gradeup

Syntax Analysis and Semantic Analysis Part -2

Syntax Analysis

Content:

Syntax Analysis : It is the next phase after lexical analysis in compiler design. It is a major of the front end of a compiler. The basic concepts and definition such as context free grammar, parse tree and non- ambiguous grammars are crucial to the parsing.

Role Of The Parser:

It plays an important role in the compiler design. It performs the following tasks.

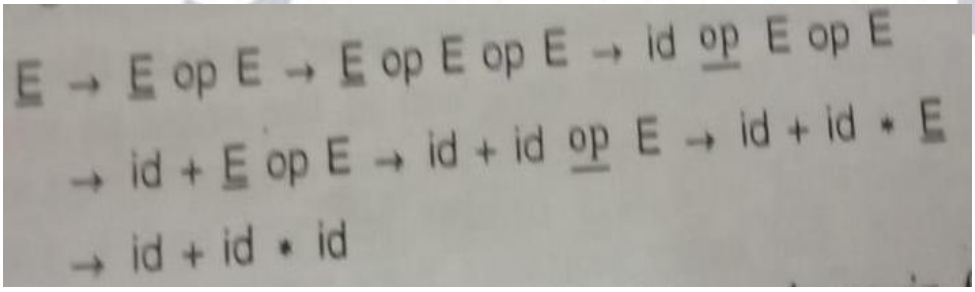
- It obtains a string of tokens from the lexical analyzer.
- It groups the tokens appearing in the input in order to identify larger structures in the program. This process is done to verify that the string can be generated by the grammar for the source language.
- It should report any syntax error in the program.
- It should also recover from the errors so that it can continue to process the rest of the input.

Basic Issues In Parsing:

- The specification of syntax of a programming language. This specification should be precise and unambiguous. It must cover all the syntactic details of the language. The syntax of programming language constructs can be described using context free grammar which we will introduce later.
- The representation of the input after it has been successfully parsed. This information is important as it is required by the subsequent phases of the compiler. Parse tree may be used to represent the parsed tree.
- The parsing algorithm. There are parsing algorithms which have been developed and standardized over the years. These algorithms fall in two categories top-down and bottom-up.

There are special tools available which can automatically generate a parser for a given grammar. YACC is a compiler generation tool available on UNIX which generates a bottom up parser. BISON is another such tool available in the GNU public domain software. Llama generate a top-down LL(1) parser and Occs generates a bottom-up LALR(1) parser.

Context Free Grammar:



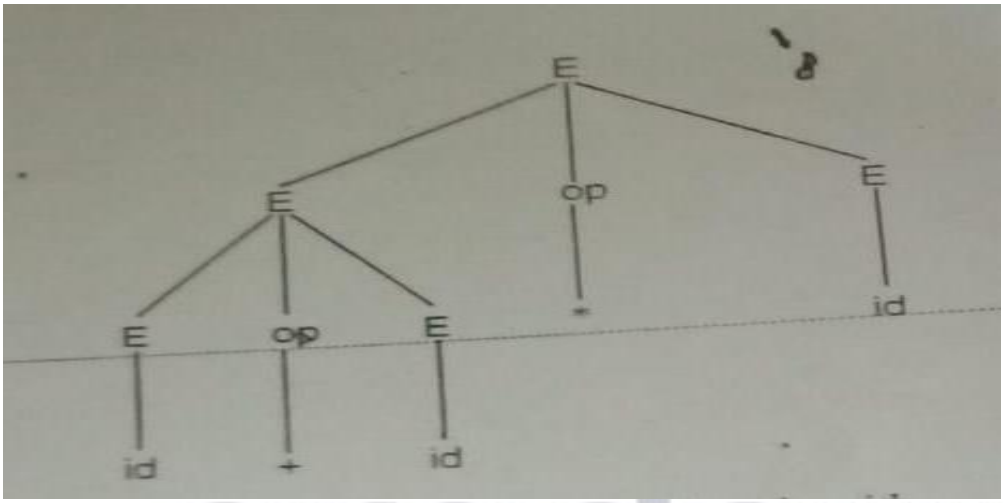
The syntax of a language may be specified using a notation called context free language. A context free grammar consists of terminals, non-terminals, a start symbol and production rules.

Derivation and Parse Trees:

A parse tree is an equivalent form of showing a derivation which represents a derivation graphically or pictorially.

Example: Let us draw the parse tree for the string id + id * id using the CFG. The leftmost derivation for the given string is as follows

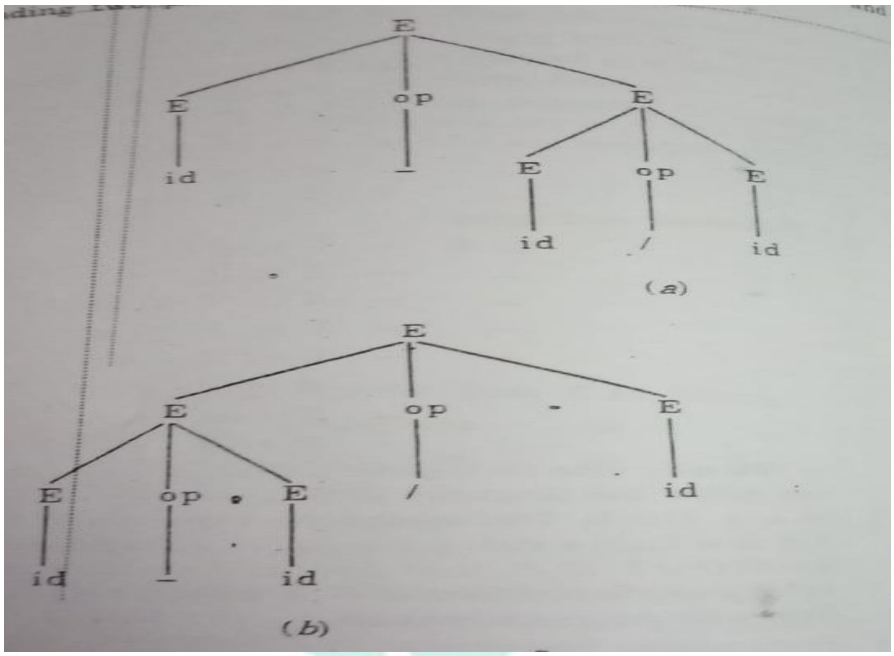
The parse tree for id + id * id



Ambiguity :

It is quite possible that a context free grammar G may produce more than one parse tree for some sentence. Such grammar is said to be ambiguous. A grammar is ambiguous when the same non-terminal appears twice on a right hand side.

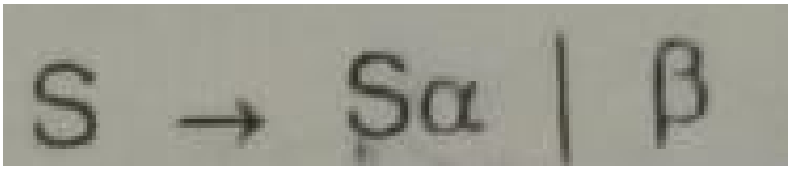
Example: Let us consider the grammar again. The sentence id – id/id has two different leftmost derivations and corresponding two parse trees



Eliminating Left Recursion:

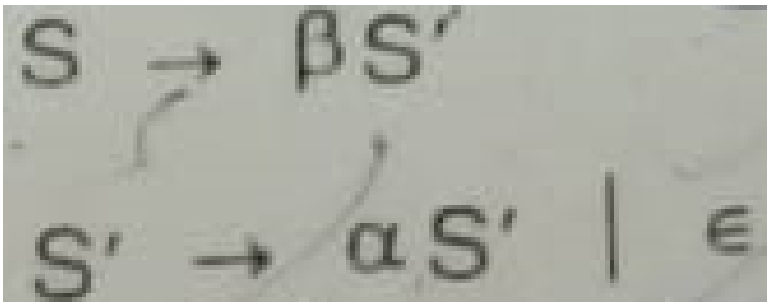
There are many syntactic features of a programming language that are expressed using recursive rules. The recursion involved may be left recursion or right recursion. Though both forms are equivalent in expressiveness, left recursive grammars are not suitable for top-down parsers.

A grammar is left recursive if the first symbol in the right hand side of a rule is the same non-terminal as that in the left hand side. We can also say that a grammar is left recursive if there exists a non-terminal S such that S=> S



Fortunately, we can always eliminate left recursion from a grammar by applying certain transformations in such a way that the translated grammar recognizes the same input strings. Consider a production which is self left recursive as follows

We can make it non-recursive by rewriting the production as:



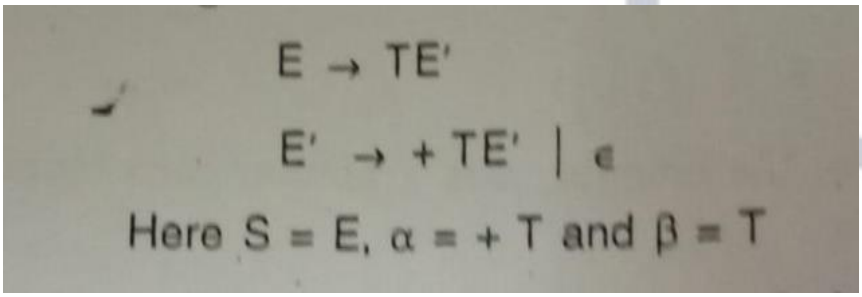
We can apply this translation rule to the following production

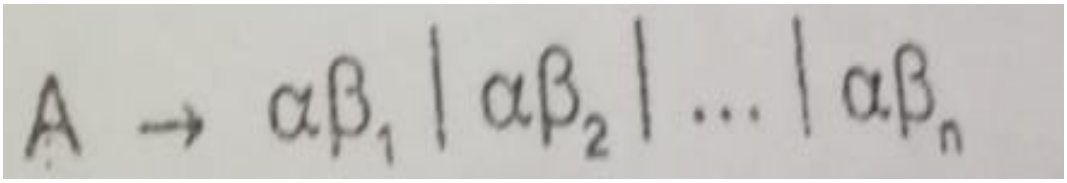
E→ E + T|T

To get

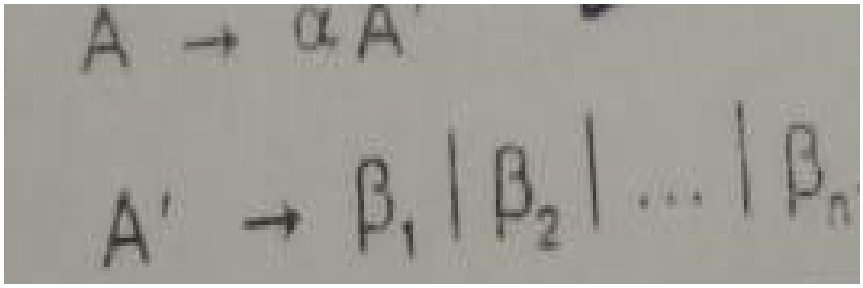
Eliminating Left Factoring:

Left factoring is a process which isolates the common parts of two productions into a single production. Any single production of the form





Can be represented by



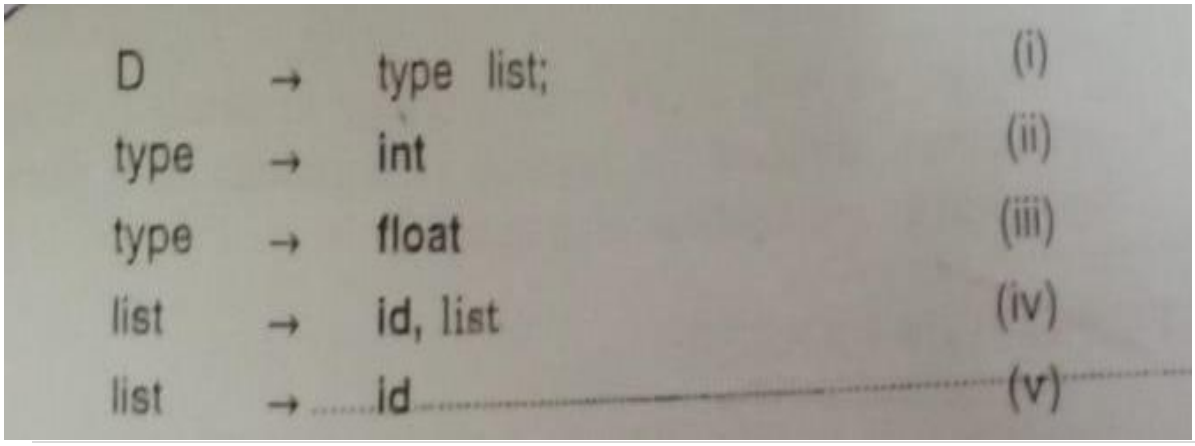
Parsing: It scans an input token stream, from left to right, and groups tokens in order nto check the syntax of an input string by identifying a derivation by using the production rules of the grammar. We can define a parser formally as a program p, for a context free grammar G, which takes a string w as input and outputs

1. A parse tree for w is a sentence of G i.e.
2. Or an error massage if w is not a sentence. It may also provide information about the type of error and the location of the error.

There are two basic approaches to parsing. Each corresponds to one of the two possible ways in which a parse tree can be constructed.

1. Top-down parsers create the parse tree from the root and expand it till all the leaves are reached.
2. Bottom-up parsers create the parse tree from leaves upwards to the root.

Top-Down Parsers:

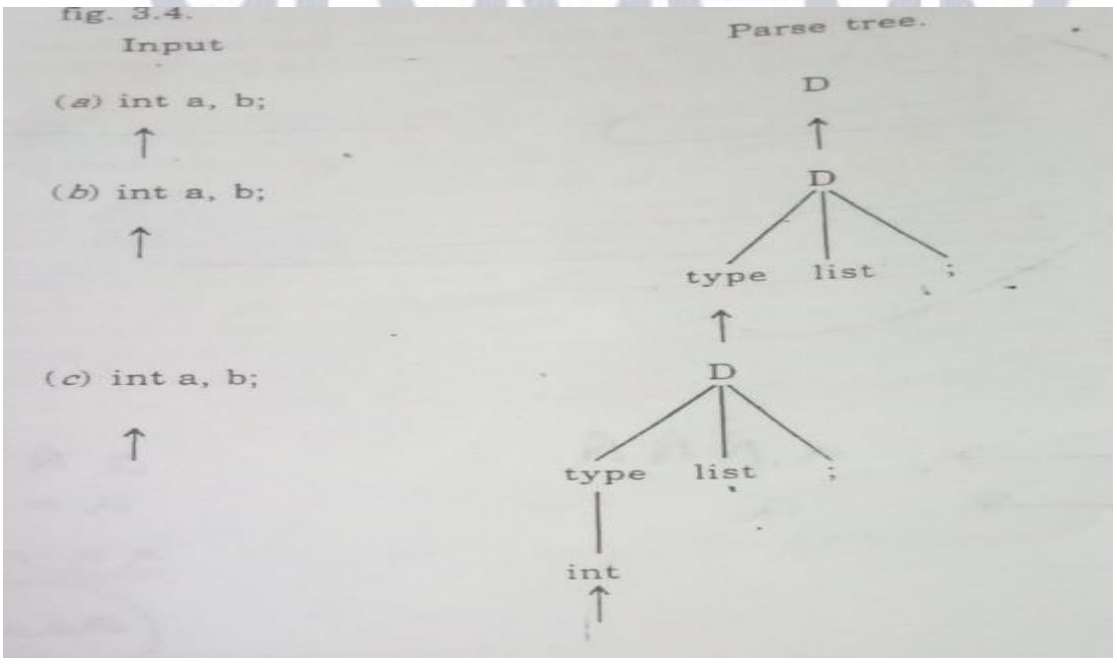


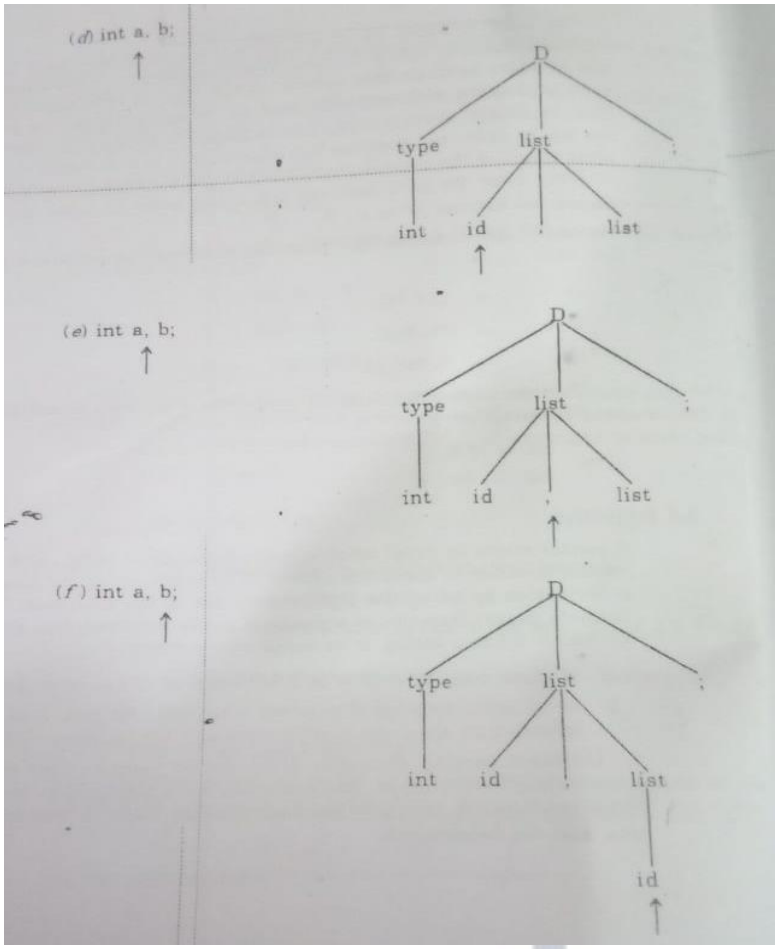
It may be considered as an attempt to construct a parse tree starting from the root and create the nodes of the parse tree in preorder. This means it starts the derivation from the start symbol of the grammar. It is desirable to construct a derivation that produces terminals in the left to right fashion in the sentential forms as the input tokens are scanned from left to right.

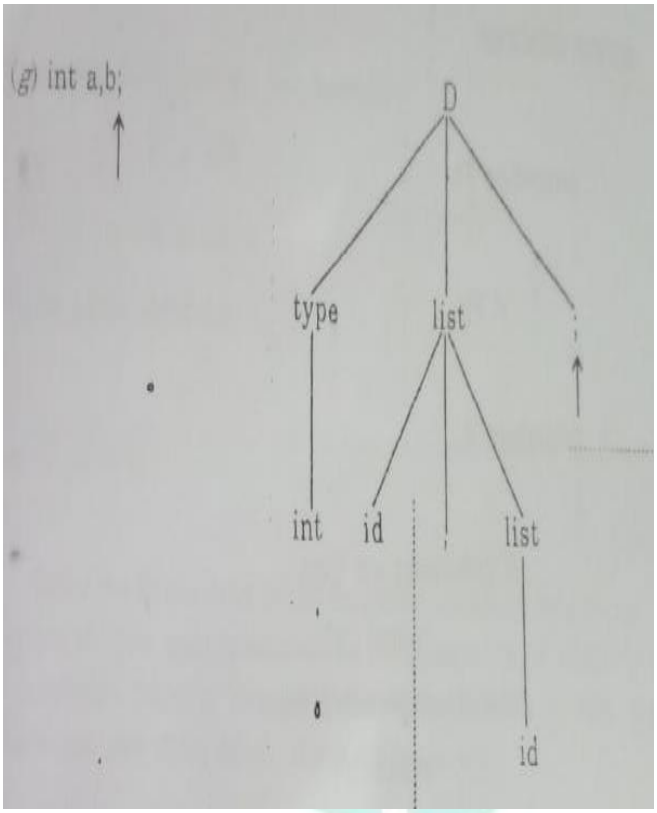
It starts constructing the left-most derivation from the start symbol. If the leftmost non-terminal has more than one production rule, a selection may be made depending on whether backtracking is permitted or not. If back-tracking is permitted, parser can make repeated scans of the input.

Example: Consider the following context free grammar which generates a subset of the types used in C language.

Let int a,b; be the sentence to be parsed using a top down parser. The parser would start construction of a parse tree by the root labeled with the starting non-terminal D and repeatedly selecting one of the production for the non-terminal node n and constructing children at n for the right side of the production. It then finds the next node at which a sub tree is to be constructed.





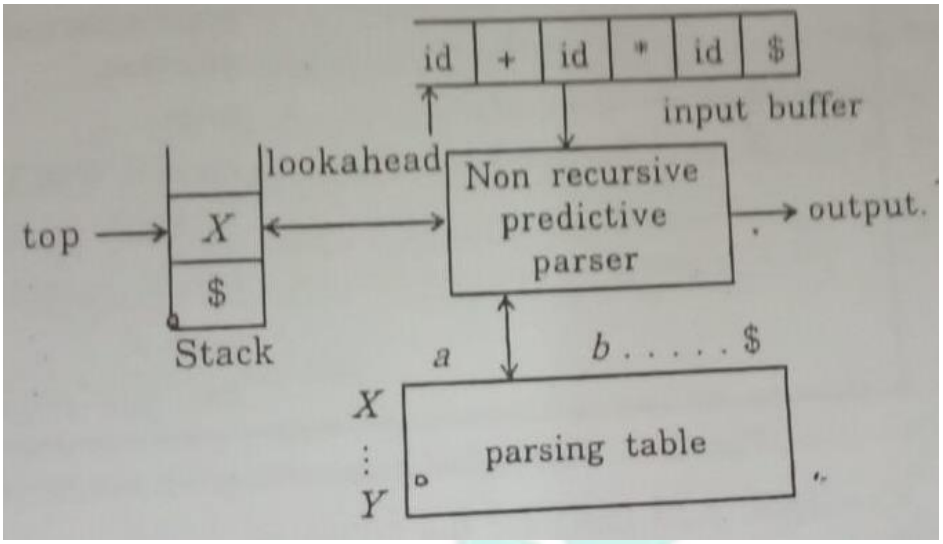


The current token being scanned is referred to as the look ahead symbol.

Recursive Descent Parser: It is a top down method of syntax analysis which uses a collection of recursive procedures to process the input. A procedure is associated with each non-terminal of the grammar.

Predictive Parsing: A special form of recursive descent parsing called predictive parsing in which the look ahead symbol unambiguously determines the procedure selected for each non-terminal and no back-tracking occurs. A parse tree for the input is implicitly defined by the sequence of procedures called in processing the input.

Non Predictive Parser: It uses a parsing table which indicates which alternative of the rule to be used for making the prediction for a given non-terminal and the first terminal symbol which that a non-terminal is to produce. The parser uses its own stack to store those non-terminal in the present sentential form for which no prediction has been made



yet.

The parser acts on the basis of two symbols. First is X, the top symbol of the stack and t, the current look ahead symbol. The various possibilities are

1. If the stack is empty and the look ahead symbol is also $, the parser halts. Thus X = t = $ is the condition for successful parsing.

2. If the symbol at the top of the stack is a terminal and is same as the look ahead symbol i.e. X = t= $, the parser pops X and advances the look ahead pointer so that next input symbol becomes the current look ahead symbol.

3. If the symbol at the top of the stack is a non-terminal, the parser refers to the entry table[X,t]. If the entry is a production X ABC, the parser replaces X on top of the stack with the right hand side of the production in the reverse order.

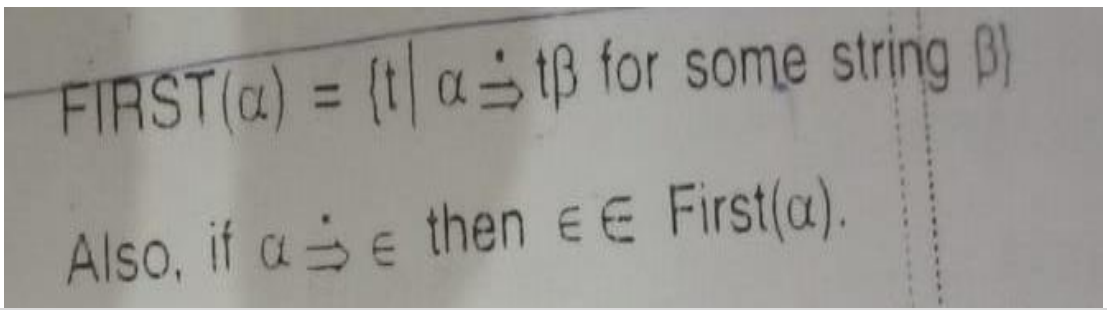
top of stack	Lookahead symbol	parser action
$	$	parsing successful, halt.
t	t	pop t, advance lookahead point continue.
t	r	error.
X	t	consult TABLE [X, t]
		if TABLE [X, t] = X → ABC
		pop X; push C; push B;
		push A;
		else give error message.

Construction of Predictive parsing table: For the construction of parsing table we must know the terminal symbols for which a particular rule should be selected.

First(α) may be defined as the collection of terminal symbols that began the strings derived from α i.e.

We can use the following rules to find the first (X)for all grammar symbols X until no more terminal can be added to it.

1. For a terminal t, first(t) ={t}.
2. For a non-terminal X, if X ⟶ ∃ then add ∃ to first(x)
3. → If X $A_{1'}$, A_2'..... A_n first(x)= first (x) U



Another function called follow, is used to trap the ∃productions. Follow (x) is defined to be a set of terminal symbols such that any of these terminals can appear immediately to the right of A in some sentential form.

LL(1) Grammar: LL(1) parsers for which such parsers can be constructed are known as LL(1) grammar. The first L stands for left to right scanning of input. The second L indicates that it produces leftmost derivation and 1n is the number of look ahead symbols.

The distinctive properties of LL(1) grammar are as follows:

1. No ambiguous of left recursive grammar can be LL(1).
2. All the entries in the parsing table are unique for an LL(1)grammar.
3. A grammar G is LL(1) if and only if the following conditions hold for two distinct productions A → α| **β**

a) First (α) ∩ first(β**)** ≠{a} where a is some terminal symbol of the grammar

b) First (α) ∩ first(β**)** ≠ ∃ i.e at most one of α and β can derive the empty string.

Bottom Up Parsing: It is another parsing strategy in which we start with the input string and try to obtain the start symbol of the grammar using successive reductions. If we could reduce the input string to the start symbol, the parse is successful otherwise it is unsuccessful. In this, we can create a parse tree from leaves upwards. The symbol in the input are placed at the leaf nodes of the tree.

The most important aspect of bottom up parsing is the process of detecting handless and then using them in reductions.

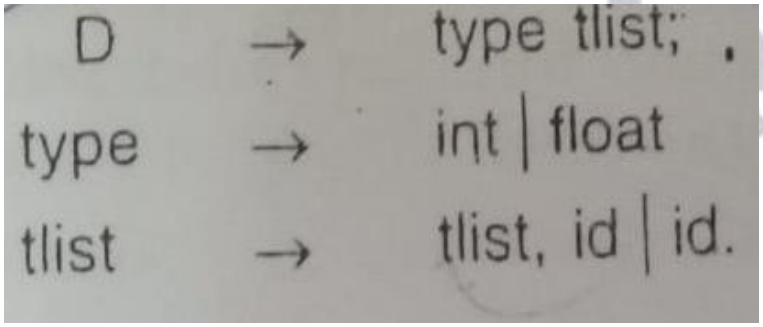
Shift Reduce Parser: It use the principle of bottom up parsing. The various data structures required for a shift reduce parser are as follows:

1. A stack to hold grammar symbols and to detect handles.
2. An input buffer to hold the input string to be parsed.
3. A data structure for storing and accessing the left hand side and right hand side of production rules.

It performs the shift and reduce actions repeatedly until an accept or an error condition is found as follows:

1. It keeps moving input symbols from the buffer on to the stack, one symbol at the time. Moving the next input symbol on the top of the stack is known as shift action.
2. Once the handle is on the top of the stack, it detects the left end of the handle within the stack and reduces it by an appropriate rule.
3. The parser announces a successful parse if the stack contains the start symbol and the input buffer is empty. This is known as accept action.
4. In case parser is not able to either shift or reduce or accept, it announces that a syntax error has occurred and calls an error recovery routine. This is known as error action.

Example: Let us consider the following grammar

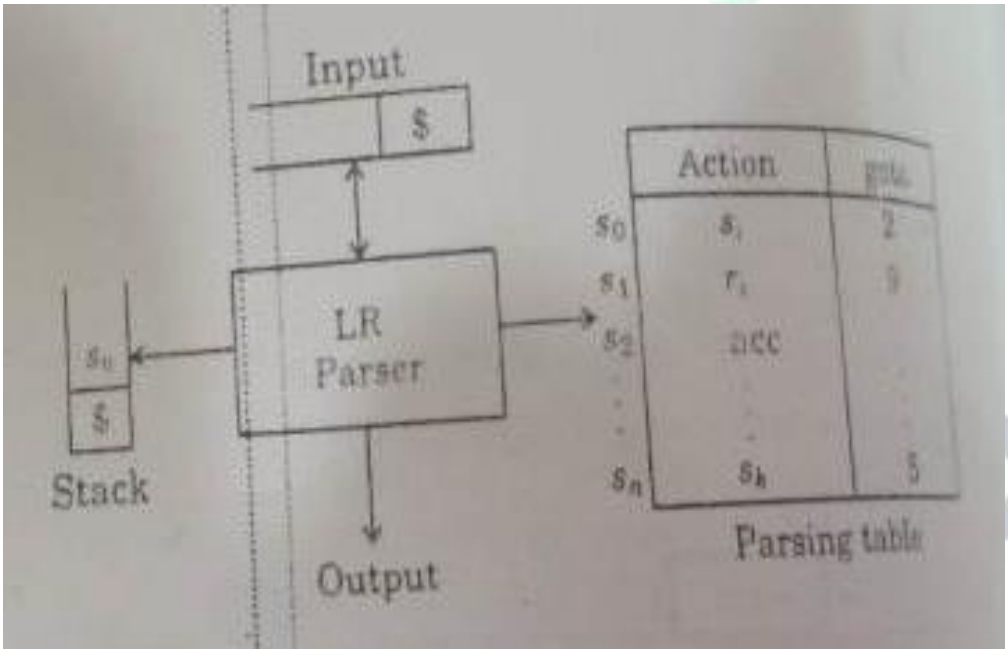


The set of the prefixes of right sentential forms which can appear on the stock of a shift reduce parser are called viable prefixes. We can say that s viable prefix of a grammar is defined to be the prefix of right sentential forms that do not contain any symbols to a right of a handle. The properties of viable prefixes are as follows:

1. We can construct a right most sentential form by adding terminal symbols to viable prefixes.
2. It either contain a handle or part of a handle or parts of more than one handle.

LR Parser: It is a bottom up parsing which can be used to parse a large class of context free grammars. This technique is called LR(K) parsing where L stands for left to right scanning of the input, R stands for constructing right most derivation in reverse and K for the number of input symbols of look ahead which are used to make parsing decisions.

Structure of a LR Parser: The components of an LR parser are a stack, an input, an output and a parsing table having two parts, action and go to.



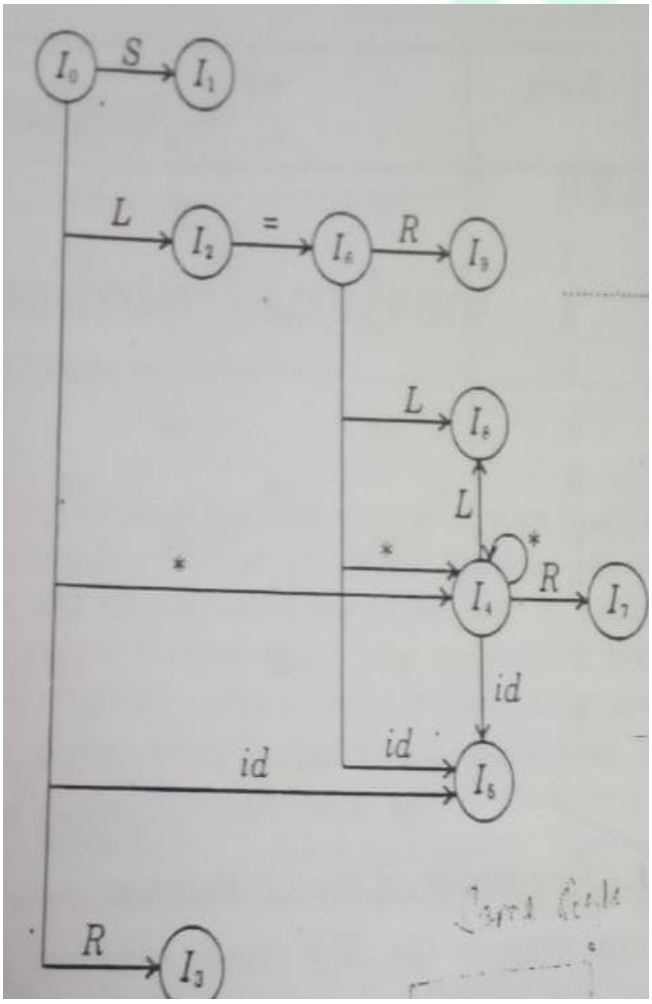
The parser uses extra symbols known as states which summarizes the information contained in the stack below it. Both grammar symbols and state symbols can appear on the stack but the top of the stack always contain a state which is used to index the parsing table and to determine the shift reduce parsing decision in combination with the current input symbol.

SLR Parsing table: SLR(1) or simple LR(1) is the simplest parser in the LR parser family. A grammar for which SLR parsers can be constructed is said to be an SLR grammar.

Canonical LR(1) Parser: LR(1) item which includes a terminal symbol as a second component. The 1 in LR(1) refers to the length of the second component which is called the look ahead symbol.

LALR (1) Parsers: It stands for look ahead LR. This parser has size advantage of a SLR parser as the tables obtained by it are considerably smaller than the canonical LR tables. It has the look ahead capability like a LR parser. The number of states of SLR and LALR parsing tables are always the same.

We may note the following facts about the behavior of LALR(1) parsing table.

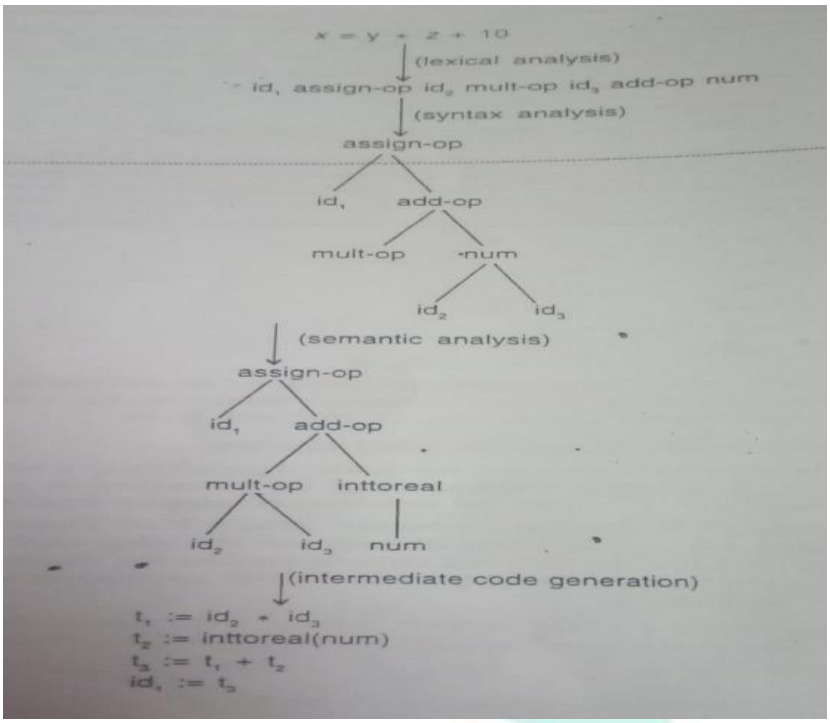


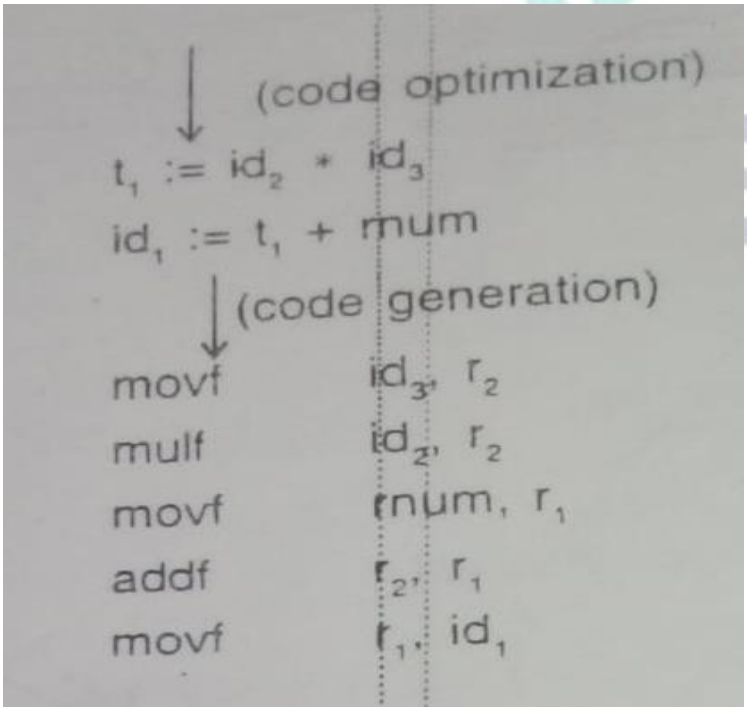
1. A LALR(1) parsing is always free of shift reduce conflicts provided the corresponding LR(1) table is shift-reduce conflict free.
2. A LALR(1) parsing table may have a reduce conflict even if the corresponding LR(1) table is conflict free.
3. For a sentence of the language, both LALR and LR parsers produces same sequence of shifts and reduces.
4. For an erroneous input, a LALR parser may continue to perform reductions even after the LR parser has caught the error. But it does not shift any symbol beyond the point that the LR parser declared an error.

State	action				goto		
	*	=	id	$	S	R	L
0	s4		s5		1	3	2
1				acc			
2		s6		r5			
3				r2			
4	s4		s5			7	8
5		r4		r4			
6	s4		s5			9	8
7		r3		r3			
8		r5		r5			
9				r1			

Comparison of the LR parsers: We can compare the SLR parser, canonical LR parser and LALR parser with respect to their size, capability and the class of grammars they can handle as follows:

1. SLR parser is easiest to construct. Both LR and LALR parsers are expensive to construct in terms of time and space.
2. LALR parser has the same size as that of a SLR parser. Canonical LR parser has the largest size among these parsers.
3. SLR uses follow information to guide reductions. In case of LR and LALR parsers, the look ahead are associated with the items and they make use of the left context available to the parser.
4. LR grammar is a larger subclass of context free grammars as compared to that of SLR and LALR grammars. LALR is applicable to a wide class of grammars than SLR grammar.
5. LR parser is most powerful among the family of bottom up parsers. LALR is intermediate in power but most of the syntactic features of programming languages can be expressed in this grammar. SLR is least powerful and may fail to produce a parsing table for certain grammars on which the other methods succeed.
6. 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 but it is not so in LALR.





gradeup SUPER

Gradeup UGC NET

Super Superscription

Features:

1. 7+ Structured Courses for UGC NET Exam
2. 200+ Mock Tests for UGC NET & MHSET Exams
3. Separate Batches in Hindi & English
4. Mock Tests are available in Hindi & English
5. Available on Mobile & Desktop

Gradeup Super Subscription, Enroll Now

gradeup.co