Module II: Syntax Analysis:

Syntax Analysis: Introduction, Context-Free Grammars, Writing a Grammar, Top-Down Parsing, Recursive and Non recursive top down parsers, Bottom-Up Parsing,

Introduction to LR Parsing: Simple LR, More Powerful LR Parsers, Using Ambiguous Grammars, Parser Generators.

Syntax analysis:

- In syntax analysis phase the source program is analyzed to check whether if conforms to the source language's syntax, and to determine its phase structure.
- This phase is often separated into two phases:
- Lexical analysis: which produces a stream of tokens?
- Parser: which determines the phrase structure of the program based on the context-free grammar for the language?

Parsing:

- It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

- Graphical representation of a derivation or deduction is called a parse tree.
- Each interior node of the parse tree is a non-terminal;

the children of the node can be terminals or nonterminals.

- A **parse tree** is the graphical representation of the structure of a sentence according to its grammar.

Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer.

If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not.

ROLE OF THE PARSER:
Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar.

The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer.

The tree reflects the sequence of derivations or reduction used during the parser.

Hence, it is called parse tree.

If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string.

Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:
By deriving a string from a non-terminal or
By reducing a string of symbol to a non-terminal.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

PARSING:

There are two main kinds of parsers in use, named for the way they build the parse trees:

Top-down: A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
A parser can start with the start symbol and try to transform it to the input string.
Example : LL Parsers.

Bottom-up: A Bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together.
A parser can start with input and attempt to rewrite it into the start symbol.
Example : LR Parsers.
In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

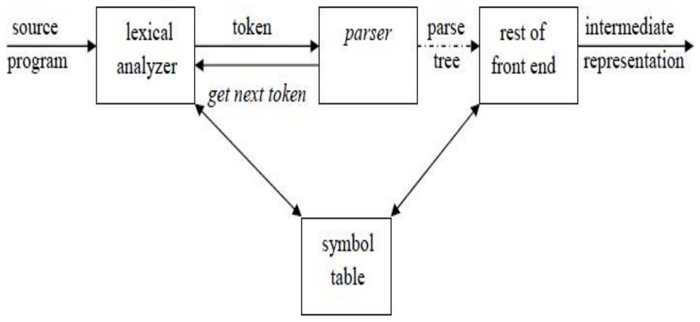
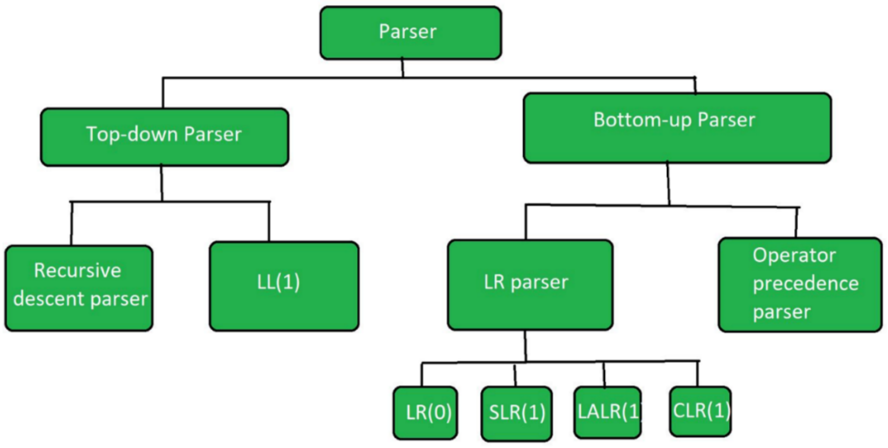
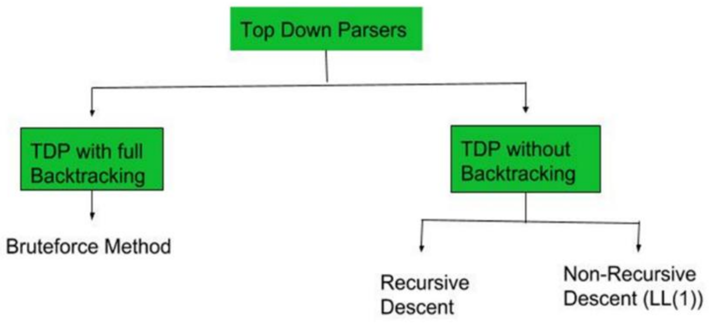


Fig . Position of parser in compiler model.

A top-down parser builds the parse tree from the top down, starting with the start non-terminal.
There are two types of Top Down Parsers:
Top Down Parser with Backtracking
Top Down Parsers without Backtracking



Top Down Parsers without Backtracking can further be divided into two parts:



Context-free Grammars:

A grammar consists of a number of productions.
Each production has an abstract symbol called a nonterminal as its left-hand side, and a sequence of one or more nonterminal and terminal symbols as its right-hand side.
For each grammar, the terminal symbols are drawn from a specified alphabet.
Starting from a sentence consisting of a single distinguished nonterminal, called the goal symbol, a given context-free grammar specifies a language, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

CONTEXT FREE GRAMMARS:

Inherently recursive structures of a programming language are defined by a context-free Grammar.
Formally, a context-free grammar G is a 4-tuple G = (V, T, P, S), where:
V is a finite set of variables (or nonterminals). These describe sets of “related” strings.
T is a finite set of terminals (set of tokens)
P is a finite set of productions rules in the following form A → α where A ∈ V is a variable, and α ∈ (V ∪ T)* is a sequence of terminals and nonterminals.
S ∈ V is the start symbol. (one of the non-terminal symbol)

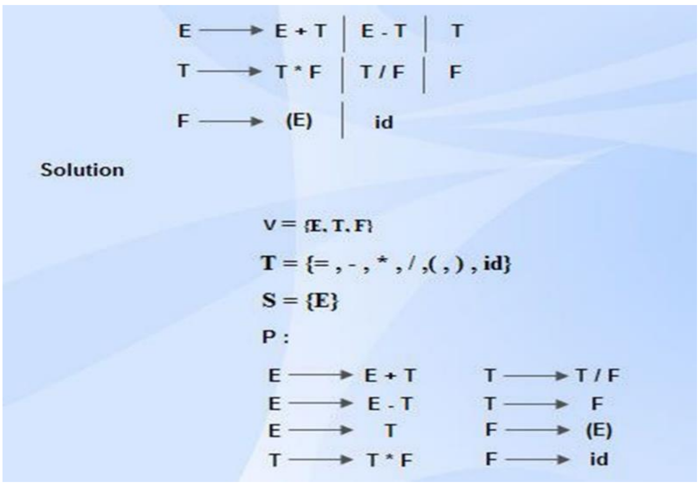
Example of CFG:
E ==>EAE | (E) | -E | id A==> + | - | * | / |
Where E, A are the non-terminals while id, +, *, -, /,(,) are the terminals.

Conventions:

Terminals are symbols from which strings are formed.
Lowercase letters i.e., a, b, c.
Operators i.e.,+,-,*·
Punctuation symbols i.e., comma, parenthesis.
Digits i.e. 0, 1, 2, · · · ,9.
Boldface letters i.e., id, if.

Non-terminals are syntactic variables that denote a set of strings.
Uppercase letters i.e., A, B, C.
Lowercase italic names i.e., expr , stmt.
Start symbol is the head of the production stated first in the grammar.

Production is of the form LHS ->RHS (or) head -> body, where head contains only one non-terminal and body contains a collection of terminals and non-terminals.
(eg.) Let G be



$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid id$

Solution

V = {E, T, F}

T = {= , - , * , / ,(,) , id}

S = {E}

P :

$E \rightarrow E + T$ $\quad T \rightarrow T / F$

$E \rightarrow E - T$ $\quad T \rightarrow F$

$E \rightarrow T$ $\quad F \rightarrow (E)$

$T \rightarrow T * F$ $\quad F \rightarrow id$

Example of context-free grammar:

The following grammar defines simple arithmetic expressions:

expr→ expr op expr
expr → (expr)
expr→ - expr
expr→ id
op → +
op → -
op → *
op → /
op→ ↑

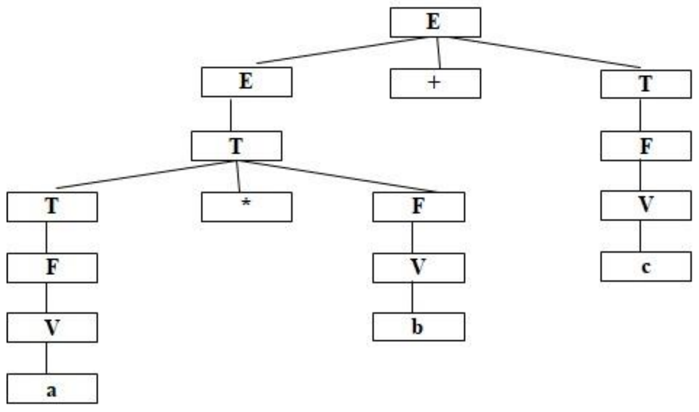
In this grammar,
id + - * / ↑ ( ) are terminals.
expr , op are non-terminals.
expr is the start symbol.
Each line is a production.

Ambiguity:

Example:

Let the production P is:
E→ T | E+T
T→ F | T*F
F→ V | (E)
V→ a | b | c |d

- The parse tree may be viewed as a representation for a derivation that filters out the choice regarding the order of replacement.
- Parse tree for a * b + c



grammar that produces more than one parse for some sentence is said to be ambiguous grammar.
Example : Given grammar G : E→ E+E | E*E | (E) | - E | id
The sentence id+id*id has the following two distinct leftmost derivations:

E→ E+ E	E→ E* E
E→ id + E	E→E+ E * E
E→ id + E * E	E→ id + E * E
E→ id + id * E	E→ id + id * E
E→ id + id * id	E→ id + id * id

- The two corresponding trees are,

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

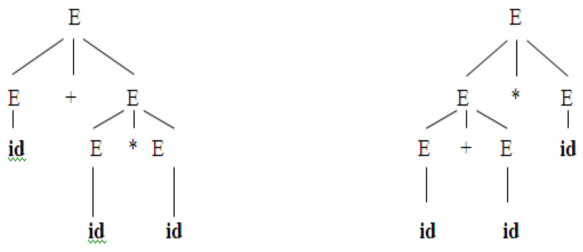


Fig. Two parse trees for id+id*id

Ambiguity in Grammar:

- A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.
- If the grammar is not ambiguous, then it is called unambiguous.
- If the grammar has ambiguity, then it is not good for compiler construction.
- No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

Example 1:

Let us consider a grammar G with the production rule

E → I

E → E + E

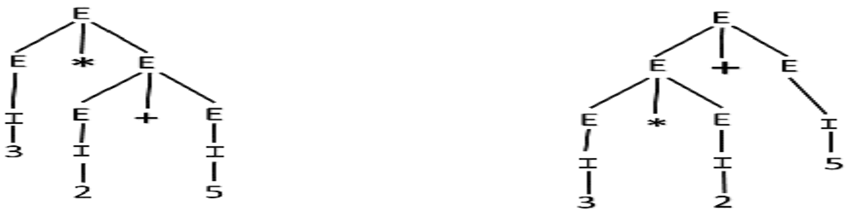
E → E * E

E → (E)

I → ε | 0 | 1 | 2 | ... | 9

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:



Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

Eliminating ambiguity:

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

- Consider this example, G: *stmt*→**if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**
- This grammar is ambiguous since the string **if E1 then S1 | if E1 then S1 else S2** has the following two parse trees for leftmost derivation

To eliminate ambiguity, the following grammar may be used:

stmt→*matched_stmt* | *unmatched_stmt*

matched_stmt→**if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt→**if** *expr* **then** *stmt* | **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

Unambiguous Grammar:

A grammar can be unambiguous if the grammar does not contain ambiguity that means if it does not contain more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.

To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:

1. If the left associative operators (+, -, *, /) are used in the production rule, then apply left recursion in the production rule.

Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example,

1. X → Xa

2. If the right associative operates(^) is used in the production rule then apply right recursion in the production rule.

Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example,

1. X → aX

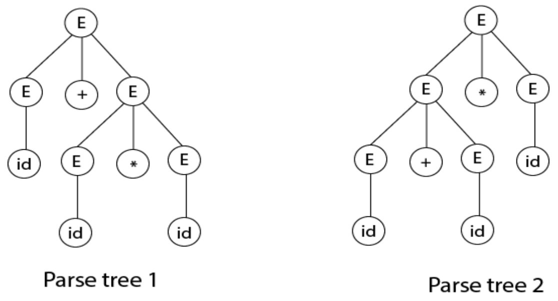
Example:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

1. E → E + E
2. E → E * E
3. E → id

Solution:

Let us derive the string "id + id * id"



As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → id

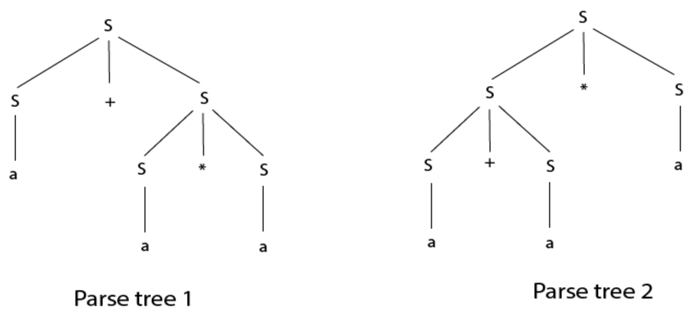
Example:

Check that the given grammar is ambiguous or not. Also, find an equivalent unambiguous grammar.

1. S → S + S
2. S → S * S
3. S → S ^ S
4. S → a

Solution:

The given grammar is ambiguous because the derivation of string aab can be represented by the following string:



Unambiguous grammar will be:

- S → S + A | A
- A → A * B | B
- B → C ^ B | C
- C → a

REGULAR GRAMMAR:

Regular grammar is a context-free grammar in which every production is restricted to one of the following forms:

$A \rightarrow aB$, or

$A \rightarrow w$, where A and B are the nonterminals, a is a terminal symbol, and w is in T^*.

The ∈-productions are permitted as a special case when L(G) contains ∈.

This grammar is called "regular grammar," because if the format of every production in CFG is restricted to $A \rightarrow aB$ or $A \rightarrow a$, then the grammar can specify only regular sets.

REGULAR EXPRESSION	**CONTEXT-FREE GRAMMAR**
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal.
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

Left Recursion, Left-factoring:

There are four categories in writing a grammar :

1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

Eliminating Left Recursion:

Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α. Or

A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.

Top-down parsing methods cannot handle left-recursive grammars.

Hence, left recursion can be eliminated as follows:

If there is a production $A \rightarrow A\alpha \mid \beta$ it can be replaced with a sequence

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

without changing the set of strings derivable from A.

Example 1:
Consider the following grammar and eliminate left recursion-
E → E + E / E x E / a

Solution-
The grammar after eliminating left recursion is-
E → aA
A → +EA / xEA / ∈

Example 2:
Consider the following grammar and eliminate left recursion-
A → ABd / Aa / a
B → Be / b

Solution-
The grammar after eliminating left recursion is-
A → aA'
A' → BdA' / aA' / ∈
B → bB'
B' → eB' / ∈

Algorithm to eliminate left recursion:

```
1. Arrange the non-terminals in some order A1, A2 . . . An.
2. for i := 1 to n do begin
       for j := 1 to i-1 do begin
       replace each
       production of the
       form Ai → Aj γ
           by the productions Ai→δ1
                           γ | δ2γ | . . . | δk γ
       where Aj→δ1 | δ2 | . . . | δk are all the current Aj-productions;
       end
       eliminate the immediate left recursion among the Ai-productions
   end
```

Left factoring:

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- It consists in "factoring out" prefixes which are common to two or more productions.
- When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production A → αβ1 | αβ2 , it can be rewritten as
A →αA'
A' → β1 | β2|ε
Consider the grammar , G : S → iEtS | iEtSeS | a
E→b
Left factored, this grammar becomes
S→ iEtSS' | a
S'→ eS |ε
E→ b

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

RECURSIVE DESCENT PARSING:

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string.
Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right.
It uses procedures for every terminal and non-terminal entity.
This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking.
But the grammar associated with it (if not left factored) cannot avoid back-tracking.
A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.
This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

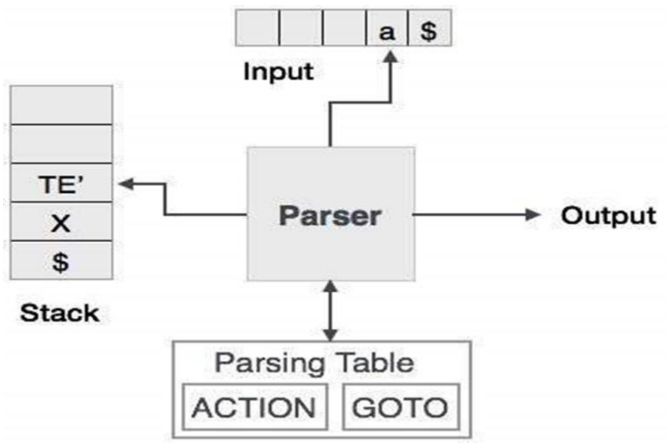
This approach is known as recursive descent parsing, also known as LL(k) parsing where the
first L stands for left-to-right, the
second L stands for leftmost-derivation, and
k indicates k-symbol lookahead.

Therefore, a parser using the single-symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser.

Here the
1st L represents that the scanning of the Input will be done from Left to Right manner and
second L shows that in this Parsing technique we are going to use Left most Derivation Tree. and finally the 1 represents the number of look ahead, means how many symbols are you going to see when you want to make a decision.

Predictive Parser:

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string.
The predictive parser does not suffer from backtracking.
To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols.
To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Both the stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed.
The parser refers to the parsing table to take any decision on the input and stack element combination.

LL(1) grammar:

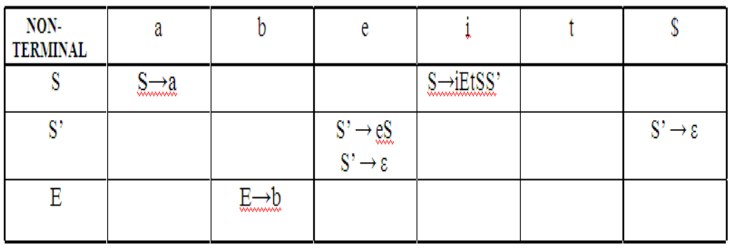
The parsing table entries are single entries.
So each location has not more than one entry.
This type of grammar is called LL(1) grammar.

Consider this following grammar:
S→iEtS | iEtSeS| a
E→b

After eliminating left factoring, we have
S→iEtSS'|a
S'→ eS | ε
E→b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.
S→iEtSS'|a
S'→ eS | ε
E→b
FIRST(S) = { i, a }
FIRST(S') = {e, ε }
FIRST(E) = { b}
FOLLOW(S) = { $,e }
FOLLOW(S') = { $,e }
FOLLOW(E) = {t}

Parsing table:



NON-TERMINAL	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S'→eS S'→ε			S'→ε
E		E→b				

Since there are more than one production, the grammar is not LL(1) grammar.

Steps required for predictive parsing.

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

It is possible to build a nonrecursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls.

The key problem during predictive parsing is that of determining the production to be applied for a nonterminal.

The Non-Recursive parser in figure looks up the production to be applied in parsing table.

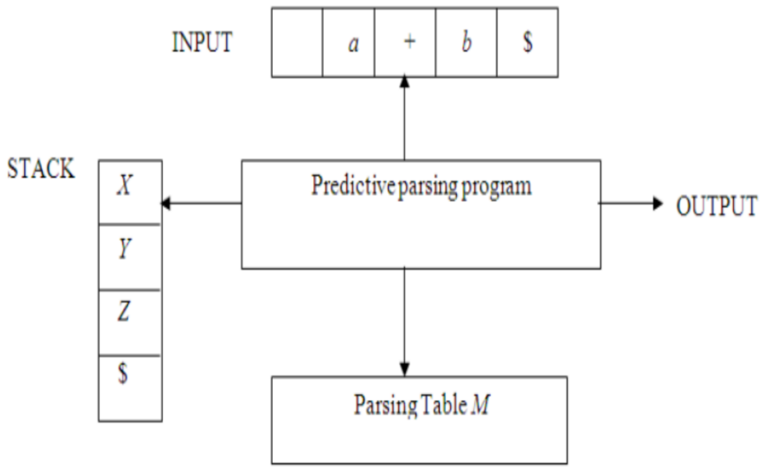


Fig. 2.4 Model of a nonrecursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream.
The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.
The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $.
The parsing table is a two dimensional array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $.
The parser is controlled by a program that behaves as follows.
The program considers X, the symbol on the top of the stack, and a, the current input symbol.
These two symbols determine the action of the parser.

There are three possibilities.

1. If X= a=$, the parser halts and announces successful completion of parsing.
2. If X=a!=$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the program consults entry M[X , a] of the parsing table M.

This entry will be either an X-production of the grammar or an error entry.
If, for example, M[X , a]={X- >UVW}, the parser replaces X on top of the stack by WVU(with U on top).
As output, we shall assume that the parser just prints the production used;
any other code could be executed here.
If M[X , a]=error, the parser calls an error recovery routine.

Error Recovery in Predictive Parsing:

We know that the Predictive parser performs Left most derivative while parsing the given sentence. Now the given sentence may be a valid sentence or an invalid sentence with respect to the specified grammar. An error is detected during the predictive parsing when the terminal on top of the stack does not match the next input symbol, or when nonterminal X on top of the stack, then the present input symbol is a, and the parsing table entry M [X, a] is considered empty.

It is also possible that An error may occur while performing predictive parsing (LL(1) parsing)

When the terminal symbol is at top of the stack and it does not match the current input symbol. In LL(1) parsing If the top of the stack is a non-terminal A, then the present input symbol is **a**, and the parsing table entry M [A, a] is considered empty.

Predictive parsing table construction:
The construction of a predictive parser is aided by two functions associated with a grammar G :
1. FIRST
2. FOLLOW

Rules for first( ):
1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → Y1 Y2…Yk is a production,

then place a in FIRST(X) if for some i, a is in FIRST(Yi),
and ε is in all of FIRST(Y1),…,FIRST(Yi-1);
that is, Y1,….Yi-1=> ε.
If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):
1. If S is a start symbol, then FOLLOW(S) contains $.

2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).

3. If there is a production A → αB, or a
   production A → αBβ
   where FIRST(β) contains ε,
   then everything in FOLLOW(A) is in FOLLOW(B).

Examples:
Consider the following grammar :
E→E+T|T
T→T*F|F
F→(E)|id
After eliminating left-recursion the grammar is
E →TE'
E' → +TE' | ε
T →FT'
T' → *FT' | ε
F→ (E)|id
First( ) :
FIRST(E) = { (, id}
FIRST(E') ={+ , ε }
FIRST(T) = { (, id}
FIRST(T') = {*, ε }
FIRST(F) = { (, id }

Follow( ):
FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Predictive parsing Table:

NON-TERMINAL	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ε	E’→ε
T	T→FT’			T→FT’		
T’		T’→ε	T’→*FT’		T’→ε	T’→ε
F	F→id			F→(E)		

Stack Implementation:

stack	Input	Output
$E	id+id*id $	
$E’T	id+id*id $	E→TE’
$E’T’F	id+id*id $	T→FT’
$E’T’id	id+id*id $	F→id
$E’T’	+id*id $	
$E’	+id*id $	T’ → ε
$E’T+	+id*id $	E’ → +TE’
$E’T	id*id $	
$E’T’F	id*id $	T→FT’
$E’T’id	id*id $	F→id
$E’T’	*id $	
$E’T’F*	*id $	T’ → *FT’
$E’T’F	id $	
$E’T’id	id $	F→id
$E’T’	$	
$E’	$	T’ → ε
$	$	E’ → ε

Bottom up parsing:

Bottom up parsing - Shift Reduce parsing

Bottom-Up Parser : Constructs a parse tree for an input string beginning at the leaves(the bottom) and working up towards the root(the top)

String
Reduced to
Start Symbol

Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.
Bottom-up parsing is also known as shift-reduce parsing because its two main actions are shift and reduce.
At each shift action, the current symbol in the input string is pushed to a stack.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

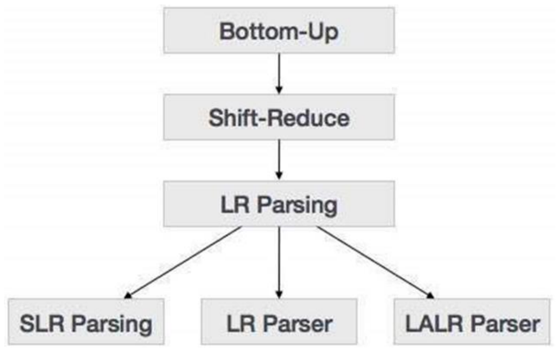
At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.

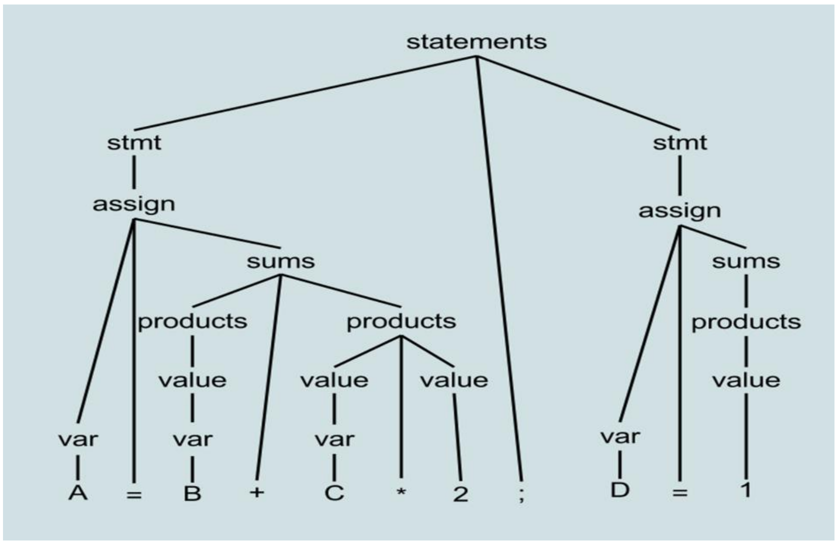
Bottom-up parsing is more general than top-down parsing
–And just as efficient
–Builds on ideas in top-down parsing
–Preferred method in practice

Also called LR parsing
–L means that tokens are read left to right
–R means that it constructs a rightmost derivation

LR parsers don’t need left-factored grammars and can also handle left-recursive grammars

The image given below depicts the bottom-up parsers available:

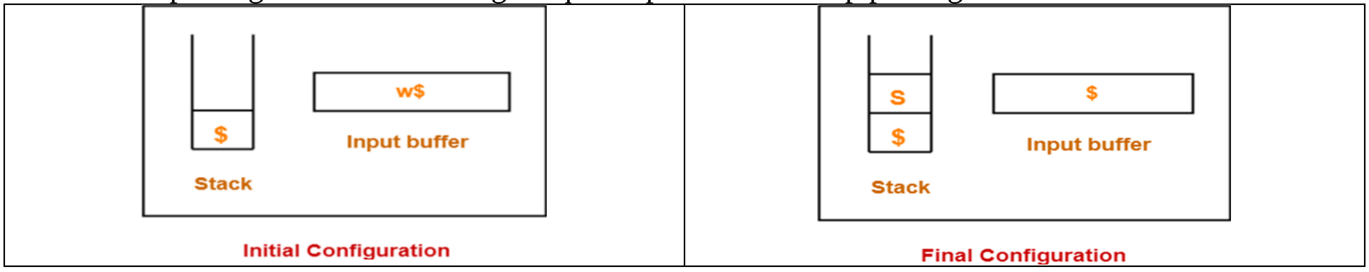




Typical parse tree for A = B + C*2; D = 1

Shift-Reduce Parsing

Shift-reduce parsing uses the following unique steps for bottom-up parsing.



P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

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

Accept:

- If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept.
- When accept action is obtained, it is means successful parsing is done.

Error:

- This is the situation in which the parser can neither perform shift action nor reduce action and not even accept.

<u>Rules To Remember</u>

- It is important to remember the following rules while performing the shift-reduce action-
- If the priority of incoming operator is more than the priority of in stack operator, then shift action is performed.
- If the priority of in stack operator is same or less than the priority of in stack operator, then reduce action is performed.
  - **Example 1 –** Consider the grammar
    S –> S + S
    S –> S * S
    S –> id
    Perform Shift Reduce parsing for input string “id + id + id”.

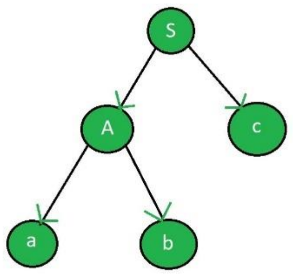
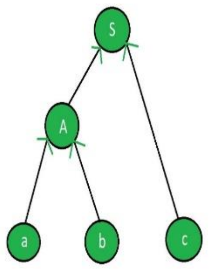
Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	+id+id$	Reduce by S --> id
$S	+id+id$	Shift
$S+	id+id$	Shift
$S+id	+id$	Reduce by S --> id
$S+S	+id$	Shift
$S+S+	id$	Shift
$S+S+id	$	Reduce by S --> id
$S+S+S	$	Reduce by S --> S+S
$S+S	$	Reduce by S --> S+S
$S	$	Accept

Example 2 – Consider the grammar
E –> 2E2
E –> 3E3
E –> 4
Perform Shift Reduce parsing for input string “32423”.

Stack	Input Buffer	Parsing Action
$	32423$	Shift
$3	2423$	Shift
$32	423$	Shift
$324	23$	Reduce by E --> 4
$32E	23$	Shift
$32E2	3$	Reduce by E --> 2E2
$3E	3$	Shift
$3E3	$	Reduce by E --> 3E3
$E	$	Accept

Bottom up parsing – LR Parsers

Difference between LL and LR parser

LL parser	LR parser
LL Parser includes both the recursive descent parser and non-recursive descent parser. Its one type uses backtracking while another one uses parsing table. Theses are top down parser. **Example:** Given grammar is S -> Ac A -> ab where S is start symbol, A is non-terminal and a, b, c are terminals. **Input string:** abc Parse tree generated by LL parser:	LR Parser is one of the bottom up parser which uses parsing table (dynamic programming) to obtain the parse tree form given string using grammar productions. **Example:** In the above example, parse tree generated by LR parser:
	

LL	LR
First L of LL is for left to right scan and second L is for leftmost derivation.	First L of LR is for left to right and R is for rightmost derivation.
Using LL parser, parser tree is constructed in top down manner.	Using LR parser, Parser tree is constructed in bottom up manner.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty i.e contains $ only	Starts with an empty stack i.e contains $ only
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.

Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal of left side of a production, off the stack, and pushes the corresponding right hand side of the production.	Tries to recognize a right hand side of a production on the top of stack, pops it, and pushes the corresponding nonterminal of production to stack.
i.e. Expands the non-terminals.	i.e. Reduces to the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.
It may use backtracking or dynamic programming.	It uses dynamic programming.
Example: LL(0), LL(1)	**Example:** LR(0), SLR(1), LALR(1), CLR(1)

LR Parser

- The LR parser is a non-recursive, shift-reduce, bottom-up parser.
- It uses a wide class of context-free grammars which makes it the most efficient syntax analysis technique.
- Also known as LR(k) parsers, "L" stands for left-to-right scanning of the input.
- "R" stands for constructing a right most derivation in reverse.
- "K" is the number of input symbols of the look ahead used to make decision.
- LR parsers are classified into four types: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

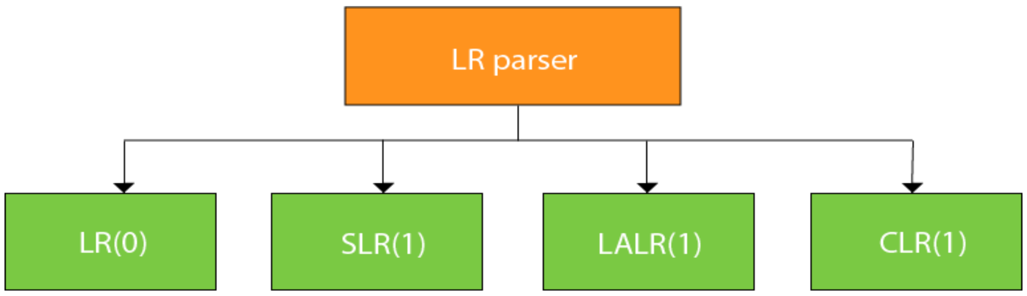


Fig: Types of LR parser

- There are three widely used algorithms available for constructing an LR parser:
- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- LR(1) – LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

LR algorithm:

- The LR algorithm requires stack, input, output and parsing table.
- In all type of LR parsers, input, output and stack are same but parsing table is different.

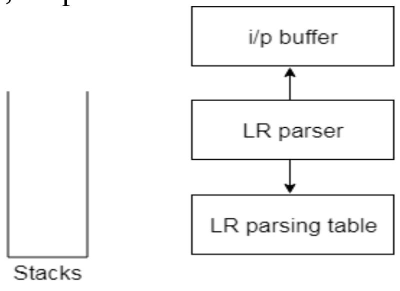


Fig: Block diagram of LR parser

In input buffer is used to indicate the string to be parsed followed by a $ Symbol.
A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.
Parsing table is a two dimensional array.
It contains two parts: Action part and Go To part.

LR (1) Parsing:
Various steps involved in the LR (1) Parsing:

- For the given input string write a context free grammar.
- Check the ambiguity of the grammar.
- Add Augment production(s) in the given grammar.
- Create Canonical collection of LR (0) items.
- Draw a data flow diagram for DFA.
- Construct a LR (1) parsing table.

Augment Grammar
Augmented grammar G` will be generated if we add one more production in the given grammar G. It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Example
Given grammar

$S \rightarrow AA$
$A \rightarrow aA \mid b$

The Augment grammar G` is represented by
$S` \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA \mid b$

Canonical Collection of LR(0) items

- An LR (0) item is a production G with dot at some position on the right side of the production.
- LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.
- In the LR (0), we place the reduce node in the entire row.

Example

Given grammar:

1. $S \rightarrow AA$
2. $A \rightarrow aA \mid b$

Add Augment Production and insert '•' symbol at the first position for every production in G

1. S` → •S
2. S → •AA
3. A → •aA
4. A → •b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal.
So, the I0 State becomes

I0 = S` → •S
S → •AA

Add all productions starting with "A" in modified I0 State because "•" is followed by the non-terminal.

So, the I0 State becomes.
I0= S` → •S
S → •AA
A → •aA
A → •b

I1 State:
I1= Go to (I0, S) = closure (S` → S•) = S` → S•

Here, the Production is reduced so close the State.
I1= S` → S•
I2 State:
I2= Go to (I0, A) = closure (S → A•A)

Add all productions starting with A in to I2 State because "•" is followed by the non-terminal.

So, the I2 State becomes
I2 =S→A•A
A → •aA
A → •b
Go to (I2,a) = Closure (A → a•A) = (same as I3)
Go to (I2, b) = Closure (A → b•) = (same as I4)

I3 State:
I3= Go to (I0,a) = Closure (A → a•A)

Add productions starting with A in I3.
A → a•A
A → •aA
A → •b

Go to (I3, a) = Closure (A → a•A) = (same as I3)
Go to (I3, b) = Closure (A → b•) = (same as I4)

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

I4, I5 and I6 States:

I4= Go to (I0, b) = closure (A → b•) = A → b•

I5= Go to (I2, A) = Closure (S → AA•) = SA → A•

I6= Go to (I3, A) = Closure (A → aA•) = A → aA•

I0 State:

S` → •S
    S → •AA
    A → •aA
    A → •b

I1 State:

I1= S` → S•

I2 State:

S→A•A
    A → •aA
    A → •b

I3 State:

A → a•A
A → •aA
A → •b

I4 State:

A → b•

I5 State:

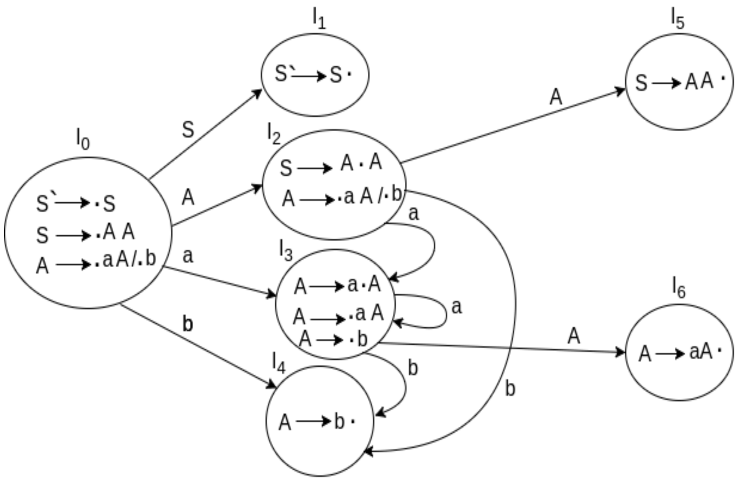
SA → A•

I6 State:

A → aA•

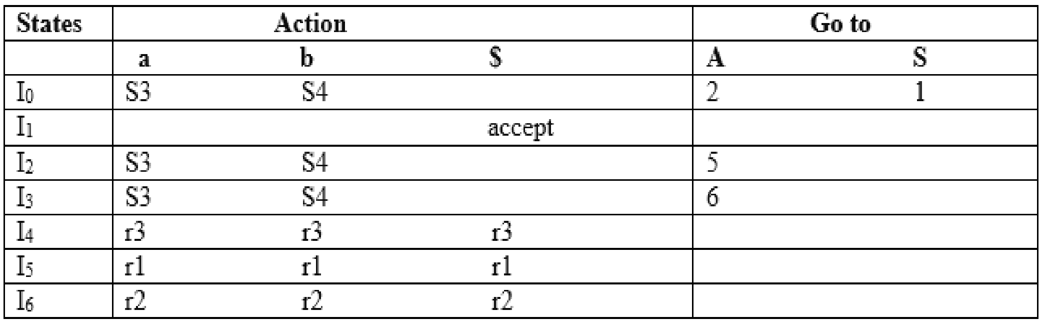
Drawing DFA:

The DFA contains the 7 states I0 to I6.



LR(0) Table:

- If a state is going to some other state on a terminal then it correspond to a shift move.
- If a state is going to some other state on a variable then it correspond to go to move.
- If a state contain the final item in the particular row then write the reduce node completely.



States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

Explanation:

- I0 on S is going to I1 so write it as 1.
- I0 on A is going to I2 so write it as 2.
- I2 on A is going to I5 so write it as 5.
- I3 on A is going to I6 so write it as 6.
- I0, I2and I3on a are going to I3 so write it as S3 which means that shift 3.
- I0, I2 and I3 on b are going to I4 so write it as S4 which means that shift 4
- I4, I5 and I6 all states contains the final item because they contain • in the right most end.
- So rate the production as production number.

Productions are **numbered** as follows:

1. S → AA ... (1)
2. A → aA ... (2)
3. A → b ... (3)

- I1 contains the final item which drives(S` → S•), so action {I1, $} = Accept.
- I4 contains the final item which drives A → b• and that production corresponds to the production number 3 so write it as r3 in the entire row.
- I5 contains the final item which drives S → AA• and that production corresponds to the production number 1 so write it as r1 in the entire row.
- I6 contains the final item which drives A → aA• and that production corresponds to the production number 2 so write it as r2 in the entire row.

Construct LR (1) parsing table for the following grammar.

S->CC
C->cC
C->d

Construct LR (1) parsing table:

Let the given grammar is.
S->CC
C->cC
C->d

- Add Augment Productions, insert '•' symbol at the first position for every production in G and also add the lookahead.

1. S` → •S, $
2. S → •CC, $
3. C → •cC, c/d
4. C → •d, c/d

The canonical collection of items sets is:

I0 State:

- Add Augment production to the I0 State and Compute the Closure
  I0 = Closure (S` →•S)
- Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes
  I0 = S` →•S, $
  S →•CC, $
- Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0 = S` →•S, $
S →•CC, $
C →•cC, c/d
C →•d, c/d

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $

I2= Go to (I0, C) = closure (S→ C•C, $)

- Add all productions starting with C in I2 State because "." is followed by the non-terminal.
- So, the I2 State becomes

I2 = S → C•C, $
C →•cC, $
C →•d, $

I3= Go to (I0, c) = Closure (C→ c•C, c/d)

- Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes
  I3= **C** → c•C, c/d
  C →•cC, c/d
  C →•d, c/d

I4= Go to (I0, d) = closure (C → d•, c/d) = C → d•, c/d

Go to (I3, c) = Closure (C → c•C, c/d) = (same as I3)
Go to (I3, d) = Closure (C → d•, c/d) = (same as I4)

I5= Go to (I2, C) = Closure (S → CC•, $) =S → CC•, $

I6= Go to (I2, c) = Closure (C → c•C, $)

- Add all productions starting with C in I6 State because "." is followed by the non-terminal.
- So, the I6 State becomes
  I6 = C → c•C, $
  C →•cC, $
  C →•d, $

Go to (I6, c) = Closure (C → c•C, $) = (same as I6)
Go to (I6, d) = Closure (C → d•, $) = (same as I7)

I7= Go to (I2, d) = Closure (C → d•, $) = C → d•, $

I8= Go to (I3, C) = Closure (C → cC•, c/d) = C → cC•, c/d

I9= Go to (I6, C) = Closure (C → cC•, \$) = C → cC•, \$

The LR(1) items for *G* is as follows:

State I0. **[*S′* → · *S*, \$]** **[*S* → · *C C*, \$]** **[*C* → · c *C*, c]** **[*C* → · c *C*, d]** **[*C* → · d, c]** **[*C* → · d, d]** **On *S* goto 1** **On *C* goto 2** **On c goto 3** **On d goto 4**	**State I1.** **[*S′* → *S* ·, \$]** **State I2.** **[*S* → *C* · *C*, \$]** **[*C* → · c *C*, \$]** **[*C* → · d, \$]** **On *C* goto 5** **On c goto 6** **On d goto 7**	**State I3.** **[*C* → c · *C*, c]** **[*C* → c · *C*, d]** **[*C* → · c *C*, c]** **[*C* → · c *C*, d]** **[*C* → · d, c]** **[*C* → · d, d]** **On *C* goto 8** **On c goto 3** **On d goto 4**
State 4. **[*C* → d ·, c]** **[*C* → d ·, d]** **State 5.** **[*S* → *C C* ·, \$]**	**State 6.** **[*C* → c · *C*, \$]** **[*C* → · c *C*, \$]** **[*C* → · d, \$]** **On *C* goto 9** **On c goto 6** **On d goto 7**	**State 7.** **[*C* → d ·, \$]** **State 8.** **[*C* → c *C* ·, c]** **[*C* → c *C* ·, d]** **State 9.** **[*C* → c *C* ·, \$]**

Given Productions are numbered as follows:

- S → CC ... (1)
- C → cC ... (2)
- C → d ... (3)

The LR(1) parsing table is as follows:

	Actions			Goto	
	c	d	\$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Construct the predictive parsing table for the following grammar
E - > E+T /T
T-> T*F/F
F-> (E)/id

Predictive parsing table construction:

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

The following are the steps in constructing **parsing table.**

Step 1: Elimination of left recursion, left factoring and ambiguous grammar.

Step 2: Construct FIRST() and FOLLOW() for all non-terminals in the resulting grammer. These functions will indicate proper entries in the table for a grammar G.

Step 3: Construct predictive parsing table as described below.

- The parsing table is a two dimensional array M[A, a] where A is a nonterminal, and a is a terminal or the symbol $.

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST
2. FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → Y1 Y2…Yk is a production,

then place a in FIRST(X) if for some i, a is in FIRST(Yi),
and ε is in all of FIRST(Y1),…,FIRST(Yi-1);
that is, Y1,….Yi-1=> ε.
If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A → αB, or a
   production A → αBβ
   where FIRST(β) contains ε,
   then everything in FOLLOW(A) is in FOLLOW(B).

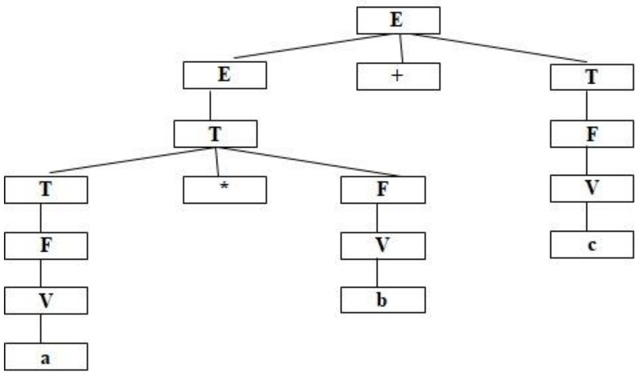
Consider the following grammar:
E→E+T|T
T→T*F|F
F→(E)|id

After eliminating left-recursion the grammar is:
E →TE’
E’ → +TE’ | ε
T →FT’
T’ → *FT’ | ε
F→ (E)|id

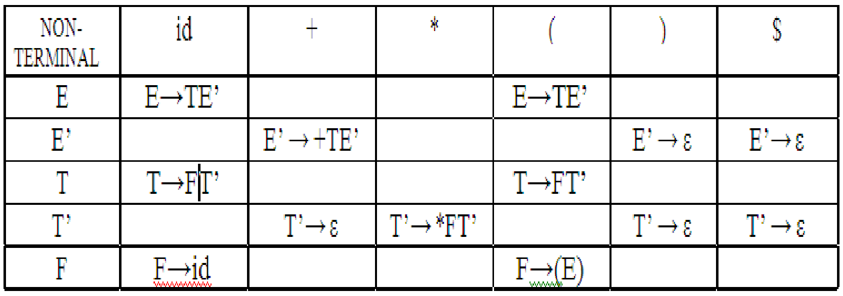
First( ) :
FIRST(E) = { (, id}
FIRST(E’) ={+ , ε }
FIRST(T) = { (, id}
FIRST(T’) = {*, ε }
FIRST(F) = { (, id }

Follow( ):
FOLLOW(E) = { $,) }
FOLLOW(E’) = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T’) = { +, $,) }

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



Predictive parsing Table:



NON-TERMINAL	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Compute FIRST and FOLLOW sets for all non terminals and construct the LL(1) Parsing table in the following grammar

bexpr-> bexpr or bterm / bterm
bterm-> bterm and bfactor / bfactor
bfactor-> not bfactor / (bexpr) / true / false

FIRST and FOLLOW sets:

- Given Grammar:
  bexpr-> bexpr **or** bterm / bterm
  bterm-> bterm **and** bfactor / bfactor
  bfactor-> **not** bfactor / (bexpr) / true / false

Since the given grammar contains left recursion, after removing left recursion:

bexpr → bterm E'
E' → or bterm E'
→ ε
T' → and bfactor T'
→ ε
bterm → bfactor T'
bfactor → not bfactor
| (bexpr)
| true
| false

First and Follow for the non-terminals:

First(bexpr) = First(bterm) = First (bfactor) = {not, (, true, false} First(E') = {or, ε}
First(T') = {and, ε}
Follow(bexpr) = {$,)}
Follow(E') = Follow(bexpr) = {$,)}
Follow(bterm) = First(E') U Follow(E') = {or,), $}
Follow(T') = Follow(bterm) = {or,), $}

Follow(bfactor) = First(T') U Follow(bterm) = {and, or,), $}

Construct the parse table:

	or	and	not	(	)	True/false	$
bexpr			bexpr → bterm E'	bexpr → bterm E'		bexpr → bterm E'	
E'	E' → or bterm E'				E'→ ε		E' → ε
bterm			bterm → bfactor T'	bterm → bfactor T'		bterm → bfactor T'	
T'	T'→ ε	T' → and bfactor T'			T'→ ε		T'→ ε
bfactor			bfactor → not bfactor	bfactor → (bexpr)		bfactor → true/false	

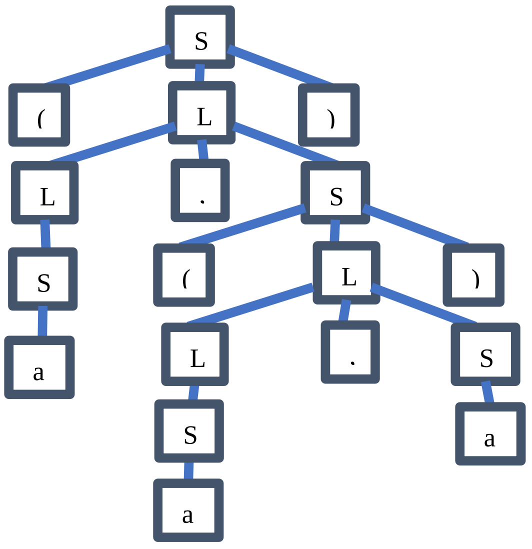
Consider the Grammar and solve the following sentence

S-> (L) / a
L-> L, S / S

i) What are the terminals, non terminals and start symbol
ii) Find the parse tree for the sentence (a, (a, a))
iii) Construct the LMD for the sentence (a, (a, a))
iv) Construct the RMD for the sentence (a, (a, a))

i)	start symbol	=	S
	non terminals	=	S, L
	terminals	=	(
			)
			a
			,

Find the parse tree for the sentence (a,(a, a))



ii) LMD	iii) RMD
S-> (L)	S-> (L)
-> (L , S)	-> (L , S)
-> (S , S)	-> (L , (L))
-> (a , S)	-> (L , (L , S))
-> (a , (L))	-> (L , (L , a))
-> (a , (L , S))	-> (L , (S , a))
-> (a , (S , S))	-> (L , (a , a))
-> (a , (a , S))	-> (S , (a , a))
-> (a , (a , a))	-> (a , (a , a))

Construct LR (0) parsing table for the following grammar.
E - > E+T /T
T-> T*F/F
F-> id

- An LR (0) item is a production G with dot at some position on the right side of the production.
- LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.
- In the LR (0), we place the reduce node in the entire row.

Add Augment Production and insert '•' symbol at the first position for every production in G
S’ -> E
E - > E+T
E - > T
T-> T*F
T-> F
F-> id

I0 State:

Add Augment production to the I0 State and Compute the Closure
I0 = Closure (S` → •E)
Add all productions starting with E in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes
I0 = S` → •E
E → •E + T
E → •T
Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.
I0= S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I1= Go to (I0, E) = closure (S` → E•, E → E• + T)
I2= Go to (I0, T) = closure (E → T•T, T• → * F)
I3= Go to (I0, F) = Closure (T → F•) = T → F•
I4= Go to (I0, id) = closure (F → id•) = F → id•
I5= Go to (I1, +) = Closure (E → E +•T)
Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes
I5 = E → E +•T
T → •T * F
T → •F
F → •id
Go to (I5, F) = Closure (T → F•) = (same as I3)
Go to (I5, id) = Closure (F → id•) = (same as I4)

I6= Go to (I2, *) = Closure (T → T * •F)
Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes
I6 = T → T * •F
F → •id
Go to (I6, id) = Closure (F → id•) = (same as I4)

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

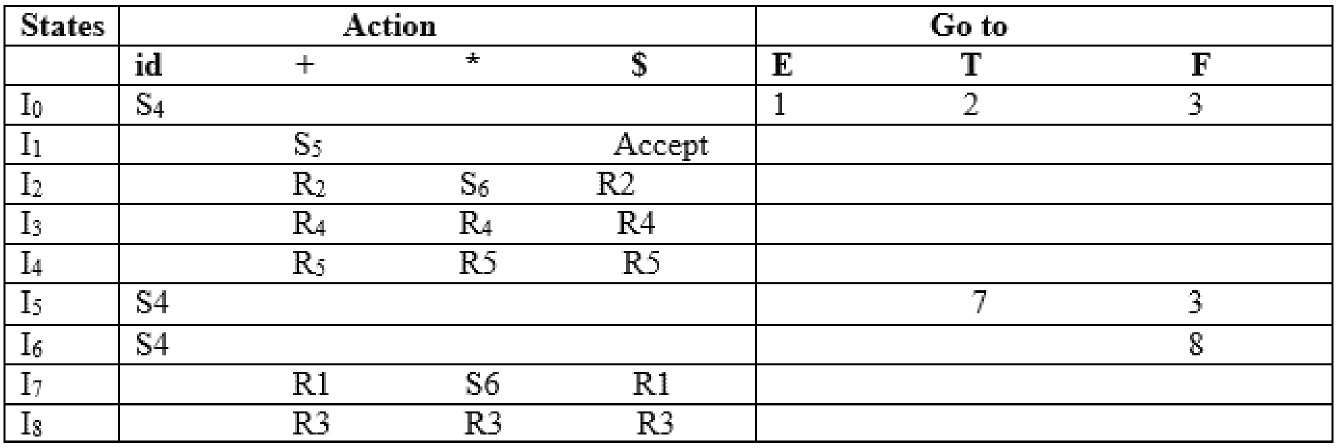
I7= Go to (I5, T) = Closure (E → E + T•) = E → E + T•
I8= Go to (I6, F) = Closure (T → T * F•) = T → T * F•

First and Follows:

First (E)	= First (E + T) ∪ First (T)
First (T)	= First (T * F) ∪ First (F)
First (F)	= {id}
First (T)	= {id}
First (E)	= {id}
Follow (E)	= First (+T) ∪ {$} = {+, $}
Follow (T)	= First (*F) ∪ First (F)
	= {*, +, $}
Follow (F)	= {*, +, $}

- I1 contains the final item which drives S → E• and follow (S) = {$}, so action {I1, $} = Accept
- I2 contains the final item which drives E → T• and follow (E) = {+, $}, so action {I2, +} = R2, action {I2, $} = R2
- I3 contains the final item which drives T → F• and follow (T) = {+, *, $}, so action {I3, +} = R4, action {I3, *} = R4, action {I3, $} = R4
- I4 contains the final item which drives F → id• and follow (F) = {+, *, $}, so action {I4, +} = R5, action {I4, *} = R5, action {I4, $} = R5
- I7 contains the final item which drives E → E + T• and follow (E) = {+, $}, so action {I7, +} = R1, action {I7, $} = R1
- I8 contains the final item which drives T → T * F• and follow (T) = {+, *, $}, so action {I8, +} = R3, action {I8, *} = R3, action {I8, $} = R3.

LR (0) Table:



States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S4				1	2	3
I_1		S5		Accept			
I_2		R2	S6	R2			
I_3		R4	R4	R4			
I_4		R5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

SLR Parser

SLR (1) Parsing:

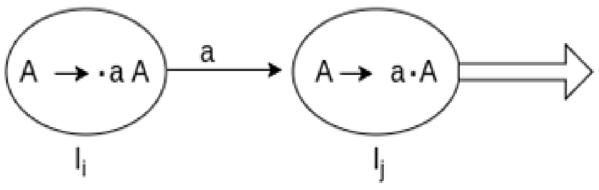
- SLR (1) refers to simple LR Parsing.
- It is same as LR(0) parsing.
- The only difference is in the parsing table.
- To construct SLR (1) parsing table, we use canonical collection of LR (0) items.
- In the SLR (1) parsing, we place the reduce move only in the follow of left hand side.

Various steps involved in the SLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a SLR (1) parsing table

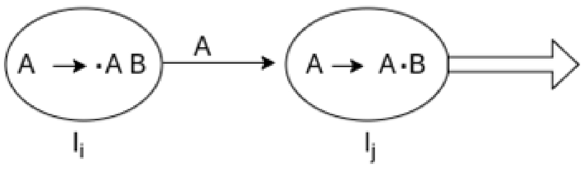
SLR (1) Table Construction:

- The steps which use to construct SLR (1) Table is given below:
- If a state (I_i) is going to some other state (I_j) on a terminal then it corresponds to a shift move in the action part.



States	Action		Go to
	a	$	A
I_i	Sj		
I_j			

- If a state (I_i) is going to some other state (I_j) on a variable then it correspond to go to move in the Go to part.

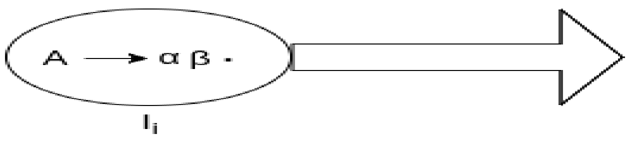


States	Action		Go to
	a	$	A
I_i			j
I_j			

- If a state (I_i) contains the final item like A → ab• which has no transitions to the next state then the production is known as reduce production.
- For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers.

Example

1. S -> •Aa
2. A ->αβ•
3. Follow(S) = {$}
4. Follow (A) = {a}



States	Action			Go to	
	a	b	$	S	A
I_i	r2				

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

SLR (1) Grammar
S → E
E → E + T | T
T → T * F | F
F → id
Add Augment Production and insert '•' symbol at the first position for every production in G
S`→•S
S →•E
E →•E + T
E →•T
T →•T * F
T →•F
F →•id

I0 = Closure (S` →•S)

- Add all productions starting with S and E in to I0 State because "." is followed by the non-terminal.
- So, the I0 State becomes
  I0 = S` →•S
  S →•E
  E →•E + T
  E →•T
- Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal.
- So, the I0 State becomes.

I0= S` →•E
E →•E + T
E →•T
T →•T * F
T →•F
F →•id

I1= Go to (I0, E) = closure (S` → E•, E → E• + T)
I2= Go to (I0, T) = closure (E → T•, T→T• * F)
I3= Go to (I0, F) = Closure (T → F•) = T → F•
I4= Go to (I0, id) = closure (F → id•) = F → id•
I5= Go to (I1, +) = Closure (E → E +•T)

- Add all productions starting with T and F in I5 State because "." is followed by the non-terminal.
- So, the I5 State becomes
- **I5 =** E → E +•T
  T →•T * F
  T →•F
  F →•id
- Go to (I5, F) = Closure (T → F•) = (same as I3)
  Go to (I5, id) = Closure (F → id•) = (same as I4)

I6= Go to (I2, *) = Closure (T → T * •F)

- Add all productions starting with F in I6 State because "." is followed by the non-terminal.
  So, the I6 State becomes
  I6 = T → T * •F
  F →•id
- Go to (I6, id) = Closure (F → id•) = (same as I4)

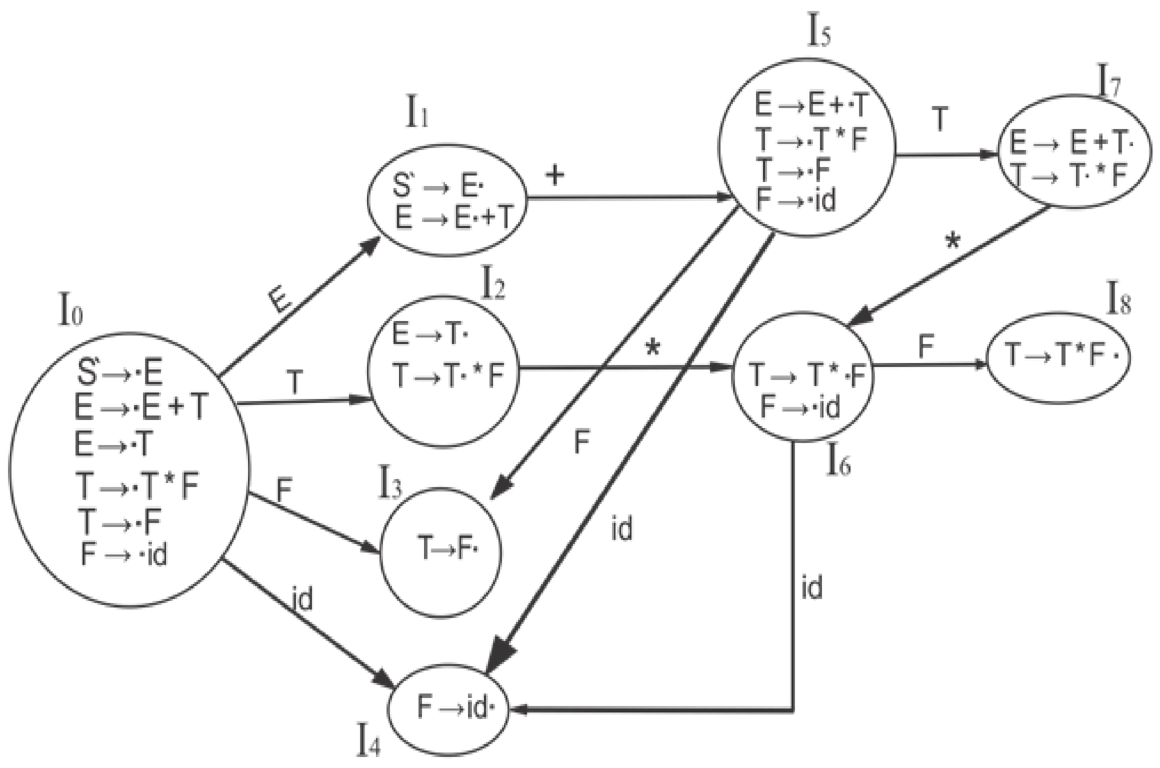
P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

I7= Go to (I5, T) = Closure (E → E + T•) = E → E + T•, T →T• * F

I8= Go to (I6, F) = Closure (T → T * F•) = T → T * F•

- Go to (I7, *) = Closure (T →T*• F, F →•id)= (same as I6)

Drawing DFA:



SLR (1) Table:

States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Explanation:

- First (E) = First (E + T) ∪ First (T)
  First (T) = First (T * F) ∪ First (F)
  First (F) = {id}
  First (T) = {id}
  First (E) = {id}
- Follow (E) = First (+T) ∪ {$} = {+, $}
  Follow (T) = First (*F) ∪ Follow (E)

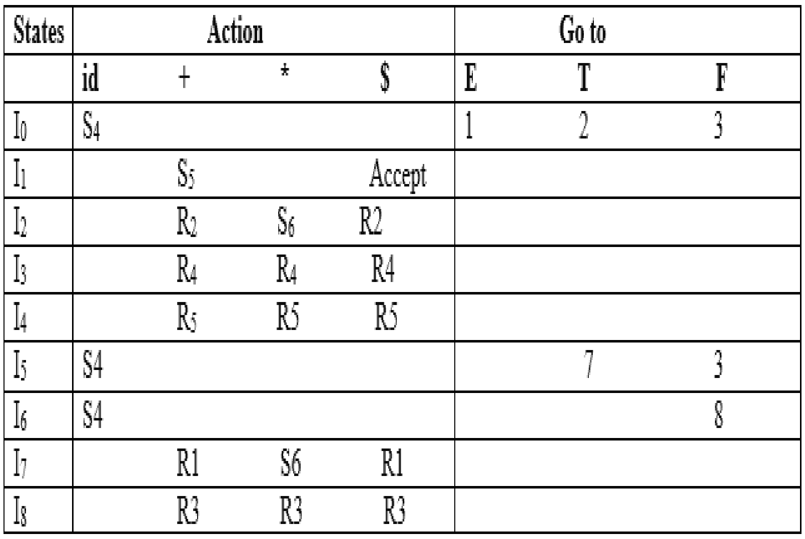
P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

= {*, +, $}

Follow (F) = {*, +, $}

- I1 contains the final item which drives S → E• and
  follow (S) = {$},
  so action {I1, $} = Accept
- I2 contains the final item which drives E → T• and
  follow (E) = {+, $},
  so action {I2, +} = R2,
  action {I2, $} = R2
- I3 contains the final item which drives T → F• and
  follow (T) = {+, *, $},
  so action {I3, +} = R4,
  action {I3, *} = R4,
  action {I3, $} = R4
- I4 contains the final item which drives F → id• and
  follow (F) = {+, *, $},
  so action {I4, +} = R5,
  action {I4, *} = R5,
  action {I4, $} = R5
- I7 contains the final item which drives E → E + T• and
  follow (E) = {+, $},
  so action {I7, +} = R1,
  action {I7, $} = R1
- I8 contains the final item which drives T → T * F• and
  follow (T) = {+, *, $},
  so action {I8, +} = R3,
  action {I8, *} = R3,
  action {I8, $} = R3.

SLR (1) Table:



States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

LALR (1) Parsing

- LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.
- In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items.
- LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Example
LALR (1) Grammar

1. S → AA
2. A → aA
3. A → b

- Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:

- Add Augment production to the I0 State and Compute the Closure
  I0 = Closure (S` →•S, $)
- Add all productions starting with S in to I0 State because "•" is followed by the non-terminal.
- So, the I0 State becomes
  I0 = S` →•S, $
  S →•AA, $
- Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal.
- So, the I0 State becomes.

I0= S` →•S, $
S →•AA, $
A →•aA, a/b
A →•b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S**•, $**
I2= Go to (I0, A) = closure (S → **A•A, $**)

- Add all productions starting with A in I2 State because "•" is followed by the non-terminal.
- So, the I2 State becomes
- **I2=** S → A•A, $
  A →•aA, $
  A →•b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

- Add all productions starting with A in I3 State because "•" is followed by the non-terminal.
- So, the I3 State becomes
  I3= A → a•A, a/b

A →•aA, a/b
A →•b, a/b
Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)
I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $
I6= Go to (I2, a) = Closure (A → a•A, $)

- Add all productions starting with A in I6 State because "•" is followed by the non-terminal.
- So, the I6 State becomes
  I6 = A → a•A, $

A →•aA, $
A →•b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, $) =A → aA•, $

- If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = { A → a•A, a/b
A →•aA, a/b
A →•b, a/b
}

I6= { A → a•A, $
A →•aA, $
A →•b, $
}

- Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { A → a•A, a/b/$
A →•aA, a/b/$
A →•b, a/b/$
}

I4= A → b•, a/b

I7= A → b•, $

- The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 ={ A → b•, a/b/$ }

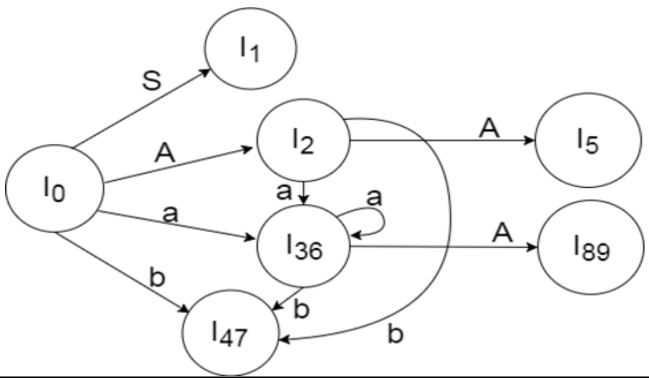
I8= A → aA•, a/b

I9= A → aA•, $

- The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = { A → aA•, a/b/$ }

Drawing DFA:



CLR (1) Parsing:

- CLR refers to canonical lookahead.
- CLR parsing use the canonical collection of LR(1) items to build the CLR(1) parsing table.
- CLR(1) parsing table produces the more number of states as compare to the SLR(1) parsing.
- In the CLR(1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR(1) items
- Draw a data flow diagram (DFA)
- Construct a CLR(1) parsing table

LR(1) item

- LR(1) item is a collection of LR(0) items and a look ahead symbol.

LR(1) item = LR(0) item + look ahead

- The look ahead is used to determine that where we place the final item.
- The look ahead always add $ symbol for the argument production.

Example

CLR(1) Grammar

1. S → AA
2. A → aA
3. A → b

- Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:

- Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` →•S)

- Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` →•S, $
S →•AA, $

- Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` →•S, $
S →•AA, $
A →•aA, a/b
A →•b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S→ A•A, $)

- Add all productions starting with A in I2 State because "." is followed by the non-terminal.
- So, the I2 State becomes

I2= S → A•A, $
A →•aA, $
A →•b, $

I3= Go to (I0, a) = Closure (A→ a•A, a/b)

- Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

I3= A → a•A, a/b
A →•aA, a/b
A →•b, a/b

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b

- Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
  Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

- Add all productions starting with A in I6 State because "." is followed by the non-terminal.
- So, the I6 State becomes

I6 = A → a•A, $
A →•aA, $
A →•b, $

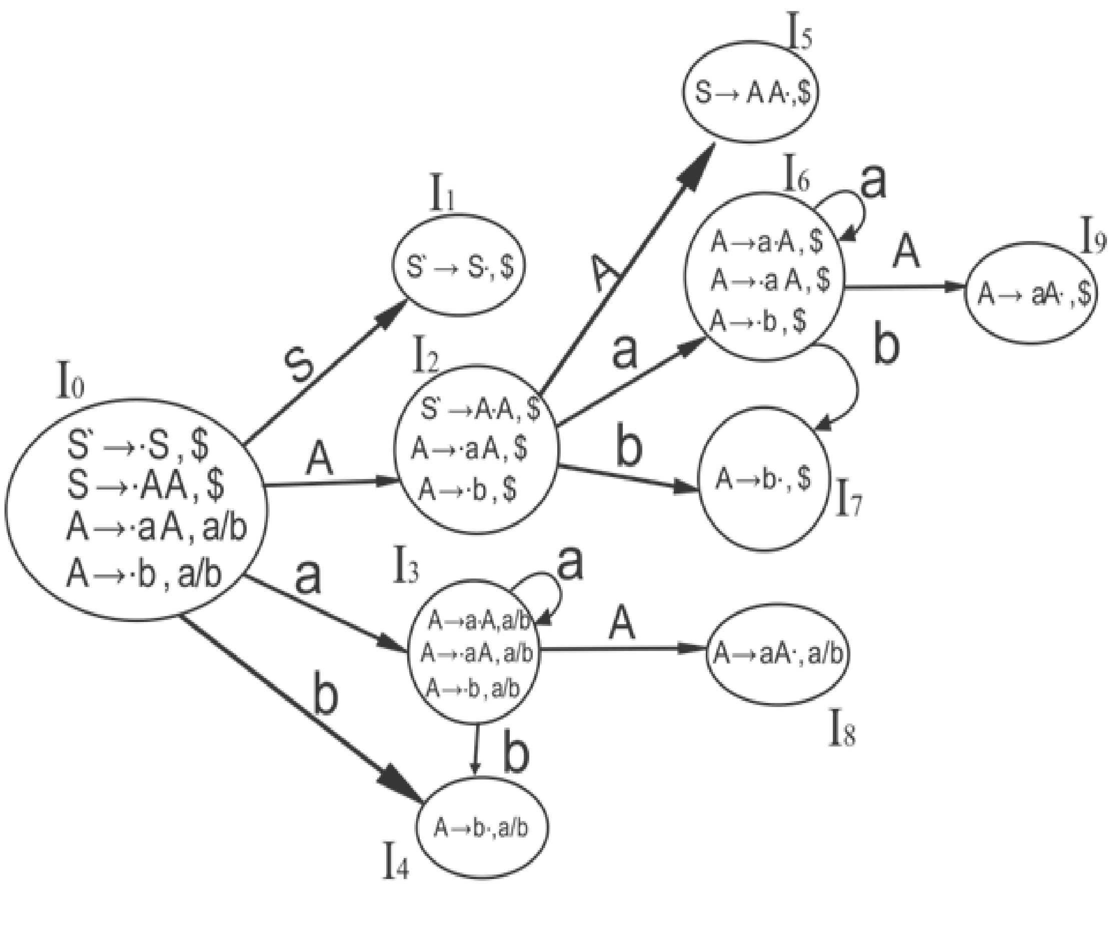
- Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
  Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

Drawing DFA:



P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

CLR (1) Parsing table:

Sates	**Action**			**Go to**	
	a	b	$	S	A
I0	S3	S4		1	2
I1			Accept		
I2	S6	S7			5
I3	S3	S4			8
I4	R3	R3			
I5			R1		
I6	S6	S7			9
I7			R3		
I8	R2	R2			
I9			R2		

Productions are numbered as follows:

- S → AA ... (1)
- A → aA .... (2)
- A → b ... (3)

- The placement of shift node in CLR (1) parsing table is same as the SLR (1) parsing table.
- Only difference in the placement of reduce node.
- I4 contains the final item which drives (A → b•, a/b),
  so action {I4, a} = R3,
  action {I4, b} = R3.

I5 contains the final item which drives (S → AA•, $),
so action {I5, $} = R1.

I7 contains the final item which drives (A → b•,$),
so action {I7, $} = R3.

I8 contains the final item which drives (A → aA•, a/b),
so action {I8, a} = R2,
action {I8, b} = R2.

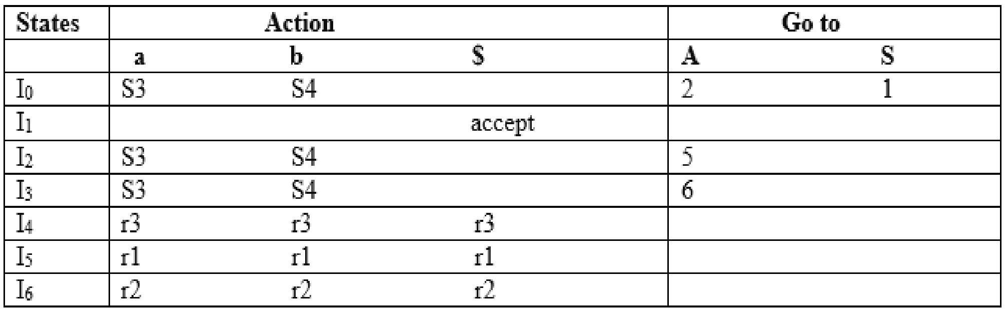
I9 contains the final item which drives (A → aA•, $),
so action {I9, $} = R2.

Shift-Reduce and Reduce-Reduce Conflicts

Shift-Reduce Conflict:

- The Shift-Reduce Conflict is the most common type of conflict found in grammars.
- It is caused when the grammar allows a rule to be reduced for particular token, but, at the same time, allowing another rule to be shifted for that same token.
- As a result, the grammar is ambiguous since a program can be interpreted more than one way.
- This error is often caused by recursive grammar definitions where the system cannot determine when one rule is complete and another is just started.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)



States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

If the grammar is not **SLR**(1), then there may be more than one entry in the parsing LR table.
If both a "**shift**" action and "**reduce**" action occur in the same entry, and the parsing process consults that entry, then a **shift-reduce conflict** is said to occur

The Lookahead Set:

- The Lookahead Set is used by the LALR construction algorithm to determine when to "reduce" a rule.
- When a parsing is complete - (e.g. the . is past the last symbol), the LALR algorithm reduces the rule for each token in the set.
- This information is stored as a series of "reduce" actions in the LALR state.
- When a token is read by the LALR algorithm, it looks up token in the current state and then performs the associated action.
- If an entry exists with a "shift" action, the system will push the token on an internal stack and jump to the specified state.
- If a "reduce" action is found, the associated rule is reduced and passed to the developer.
- If the token is not found in the state, a syntax error occurs.
- Naturally, there can only be one action for any given state.
- For example, if a programming language contains a terminal for the reserved word "while", only one entry for "while" can exist in the state.
- A shift-reduce action, therefore, is caused when the system does not know if to "shift" or "reduce" for a given token.

Example:

Given Grammar
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$
Right-Most Derivation of given input string id+id*id
$E \Rightarrow E+T$
$\Rightarrow E+T*F$
$\Rightarrow E+T*id$
$\Rightarrow E+F*id$
$\Rightarrow E+id*id$
$\Rightarrow T+id*id$
$\Rightarrow F+id*id$
$\Rightarrow id+id*id$

A Stack Implementation of a Shift-Reduce Parser

- There are four possible actions of a shift-parser action:
  1. Shift : The next input symbol is shifted onto the top of the stack.
  2. Reduce: Replace the handle on the top of the stack by the non-terminal.
  3. Accept: Successful completion of parsing.
  4. Error: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker $.

Handle:

- Informally, a "handle" of a string is a substring that matches the right side of the production, and whose reduction to nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation

But not every substring matches the right side of a production rule is handle.

- Formally , a "handle" of a right sentential form γ ($\equiv \alpha\beta\omega$) is a production rule $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

$$S \Rightarrow \alpha A\omega \Rightarrow \alpha\beta\omega$$

  then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta\omega$
- The string ω to the right of the handle contains only terminal symbols.

Example:

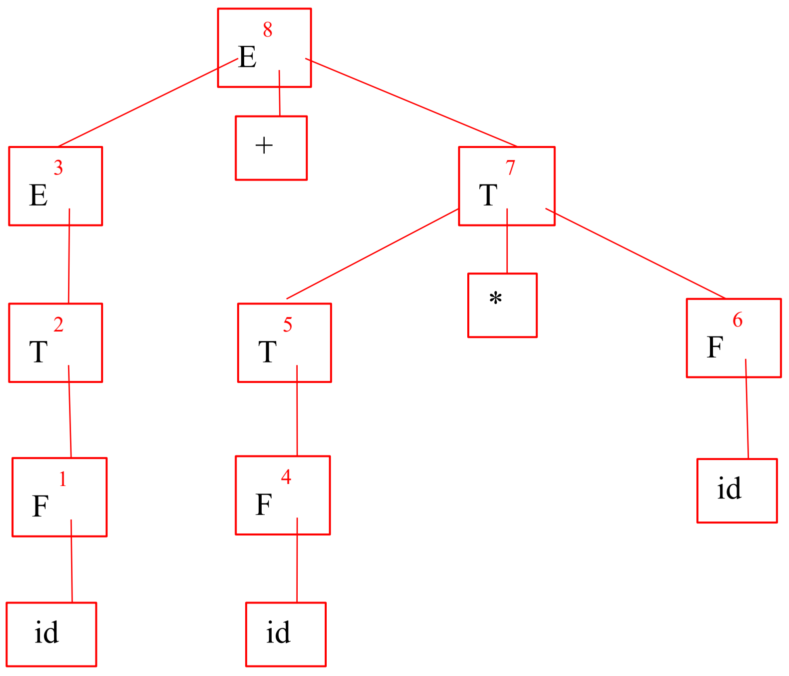
Given Grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Parse Tree



P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

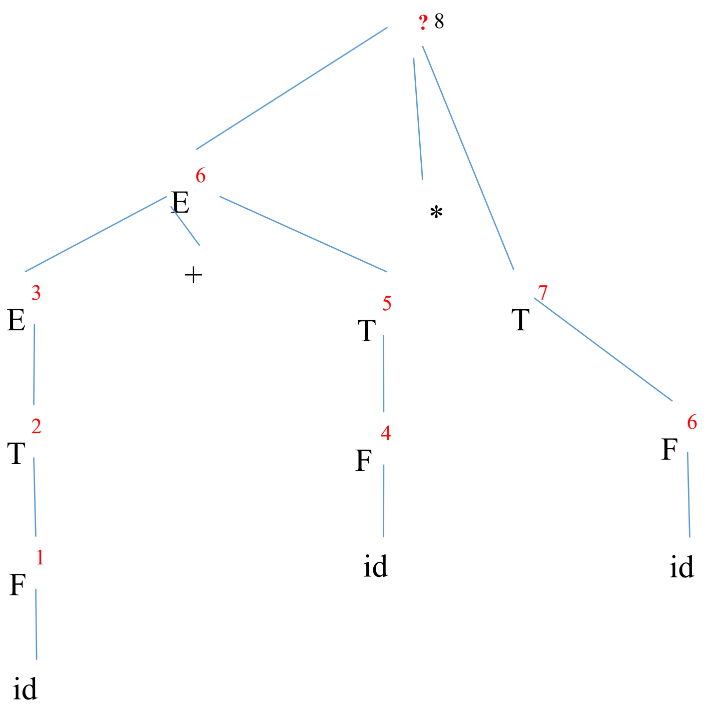
Stack	Input	Action
$	id+id*id$	Shift
$id	+id*id$	Reduce by F→id
$F	+id*id$	Reduce by T→F
$T	+id*id$	Reduce by E→T
$E	+id*id$	Shift
$E+	Id*id$	Shift
$E+id	*id$	Reduce by F→id
$E+F	*id$	Reduce by T→F
$E+T	*id$	**Shift**
$E+T*	id$	Shift
$E+T*id	$	Reduce by F→id
$E+T*F	$	Reduce by T→T*F
$E+T	$	Reduce by E →E+T
$E	$	Accept

Example:

Given Grammar
E → E+T | T
T → T*F | F
F → (E) | id

Parse Tree:

Parse Tree



P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

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
$E+T	*id$	**Reduce by E+T→E**
$**E**	*id$	Shift
$E*	id$	Shift
$E*id	$	Reduce by F→id
$E*F	$	Reduce by T→F
$E*T	$	**Error**

Reduce-Reduce Conflict

- A Reduce-Reduce error is a caused when a grammar allows two or more different rules to be reduced at the same time, for the same token.
- When this happens, the grammar becomes ambiguous since a program can be interpreted more than one way.
- This error can be caused when the same rule is reached by more than one path.

A **reduce/reduce conflict** occurs if there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar.
For **example**, here is an erroneous attempt to define a sequence of zero or more word groupings.

The Lookahead Set:

- The Lookahead Set is used by the LALR construction algorithm to determine when to "reduce" a rule.
- When a parsing is complete - (e.g. the cursor is past the last symbol), the LALR algorithm reduces the rule for each token in the set.
- This information is stored as a series of "reduce" actions in the LALR state.
- When a token is read by the LALR algorithm, it looks up token in the current state and then performs the associated action.
- If an entry exists with a "shift" action, the system will push the token on an internal stack and jump to the specified state.
- If a "reduce" action is found, the associated rule is reduced and passed to the developer.
- If the token is not found in the state, a syntax error occurs.

Example:
Given Grammar
C → AB
A → a
B → a

Stack	Input	Action
$	aa$	Shift
$a	a$	Reduce A→ a or B → a

Resolve in favor of reduce *A* → **a**, otherwise we're stuck!

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Comparison between LR(0), SLR(1), LALR(1) and CLR(1)

- $LR(0) \subset SLR(1) \subset LALR(1) \subset CLR(1)$
- If there is no RR conflict in CLR(1) then there may or may not be RR conflict in LALR(1)
- If there is no SR conflict in CLR(1) then there is no SR conflicts in LALR(1)
- Number of states in SLR(1) and LALR(1) are same, goto moves are identical, shift moves are identical, reduce moves may different else point 1 will never be satisfied.
- LALR(1) and CLR(1) both uses LR(1) items.
- LR(0) and SLR(1) both uses LR(0) items.
- The only difference between LR(0) and SLR(1) is this extra ability to help decide what action to take when there are conflicts.
- Because of this, any grammar that can be parsed by an LR(0) parser can be parsed by an SLR(1) parser.
- However, SLR(1) parsers can parse a larger number of grammars than LR(0).

The main difference of CLR(1) as compared to SLR(1) is in having extra information for deciding REDUCE moves as compared to using only the FOLLOW set.

1. CLR(1) stores extra information in each state
2. The extra information is the set of valid terminals which can cause a REDUCE move
3. This set of valid terminals will be a subset of the FOLLOW(A), where *A*
   a) is the LHS of the production used for reduction
   b) Storing extra information can lead to more number of states in CLR(1) as compared to SLR(1)

Automatic Parser Generator

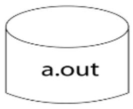
These are some points about YACC:

- YACC is an automatic tool that generates the parser program.
- As we have discussed YACC in the first unit of this tutorial so you can go through the concepts again to make things more clear.
- YACC stands for Yet Another Compiler Compiler.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

Input: A CFG- file.y
Output: A parser y.tab.c (yacc)

- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

The basic operational sequence is as follows:

		
This file contains the desired grammar in YACC format.	This file contains the desired grammar in YACC format.	Executable file that will parse grammar given in gram.Y
C Compiler	It is the c source program created by YACC.	

Handling Ambiguous Grammar : Error Recovery in Parsing

Using Ambiguous Grammar:

- It is a fact that every ambiguous grammar fails to be LR.
- How-ever, certain types of ambiguous grammars are quite useful in the specification and implementation of languages.
- For language constructs like expressions, an ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar.
- Another use of ambiguous grammars is in isolating commonly occurring syntactic constructs for special-case optimization.
- With an ambiguous grammar, we can specify the special-case constructs by carefully adding new productions to the grammar.

Ambiguity:

grammar that produces more than one parse for some sentence is said to be ambiguous grammar.
Example : Given grammar G : E→ E+E | E*E | (E) | id
The sentence id+id*id has the following two distinct leftmost derivations:

E→ E+ E	E→ E* E
E→ id + E	E→E+ E * E
E→ id + E * E	E→ id + E * E
E→ id + id * E	E→ id + id * E
E→ id + id * id	E→ id + id * id

- The two corresponding trees are,

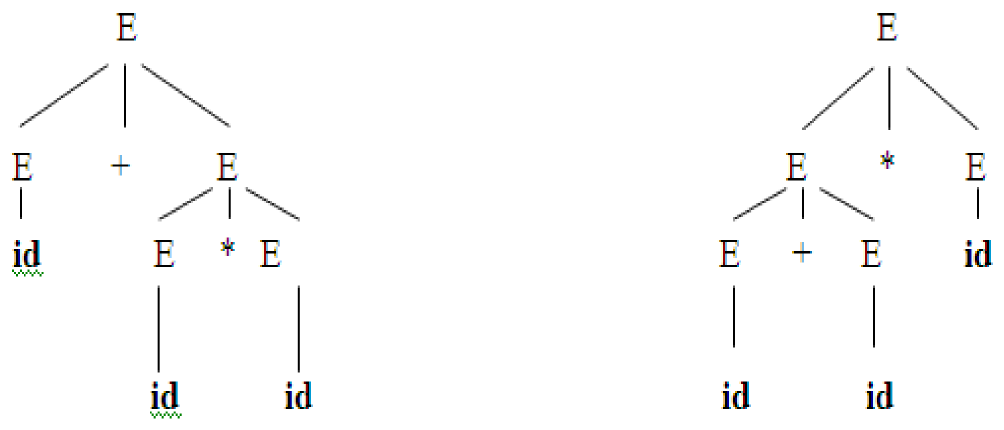


Fig. Two parse trees for id+id*id

- G : E→ E+E | E*E | (E) | id
  E → E + id | id   id + id + id
  E → id   E | id
- In order to define Association, we use Recursion

- G : E→ E+E | E*E | (E) | id
  id + id * id
  E → E + T | T
  E → T * F | F
  F → id
  E → G   F | G
- In order to define Precedence, we use levels

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Error Recovery in Parsing:

- A parser should be able to detect and report any error in the program.
- It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input.
- Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process.
- A program may have the following kinds of errors at various stages:
  1. **Lexical** : name of some identifier typed incorrectly
  2. **Syntactical** : missing semicolon or unbalanced parenthesis
  3. **Semantical** : incompatible value assignment
  4. **Logical** : code not reachable, infinite loop
- There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

- When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon.
- This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

- When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead.
- For example, inserting a missing semicolon, replacing comma with a semicolon etc.
- Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

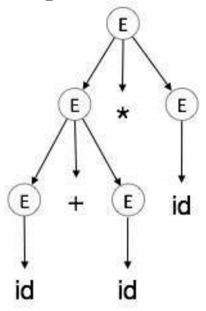
- Some common errors are known to the compiler designers that may occur in the code.
- In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

- The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.
- When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y.
- This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

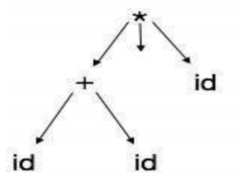
Abstract Syntax Trees:

- Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed.
- Take the following parse tree as an example

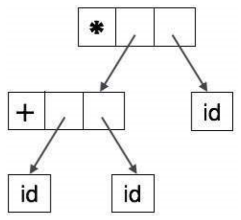


If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase.

- By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information.
ASTs are more compact than a parse tree and can be easily used by a compiler.

Find the operator precedence functions for the following grammar
E-> E+E / E*E / id with input string id + id * id.

Operator grammar
small, but an important class of grammars
we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
In an *operator grammar*, no production rule can have:
ε at the right side
two adjacent non-terminals at the right side.
Ex:

$E \to AB$	$E \to EOE$	$E \to E+E$ |
$A \to a$	$E \to id$	$E*E$ |
$B \to b$	$O \to +\|*\|/$	E/E | id
not operator grammar	not operator grammar	operator grammar

Precedence Relations:

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

a <· b	b has higher precedence than a
a =·b	b has same precedence as a
a ·> b	b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).

Using Operator-Precedence Relations:
The intention of the precedence relations is to find the handle of a right-sentential form,
<· with marking the left end,
=· appearing in the interior of the handle, and
·> marking the right hand.
In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Precedence Functions:

- Compilers using operator precedence parsers do not need to store the table of precedence relations.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b.

f(a) < g(b)	whenever a <· b
f(a) = g(b)	whenever a =· b
f(a) > g(b)	whenever a ·> b

Using Operator -Precedence Relations:

E → E+E | E*E | id

The operator-precedence table for this grammar

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

- Then the input string id+id*id with the precedence relations inserted will be:
  $ <· id ·> + <· id ·> * <· id ·> $

To Find the Handles:

1. Scan the string from left end until the first ·> is encountered.
2. Then scan backwards (to the left) over any =· until a <· is encountered.
3. The handle contains everything to left of the first ·> and to the right of the <· is encountered.

$ <· id ·> + <· id ·> * <· id ·> $	E → id	$ id + id * id $
$ <· + <· id ·> * <· id ·> $	E → id	$ E + id * id $
$ <· + <· * <· id ·> $	E → id	$ E + E * id $
$ <· + <· * ·> $	E → E*E	$ E + E * ·E $
$ <· + ·> $	E → E+E	$ E + E $
$ $		$ E $

Operator-Precedence Parsing Algorithm – Example:

stack	***input***	***action***
$	id+id*id$	$ <· id shift
$id	+id*id$	id ·> + reduce E → id
$	+id*id$	shift
$+	id*id$	shift
$+id	*id$	id ·> * reduce E → id
$+	*id$	shift
$+*	id$	shift
$+*id	$	id ·> $ reduce E → id
$+*	$	* ·> $ reduce E → E*E
$+	$	+ ·> $ reduce E → E+E
$	$	accept