UNIT 4

Syllabus: Syntax Analysis: Types of parsing, Top- Down Parsing: Recursive Descent Parsing, Predictive Parsing, Bottom-Up Parsing: SLR, CLR, LALR

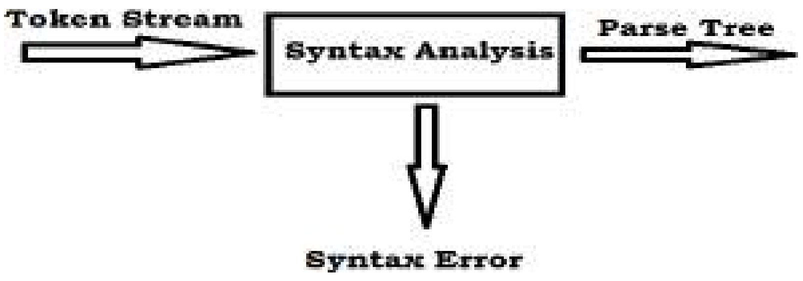
Semantic Analysis: Introduction to Syntax-Directed Definitions, Syntax-Directed Translation, Attributes, and Types of attributes Bottom-up evaluation of attributes

Intermediate-Code Generation: Types of Intermediate-Code, types of Three-Address Code, Evaluation of three address code

==

- This is the second phase of the compiler. It can also be called Syntax analyser / Syntax Phase / Syntax Verification / Parsing /Parse Tree Generator.

- Definition of a Parser: A parsing or syntax analyser is a process that takes the input string w and produces either a parse tree or generates the syntactic errors.



Role of Parser

- The parser obtains a string of tokens from lexical analyser and can verifies that the string of token names can be generated by the grammar for the source language.
- It checks for syntax errors if any and to recover from commonly occurring errors to continue the rest of the program.
- The parser constructs a parse tree and passes to the rest of the compiler for the further processing.
- Error reporting and recovery form is important part of the syntax analyser.
- The error handler in the parser has the following goals -
   i. It should report the presence of errors clearly and accurately.
   ii. It should recover from each error quickly to detect subsequent errors.
   iii. It should not slow down the processing of correct programs.
- This phase is modelled with context free grammars and the structure is recognized with push down automata or table-driven parsers.

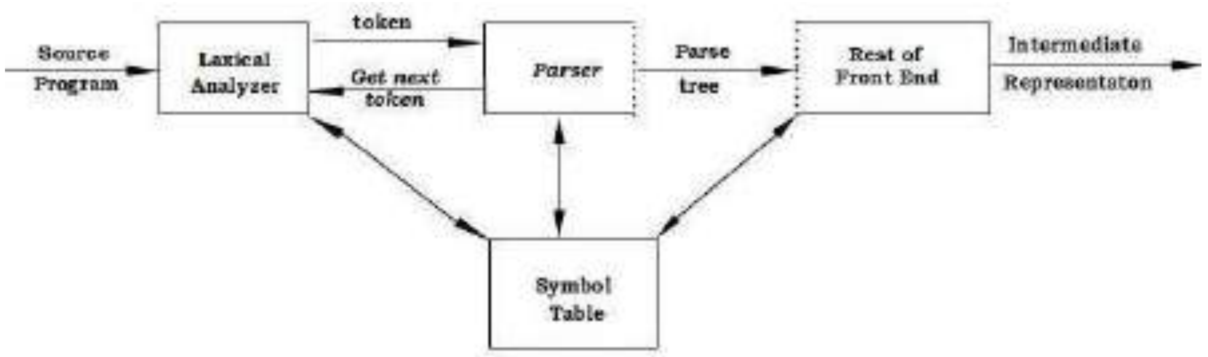


Diagram: **Position of Parser in Compiler Model**

Context Free Grammar

A context free grammar has four components G = (V,T,P,S)

V→A set of non-terminals. Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

T→A set of tokens, known as terminal symbols. Terminals are the basic symbols from which strings are formed.

P→A set of productions. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.

S→A designation of one of the non-terminals as the start symbol, and the set of strings it denotes is the language defined by the grammar.

Derivation: The process of deriving a string / sentence from a grammar is called derivation. It generates strings / sentences of a language.

Input: id + id * id

Given Grammar

E →E + E / E * E / id

It can generate two left most derivations for the sentence id + id + id. E → E + E

→id + E

→ id + E * E

→ id + id * E

→ id + id * id

There are two types of derivations

1. Left Most Derivation (LMD)   2. Right Most Derivation (RMD)

Left Most Derivation The left most derivation is a derivation in which the leftmost non terminal is replaced first from the sentential form.

Eg:

$E \rightarrow E + E$

$\rightarrow id + E$

$\rightarrow id + E * E$

$\rightarrow id + id * E$

$\rightarrow id + id * id$

Right Most Derivation The right most derivation is a derivation in which the right most non terminal is replaced first from the sentential form.

$E \rightarrow E + E$

$\rightarrow E + E * E$

$\rightarrow E + E * id$

$\rightarrow E + id * E$

$\rightarrow id + id * id$

Ambiguous Grammar

Depending on Number of Derivation trees, CFGs are sub-divided into 2 types:

- Ambiguous grammars
- Unambiguous grammars

A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one Left Most Derivation Tree (LMDT) or Right Most Derivation Tree (RMDT).

Definition: G = (V,T,P,S) is a CFG is said to be ambiguous if and only if there exist a string in T* that has more than one parse tree.

where V is a finite set of variables. T is a

finite set of terminals.

P is a finite set of productions of the form, A -> α, where A is a variable and α ∈ (V ∪ T)* S is a

designated variable called the start symbol.

Example:

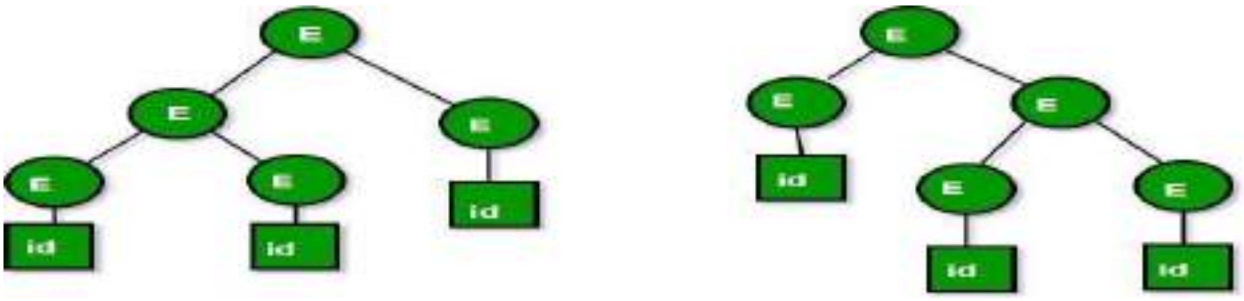
$E \rightarrow E + E$	$E \rightarrow E * E$
$\rightarrow id + E$	$\rightarrow E + E * E$

→ id + E * E → id + E * E

→ id + id * E → id + id * E

→ id + id * id → id + id * id

We can create 2 parse trees from this grammar to obtain a string id+ id*id The following are the 2 parse trees generated by left most derivation:



Both the above parse trees are derived from same grammar but both parse trees are different. Hence the grammar is ambiguous.

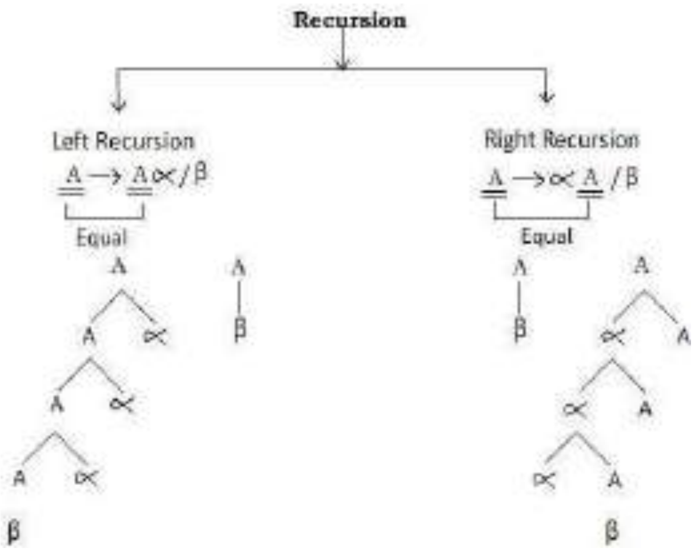
Disambiguate the grammar i.e., rewriting the grammar such that there is only one derivation or parse tree possible for a string of the language which the grammar represents.

Removing Left Recursion

A grammar is left recursive if it has a non-terminal (variable) S such that there is a derivation S -> Sα | β

where α ε(V+T)* and β ε(V+T)* (sequence of terminals and non-terminals that do not start with S)

Due to the presence of left recursion some top down parsers enter into infinite loop so we have to eliminate left recursion.



Language generated βα* Language generated α*β

A( )
{
A ( )
α *α*
}
Infinite loop may occur

A ( )
{
A ( )

}
No Problem of infinite loop

Let the productions is of the form A -> Aα1 | Aα2 | Aα3 | ….. | Aαm | β1 | β2 | …. | βn

Where no βi begins with an A. then we replace the A-productions by A -> β1 A' |

β2 A' | ….. | βn A'

A' -> α1A' | α2A' | α3A'| ….. | αmA' | ε

The nonterminal A generates the same strings as before but is no longer left recursive. Examples

1. E →E + T | T → left recursive 2.
S → S0S1S |01
3. S → (L) | x

Removing Left Factoring

A grammar is said to be left factored when it is of the form –

A -> αβ1 | αβ2 | αβ3 | …… | αβn | γ i.e the productions start with the same terminal (or set of terminals). On seeing the input **α** we cannot immediately tell which production to choose to expand A.

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top down parsing.
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen to make the right choice.

For the grammar A -> αβ1 | αβ2 | αβ3 | …… | αβn | γ The

equivalent left factored grammar will be –

A → αA' | γ

A' → β1 | β2 | β3 | …… | βn

Eg
S → iEtS / iEtSeS S
→iEtSS'
S' →ε / eS
E →b

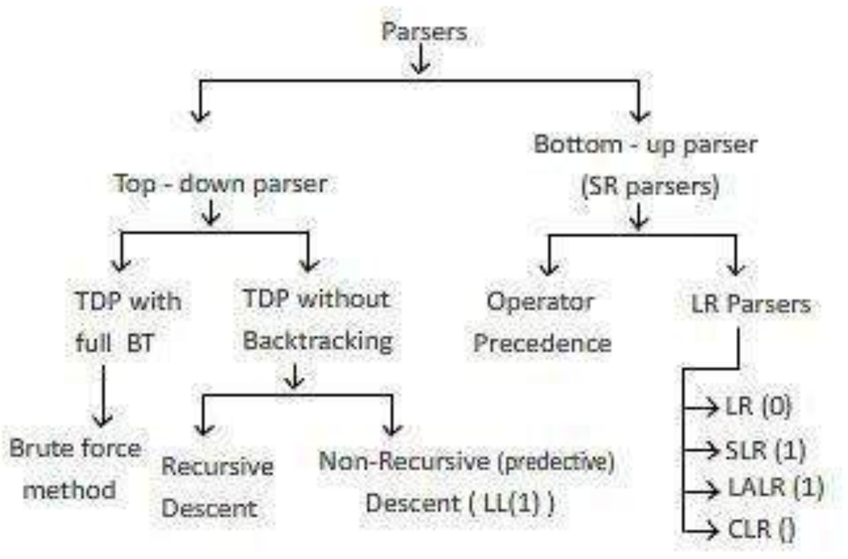
Parsing

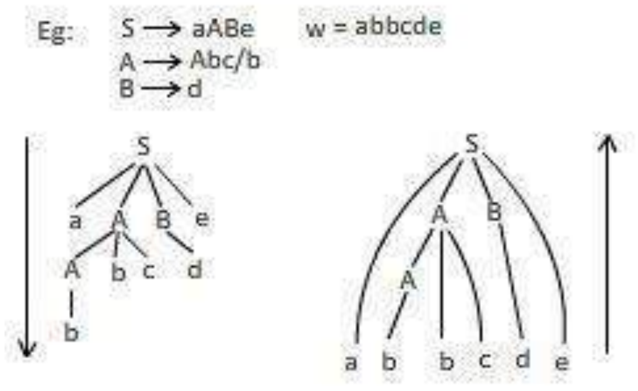
Parsing is the process of analyzing a stream of input to determine its grammatical structure for a given grammar. The task of the parser is to determine if and how the input can be derived from the start symbol within the rules of the formal grammar. This can be done in essentially two ways:

- Top-down parsing - Construction of the parse tree starts at the root /start symbol and proceeds towards leaves / terminals.

- Bottom-up parsing - Construction of the parse tree starts from the leaf nodes / terminals and proceeds towards root / start symbol.

Example



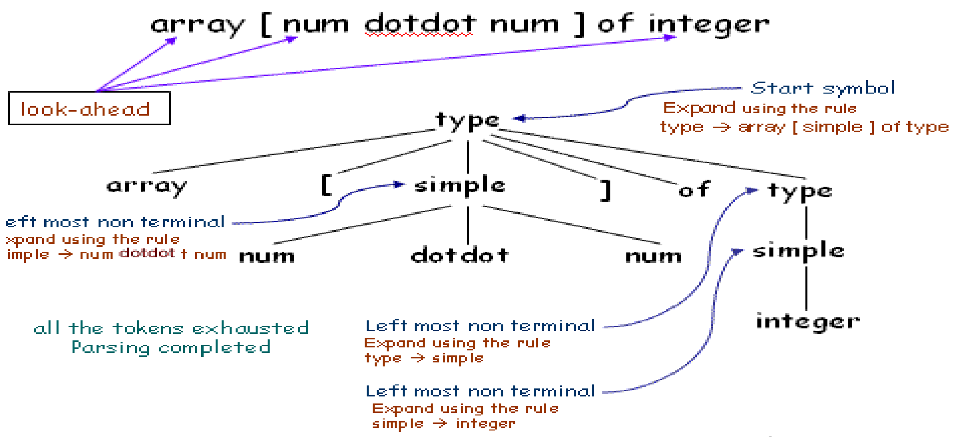


Top down Parser **Bottom up Parser**

- Construction of a parse tree is done by starting the root labelled by a start symbol
- Repeat following two steps
  - At a node labelled with non terminal A select one of the productions of A and construct children nodes.
  - Find the next node at which sub tree is constructed

Parse *array [numdotdotnum] of integer* using the grammar:

- **type $\longrightarrow$ simple**
- **| $\uparrow$ id |**
- **array [simple] of type**
- **simple $\longrightarrow$ integer | char | numdotdotnum**



- Initially, the token *array* is the look ahead symbol and the known part of the parse tree consists of the root, labelled with the starting non- terminal *type*.
- For a match to occur, non-terminal *type* must derive a string that starts with the look ahead symbol *array*. In the grammar, there is just one production of such type, so we select it, and construct the children of the root labelled with the right side of the production.
- In this way we continue, when the node being considered on the parse tree is for a terminal and the terminal matches the look ahead symbol, then we advance in both the parse tree and the input.
- The next token in the input becomes the new look ahead symbol and the next child in the parse tree is considered.

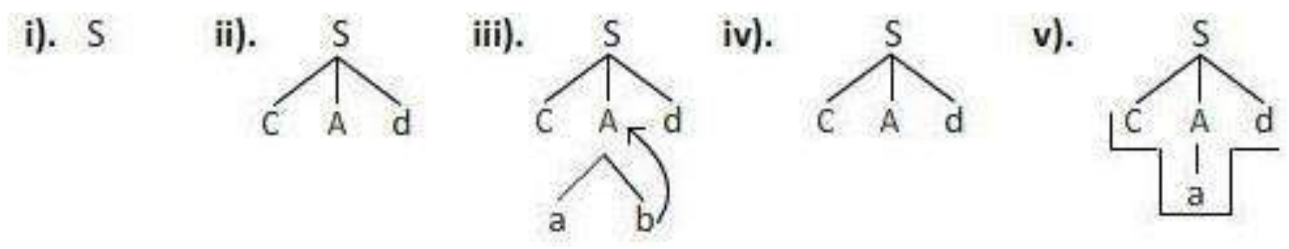
BackTracking

- A back tracking parser will have different production rules to find the match for the string by backtracking each time.
- It is powerful than predictive parsing.
- But it is slower and requires exponential time.
- Hence, it is not preferred for practical compilers.

Example

$S \longrightarrow cAd$

$A \longrightarrow ab \ / \ a$ Input String w =cad



Advantages: It is very easy to implement
Disadvantages: Time consuming

Recursive Descent Parser

- The parser uses collection of recursive procedures for parsing the given input string is called Recursive Descent Parser.
- The context free grammar is used to build recursive routines.
- The RHS of the production rule is directly converted to a program.
- For each non terminal, a separate procedure is written and body of the procedure (code) is RHS of the corresponding non terminal.

Steps for construction of RD Parser

1. If the i/p symbol is non terminal, call a procedure corresponding the non terminal is made.
2. If the output symbol is terminal then it is matched with the look ahead from input. The look ahead pointer points to next symbol.
3. If the production rule has many alternatives, then combine them into a single body of procedure.
4. The parser should be activated by a procedure corresponding to the start symbol.

Example:

$E \longrightarrow iE'$
$E' \longrightarrow +i\,E' \mid \varepsilon$

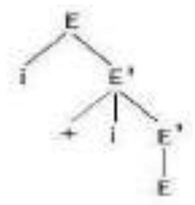
Procedures

```
E( )
{
        if (l == 'i')
        {
        match (  'i ' );
        E' ( );

        }

}
```

```
E'( )
{
        if (l == '+')
        {
match ('+');
        match ('i');
        E' ( );
        }
else return;
}
```

```
Match (char t)
{
        if (l  == t)
        l  =  ggetcvhar();
        else
        printf ("ERROR");
}
```

```
main( )
{
E ( );
if( l ==  '$')
printf(" Parsing Successful  ");

}
```



Recursive Descent Parser

Shift Reduce Parser

The parser constructs a parse tree from leaves to root. Thus, it works on the same principle of bottom up parser. The parser requires the following data structures.

1. Input Buffer: It stores the input string.
2. Stack: It is used to store and access the LHS and RHS of rules.

Initial configuration of Shift Reduce Parser is as follows

Stack

$	S

Input Buffer

w	$

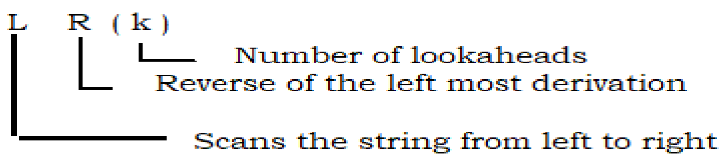
Parsing Algorithm (Actions)

1. Shift: Moving of the symbols from input buffer onto the stack. This action is called reduce action.
2. Reduce: If the handle appears on the top of the stack then reduction of it by appropriate rule. That means RHS of rule is popped off and LHS is pushed in. This action is called reduce action.
3. Accept: If stack contains start symbol(S) and input buffer is empty ($) then the action is called accept. When accept state is obtained in the process of parsing, it means that a successful parsing is done.
4. Error: It is a situation in which parser can't either shift or reduce the symbols, it cannot even perform the accept action is called error.

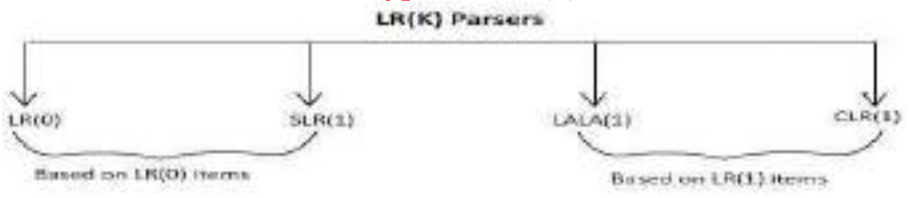
Handle: The RHS of a production is called Handle. Substitution of RHS that matches the right side of the production is called Handle.

Handle Pruning: In the derivation, replacing the handle with its LHS of a production is called Handle Pruning.

LR(k) Parsers(Bottom up parser)

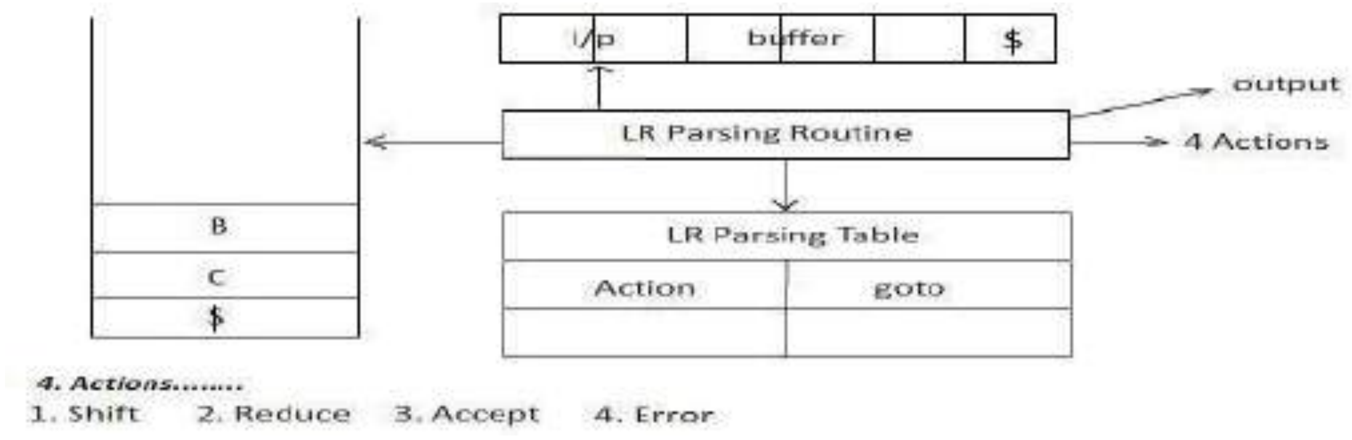

L R (k)
Number of lookaheads
Reverse of the left most derivation
Scans the string from left to right

Types of LR (k) Parsers

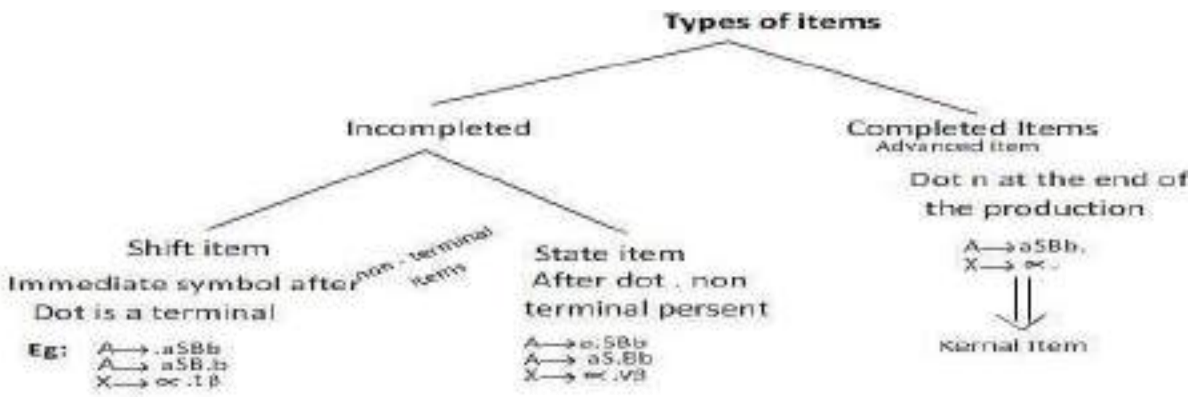


- Every LR (0) parser is SLR (1) parser but may not be reverse.
- Every SLR (1) parser is LALR (1) parser but may not be reverse.
- Every LALR (1) parser is CLR (1) parser but may not be reverse.
- Every LL (1) parser is also LR (1) but every LR (1) need not be LL (1).

Bottom up parsing Configuration



Types of Iterms



Handle: RHS of a production is said to be handle.

A → *a AB e*

LHS Handle

Closure ( ): When there is a dot on left of a variable then add all the productions of that variable.

S' → .*S*

S → .*AA*

A → .*aA*

A → . *b*

Go to ( ) : Moving the dot from the left of the variable to its right

go to (s) = S' → *S*.

LR (0) Parser

- LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars. In the LR parsing, "L" stands for left-to-right scanning of the input.
  ... LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.
  - 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.
  - To construct a parsing table for LR (0) and SLR (1), we use canonical collection of LR (0) items.
  - To construct a parsing table for LALR (1) and CLR (1), we use canonical collection of LR (1) items.

Eg: Construct LR (0) parsing table for the given grammar

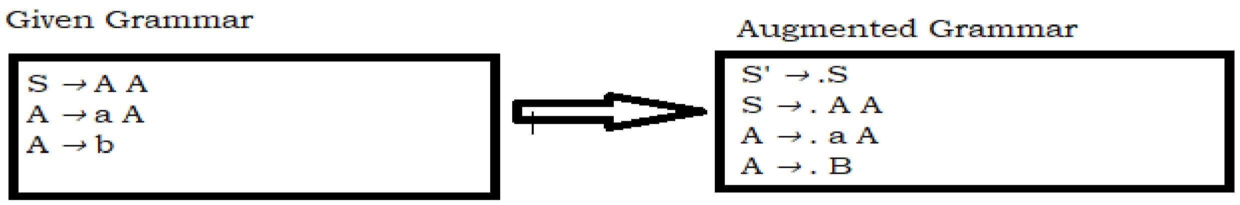
S → *A A*

A → *a A*

A → *b*

Solution:

Step 1: Constructing augmented grammar.



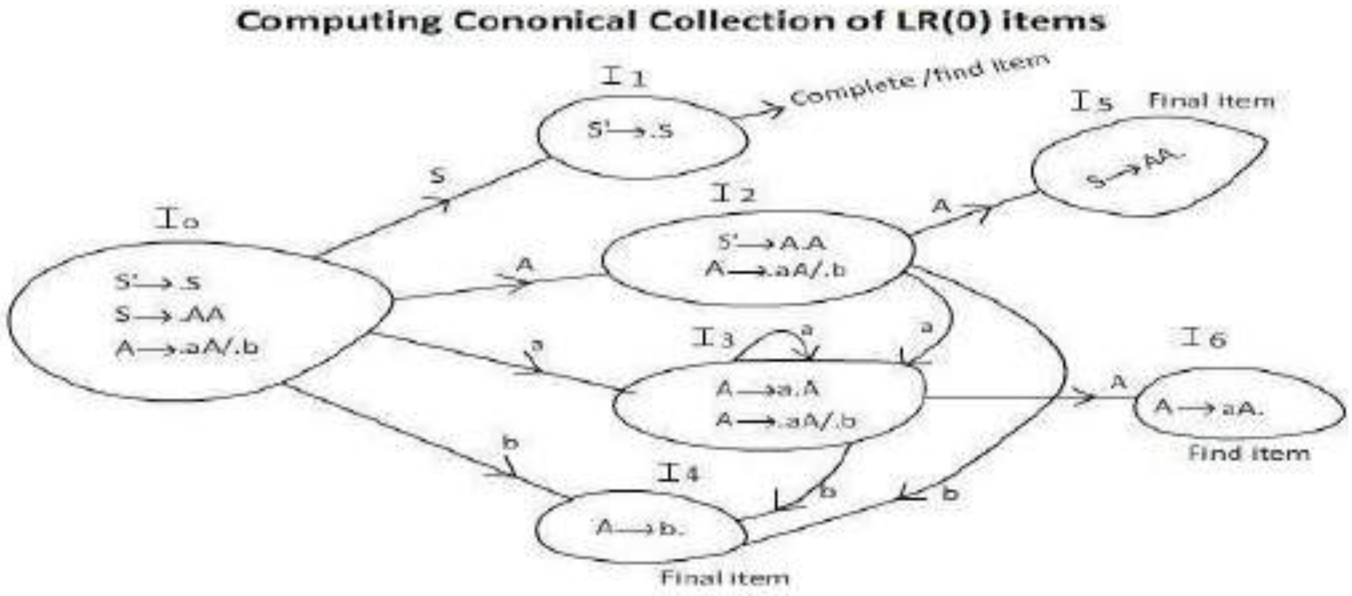
Given Grammar

S → A A
A → a A
A → b

Augmented Grammar

S' → .S
S → . A A
A → . a A
A → . B

Step 2:

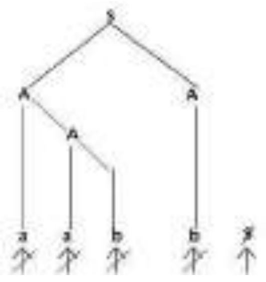


Note: States in the FA is the number of rows in the parsing table. Step 3:

Constructing parsing table for LR (0) parser

States	Action Part			Go to Part	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5	R1	R1	R1		
6	R2	R2	R2		

Step 4: Parsing the input string w = a a b b $



Stack:

$	~~a~~	~~3~~	~~a~~	~~3~~	~~b~~	~~4~~	~~A~~	~~6~~	~~A~~	~~6~~	~~A~~	~~2~~	~~b~~	~~4~~	~~A~~	~~5~~	S	1

SR Conflicts: LR(0): Puts reduce production in the entire row, for every terminal. If some column in the action part (a terminal) contains both, shift and reduce moves, it's a "shift-reduce" (s/r) conflict.
Reduce-Reduce Conflicts. 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.

SLR (1) Parser

- The SLR parser is similar to LR(0) parser except that the reduced entry. The reduced productions are written only in the FOLLOW of the variable whose production is reduced.

Construction of SLR parsing table –

- Construct C = { I_0, I_1, ……. I_n}, the collection of sets of LR(0) items for G'.
- State i is constructed from Ii. The parsing actions for state i are determined as follow :
  - If [A -> ?.a?] is in I_i and GOTO(I_i , a) = I_j , then set ACTION[i, a] to "shift j". Here a must be terminal.
  - If [A -> ?.] is in I_i, then set ACTION[i, a] to "reduce A -> ?" for all a in FOLLOW(A); here A may not be S'.
  - Is [S -> S.] is in I_i, then set action[i, $] to "accept". If any conflicting actions are generated by the above rules we say that the grammar is not SLR.
- The goto transitions for state i are constructed for all non-terminals A using the rule:
  if GOTO(I_i , A) = I_j then GOTO [i, A] = j.
- All entries not defined by rules 2 and 3 are made error.

Eg: Construct SLR (1) parsing table for the given grammar S $\rightarrow A\ A$

$A \rightarrow a\ A$

$A \rightarrow b$

Solution:

Step 1: Constructing augmented grammar.

S' $\rightarrow .S$

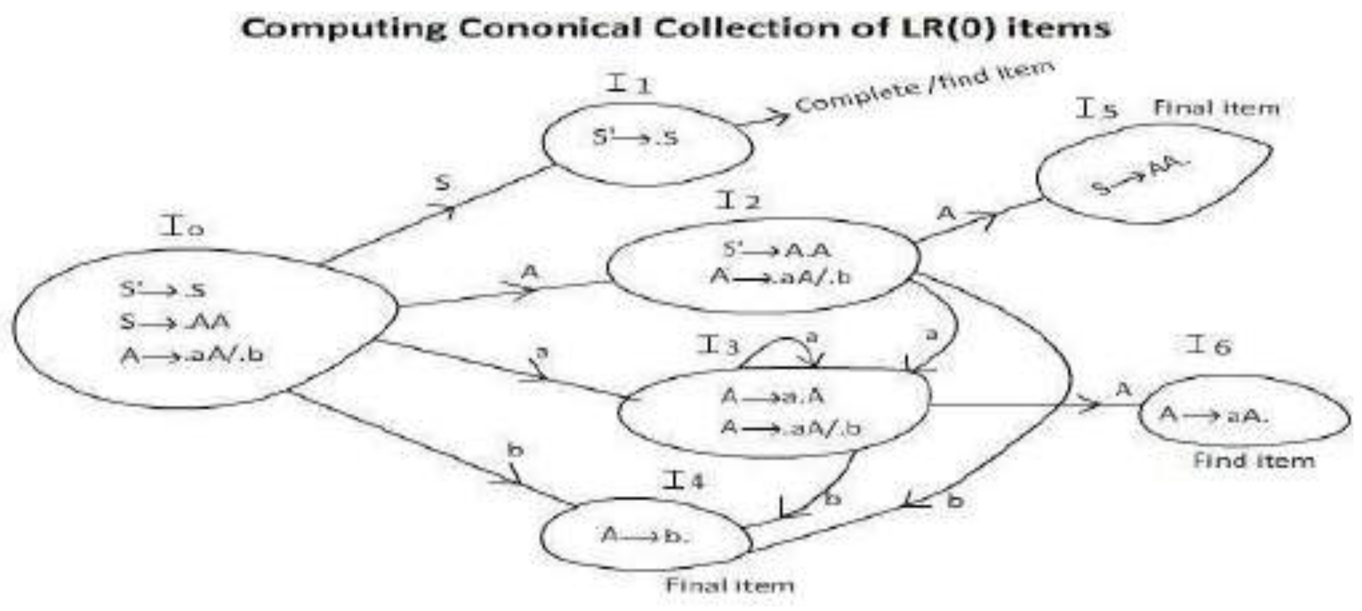
$S \rightarrow .\ A\ A$

$A \rightarrow .a\ A$

$A \rightarrow .b$

Step 2: Computing LR (0) items

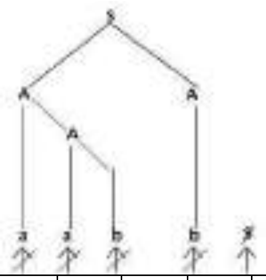
S



Step 3: Constructing Parsing table for SLR(1) Parser

States	Action Part			Go to Part	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Step 4: Parsing the string w = a a b b



$	~~a~~	~~3~~	~~a~~	~~3~~	~~b~~	~~4~~	~~A~~	~~6~~	~~A~~	~~6~~	~~A~~	~~2~~	~~b~~	~~4~~	~~A~~	~~5~~	S	1

Thus the string is parsed.

CLR (1) Parser

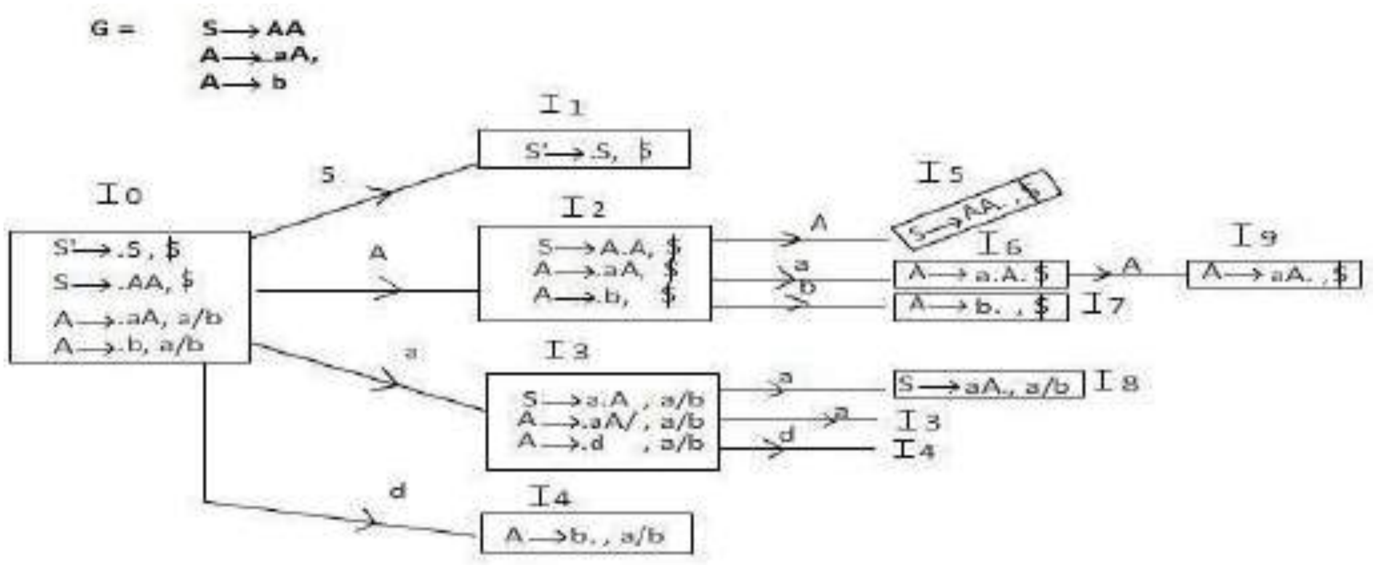
- CLR refers to canonical look-a-head. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table.
- CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.
- In the CLR (1), we place the reduce node only in the look-a-head symbols.
- In CLR parsing we will be using LR(1) items. LR(k) item is defined to be an item using look-a-heads of length k. So, the LR(1) item is comprised of two parts : the LR(0) item and the look-a-head associated with the item. LR(1) parsers are more powerful parser.

Construction of CLR parsing table-

Input – augmented grammar G'

1. Construct C = { I_0, I_1, ……. I_n} , the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow :
   i) If [A -> ?.a?, b] is in I_i and GOTO(I_i , a) = I_j, then set ACTION[i, a] to "shift j". Here a must be terminal.
   ii) If [A -> ?. , a] is in I_i , A ≠ S, then set ACTION[i, a] to "reduce A -> ?".
   iii) Is [S -> S. , $] is in I_i, then set action[i, $] to "accept".
   If any conflicting actions are generated by the above rules we say that the grammar is not CLR.
3. The goto transitions for state i are constructed for all non-terminals A using the rule: if GOTO(I_i, A) = I_j then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.

Eg.Construct CLR (1) Parsing Table for the Given Grammar

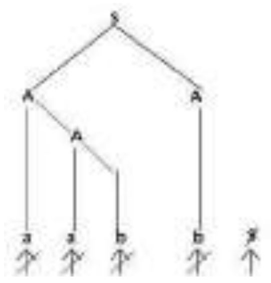


Parsing Table for CLR(1)

States	Action Part			Go to Part	
	a	b	$	S	A

0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6					9
7			R3		
8	R2	R2			
9			R2		

Parsing the input string w = a a b b $



Stack:

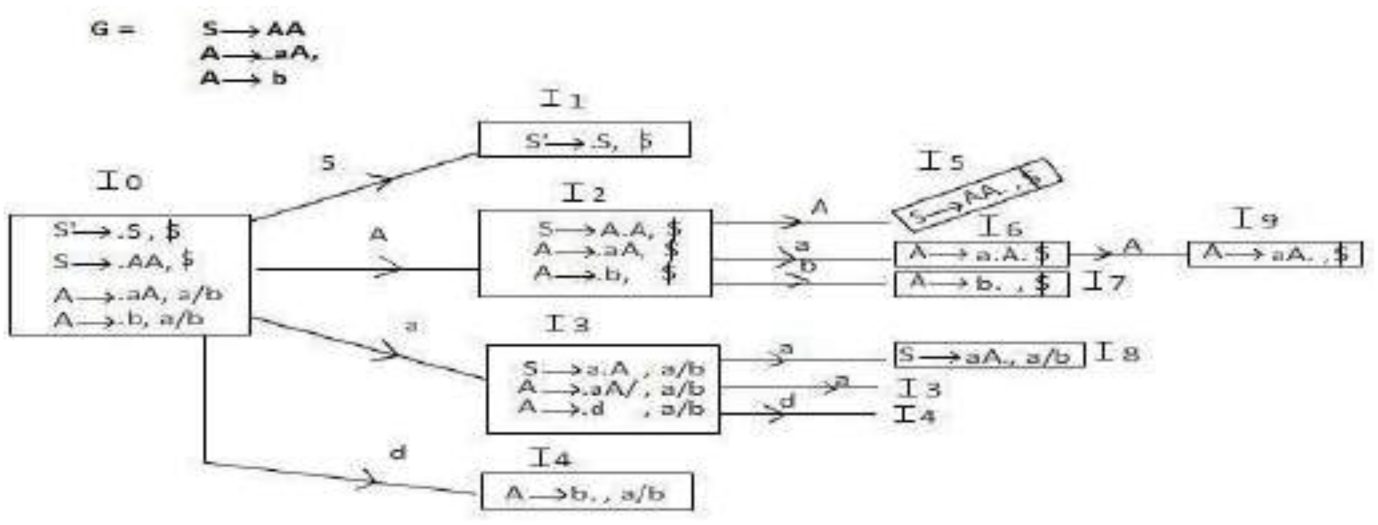
$	~~0~~	~~a~~	~~3~~	~~a~~	~~3~~	~~b~~	~~4~~	~~A~~	~~8~~	~~A~~	~~8~~	~~A~~	~~2~~	~~b~~	~~7~~	~~A~~	~~5~~	S	1

SR Conflict: If a state has one reduction and there is a shift from that state on a terminal same as the look-a-head of the reduction then it will lead to multiple entries in parsing table thus a conflict. LALR parser is same as CLR parser with one difference.

LALR (1) Parser

1. LALR means look-ahead parser. It is a bottom up parser. The idea of the parser is to build LALR parsing table.
2. LALR parser slightly less powerful than CLR but more powerful than SLR parser.
3. Number of rows in LALR parser is almost equal to SLR and less than or equal to CLR parser.
4. CLR Parser avoids conflicts in the parsing table. But it produces a greater number of states when compared to SLR parser. Hence more space is occupied by the table in memory.
5. Since it is as efficient as CLR parser, LALR parsing can be used. The parser obtains smaller size parsing tables than CLR parse tables.
6. This is because LALR parse tables are constructed from LR (1) items.
7. The LR (1) items that have same productions but different look-a-heads are combined to form a single set of items.
8. Shift-Reduce conflicts may not be taken place but Reduce-Reduce conflicts may occur.

Eg: Construct LALR (1)Parsing table for the grammar given



In the above diagrams, some states have got the same items. They can be merged. For example, I_3 and I_6 have same items. They can be merged asI_{36}.

In the same way, I_4 and I_7have the same items. They can also be merged as I_{47}

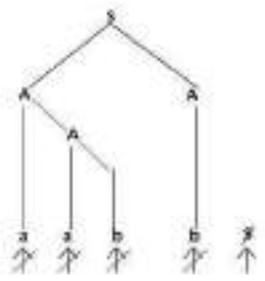
Similarly, the states I_8 and I_9 differ only in their look-a-heads. Hence I_8 and I_9 are combined to form the state I_{89}.

LALR (1) Parsing Table

States	Action Part			Go to Part	
	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

Thus, the resultant LALR parse table after merging states 3 and 6, 4 and 7, 8 and 9 is formed from LR(1) items.
The merger states can never produce a shift-reduce conflicts. However, it can produce a RR conflict.
The number of rows has been reduced from 10 to 7 rows.

Parsing the input string w = a a b b $



Stack:

$	~~0~~	~~a~~	~~36~~	~~a~~	~~36~~	~~b~~	~~47~~	~~A~~	~~89~~	~~A~~	~~89~~	~~A~~	~~2~~	~~b~~	~~47~~	~~A~~	~~5~~	S	1

RR Conflict: This reduces the power of the parser because not knowing the look-a-head symbols can confuse the parser as to which grammar rule to pick next, resulting in reduce/reduce conflicts. All conflicts that arise in applying a LALR(1) parser to an unambiguous LR(1) grammar are reduce/reduce conflicts.

Syntax Directed Definition

A *syntax-directed definition* (SDD) is a context-free grammar with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

Example

- If *X* is a symbol and *a* is one of its attributes, we write *X. a* to denote the value of a ata particular parse-tree node labeled *X.* Attributes may be numbers, types, table references or strings.

Syntax Directed Definition: It is a special form of is a kind of abstract specification. Productions with semantic rules are known as Syntax Directed Definition.

Eg: E→E1+TE.value=E1.value|T.value|+

Syntax Directed Translations: Productions with semantic actions embedded with production bodies.

Eg:E→E1+T{print'+'}

A parser builds parse trees in the syntax analysis phase. The plain parse-tree is of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar that makes the rule so the languages do not accommodate how to interpret them

Example: E → E +T

The above CFG production has no semantic rule associated with it and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types and their relations with each other. Semantic analysis help whether the syntax structure constructed in the source

program derives any meaning or not.
Grammar with Semantic rules is called Syntax Directed Definition.

SyntaxDirectedDefinitions=CFG+Semanticrules

For example: int a="value";

Should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but Semantic analysis should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

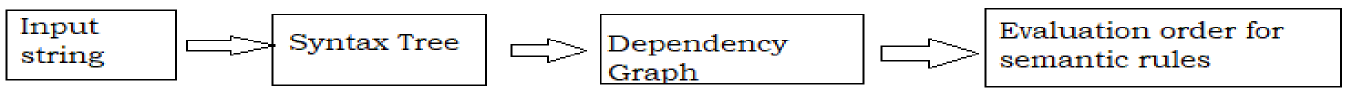
Semantic Errors

Wehavementionedsomeofthesemanticserrorsthatthesemanticanalyserisexpectedto recognize:

- *Type mismatch*
- *Undeclared variable*
- *Reserved identifier misuse.*
- *Multiple declaration of variable in a scope.*
- *Accessing an out of scope variable.*
- *Actual and formal parameter mismatch.*

TheSDDisakindofabstractspecification.SyntaxDirectedDefinitionsisanaugmentedcontext free grammar generated. It means the set of attributes are associated with each terminal and non-terminal. The attributes may be a number, string, memory location or a type.

The conceptual view of syntax directed translation is like this.



Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information means some attributes is appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help to specify the syntax and semantics of a programming anguage. Attribute grammar can pass values or information among the nodes of a tree.

Example:
E→E+T{E.value=E.value+T.value}

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added to get her and the result is copied to then on-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values.

They can be broadly divided into two categories: synthesized attributes and inherited attributes.

Synthesized attributes

The parent node attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S→ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

E → E + T, the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

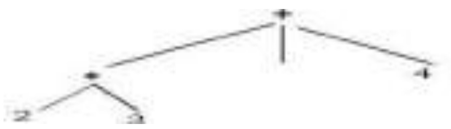
Construct Syntax Directed Translation for the given

grammarE → E+T
E→T
T → T *
FT →F
F→ num

Solution
Step 1: Context free grammar + semantic actions are said to be syntax Definition Translation.
Semantic actions are to be written in between curly braces.

Contest Free Grammar	Semantic Actions
E →E +T	{E.value=E.value+T.value}
E→ T	{E.value=T.value}
T → T * F	{T.value=T.value*T.value}
T→ F	{T.value=F.value}
F→ num	{F.value=num.lexicalvalue}

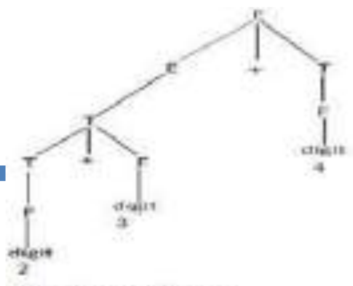
Step2AbstractSyntaxTree
Abstract syntax tree means building a tree without symbols.



Abstract Syntax

TreeStep3: Constructing Parse Tree for the above

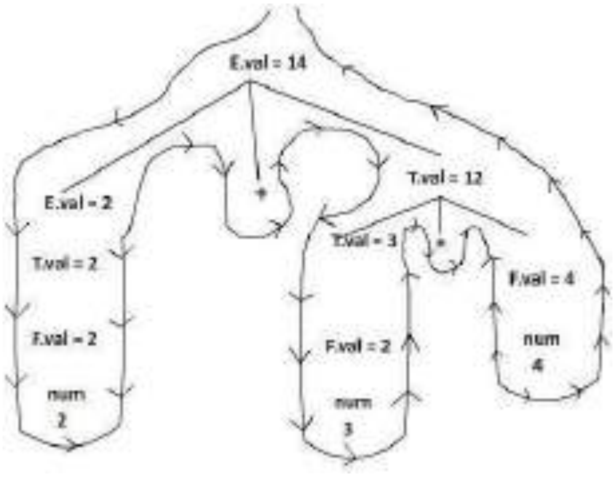
grammar



Parse Tree

Step4:Annotated Parse Tree/Decorated Parse Tree: The syntax directed definition is to be written with suitable semantic action for corresponding production rule of the given grammar.

The annotated tree can also be termed as decorated parse tree.



Annotated Parse Tree

Inherited attributes

Unlike synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S→ABC

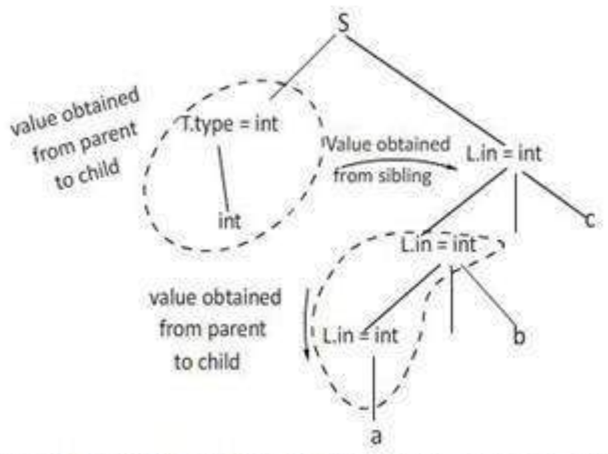
A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B

Annotate the parse tree for the computation of inherited attributes for the given string int a, b, c.

Solution

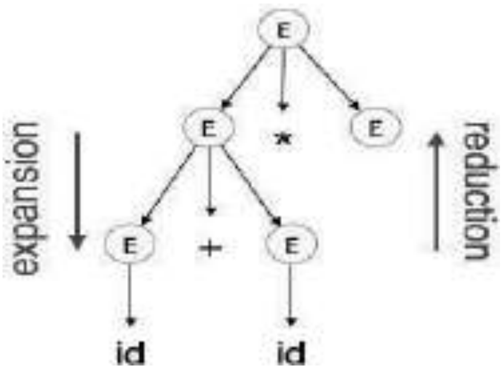
Step1. The Syntax Directed Definition for the given grammar.

Productions	SemanticRules
S→TL	L.in.T.type
T→ int	T.type =int
T→float	T.type =float
T→char	T.type=char
T →double	T.type =double
L →L1,id	L1.in=L.in
L →id	Id.entry=L.inh


S
value obtained from parent to child
T.type = int
int
Value obtained from sibling
L.in = int
c
L.in = int
value obtained from parent to child
L.in = int
b
a

Annotated Parse Tree showing values at each node

Expansion: When a non-terminal is expanded to terminals as per a grammaticalrule



Reduction: When a terminal is reduced to its corresponding non-terminal according to grammar rules, Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules.

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST Abstract Syntax Tree from its previous stage syntax analysis.

Semantic analyzer attaches attribute information with AST, which are called AttributedAST.

Attributes are two tuple values, <attribute name, attribute value>int

value =5;

<type,“integer“>

<presentvalue,“5”>

For every production, we attach a semantic rule.

Dependency Graph

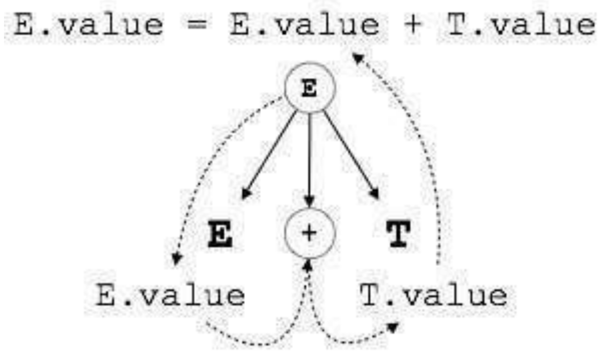
The basic idea behind dependency graphs is for compiler to look for various kinds if dependence among statements to prevent their execution in wrong order i.e. the order that changes the meaning of the program. This helps it to identify various parallel is able components in the program.

For rule X → YZ the semantic action is given by X.x = f(Y.y, Z.z) then synthesized attribute is X.x and X.x depends upon attributes Y.y and Z.z

S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production.

- Sattributeddefinitionisonesuchclassofsyntaxdirecteddefinitionwithsynthesizedattributesonly.
- Synthesized attributes are evaluated in bottom up fashion.
- A stack is maintained to keep track of values of synthesized attributes associated with the grammar symbols on its stack. This stack is often termed as parser stack.



As in the above diagram, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

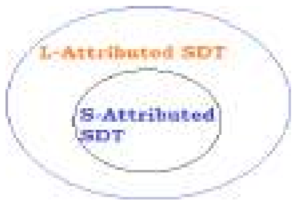
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

Example S→ ABC

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



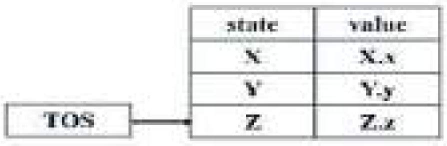
We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

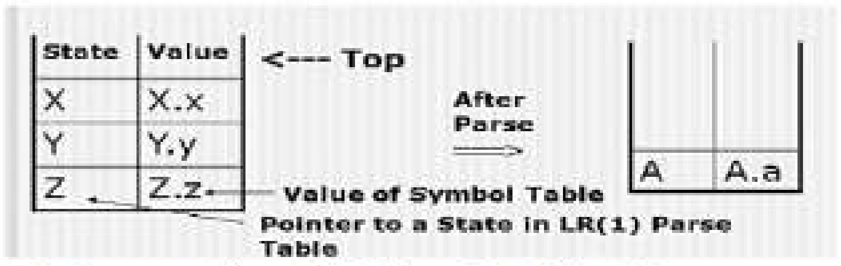
Intermediate Code: Compilers generate easy to represent form of source language that is called intermediate language.

Bottom up Evaluation of Attributes:

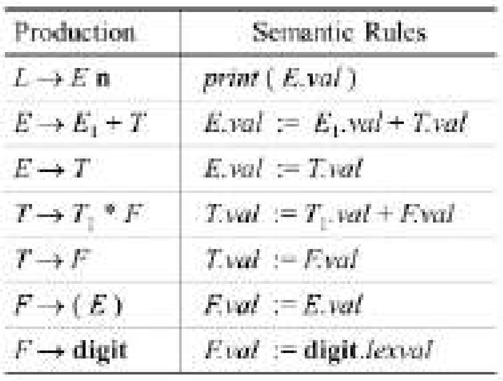
- Syntax-directed definitions with only synthesized attributes(S- attribute) can be evaluated by a bottom up parser (BUP) as input is parsed
- In this approach, the parser will keep the values of synthesized attributes associated with the grammar symbol on its stack.
- The stack is implemented as a pair of state and value.
- When a reduction is made ,the values of the synthesized attributes are computed from the attribute appearing on the stack for the grammar symbols
- implementation is by using an LR parser (e.g. YACC)

• **e.g. A => XYZ and A.a := f (X.x, Y.y, Z.z)**



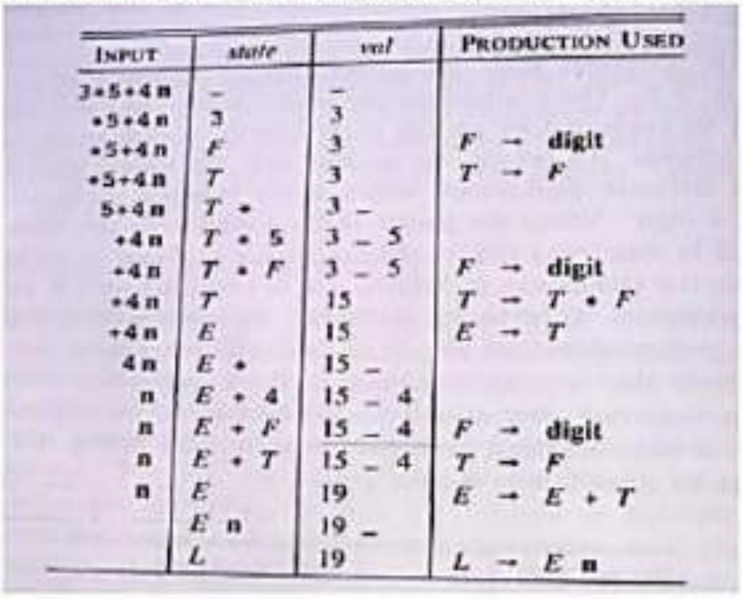


Consider again the syntax-directed definition of the desk calculator.



Production	Semantic Rules
$L \rightarrow E$ **n**	*print* (*E.val*)
$E \rightarrow E_1 + T$	*E.val* := E_1*.val* + *T.val*
$E \rightarrow T$	*E.val* := *T.val*
$T \rightarrow T_1 * F$	*T.val* := T_1*.val* + *F.val*
$T \rightarrow F$	*T.val* := *F.val*
$F \rightarrow (E)$	*F.val* := *E.val*
$F \rightarrow$ **digit**	*F.val* := **digit**.*lexval*

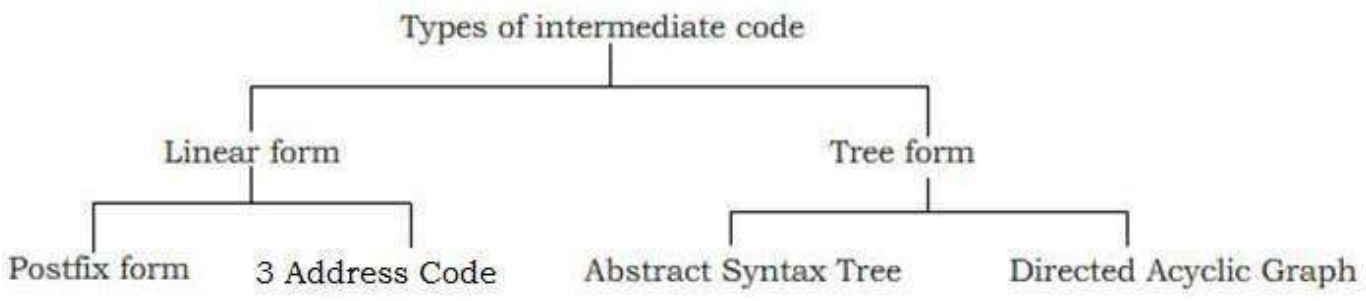
- The following Figure shows the moves made by the parser on input 3*5+4n.
  - Stack states are replaced by their corresponding grammar symbol;
  - Instead of the token digit the actual value is shown.



INPUT	state	val	PRODUCTION USED
3*5+4n	-	-	
*5+4n	3	3	
*5+4n	F	3	$F \rightarrow$ digit
*5+4n	T	3	$T \rightarrow F$
5+4n	T *	3 _	
+4n	T * 5	3 _ 5	
+4n	$T * F$	3 _ 5	$F \rightarrow$ digit
+4n	T	15	$T \rightarrow T * F$
+4n	E	15	$E \rightarrow T$
4n	E +	15 _	
n	E + 4	15 _ 4	
n	$E + F$	15 _ 4	$F \rightarrow$ digit
n	$E + T$	15 _ 4	$T \rightarrow F$
n	E	19	$E \rightarrow E + T$
	E n	19 _	
	L	19	$L \rightarrow E$ n

17

Types of Intermediate code



Intermediate code is broadly divided into two forms - Linear form and Tree form.

Linear form is again divided into two types – Post fix form and 3 address code.

Tree form is also divided into two types –Abstract Syntax Tree and Directed Acyclic Graph (DAG).

Postfix or Polish Notation: In this form the operator is associated with the corresponding operands. This is the most natural way of representation in expression evaluation. In this notation the operator occurs fir and then operands are placed.

Example: (a +b) * (c + d)

ab +c d + *

Construct the given expression into postfix notation / Polish notation

Expression: (a + (b * c)) ^ d – e / (f + g)

t1 = b c *

t2 = a t1 +

t3 = t2 d ^

t4 = f g +

t5 = e t4

t6 = t3 t5 –

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

- Each instruction should have maximum three addresses.
- Each instruction should have only one operator at its right side.

Implementation of Three Address Code

There are 3 representations of three address code.

1. Quadruple
2. Triples
3. Indirect Triples

Quadruple

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantages

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example Consider expression a = b * – c + b * – c.

The three address code is:

t1 = uminus c
t2 = b * t3
t3 = uminus * t3
t4 b * t4
t5 = t2 + t4
a = t5

#	op	Arg1	Arg2	result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	T4
(4)	+	22	t4	t5
(5)	=	t5		a

Quadruple Representation

Triples

This representation doesn’t make use of extra temporary variable to represent a single operation instead when a reference to another triple’s value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example Consider expression a = b * – c + b * – c

	#	op	Arg1	Arg2
	(0)	uminus	c	
	(1)	*	b	B
	(2)	uminus	c	
	(3)	*	b	B
	(4)	+	(1)	(3)
	(5)	=	a	(4)

Triples Representation

Indirect Triples

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example Consider expression a = b * – c + b * – c

List of pointers to the table

#	op	Arg1	Agr2		Pointers	op
(1)	uminus	C			100	(1)
(2)	*	(14)	B		200	(2)
(3)	uminus	C			300	(3)
(4)	-	(16)	B		400	(4)
(5)	+	(15)	(17)		500	(5)
(6)	=	a	(18)		600	(6)

Abstract Syntax Tree: It is condensed / compact version of parse tree. In this tree nosymbols are present except terminals.

Construct an Abstract Syntax Tree for the given expression

Input string x = - a * b + - a * b

Abstract Syntax Tree

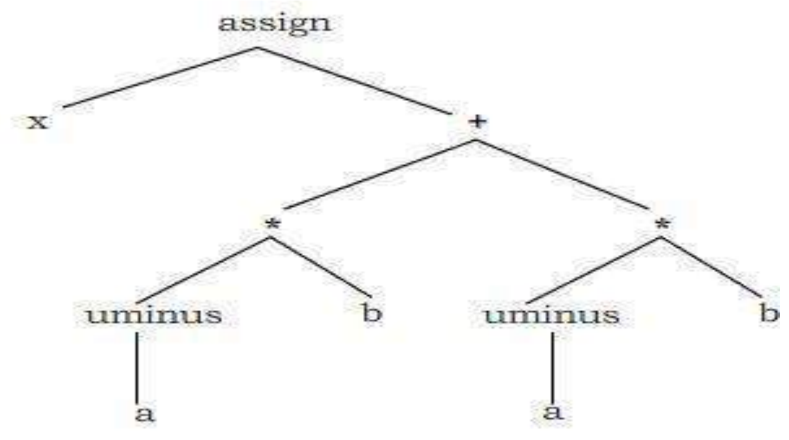
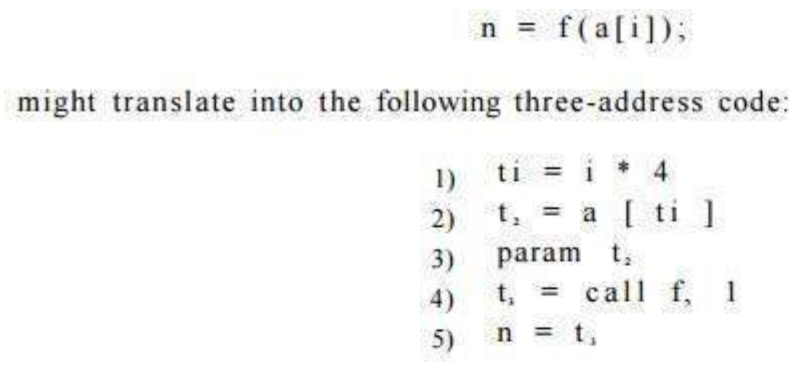


Fig: Syntax Tree

integers to integers. Then, the assignment



```
n = f(a[i]);
```

might translate into the following three-address code:

```
1)  ti = i * 4
2)  t2 = a [ ti ]
3)  param  t2
4)  t3 =  call  f,  1
5)  n = t3
```

The first two lines compute the value of the expression a [i] into temporary t2.

Line 3 makes t2 an actual parameter for the call.

On line 4 of f with one parameter.

Line 5 assigns the value returned by the function call to t3 .

Line 6 assigns the returned value to n.

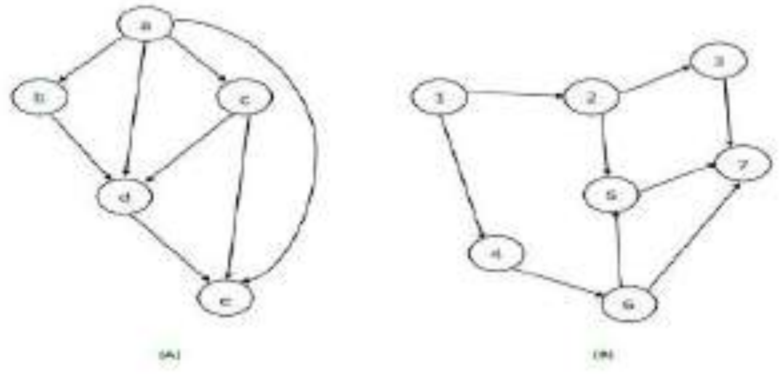
$$
\begin{aligned}
D &\rightarrow \text{define } T \text{ id } (F) \{ S \} \\
F &\rightarrow \varepsilon \mid T \text{ id}, F \\
A &\rightarrow \text{return } E; \\
E &\rightarrow \text{id } (A) \\
A &\rightarrow \varepsilon \mid E, A
\end{aligned}
$$

Adding functions to the source language

The productions allow function definitions and function calls. Non terminal D and T generate declarations and types, respectively. A function definition generated by D consists of keyword define, a return type, the function name, formal parameters in parentheses and a function body consisting of a statement.

Directed Acyclic Graph :
The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

- Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
- The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
- DAG is an efficient method for identifying common sub-expressions.
- It demonstrates how the statement's computed value is used in subsequent statements.
  - **Examples of directed acyclic graph :**



Directed Acyclic Graph Characteristics :
A Directed Acyclic Graph for Basic Block is a directed acyclic graph with the following labels on nodes.

- The graph's leaves each have a unique identifier, which can be variable names or constants.
- The interior nodes of the graph are labelled with an operator symbol.
- In addition, nodes are given a string of identifiers to use as labels for storing the computed value.
- Directed Acyclic Graphs have their own definitions for transitive closure and transitive reduction.
- Directed Acyclic Graphs have topological orderings defined.

Algorithm for construction of Directed Acyclic Graph :
There are three possible scenarios for building a DAG on three address codes:
Case 1 – x = y op z
Case 2 – x = op y
Case 3 – x = y
Directed Acyclic Graph for the above cases can be built as follows :

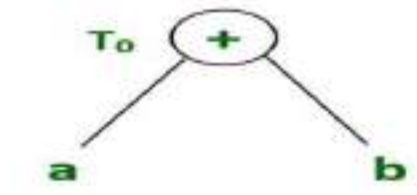
Step 1 –
- If the y operand is not defined, then create a node (y).
- If the z operand is not defined, create a node for case(1) as node(z).

Step 2 –
- Create node(OP) for case(1), with node(z) as its right child and node(OP) as its left child (y).
- For the case (2), see if there is a node operator (OP) with one child node (y).
- Node n will be node(y) in case (3).

Step 3 –
Remove x from the list of node identifiers. Step 2: Add x to the list of attached identifiers for node n.
Example :

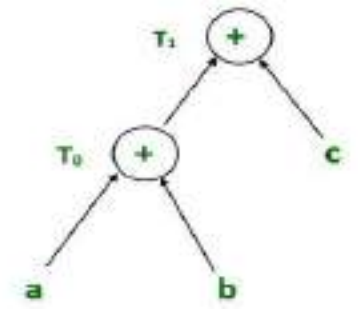
$T_0 = a + b$ —Expression 1
$T_1 = T_0 + c$ —-Expression 2
$d = T_0 + T_1$ —–Expression 3

Expression 1 : **$T_0 = a + b$**

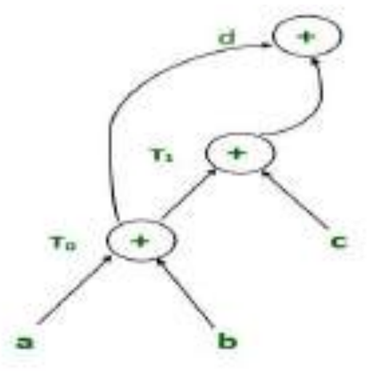


•

Expression 2: $T_1 = T_0 + c$

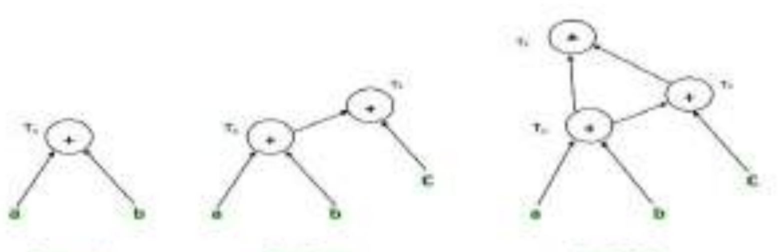


Expression 3 : **$d = T_0 + T_1$**



Example :

```
T1 = a + b
T2 = T1 + c
T3 = T1 x T2
```



$T_1 = a + b$ $T_2 = T_1 + c$ $T_3 = T_1 * T_2$

Application of Directed Acyclic Graph:

- Directed acyclic graph determines the subexpressions that are commonly used.
- Directed acyclic graph determines the names used within the block as well as the names computed outside the block.
- Determines which statements in the block may have their computed value outside the block.
- Code can be represented by a Directed acyclic graph that describes the inputs and outputs of each of the arithmetic operations performed within the code; this representation allows the compiler to perform common subexpression elimination efficiently.
- Several programming languages describe value systems that are linked together by a directed acyclic graph. When one value changes, its successors are recalculated; each value in the DAG is evaluated as a function of its predecessors.