___

CD: COMPILER DESIGN

INSTRUCTOR: Ms. SHWETA TIWARI
shwetatiwari08@ recabn.ac.in
shwetatiwari08aug@ gmail.com
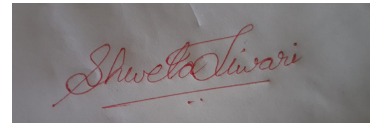
September 2022

TOPIC On : UNIT-2 BOTTOM-UP PARSERS

___

By SHWETA TIWARI

Under On: Basic Parsing Techniques

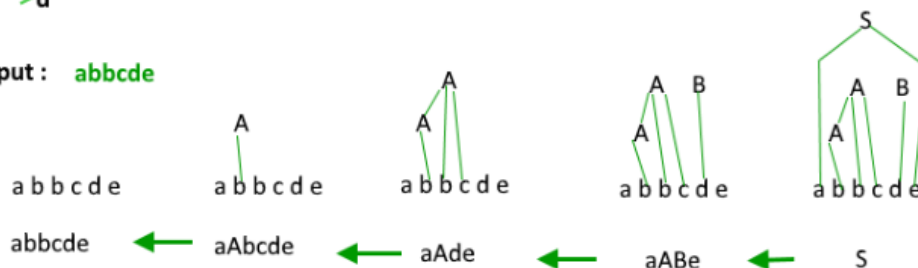# *TOPIC On :* *UNIT-2 BOTTOM-UP PARSERS*

II.   BOTTOM-UP PARSER

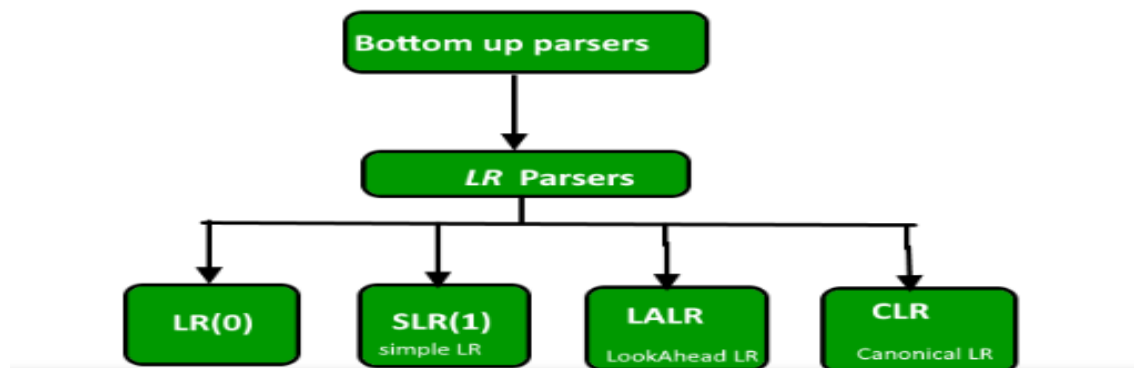**Bottom-Up Parsing:**

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.

Classification of bottom up parsers

Example:

Consider the grammar:

S → aABe A → Abc | b B → d

The string to be recognized is abbcde. We want to reduce the string to S.

Steps of reduction:

abbcde   (b,d can be reduced)

aAbcde   (leftmost b is reduced)

aAde   (now Abc,b,d qualified for reduction)

aABe   (d can be reduced) S

Each replacement of the right side of a production by the left side in the above example is called reduction, which is equivalent to rightmost derivation in reverse.

$$S \xrightarrow{rmd} aABe$$
$$\xrightarrow{rmd} aAde$$
$$\xrightarrow{rmd} aAbcde$$
$$\xrightarrow{rmd} abbcde$$

Handle:

A substring which is the right side of a production such that replacement of that substring by the production left side leads eventually to a reduction to the start symbol, by the reverse of a rightmost derivation is called a handle.

The **handle** is the substring that matches the body of a production whose reduction represents one step along with the reverse of a Rightmost derivation.
The handle of the right sequential form Y is the production of Y where the string S may be found and replaced by A to produce the previous right sequential form in RMD(Right Most Derivation) of Y.

Sentential form: S => a here, 'a' is called sentential form, 'a' can be a mix of terminals and nonterminals.
Example:
Consider Grammar : S -> aSa | bSb | ε

Derivation: S => aSa => abSba => abbSbba => abbbba

Left Sentential and Right Sentential Form:
●   A left-sentential form is a sentential form that occurs in the *leftmost derivation of some sentence.*

- A right-sentential form is a sentential form that occurs in the *rightmost derivation of some sentence.*
  Handle contains two things:

- Production

- Position

Example:

```
S  ->  aABe
A  ->  Abc | b
B  ->  d
```

Steps:

```
abbcde  :  γ = abbcde , A->b; Handle = b
aAbcde  :  γ = RHS = aAbcde , A->Abc; Handle = Abc
aAde  :  γ = aAde , B->d; Handle = d
aABe  :  γ = aABe, S-> aABe; Handle = aABe
```

Note- *Handles are underlined in the right-sentential forms.*
Is the leftmost substring always handled?

No, choosing the leftmost substring as the handle always, may not give correct SR(Shift-Reduce) Parsing.

Handle Pruning:

Removing the children of the left-hand side non-terminal from the parse tree is called **Handle Pruning**.
A rightmost derivation in reverse can be obtained by handle pruning.

Steps to Follow:
- Start with a string of terminals 'w' that is to be parsed.
- Let w = γ_n, where γ_n is the nth right sequential form of an unknown RMD.
- To reconstruct the RMD in reverse, locate the handle βn in γ_n. Replace βn with LHS of some $A_n \dashrightarrow$ βn to get $(n-1)^{th}$ RSF γ_{n-1}. Repeat.

Example:
S -> aABe

A -> Abc | b

B -> d

Right Sentential Form	Handle	Reducing Production
abbcde	b	A->b
aAbcde	Abc	A->Abc
aAde	d	B->d
aABe	aABe	S->aABe
S(Root)		

Table: Reductions made by Parser

Shift-Reduce Parsing

A general type of bottom-up parser is a shift-reduce parser.

Shift-Reduce Parsing:

A general shift-reduce parsing is LR parsing. The L stands for scanning the input from left to right and R stands for constructing a rightmost derivation in reverse. Benefits of LR parsing:

1. Many programming languages use some variations of an LR parser. It should be noted that C++ and Perl are exceptions to this.
2. LR Parser can be implemented very efficiently.
3. Of all the Parsers that scan their symbols from left to right, LR Parsers detect syntactic errors, as soon as possible.

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

Shift-reduce parsing is a type of bottom -up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step**: When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

Stack Implementation of Shift-Reduce Parsing:

There are two problems that must be solved if we are to parse by handle pruning. The first is to locate the substring to be reduced in a right-sentential form, and the second is to determine what production to choose in case there is more than one production with that substring on the right side.

A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed. We use **$** to mark the bottom of the stack and also 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 is on top of the stack. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK   INPUT

$ S $

Example: The actions a shift-reduce parser in parsing the input string **abbcde**, according to the grammar

S -> aABe

A -> Abc | b

B -> d

Stack	Input	Action
$	abbcde$	shift
$a	bbcde$	shift
$ab	bcde$	reduce(A->b)
$aA	bcde$	shift
$aAb	cde$	shift
$aAbc	de$	reduce(A->Abc)
$aA	de$	shift
$aAd	e$	reduce(B->d)
$aAB	e$	shift
$aABe	$	reduce(S->aABe)
$S	$	pop
$	$	accept

Table: Configuration of Shift Reduce Parser on input **abbcde**

Example: The actions a shift-reduce parser in parsing the input string $id_1+id_2*id_3$, according to the ambiguous grammar for arithmetic expression.

Production:

E->E+E/E*E

E->E+E/id

	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 1 Configuration of Shift Reduce Parser on input id1+id2*id3

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		

Fig 2 Reductions made by Shift Reduce Parser

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, and (4) error.

- In a **shift** action, the next input symbol is shifted onto the top of the stack.
- In a **reduce** action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.
- In an **accept** action, the parser announces successful completion of parsing.
- In an **error** action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Figure3 represents the stack implementation of shift reduce parser using unambiguous grammar.
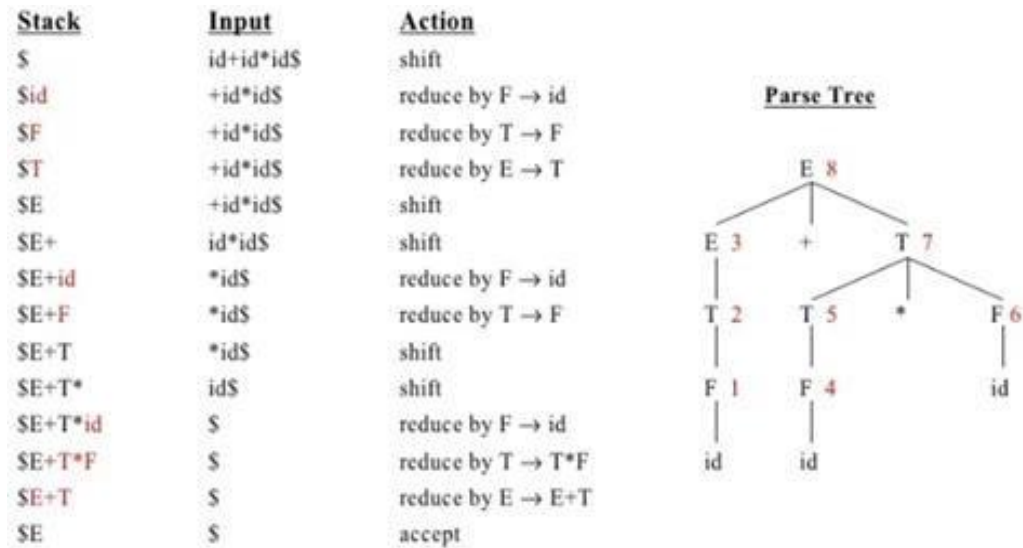
Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

Parse Tree

Fig 3 A stack implementation of a Shift-Reduce parse

************************************THE END*************************