$ then stmt" on the stack to "stmt" coz it is also possible to shift "else" and then look for another "stmt to complete the alternative "if exp$ then stmt else stmt" which can be Reduced to "stmt". Thus, we can not tell whether to shift(or) Reduce.

Another conflict occurs when we know we have a handle, but the stack contents and next i/p symbol are not sufficient to determine which Production should be used in a Reduction.

Example:

1, S → 0S1 | 01, indicate the handle in each of the Following Right-sentential form.

a, 000111 b, 00S11

2, S → SS+ | SS* | a

a, SSS+a*+, b, SSa*a+, c, aaa*a++.

Example: Consider the following grammer

1) S → TL;
T → int | float
L → L, id | id

Parse the input string int id,id; using shift-Reduce parser.

2) S → (L) | a
L → L, S | S , w = (a, (a, a))

3) S → 0S0 | 1S1 | 2 w = 10201

17/9/18 - E_3 - Sec-4

5, 7, 11, 27, 30, 31, 36, 46, 47, 50, 54, 57, 58, 59, 61, 62, 63

17/9/18 - E_3 - Sec-5

4, 11, 15, 16, 27, 30, 32, 33, 46, 47, 48, 53, 57, 59, 61, 62, 63.

Operator Precedence Parser:-

** A grammar G is said to be operator precedence if it posses following properties.

1, No production on the Right side is ε. (i.e, A→ε)

2, there should not be any production rule possessing TWO adjacent non-terminals at the Right hand side. (i.e, S→ABC)

Consider the grammar for arithmetic expressions.

E → EAE / (E) / -E / id

A → + | - | * | / | ^

This grammar is not an operator precedent grammar as in the production Rule. E → EAE

It contains two consecutive non-terminals. Hence first we will convert it into equivalent operator precedence grammar by removing A.

E → E+E | E-E | E*E | E/E | E^E

E → (E) | -E | id

** In operator predence parsing we will first define precedence Relations <· ≐ and ·> between pair of terminals. the meaning of these relations is.

P <· Q	P gives more precedence than Q
P = Q	P has same precedence as Q
P ·> Q	P takes precedence over Q.

Now by considering the precedence relation b/w the arithmetic operators we will construct the operator precedence table. The operators precedences we have considered are id, +, *, $.

precedence Relation Table

	id	+	*	$
id	-	.>	.>	.>
+	<.	.>	.>	.>
*	<.	.>	.>	.>
$	<.	<.	<.	-

Now Consider the string.

id+id*id.

Advantages:

This type of parsing is simple to implement.

Disadvantages:

1. The operator like minus has two different precedence (unary and binary). Hence it is hard to handle tokens like minus sign.
2. This kind of parsing is applicable to only small class of grammars.
3. This is not an efficient parser.

Constructing SLR - parser:

* The SLR method begins with LR(0) items and LR(0) automata, we augmented G to produce G', with a new start symbol S'.
* From G', we construct C, the canonical collection of sets of items for G' together with the GOTO function.
* The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm. It requires us to know FOLLOW(A) for each nonterminal A of a grammar.

Algorithm for 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 Result from 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 $[S' \to S \cdot]$.

* the parsing table consisting of the ACTION and GOTO functions determined by Algorithm is called the SLR(1) table for G.

* An LR parser using the SLR(1) table for G is called the SLR(1) parser for G, and a grammar having an SLR(1) parsing Table is said to be SLR(1). We usually omit the "(1)" after the "SLR", since we shall not deal here with parsers having more than one symbol of lookahead.

* Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions.

$S \to L = R \mid R$

$L \to *R \mid id$

$R \to L$

—the canonical collection of sets of LR(0) items for a grammar

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

I_1 : $S' \to S.$

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

I_3 : $S \to R.$

I_4 : $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$

I_5 : $L \to id.$

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

I_7 : $L \to *R.$

I_8 : $R \to L.$

I_9 : $S \to L = R.$

* Consider the set of items I_2. The first item in this set makes ACTION[2, =] be "shift 6". Since FOLLOW(R) contains = (to see why, consider the derivation $S \Rightarrow L = R \Rightarrow *R = R$), the second item sets ACTION[2, =] to "Reduce $R \to L$."

Since there is both a shift and Reduce entry in ACTION [2, =], State 2 has a shift/Reduce conflict on input symbol =.

STATE	ACTION			GOTO		
	*	=	id	S	L	R
I_0						
I_1						
I_2						
I_3						
I_4						
I_5						
I_6						
I_7						
I_8						
I_9						

* above Grammar is not ambiguous, this Shift/Reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to ~~Remember~~ Remember enough left context to decide what action the parser should take on input =, having seen a string Reducible to L.

Viable prefixes:

* the prefixes of Right sentential forms that can appear on the stack of a shift Reduce parser are called viable prefixes.

* a viable prefix is a prefix of a Right~~most~~ sentential form that does not continue past the Right end of the Right most handle of that sentential form. By this definition, it is always possible to add terminal symbols to end of a viable prefix to obtain a Right-sentential form.

* the stack containing must be a prefix of a Right-sentential form. if the stack holds α and the Rest of the Input is x, then a sequence of Reductions will take αx to S. in terms of derivation, $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha x$.

Not all prefixes of Right-sentential forms can appear on the stack, however, since the parser must not shift past handle.

$$E \Rightarrow E * id \Rightarrow E * id$$

Then, at various times during the parse, the stack will hold (, (E, and (E), but it must not hold (E) *, since (E) is a handle, which the parser must Reduce to F before shifting *.

Importance: The entire SLR parsing algorithm based on the idea that the LR(0) automaton can Recognise Viable prefixes and Reduce them appropriately.

$E \Rightarrow T \Rightarrow T*F \Rightarrow T*id \Rightarrow F*id \Rightarrow id*id$.

(id* → it can not be viable prefix)

25/09/18 - $\mathcal{E}_3$ - sec-4

3, 10, 12, 15, 31, 42, 47, 55, 56,

25/09/18 - $\mathcal{E}_3$ - sec-3

5, 14, 16, 17, 19, 22, 24, 27, 28, 30, 34, 37, 42, 43, 44, 48, 53, 55, 59, 60, 61, 63.

LR Parsing:-

** . LR parsers are bottom-up parsers.

** This is the most efficient method of bottom-up parsing which can be used to parse the large class of CFG.

** This method is also called LR(k) parsing. Here

L — left to Right scanning

R — Rightmost derivation in Reverse

K - is number of I/p symbols of lookahead that are used in making parsing decisions.

when K is omitted, K is assumed to be 1.
(not specified)

LR parsers are advantageous for various Reasons:

* The LR parsing method is the most general non-backtracking shift-Reduce parsing method.

* 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 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.

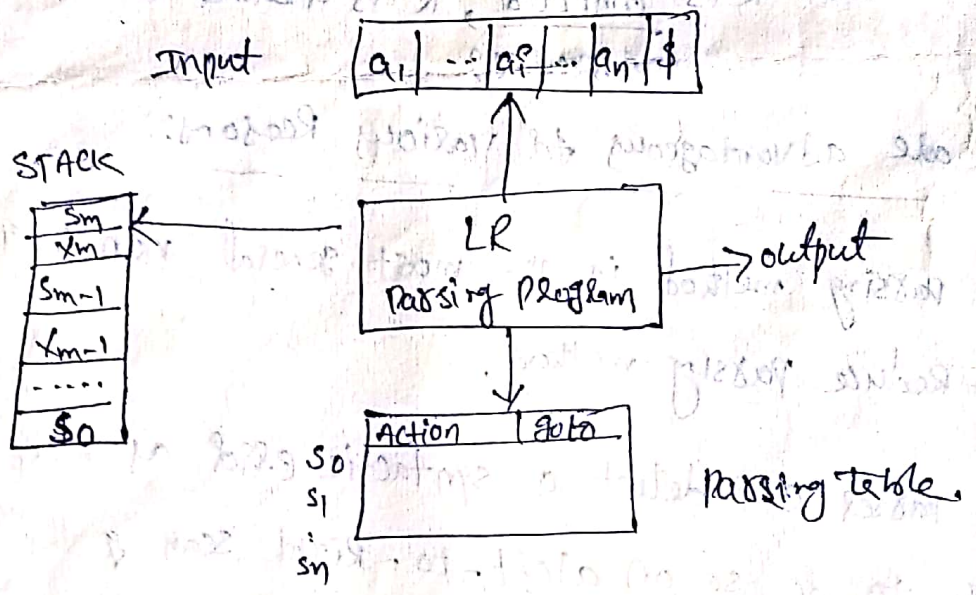
LR parsers have some limitations:-

* It is complex to construct an LR parser by hand for a programming language grammar.

model of an LR parser (or) LR Parsing Algorithm:-

The LR-parsing method consists of An Input, An output, stack, parsing program, and parsing table.

The parsing table comprised of two parts i.e., action and goto.



→ 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 $S_0X_1S_1X_2 \cdots X_mS_m$, where S_m is on top. Each X_i is a Grammar symbol and each S_i is a symbol called a state.

* The parsing table consists of two parts, a parsing action function called action and a goto function called goto.

* the combination of the state symbol on top of the stack and current input symbol are used to index the parsing table and determine the shift-reduce parsing actions.

(or)

Parsing program works on following line.

1. It initialises the stack with start symbol and invokes scanner (lexical analyser) to get next token.
2. it determines $\frac{Sm}{Sg}$ the state currently on the top of the stack and a_i the current input symbol.
3. It then consults action $[s_m, a_i]$, the parsing action table entry for state s_m and input a_i, which has the one of four values.

   1) shift s, where s is a state
   2) Reduce by a grammar production $A \rightarrow \beta$.
   3) Accept
   4) Error (syntatic error).

Items and the LR(0) Automaton:—

* How does a shift-Reduce parser know when to shift and when to Reduce

with stack contents $T and next i/p symbol *. How does the parser know that T on the top of the stack is not a handle, so the appropriate action is to shift and not to Reduce T to E.

An LR parser makes shift-Reduce decisions by maintaining states to keep track of where we are in a parse.

States represent sets of "items". An LR(0) item of a Grammar G is a production of G with a dot at some position of the body. The production $A \to XYZ$ yields the 4 items

$A \to \cdot XYZ$ — indicates that we hope to see a string derivable from XYZ next on the i/p

$A \to X \cdot YZ$ → that we have just seen on the i/p a string derivable from X and that we hope to see a string derivable from YZ.

$A \to XY \cdot Z$

$A \to XYZ \cdot$ → that we have seen the body XYZ and that it may be time to Reduce XYZ to A.

The production $A \to \varepsilon$ generates only one item, $A \to \cdot$.

One collection of sets of LR(0) items, called canonical LR(0) collection provides the basis for constructing D.F.A, that is used to make parsing decisions. Such an automaton is called LR(0) automaton.

* To construct the canonical 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 acceptance of the input. Acceptance occurs when and only when the parser is about to Reduce by S' → S.

Closure of Item sets:

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

1. Initially, add every item in I 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 CLOSURE(I), if it is not already there. Apply this Rule until no more new items can be added to CLOSURE(I).

ie, $A \to \alpha \cdot B\beta$ in CLOSURE(I) indicates that, at some point in the parsing process, we think we might next see a substring derivable from $B\beta$ as input.

the substring derivable from $B\beta$ will have a prefix derivable from B by applying one of the B-productions.

Kernel items: the initial item, $S' \to \cdot S$, and all items whose dots are not at the left end.

Non-kernel items: all items with their dots at the left end, except for $S' \to S$.

Function GOTO:

Function is 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 $[A \to aX \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I.

* GOTO function is used to define the transitions in the LR(0) automaton for a grammar. the states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for under input X.