Table of Content

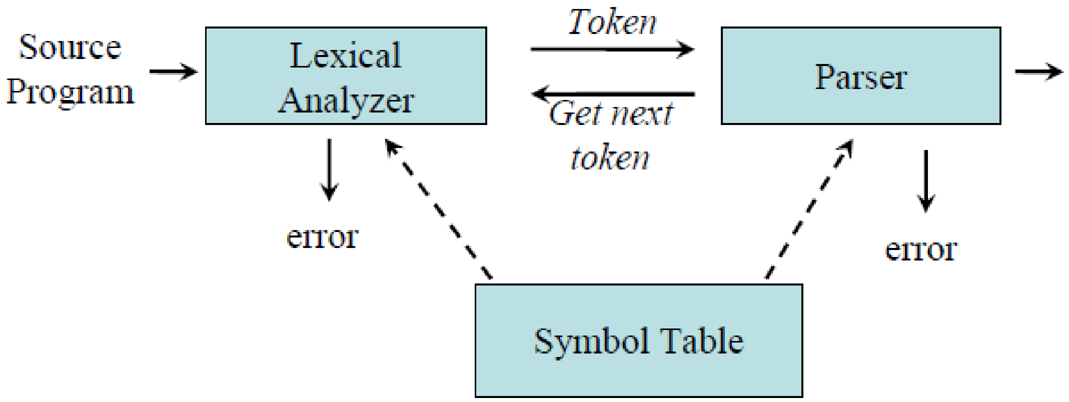
Unit-2

2.1 Lexical Analyzer

2.1.1 Role of Lexical Analysis

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser. Another main task of the lexical analyzer is to strip out white spaces and other unnecessary character from the source program while generating the tokens. Last but not the least another main task is to correlate error message from the compiler with the source program.

These interactions are suggested in Figure. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line

number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro – preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Why separate Lexical Analyzer is needed?

a) Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.

b) Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.

c) Specialized buffering techniques for reading input characters can speed up the compiler significantly.

d) Compiler portability is enhanced. Input – Device – Specific peculiarities can be restricted to the lexical analyzer.

2.1.2 Tokens, Lexemes and Patterns

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

2.1.2.1 Attribute of Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

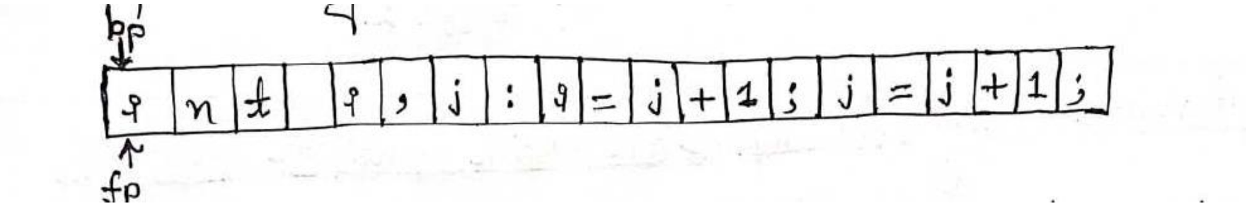
The token name and associated attribute values for the following statement.

A=B*C+2

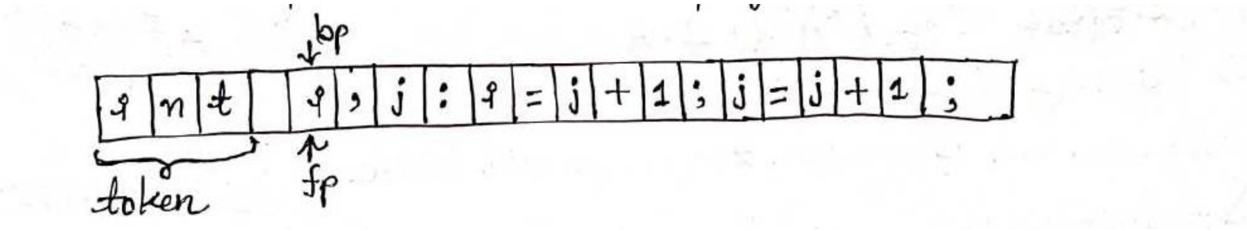
<id, pointer to symbol table entry for A>
<assignment_op>
<id, pointer to symbol table entry for B>
<mult_op>
<id, pointer to symbol-table entry for C>
<add_op>
<number, interger value 2>

2.1.3 Input Buffering

Reading character by character from secondary storage is slow process and time consuming, so we use buffer technique to eliminate this problem and increase efficiency. The lexical analyzer scans the input from left to right one character at a time. It uses two pointers" bp" (begin pointer) and" fp" (forward pointer) to keep track of the pointer of input scanned. Initially both the pointers point to the first character of the input string.



The forward pointer moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of the lexeme. In above example as soon as fp encounters a blank space the lexeme 'int' is identified. Then both bp and fp are set at next token as shown in the figure below. This process will be repeated for the whole program.



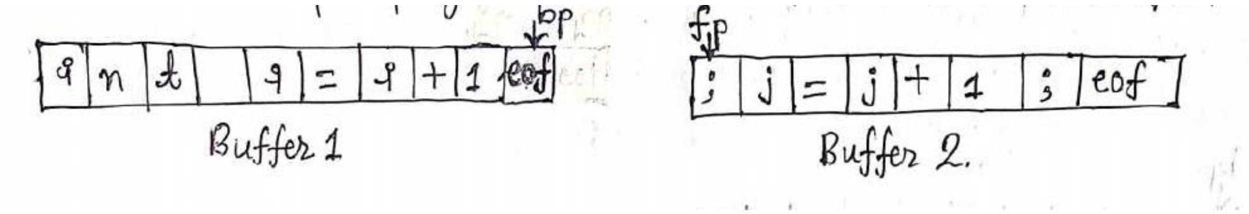
2.1.3.1 One Buffer Scheme:

In this scheme, only one buffer is used to store the input string. The problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan the rest of the lexeme the buffer must be refilled.

2.1.3.2 Two Buffer Scheme:

In this scheme two buffer are used to store input string and they are scanned alternately. When end of the current buffer is reached the other buffer is filled. The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

Initially both the bp and fp are pointing to the first character of first buffer. Then the fp move forward in search of end of lexeme. As soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify, the boundary of first buffer end of buffer character (eof) should be placed at the end of the first buffer, similarly for second buffer also. Alternatively, both the buffers can be filled up until end of the input program and stream of tokens is identified.



2.1.4 Specification of Tokens

2.1.4.1 Alphabet:

An alphabet is a finite, nonempty set of symbol. Conventionally, we use the symbol Σ for an alphabet. Common alphabet includes:

Σ ={0,1}, this is the binary alphabet.

Σ ={a,b,….z}, this is the set of all lower-case letters.

Power of an alphabet

If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet using an exponential notation. We define Σk to be the set of strings of length k, each of whose symbol is in Σ

For example: if {a,b,c}

Σ^0=∈

Σ^1={a,b,c}

The set of all string over an alphabet is conventionally denoted by Σ*

Example Σ={0,1}

Σ*={0,1}*={∈,0,1,00,01,10,11,……}

Sometimes, we wish to exclude the empty string from the set of strings. The set of nonempty string from alphabet Σ is denoted by Σ^+.

Concentration of String

Let x and y be strings. Then xy denotes the concentration of x and y, that is the string formed by making a copy of x and following it by a copy of y.

Example: let x=1101 and y=0011 then xy=11010011

2.1.4.2 Strings:

A string is a finite sequence of symbols chosen from some alphabet.

For example, 01101 is a string from the binary alphabet Σ ={0,1}

- **Empty String**

The empty string is the string with zero occurance of symbol. It is denoted by ∈

- **Length of String**

The number of positions in a string is called its length. The notation for the length of a string S is |S|. For example, |01101|=5

2.1.4.3 Language

A language is a specific set of strings over some fixed alphabet Σ

Example:

a) ϕ the empty set is language.

b) {ε} the set containing empty string is a language

c) The set of well-formed c program is a language.

d) The set of all possible identifier is a language.

Operation on languages:

- Concatnation: L1L2={s1s2|s1 ∈ L1 and s2 ∈ L2}
- Union: L1UL2={s|s ∈ L1 or s ∈ L2}
- Exponential: L^0={∈} L^1=L L^2=LL
- Kleen Closure: L*= $\bigcup_{i=0}^{\infty} \boldsymbol{Li}$

- Positive Closure $L^{+}=\bigcup_{i=1}^{\infty} Li$

2.1.4.4 Regular Expression

- Regular expression are the algebric expressions that are used to describe token of a programming language.
- Let Σ be an alphabet, the regular expression over the alphabet Σ are defined inductively as follows:
- **Basic Step:**
- ɸ is a regular expression representing the empty language i.e L(ɸ)= ɸ
- ∈ is a regular expression representing the language of empty string {∈} i.e; L(∈)={∈}
- If 'a' is a symbol in Σ, then 'a' is a regular expression representing the language {a} i.e; L(a)={a}

Example:

- 1(1+0) *0 denotes the language of all string that begins with a '1' and ends with a '0'.
- (1+0) *00 denotes the language of all strings that ends with 00.

2.1.4.5 Regular Definition

To write regular expression for some languages can be difficult because their regular expressions can be quite complex. In those cases, we may use *regular definitions*. The regular definition is a sequence of definitions of the form,

d1 → r1

d2 → r2

…………….

dn → rn

Where di is a distinct name and ri is a regular expression over symbols in Σ∪ {d1, d2... di-1}

Where, Σ = Basic symbol and {d1, d2... di-1} = previously defined names.

Example:

Regular definition for specifying identifiers in a programming language like C.

letter → A | B | C |………| Z | a | b | c |………| z

underscore → '_'

digit →0 | 1 | 2 |……………. | 9

id → (letter | underscore). (letter | underscore | digit) *

If we are trying to write the regular expression representing identifiers without using regular definition, it will be complex.

(A | B | C |………| Z | a | b | c |………| z | _ |). ((A | B | C |………| Z | a | b | c |………| z | _ |) (1 | 2 |……………. | 9)) *

2.1.5 Recognition of Tokens

To recognize tokens lexical analyzer performs following steps:

A. Lexical analyzers store the input in input buffer.
B. The token is read from input buffer and regular expressions are built for corresponding token
C. From these regular expressions finite automata is built. Usually NFA is built.
D. For each state of NFA, a function is designed and each input along the transitional edges corresponds to input parameters of these functions.
E. The set of such functions ultimately create lexical analyzer program.

2.1.6 Finite Automata

Finite Automata (FA) is the simplest machine to recognize patterns. Finite Automata is a model of a computational system, consisting of a set of states, a set of possible inputs, and a rule to map each state to another state, or to itself, for any of the possible inputs. Formal specification of machine is {Q, ∑, q, F, δ } where

Q: Finite set of states,

∑: set of Input Symbols,

q: Initial state,

F: set of Final States

δ: Transition Function

Finite automata come in two flavors:

- non-deterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and #, the empty string, is a possible label.
- Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

2.1.6.1 DFA

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where.

- Q is a finite set of states.
- ∑ is a finite set of symbols calling the alphabet.
- δ is the transition function where δ: Q × ∑ → Q
- q0 is the initial state from where any input is processed (q0 Q). ∈ Q).
- F is a set of final state/states of Q (F Q). ⊆ Q).

Example

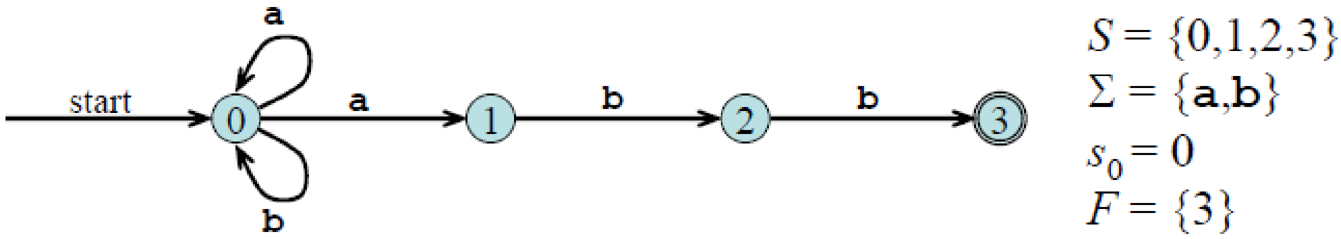


Fig: - NFA for regular expression (a + b)*a b b

2.1.6.2 ε- NFA

In NFA if a transition made without any input symbol is called ε-NFA. Here we need ε-NFA because the regular expressions are easily convertible to ε-NFA.

A ε -NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where.

- Q is a finite set of states.
- ∑ is a finite set of symbols calling the alphabet.
- δ is the transition function where δ: Q × (∑U ε)→2^Q
- q0 is the initial state from where any input is processed (q0 Q). ∈ Q).
- F is a set of final state/states of Q (F Q). ⊆ Q).

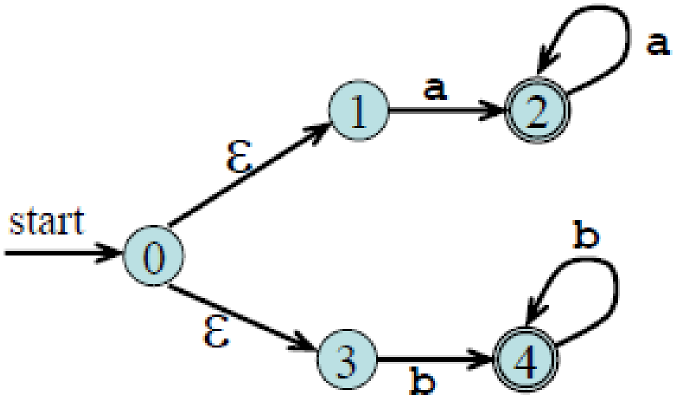


Fig: - ε-NFA for regular expression aa* +bb*

2.1.6.3 NFA

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. As it has finite number of states, the machine is called Non-deterministic Finite Machine or Non-deterministic Finite Automaton.

A NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where.

- Q is a finite set of states.
- ∑ is a finite set of symbols calling the alphabet.
- δ is the transition function where δ: Q × ∑ →2^Q
- q0 is the initial state from where any input is processed (q0 Q). ∈ Q).
- F is a set of final state/states of Q (F Q). ⊆ Q).

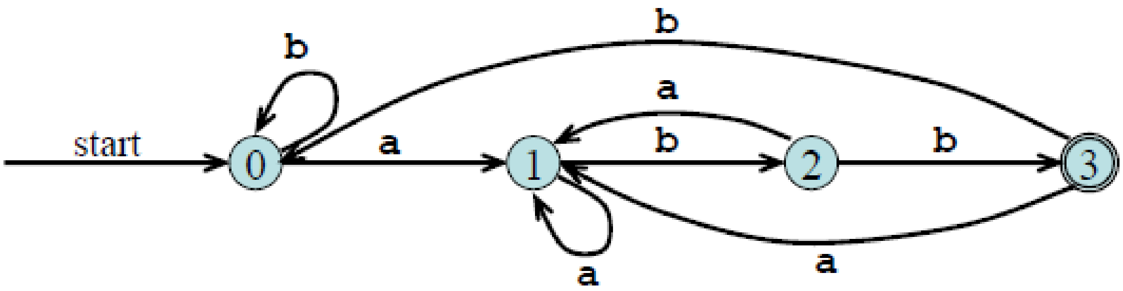
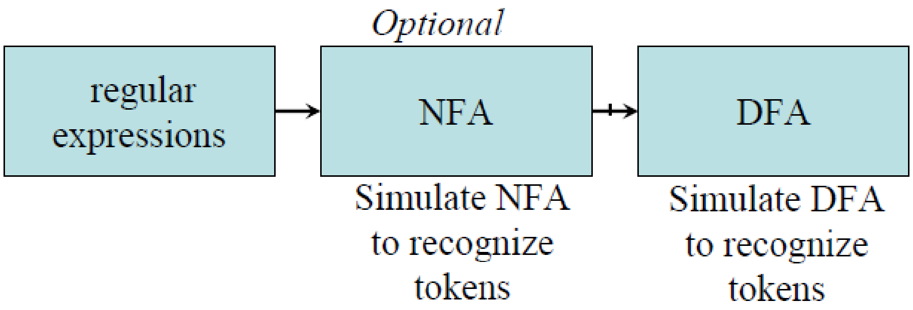


Fig:-DFA for regular expression (**a**+**b**)***abb**

2.1.7 Design of Lexical Analyzer

First, we define regular expression for tokens, then we convert them into a DFA to get a lexical analyzer for our tokens.


Optional
regular expressions
NFA
DFA
Simulate NFA to recognize tokens
Simulate DFA to recognize tokens

Algorithm1:

- Regular Expression→ NFA → DFA (two steps: first NFA, then to DFA) **(Self-Study)**

Algorithm2:

- Regular Expression → DFA (Directly convert a regular expression into a DFA)

2.1.7.1 Conversion from RE to DFA Directly

To construct a DFA directly from a regular expression, we construct its syntax tree and then compute four functions: nullable , firstpos , lastpos , and followpos , defined as follows. Each definition refers to the syntax tree for a particular augmented regular expression (r) #.

A. nullable (n) is true for a syntax-tree node n if and only if the subexpression represented by n has # in its language. That is, the subexpression can be made null or the empty string, even though there may be other strings it can represent as well.

B. firrstpos (n) is the set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the subexpression rooted at n.

C. lastpos (n) is the set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the subexpression rooted at n.

D. followpos (p), for a position p , is the set of positions q in , the entire syntax # tree such that there is some string x = a1 a2 ...... an in L (r)# , such # that for some i , there is a way to explain the membership of x in L (r)# by matching ai to position p of the syntax tree and ai +1 to position q.

Conversion steps:

a) Augment the given regular expression by concatenating it with special symbol # i.e. r □(r) #

b) Create the syntax tree for this augmented regular expression

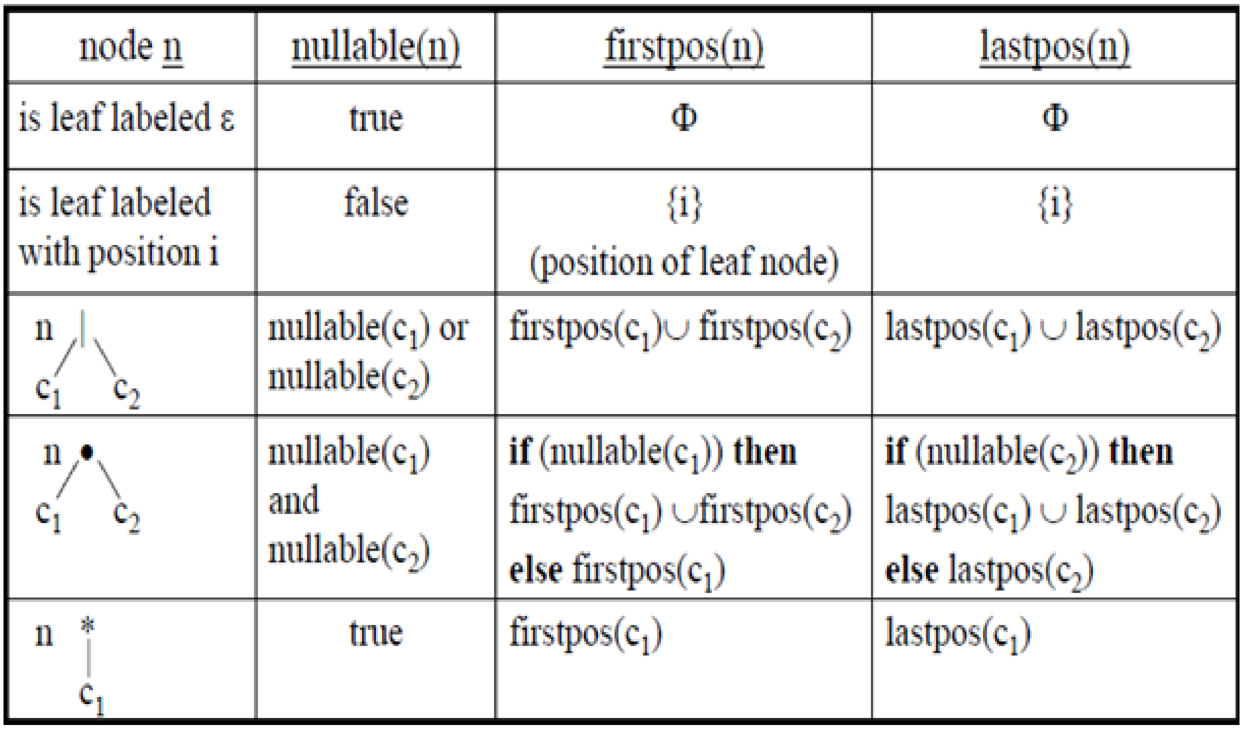
In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.

c) Then each alphabet symbol (plus #) will be numbered (position numbers)

d) Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

e) Finally construct the DFA from the *followpos*

Rules for calculating nullable, firstpos and lastpos:



node n	nullable(n)	firstpos(n)	lastpos(n)
is leaf labeled ε	true	Φ	Φ
is leaf labeled with position i	false	{i} (position of leaf node)	{i}
n $\mid$ c_1 c_2	nullable(c_1) or nullable(c_2)	firstpos(c_1)∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
n • c_1 c_2	nullable(c_1) and nullable(c_2)	**if** (nullable(c_1)) **then** firstpos(c_1) ∪firstpos(c_2) **else** firstpos(c_1)	**if** (nullable(c_2)) **then** lastpos(c_1) ∪ lastpos(c_2) **else** lastpos(c_2)
n * c_1	true	firstpos(c_1)	lastpos(c_1)

Algorithm to evaluate followpos

```
for each node n in the tree do
    if n is a cat-node with left child c1 and right child c2 then
        for each i in lastpos(c1) do
            followpos(i) := followpos(i) ∪ _firstpos(c2)
        end do
    else if n is a star-node
        for each i in lastpos(n) do
            followpos(i) := followpos(i) ∪ _firstpos(n)
        end do
    end if
end do
```

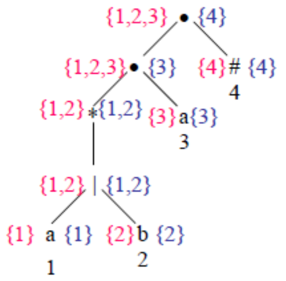
Example

Convert regular expression (a | b) * a into DFA.

Step-1: Its augmented regular expression is.

(a|b) *a#

Step-2: Construct syntax tree and calculate firstpos and followpos:



Step-3: Now we calculate followpos,

followpos(1)={1,2,3}
followpos(2)={1,2,3}
followpos(3)={4}
followpos(4)={}

Step-4: Compute State of a DFA with a help of followpos.

S_1=*firstpos(root)*={1,2,3}

mark S_1

for a: followpos(1) ∪ *followpos(3)*={1,2,3,4}=S_2 *move*(S_1,a)=S_2

for b: followpos(2)={1,2,3}=S_1 *move*(S_1,b)=S_1

mark S_2

for a: followpos(1) ∪ *followpos(3)*={1,2,3,4}=S_2 *move*(S_2,a)=S_2

for b: followpos(2)={1,2,3}=S_1 *move*(S_2,b)=S_1

Here Start state is S1 and the final state is S2.

Step-5: Design DFA

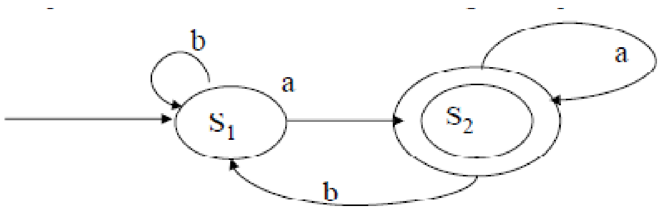


Fig: Resulting DFA of given regular expression

2.1.8 State Minimization in DFA

DFA minimization is the task of transforming a given deterministic finite automaton (DFA) into an equivalent DFA that has a minimum number of states. Here, two DFAs are called equivalent if they recognize the same regular language. Several different algorithms accomplishing this task are known and described in standard textbooks on automata theory. For each regular language, there also exists a minimal automaton that accepts it, that is, a DFA with a minimum number of states and this DFA is unique (except that states can be given different names). The minimal DFA ensures minimal computational cost for tasks such as pattern matching.

Partition the set of states into two groups:

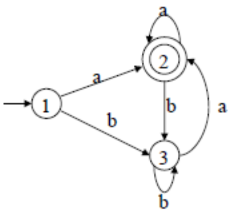
- G1: set of accepting states
- G2: set of non-accepting states

For each new group G:

partition G into subgroups such that states s1 and s2 are in the same group if for all input symbols a, states s1 and s2 have transitions to states in the same group.

- Start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

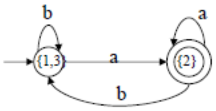
Example



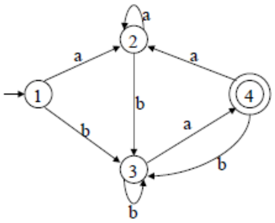
$G_1 = \{2\}$
$G_2 = \{1,3\}$
G_2 cannot be partitioned because

move(1,a)=2 move(1,b)=3
move(3,a)=2 move(2,b)=3

So, the minimized DFA (with minimum states)



Example 2:

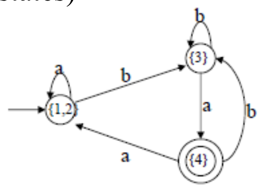


Groups: {1,2,3} {4}

{1,2} {3}
no more partitioning

a	b
1->2	1->3
2->2	2->3
3->4	3->3

So, the minimized DFA (with minimum states)

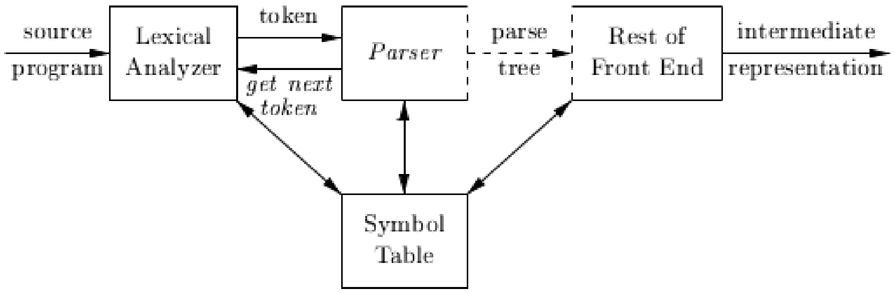


Unit 2.2

2.2 Semantic Analyzer

2.2.1 Role of Semantic Analysis

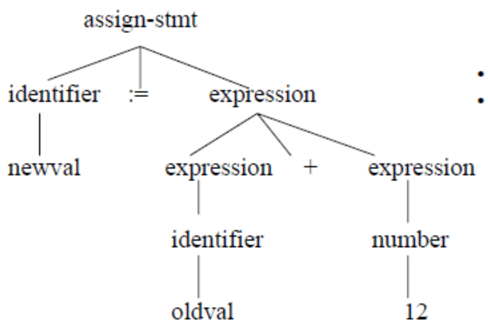
In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig, and verifies that the string of token names can be generated by the grammar for the source language. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.



A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program.

Syntax analyzer is also called the **parser.** Its job is to analyze the source program based on the definition of its syntax. It works in lockstep with the lexical analyzer and is responsible for creating a parse-tree of the source code.

Ex: newval: = oldval + 12



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

The syntax of a language is specified by a **context free grammar** (CFG). The rules in a CFG are mostly recursive. A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not. If it satisfies, the syntax analyzer creates a parse tree for the given program.

2.2.2 Context Free Grammar

Context-free grammar is a 4-tuple *G* = (*N*, *T*, *P*, *S*) where

- *T* is a finite set of tokens (*terminal* symbols)
- *N* is a finite set of *non-terminals*
- *P* is a finite set of *productions* of the form

$\alpha \rightarrow \beta$

Where,

$\alpha \in (N \cup T)^* \, N \, (N \cup T)^*$ and $\beta \in (N \cup T)^*$

- *S* ∈ _*N* is a designated *start symbol*

Programming languages usually have recursive structures that can be defined by a context-free grammar (CFG).

2.2.2.1 CFG Notational Conventions

Terminals are denoted by lower-case letters and symbols (single atoms) and **bold** strings (tokens)

a, b, c, ….. ∈T

specific terminals:

0, 1, id, +

Non-terminals are denoted by *lower-case italicized* letters or upper-case letters symbols

A, B, C… ∈*N*

specific non-terminals:

expr, *term*, *stmt*

Production rules are of the form.

A→α that is read as "A can Produce α"

Strings comprising of both terminals and non-terminals are denoted by Greek letters.

α , β , etc

2.2.3 Left most and Right-most Derivation.

2.2.3.1 Left most derivation.

If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$$

Example: Let any set of production rules in a CFG be

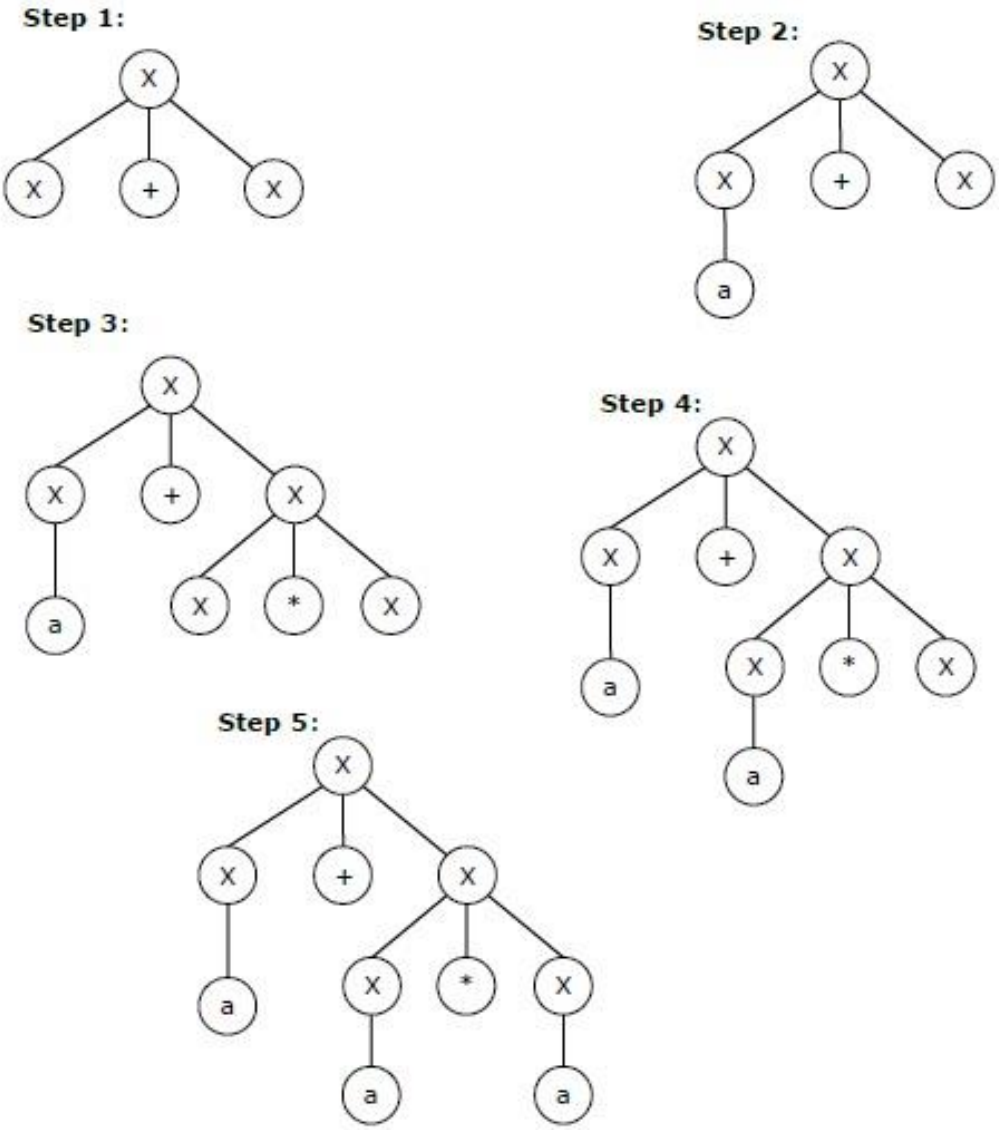
X → X+X | X*X |X| a

over an alphabet {a}.

The leftmost derivation for the string **"a+a*a"** may be −

X → X+X → a+X → a + X*X → a+a*X → a+a*a

The stepwise derivation of the above string is shown as below −



Step 1:

Step 2:

Step 3:

Step 4:

Step 5:

2.2.3.2 Right most Derivation

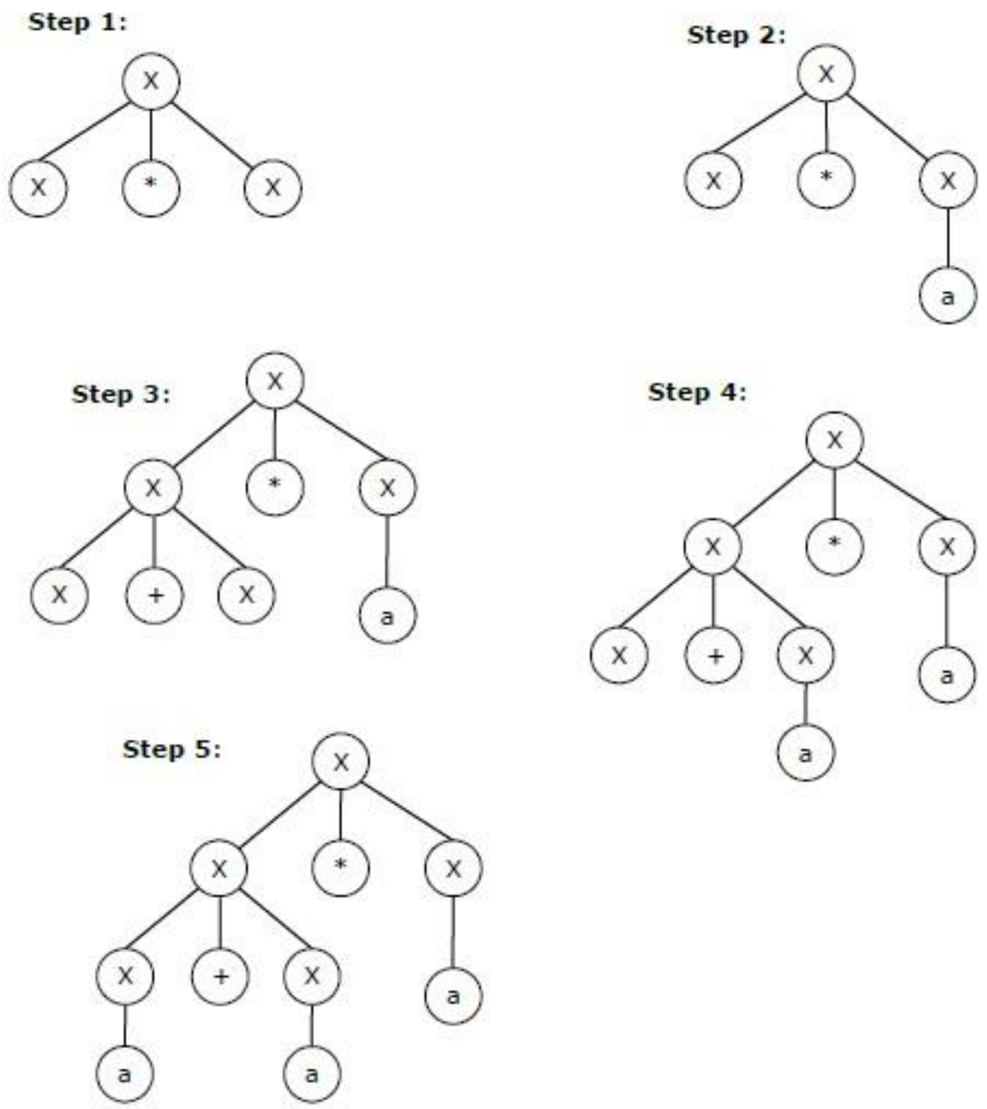
If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$

The rightmost derivation for the above string **"a+a*a"** may be −

X → X*X → X*a → X+X*a → X+a*a → a+a*a

The stepwise derivation of the above string is shown as below –



2.2.4 Parse Tree

Parse tree is the hierarchical representation of terminals or non-terminals. These symbols (terminals or non-terminals) represent the derivation of the grammar to yield input strings. In parsing, the string springs using the beginning symbol. The starting symbol of the grammar must be used as the root of the Parse Tree. Leaves of parse tree represent terminals. Each interior node represents productions of a grammar.

Rules to draw a parse tree.

- All leaf nodes need to be terminals.

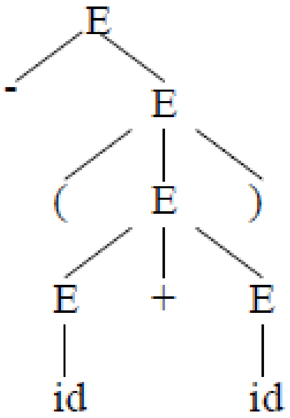
- All interior nodes need to be non-terminals.
- In-order traversal gives the original input string.

Example:

let us consider a CFG:

E→E+E _E∗E _E _−E _id _

Then the parse tree for –(id+id) is:



2.2.5 Ambiguity of grammar

A grammar is said to be ambiguous if there exists more than one left most derivation or more than one right most derivation or more than one parse tree for a given input string.

- If the grammar is not ambiguous then we call it unambiguous grammar.
- If the grammar has ambiguity then it is good for compiler construction.
- No method can automatically detect and remove the ambiguity, but we can remove the ambiguity by re-writing the whole grammar without ambiguity.

Example

Let us consider a grammar with production rules, as shown below

E = I

E = E+E

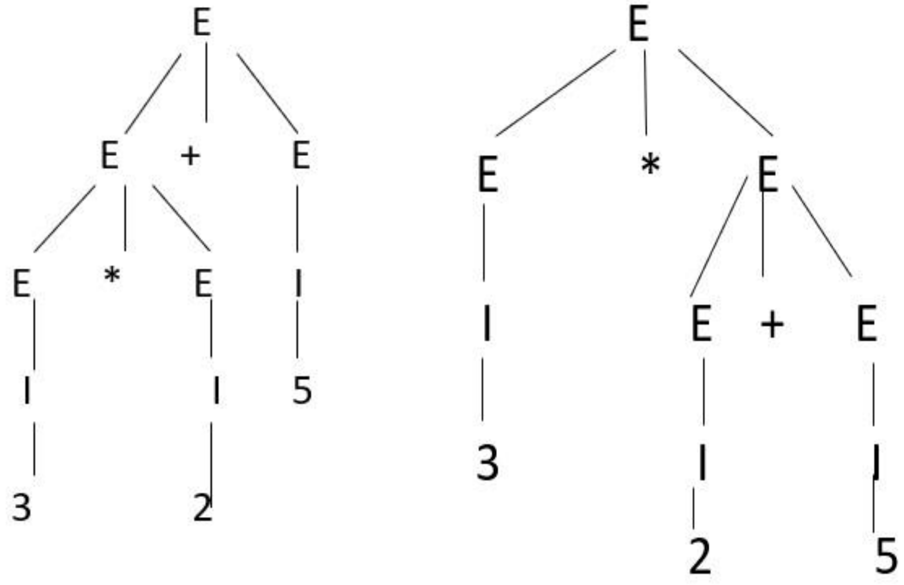
E = E*E

E = (E)

E = ε|0|1|2|3...9

Let's consider a string "3*2+5"

If the above grammar generates two parse trees by using Left most derivation (LMD) then, we can say that the given grammar is ambiguous grammar.



Since there are two parse trees for a single string, then we can say the given grammar is ambiguous grammar.

2.2.6 Left Recursion

A grammar becomes left recursive of it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol. Left recursion become problem for top-down parsers because left recursion leads to infinite loop, so we reduce or eliminate it before solving top-down parsers.

$$A \rightarrow A\ \alpha$$

$$A \rightarrow A\alpha \mid \beta$$

Eliminate immediate left recursion.

$$A \rightarrow \beta A‘$$

$$A‘ \rightarrow \alpha A‘ \mid \in$$

For general case,

$A \rightarrow A\ \alpha_1 \mid \dots \mid A\ \alpha_m \mid \beta_1 \mid \dots \mid \beta_n$ where $\beta_1 \dots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\ A' \mid \dots \mid \beta_n\ A'$

$A' \rightarrow \alpha_1\ A' \mid \dots \mid \alpha_m\ A' \mid \varepsilon$ an equivalent grammar

Example,

$$
\begin{aligned}
&E \rightarrow E+T \mid T\\
&T \rightarrow T{*}F \mid F\\
&F \rightarrow id \mid (E)\\
&\quad\Downarrow \quad \text{eliminate immediate left recursion}\\
&E \rightarrow T\,E'\\
&E' \rightarrow +T\,E' \mid \varepsilon\\
&T \rightarrow F\,T'\\
&T' \rightarrow {*}F\,T' \mid \varepsilon\\
&F \rightarrow id \mid (E)
\end{aligned}
$$

2.2.7 Left Factoring

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand. This grammar is called left factoring grammar.

Replace productions

$$A \rightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \ldots \mid \alpha\,\beta_n \mid \gamma$$

with

$$
\begin{aligned}
&A \rightarrow \alpha\,A' \mid \gamma\\
&A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n
\end{aligned}
$$

Example: Eliminate left factorial from following grammar:

S→iEiS | iEiSiS | a

B→b

Solution:

S→iEiSS' |a

S’→ε|iS

B→b

2.2.8 Parsing

The process of transforming the data from one format to another is called Parsing. This process can be accomplished by the parser. The parser is a component of the translator that helps to organise linear text structure following the set of defined rules which is known as grammar.

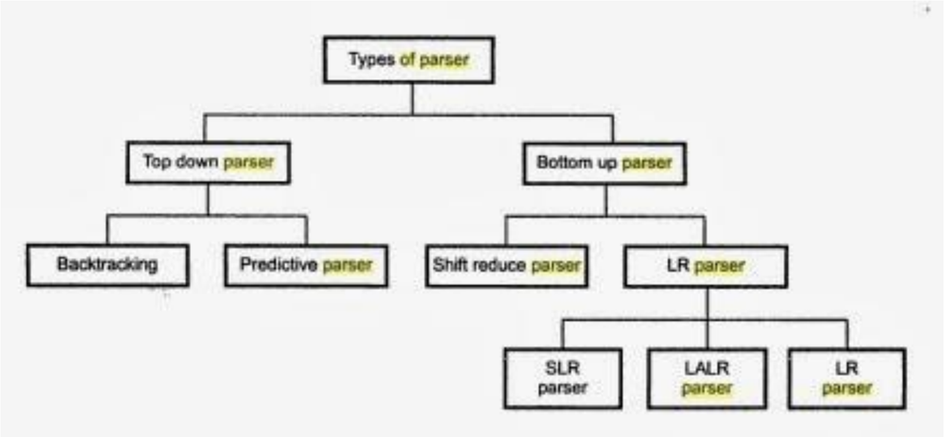


Fig: Type of Parsing

2.2.8.1 Top-Down Parser

When the parser generates a parse with top-down expansion to the first trace, the left-most derivation of input is called top-down parsing. The top-down parsing initiates with the start symbol and ends on the terminals. Such parsing is also known as predictive parsing.

a. Recursive Descent (Backtracking)

The parsing technique that starts from the initial pointer, the root node. If the derivation fails, then it restarts the process with different rules.

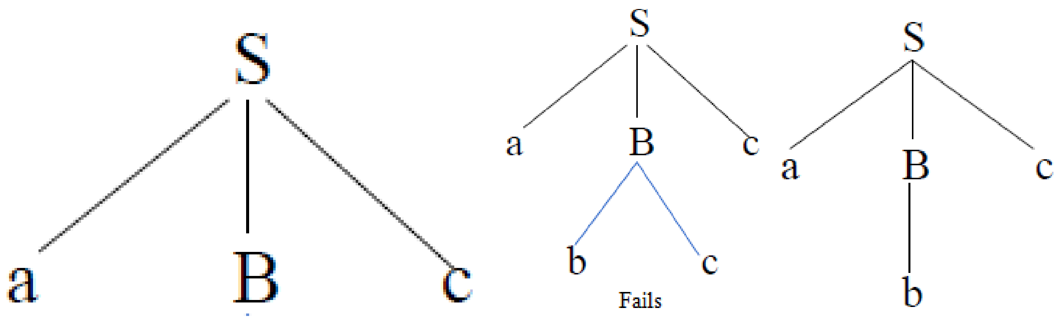
Example: Consider the grammar,

$$S \rightarrow aBc$$

$$B \rightarrow bc|b$$

And the input string “abc” parsed using Recursive-Descent parsing.

- Step 1: The first rule $S \rightarrow aBc$ to parse S
- Step 2: The next non-terminal is B and is parsed using production $B \rightarrow bc$ as,
- Step 3: Which is false and now backtrack and use production $B \rightarrow b$ to parse for B

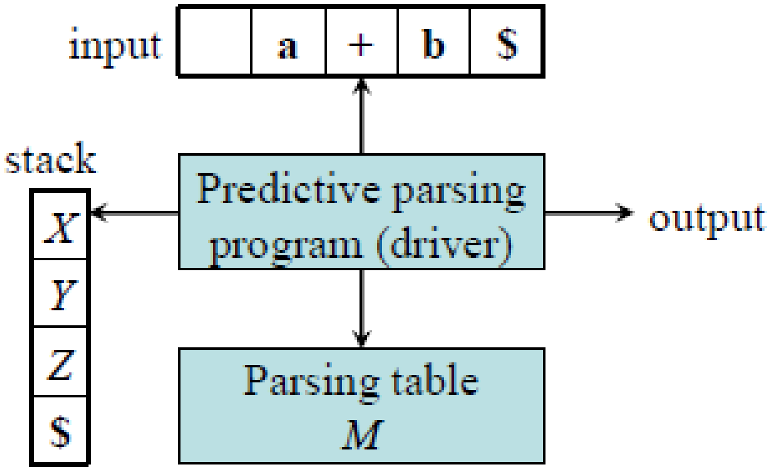


Input	Output	Rule Used
abc	S	Use S→aBc
abc	aBc	Match symbol a
bc	Bc	Use B→bc
bc	bcc	Match symbol b
c	cc	Match symbol c
ɸ	c	Dead end, backtrack
bc	Bc	Use B→b
bc	bc	Match symbol b
c	c	Match symbol c
ɸ	ɸ	accepted

b. Predictive Parsing / non-recursive Descent parsing.

- Non-Recursive predictive parsing is a table-driven parser
- A form of recursive descent parsing that does not require any back tracking is known as predictive parsing.
- It is also called LL(1) parsing table technique since we would be build a table for string to be parsed.

- It has capability to be used to replace input string.
- In the name LL(1), the first L stands for scanning the input from left to right, the second L stands for producing a leftmost derivation, and the 1 stands for using one input symbol of lookahead at each step to make parsing action decision.



Input buffer: It contains the string to be parsed followed by a special symbol $.

Stack: A stack contains a sequence of grammar symbols with $ on the bottom. Initially it contains the symbol $.

Parsing table: It is a two-dimensional array M [A, a] where 'A' is non-terminal and 'a' is a terminal symbol.

Output stream: A production rule representing a step of the derivation sequence of the string in the input buffer.

<u>Constructing LL(1) Parsing Tables</u>

- The parse table construction requires two functions: FIRST and FOLLOW.
- A grammar G is suitable for LL(1) parsing table if the grammar is free from left recursion annd left factoring.

Essential conditions to check first are as follows:

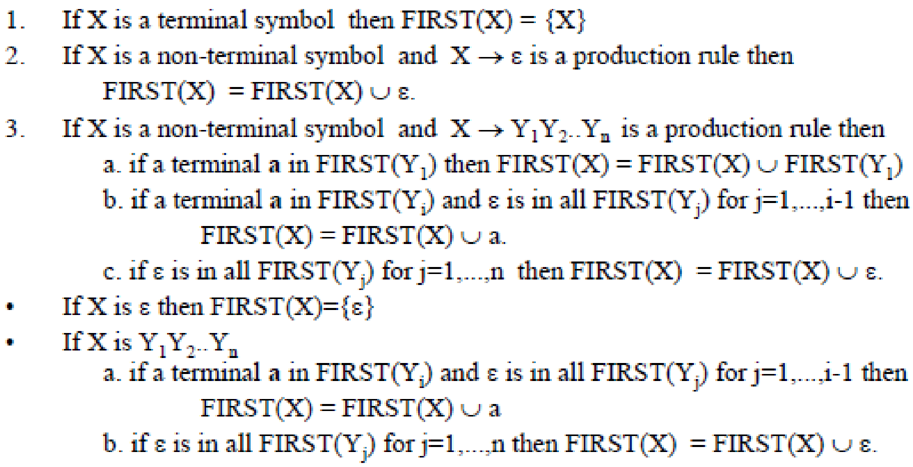
1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

Algorithm to construct LL(1) Parsing Table:

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

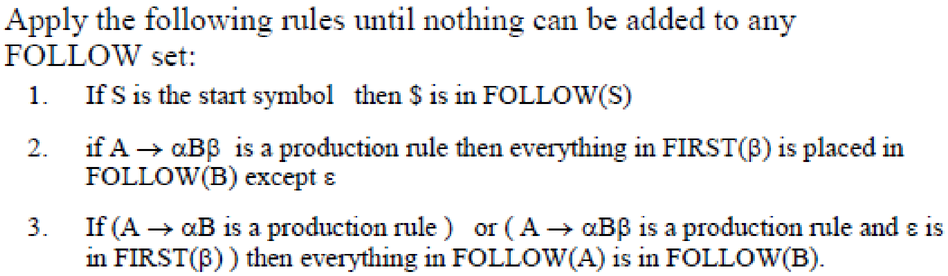
- **First():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.



1. If X is a terminal symbol then FIRST(X) = {X}
2. If X is a non-terminal symbol and X → ε is a production rule then
   FIRST(X) = FIRST(X) ∪ ε.
3. If X is a non-terminal symbol and $X \rightarrow Y_1Y_2..Y_n$ is a production rule then
   a. if a terminal a in FIRST(Y_1) then FIRST(X) = FIRST(X) ∪ FIRST(Y_1)
   b. if a terminal a in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then
      FIRST(X) = FIRST(X) ∪ a.
   c. if ε is in all FIRST(Y_j) for j=1,...,n then FIRST(X) = FIRST(X) ∪ ε.

- If X is ε then FIRST(X)={ε}
- If X is $Y_1Y_2..Y_n$
  a. if a terminal a in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then
     FIRST(X) = FIRST(X) ∪ a
  b. if ε is in all FIRST(Y_j) for j=1,...,n then FIRST(X) = FIRST(X) ∪ ε.

- **Follow():** What is the Terminal Symbol which follows a variable in the process of derivation.



Apply the following rules until nothing can be added to any FOLLOW set:

1. If S is the start symbol then $ is in FOLLOW(S)
2. if A → αBβ is a production rule then everything in FIRST(β) is placed in FOLLOW(B) except ε
3. If (A → αB is a production rule) or (A → αBβ is a production rule and ε is in FIRST(β)) then everything in FOLLOW(A) is in FOLLOW(B).

Step 3: For each production A –> α. (A tends to alpha)

- Find First(α) and for each terminal in First(α), make entry A –> α in the table.
- If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
- If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.

Example:

S→aAa|BAa|ε

A→ cA|bA| ε

B→b

First	Follow
{a,b, ε}	{$}
{c,b ε}	{a}
{b}	{c,b,a}

	a	b	c	$
S	S→aAa	S→BAa		S→ε
A	A→ε	A→bA	A→cA	
B		B→b		

Input string "bcba"

Stack	Remaining input	Action
$S	bcba$	Choose S→BAa
$aAB	bcba$	Choose B→b
$aAb	bcba$	Match b
$aA	cba$	Choose A→cA
$aAc	cba$	Match c
$aA	ba$	Choose A→bA
$aAb	ba$	Match b
$aA	a$	Choose A→ε
$a	a$	Match a
$	$	accept

Example2

E→TE'

E'→+TE'|ε

T→FT'

T'→*FT'|ε

F→id|(E)

Solution

Find their First and Follow sets:

	First	Follow
E –> TE’	{ id, (}	{ $,) }
E’ –> +TE’/ε	{ +, ε }	{ $,) }
T –> FT’	{ id, (}	{ +, $,) }
T’ –> *FT’/ε	{ *, ε }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

Now, the LL(1) Parsing Table is:

	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)		

Q. Check whether the given grammar is LL (1) or not?

S --> A | a
A --> a
Solution,
Find their First and Follow sets:

	First	**Follow**
S	{ a }	{ $ }
A	{ a }	{ $ }

Make a parser table.

	a	**$**
S	S –> A, S –> a	
A	A –> a	

Here, we can see that there are two productions in the same cell. Hence, this grammar is not feasible for LL(1) Parser.

2.2.8.2 Bottom-Up Parser

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

1. Shift Reduce Parsing

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

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.

<u>Handle</u>

A substring that can be replaced by a non-terminal when it matches its right sentential form is called a **handle**. If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Example:

E→ E+T|T
T→ T*F|F
F→ (E)|id

Right Sentential Form	**Handle**	**Reducing Production**
id1*id2	id1	F→ id
F*id2	F	T→ F
T*id2	id2	F→ id
T*F	T*F	E→ T*F

<u>Stack Implementation of Shift-Reduce Parser</u>

The stack holds the grammar symbols and input buffer holds the string w to be parsed.

1. Initially stack contains only the sentinel $, and input buffer contains the input string w$.

2. While stack not equal to $S or not ***error*** and input not $ do

    (a) While there is no handle at the top of stack, do ***shift*** input buffer and push the symbol onto stack

    (b) If there is a handle on top of stack, then pop the handle and ***reduce*** the handle with its non-terminal and push it onto stack

3. Done

<u>Parser Actions:</u>

1. Shift: The next input symbol is shifted onto the top of the stack.

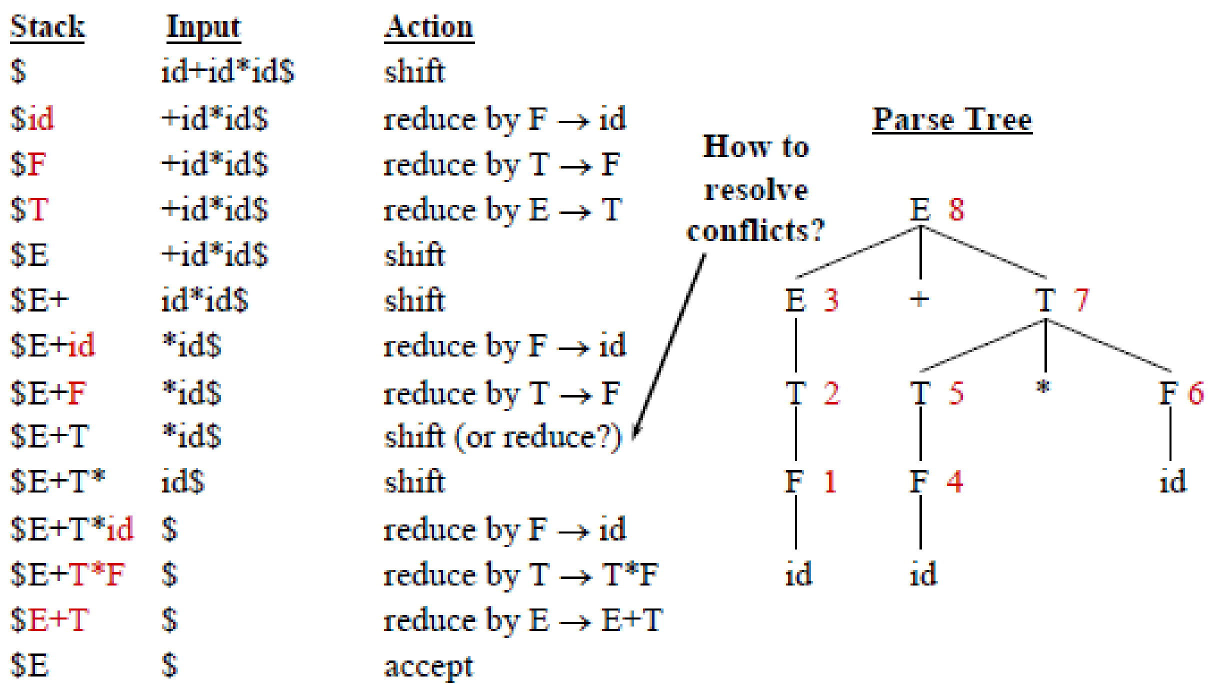
2. Reduce: Replace the handle on the top of the stack by the non-terminal.

3. Accept: Successful completion of parsing.

4. Error: Parser discovers a syntax error, and calls an error recovery routine

Example 1: Use the following grammar

E→ E+T|T
T→ T*F|F
F→ (E)|id



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 (or reduce?)
$E+T*	id$	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

How to resolve conflicts?

Parse Tree

E 8
E 3
+
T 7
T 2
T 5
*
F 6
F 1
F 4
id
id
id

Conflicts in Shift-Reduce Parsing

Some grammars cannot be parsed using shift-reduce parsing and result in *conflicts*. There are two kinds of shift-reduce conflicts:

a. shift/reduce conflict:

Here, the parser is not able to decide whether to shift or to reduce.

Example:

A→ab | abcd

the stack contains $ab, and

the input buffer contains cd$, the parser cannot decide whether to reduce $ab to $A or to shift two more symbols before reducing.

b. reduce/reduce conflict:

Here, the parser cannot decide which sentential form to use for reduction.

For example

A→bc

B →abc and the stack contains $abc, the parser cannot decide whether to reduce it to $aA or to $B.

2. LR parser

LR parser is a bottom-up parser for context-free grammar that is very generally used by computer programming language compiler and other associated tools. LR parser reads their input from left to right and produces a right-most derivation. It is called a Bottom-up parser because it attempts to reduce the top-level grammar productions by building up from the leaves. LR parsers are the most powerful parser of all deterministic parsers in practice.

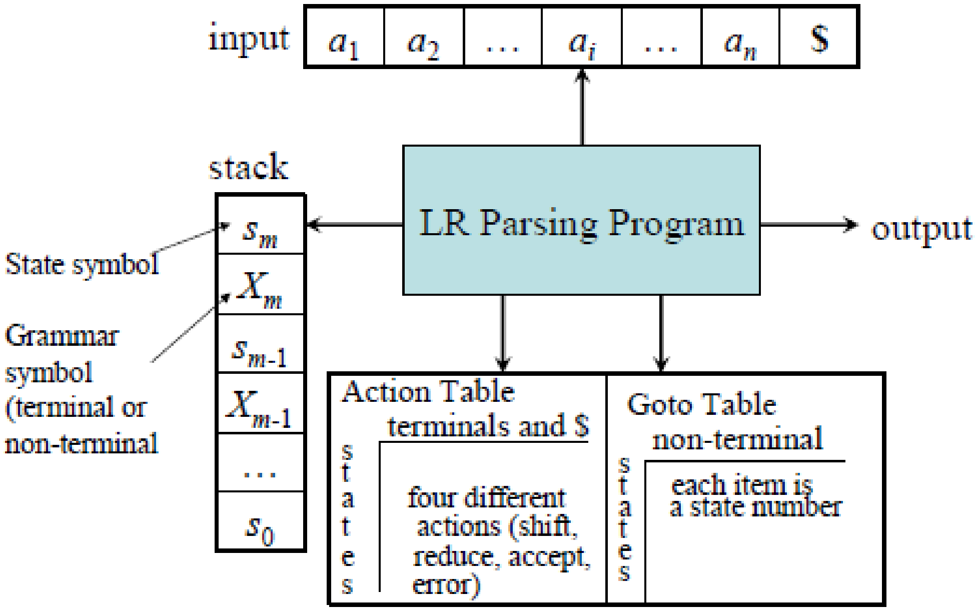


Fig: General structure of LR parser

<u>Basic Terminology used in LR parser</u>

1. Augmented grammar

If G is a grammar with start symbol S, then the augmented grammar G‘ of G is a grammar with a new start symbol S‘ and production S‘→S

Eg: the grammar,

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Its augmented grammar is;

E’→E

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (\mathrm{E}\,) \mid \mathrm{id}$

2. Canonical LR (0) collection:

An item (LR (0) item) is a production rule that contains a dot(.) somewhere in the right side of the production. For example, the production A→aAB has four items.

A→.aAB

A→ a.AB

A→aA.B

A→aAB.

A production A→ε, generates only one item A→.

3. Closure Operation

If I is a set of items for a grammar G, then closure (I) is the set of LR(0) items constructed from I using the following rules:

- Initially, every LR(0) items in I is added to closure (I).
- If A→α.Bβ is in closure (I) and B→γ is a production rule of G then add B→.γ in the closure (I) repeat until no more new LR(O) items added to closure (I)

Example:

Consider a grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (\,E\,) \mid \mathrm{id}$

Its augmented grammar is;

E'→E

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (\,E\,) \mid \mathrm{id}$

Now closure (E'→E) contains the following items:

E'→.E

$E \rightarrow .E + T$

E→.*T*

$T \rightarrow .T * F$

T→.*F*

$F \rightarrow .(E)$

F→.id

4. The goto Operation

If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:

If A → α.Xβ in I then every item in closure({A → αX.β}) will be in goto(I,X).

Example:

I = { E' → .E, E → .E+T, E → .T, T → .T*F, T → .F, F → .(E), F → .id }

goto(I,E) = *closure*({[*E'* → *E* •, *E* → *E* • + *T*]}) = { E' → E., E → E.+T }

goto(I,T) = { E → T., T → T.*F }

goto(I,F) = {T → F. }

goto(I,( ) = *closure*({[*F* →(•E)]}) = { F → (.E), E → .E+T, E → .T, T → .T*F, T → .F, F → .(E), F → .id }

goto(I,id) = { F → id. }

<u>Algorithm for Cannonical LR(0) collection</u>

```
1. Start
2. Augment the grammar by adding production S' → S
3. C = { closure({S'→.S}) }
4. repeat the followings until no more set of LR(0) items can be added to C.
   for each I in C and each grammar symbol X
       if goto(I,X) is not empty and not in C
           add goto(I,X) to C
```

1 Repeat step 4 until no new set of items are added to C.

2 Stop.

Construction of canonical LR(0) collection

Example1:

The augmented grammar is:

C'→C

C→AB

A→a

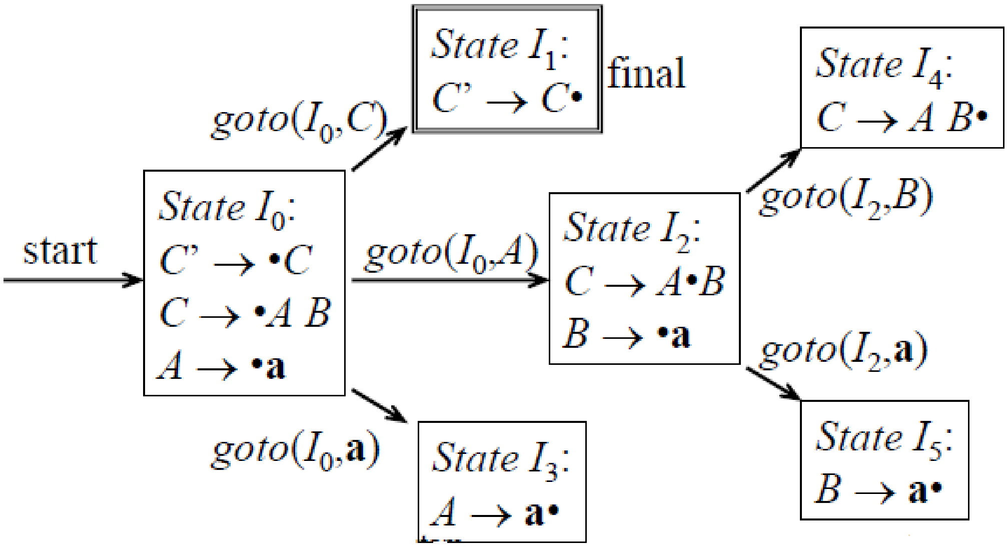
B→a

Solution:

$I0$ = *closure* ($C` \rightarrow \bullet C$)

$I1$ = goto($I0$,C) = *closure*($C` \rightarrow C\bullet$)

And so on……


State I_1:
$C' \rightarrow C\bullet$
final
State I_4:
$C \rightarrow A\ B\bullet$
goto(I_0,C)
goto(I_2,B)
start
State I_0:
$C' \rightarrow \bullet C$
$C \rightarrow \bullet A\ B$
$A \rightarrow \bullet\mathbf{a}$
goto(I_0,A)
State I_2:
$C \rightarrow A\bullet B$
$B \rightarrow \bullet\mathbf{a}$
goto(I_2,a)
goto(I_0,a)
State I_3:
$A \rightarrow \mathbf{a}\bullet$
State I_5:
$B \rightarrow \mathbf{a}\bullet$

Example2:

Given grammar.

S → AA

A → aA | b

Solution:

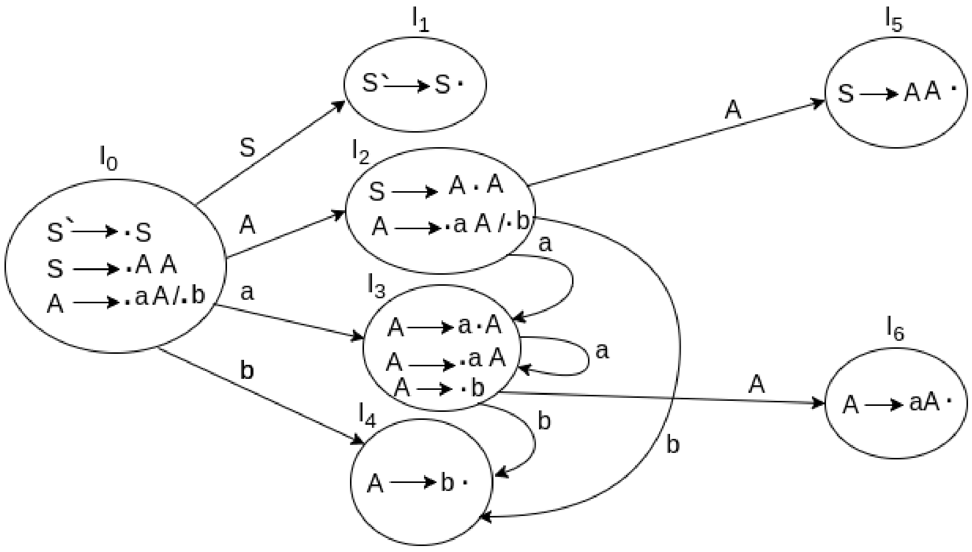
Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •S

S → •AA

A → •aA

A → •b



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
I0	S3	S4		2	1
I1			accept		
I2	S3	S4		5	
I3	S3	S4		6	
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

In Above Table

- 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.

S → AA ... (1)

A → aA ... (2)

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.

SLR Parser

SLR is simple LR. It is the smallest class of grammar having few number of states. SLR is very easy to construct and is similar to LR parsing. The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, there's a chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminal states. We can solve this problem by entering 'reduce' corresponding to FOLLOW of LHS of production in the terminating state. This is called SLR(1) collection of items.

Steps for constructing the SLR parsing table :

- Writing augmented grammar
- LR(0) collection of items to be found
- Find FOLLOW of LHS of production
- Defining 2 functions:goto[list of terminals] and action[list of non-terminals] in the parsing table

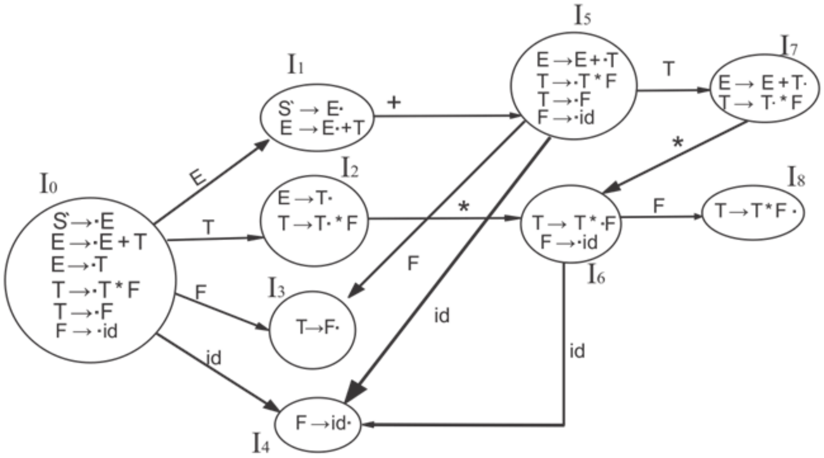
Example

S→E

E→ E + T | T

T → T * F | F

F → id



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			

In above table

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.

CLR 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 (0) 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

S → AA

A → aA

A → b

Solution

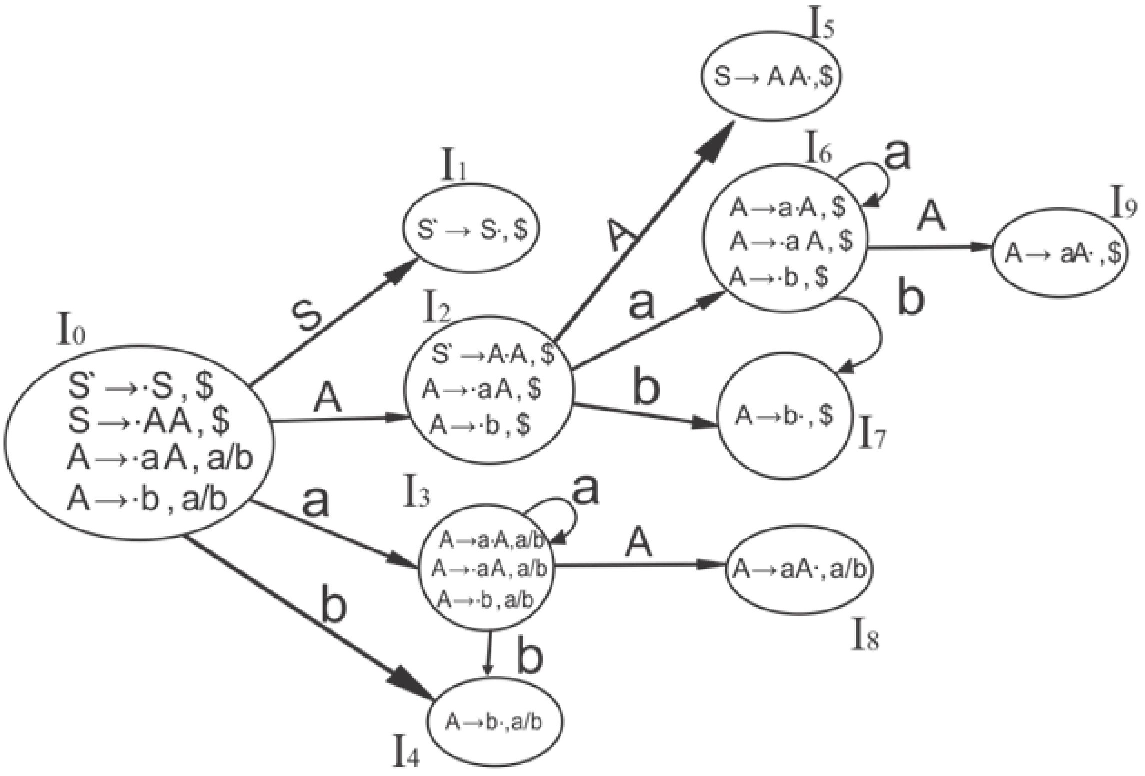
Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

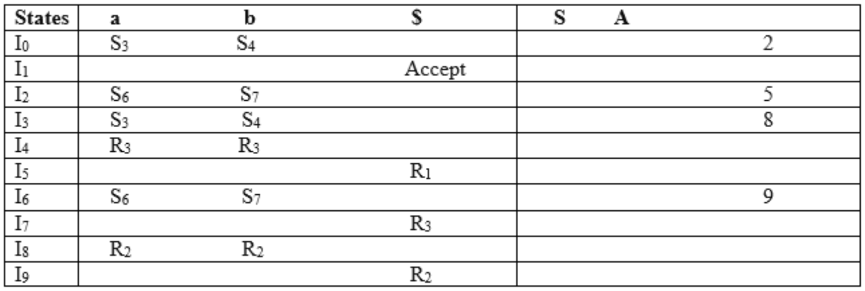
S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b





States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

In above table

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.

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

S → AA
A → aA
A → b

Solution:

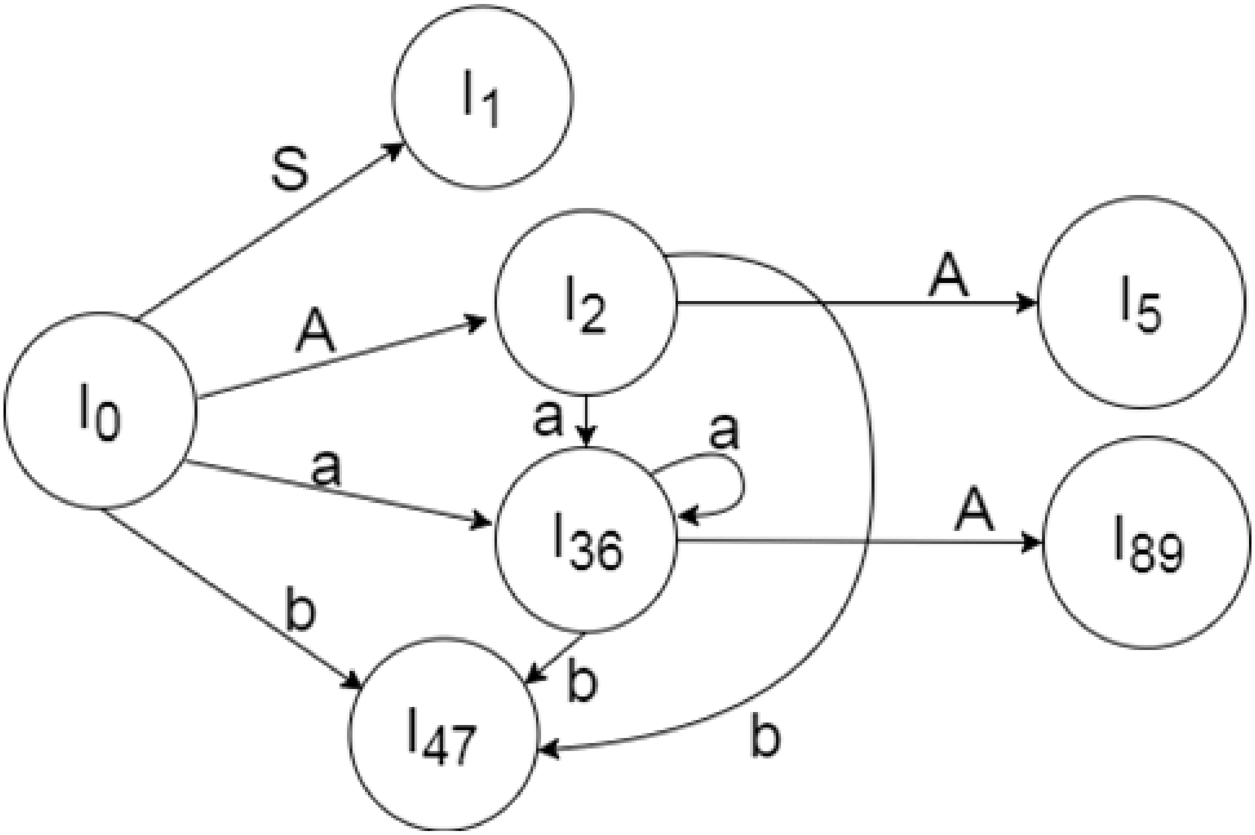
Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b



States	a	b	\$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

Top-down VS Bottom-up parsing

Top Down Parsing	**Bottom Up Parsing**
It is a parsing strategy that first looks at the higest level of the parser tree and work down the parse tree by using the rule of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and work up the parse tree by using the rules of grammar.
This parsing technique uses left most derivation.	This parsing technique use right most derivation.
It's main decision is to select what production rule to use in order to construct the string.	It's main decision is to select when to use a production rule to reduce the string to get the starting symbol.
Error detection is easy.	Error detection is difficult.
Parsing table size is small	Parsing table size is bigger.
Less power	High power.

2.3 Semantic Analysis

2.3.1 Example of Semantic Errors

Example1: Use of the non-initialized variable.

```
int I;
i++;
```

Error=the variable I is not initialized

Example2: Type incompatibility

```
Int a="hello"
```

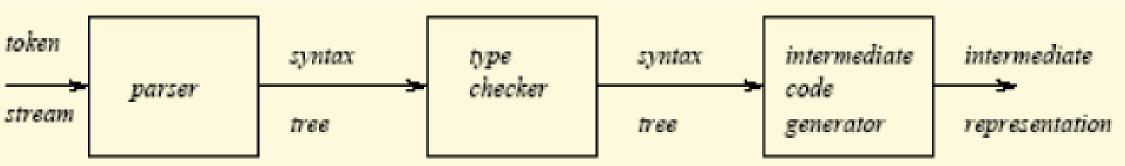
Error=the type string and inti are not compatible.

Example 3: Error in expressions

```
Char S='A'
Int a =15-S
```

Error= the '-' operator does not support type char.

2.3.2 Type Checking



Compiler must check that the source program follows both the syntactic and semantic conventions of the source language. Type checking is the process of checking the data type of different variables. The design of a type of checker for a language is based on information about the syntactic construct in the language, the notation of type, and the rules for assigning types to the language constructs.

2.3.3 Type System

- The collection of different data types and their associated rules to assign types to programming language constructs is known as type systems.
- Informal type system rules, for example —*if both operands of addition are of type integer, then the result is of type integer‖*
- A type checker implements type system.

Example Type Checking of Expressions

E → id { *E*.type = l*ookup(id.entry*) }

E → charliteral { *E*.type = *char* }

E → intliteral { *E*.type = *int* }

E → *E1* + *E2* { *E*.type = (*E1*.type == *E2*.type) ? *E1*.type : *type_error* }

E → *E1 [E2]* { *E*.type = (*E2*.type == *int* and *E1*.type == *array(s,t)*) ? t : *type_error* }

E → *E1* ↑ { *E*.type = (*E1*.type == *pointer(t)*) ? t : *type_error* }

S → _id = *E* {*S*.type = (id.type == *E*.type) ? *void* : *type_error*}

Note: the type of id is determined by : id.type = lookup(id.entry)

S → if *E* then *S*1 {*S*.type = (*E*.type == *boolean*) ? *S*1.type : *type_error*}

S → while *E* do *S*1 {*S*.type = (*E*.type == *boolean) ? S*1.type : *type_error*}

S → *S*1 ; *S*2 {*S*.type = (*S*1.type == *void* and *S*2.type == *void) ? void : type_error*}

2.3.4 Type expressions

The type of a language construct is denoted by a *type expression*.

A *type expression* can be:

A basic type

a primitive data type such as *integer, real, char, boolean*, …

A type name

a name can be used to denote a type expression.

A type constructor applies to other type expressions.

arrays: If T is a type expression, then *array(I,T)* is a type expression where I denotes index range. Ex: array(0..99, int)

products: If T1 and T2 are type expressions, then their Cartesian product *T1 X T2* is a type expression. Ex: int x int

pointers: If T is a type expression, then *pointer(T)* is a type expression. Ex: pointer(int)

functions: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression *D→R* where D is R type expressions. Ex: int → int represents the type of a function which takes an int value as parameter, and its return type is also **int.**

Example Type Checking of Expressions

- *E* → **id** { *E*.type = l*ookup(id.entry*) }
- *E* → **charliteral** { *E*.type = *char* }

- *E* → **intliteral** { *E*.type = *int* }
- *E* → *E1* + *E2* { *E*.type = (*E1*.type == *E2*.type) ? *E1*.type : *type_error* }
- *E* → *E1 [E2]* { *E*.type = (*E2*.type == *int* and *E1*.type == *array(s,t)*) ? t : *type_error* }
- *E* → *E1* ↑ { *E*.type = (*E1*.type == *pointer(t)*) ? t : *type_error* }
- *S* → **id** **=** *E* {*S*.type = (**id**.type == *E*.type) ? *void* : *type_error*}
- *Note: the type of **id** is determined by : **id**.type = lookup(**id**.entry)*
- *S* → **if** *E* **then** *S*1 {*S*.type = (*E*.type == *boolean*) ? *S*1.type : *type_error*}
- *S* → **while** *E* **do** *S*1 {*S*.type = (*E*.type == *boolean) ? S*1.type : *type_error*}
- *S* → *S*1 **;** *S*2 {*S*.type = (*S*1.type == *void* **and** *S*2.type == *void) ? void : type_error*}

2.3.5 Static versus Dynamic type Checking

Static checking**:** The type checking at the compilation time is known as static checking. Typically, syntactical errors and misplacement of data type take place at this stage.

Program properties that can be checked at compile time known as static checking.

Typical examples of static checking are:

- Type checks
- Flow-of-control checks
- Uniqueness checks
- Name-related checks

Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables. Implementations of dynamically type-checked languages runtime objects are generally associated with each other through a type tag, which is a reference to a type containing its type information. Dynamic typing is more flexible. A static type system always restricts what can be conveniently expressed. Dynamic typing results in more compact programs since it is more flexible and does not require types to be spelled out. Programming with a static type system often requires more design and implementation effort.

Languages like Pascal and C have static type checking. Type checking is used to check the correctness of the program before its execution. The main purpose of type-checking is to check the correctness and data type assignments and type-casting of the data types, whether it is syntactically correct or not before their execution.

Example: **int** x[100]; … x[i] àmost of the compilers cannot guarantee that i will be between 0 and 99

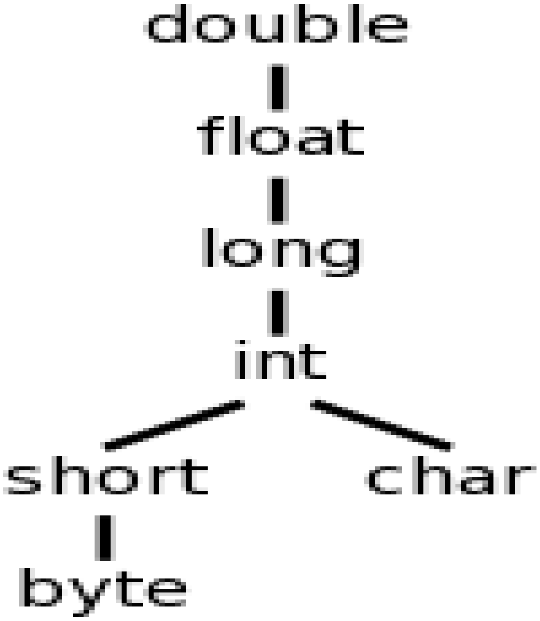
2.3.6 Type Conversion and Coercion

Type Conversion

- The process of converting data from one type to another type is known as type conversion. Often if different parts of an expression are of different types then type conversion is required.
- For example, in the expression: *z = x + y what is the type of z if x is integer and y is real?*
- *Compiler have to convert one of them to ensure that both operand of same type!*
- In many language *Type conversion* is explicit, for example using type casts i.e. must be specify as *int_to_real(x)*

Coercion

- The process of converting one type to another by compiler itself is known as coercion.
- Type conversion which happens implicitly is called *coercion*.
- Implicit type conversions are carried out by the compiler recognizing a type incompatibility and running a type conversion routine (for example, something like *inttoreal (int)*) that takes a value of the original type and returns a value of the required type.



2.3.7 Syntax-Directed Definitions

Syntax-Directed Definitions are high level specifications for translations. They hide many implementation details and free the user from having to explicitly specify the order in which translation takes place. A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol is associated with a set of attributes. This set of attributes for a grammar symbol is partitioned into two subsets **synthesized** and **inherited** attributes of that grammar.

In brief, A *syntax-directed definition* is a grammar together with *semantic rules* associated with the productions. These rules are used to compute attribute values.

Example: The syntax directed definition for a simple desk calculator

Production	**Semantic Rules**
L → E	return print(E.val)
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

Note: all attributes in this example are of the synthesized type.

2.3.8 Annotated Parse Tree

A parse tree constructing for a given input string in which each node showing the values of attributes is called an annotated parse tree.

Example:

Let‘s take a grammar,

L → E **return**

E → E1 + T

E → T

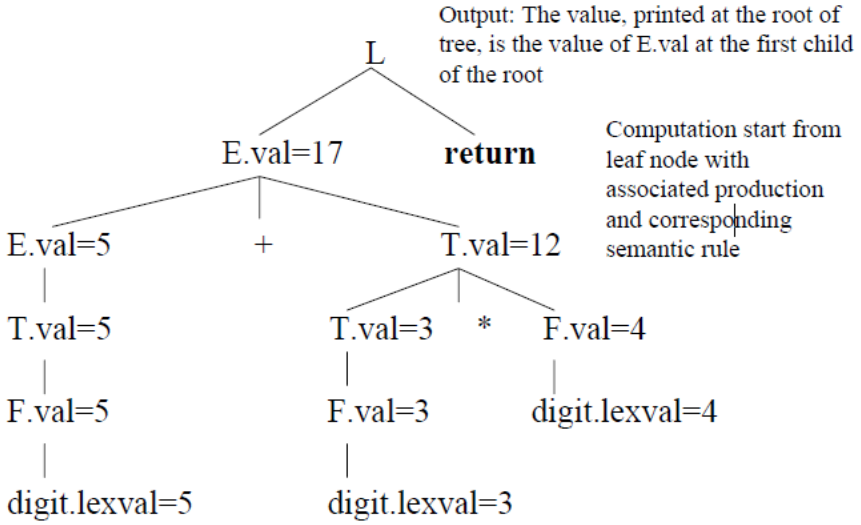
T → T1 * F

T → F

F → (E)

F → **digit**

Now the annotated parse tree for the input string 5+3*4 is,



Output: The value, printed at the root of tree, is the value of E.val at the first child of the root

Computation start from leaf node with associated production and corresponding semantic rule

2.3.9 Inherited and Synthesized Attributes:

A node in which attributes are derived from the parent or siblings of the node is called inherited attribute of that node. The attributes of a node that are derived from its children nodes are called synthesized attributes. Terminals do not have inherited attributes. A non-terminal _A‘ can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree.

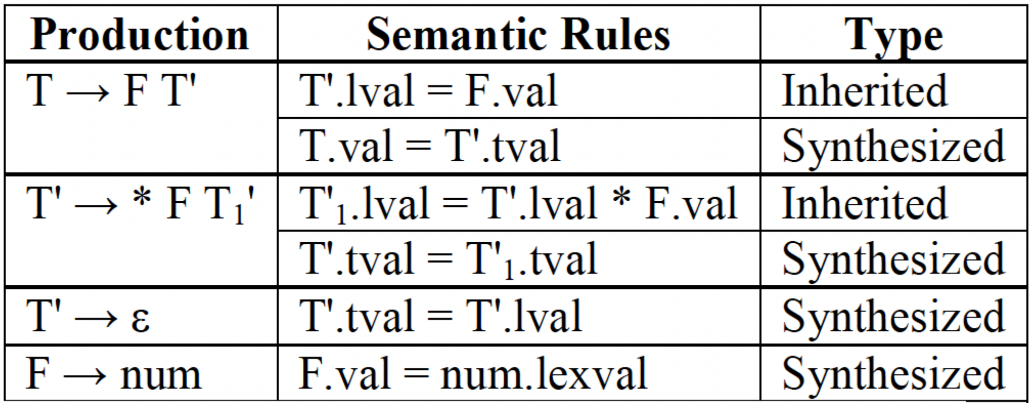
Example:

T→ FT‘

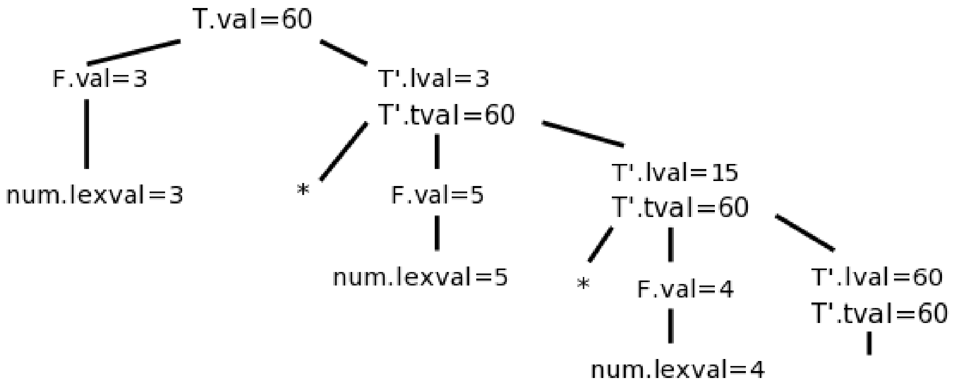
T→ T*F

T→ F

F→ num



Production	Semantic Rules	Type
T → F T'	T'.lval = F.val	Inherited
	T.val = T'.tval	Synthesized
T' → * F T_1'	T'$_1$.lval = T'.lval * F.val	Inherited
	T'.tval = T'$_1$.tval	Synthesized
T' → ε	T'.tval = T'.lval	Synthesized
F → num	F.val = num.lexval	Synthesized



2.3.10 Dependency Graph

If interdependencies among the inherited and synthesized attributes in an annotated parse tree are specified by arrows then such a tree is called dependency graph. In order to correctly evaluate attributes of syntax tree nodes, a dependency graph is useful. A dependency graph is a directed graph that contains attributes as nodes and dependencies across attributes as edges.

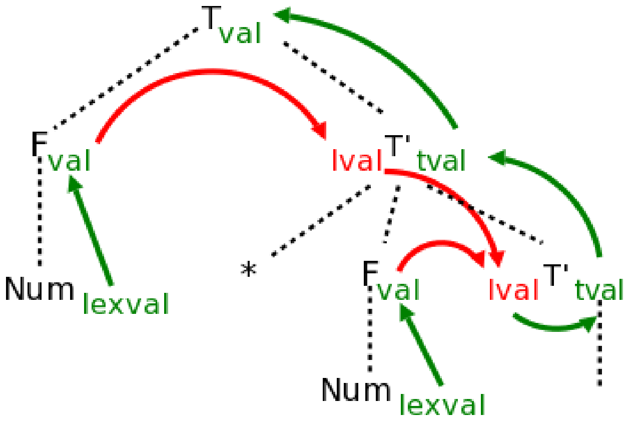
Example: let‘s take a grammar,

T → F T

T → T * F

T → F

F → num



2.3.11 S-attributed definition

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes. Semantic actions are placed in rightmost place of RHS.

Example: Bottom-up evaluation of S-Attributed definition:

Let‘s take a grammar:

L → E **n**

E → E1 + T

E → T

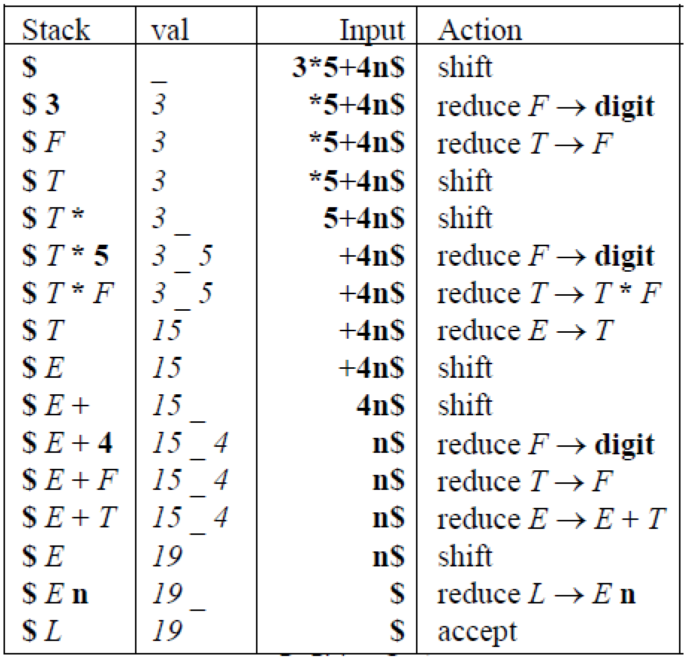
T → T1 * F

T → F

F → (E)

F → **digit**

INPUT **3*5+4 n**



Stack	val	Input	Action
$	_	**3*5+4n$**	shift
$ 3	*3*	***5+4n$**	reduce *F* → **digit**
$ *F*	*3*	***5+4n$**	reduce *T* → *F*
$ *T*	*3*	***5+4n$**	shift
$ *T* *	*3* _	**5+4n$**	shift
$ *T* * **5**	*3* _ *5*	**+4n$**	reduce *F* → **digit**
$ *T* * *F*	*3* _ *5*	**+4n$**	reduce *T* → *T* * *F*
$ *T*	*15*	**+4n$**	reduce *E* → *T*
$ *E*	*15*	**+4n$**	shift
$ *E* +	*15* _	**4n$**	shift
$ *E* + **4**	*15* _ *4*	**n$**	reduce *F* → **digit**
$ *E* + *F*	*15* _ *4*	**n$**	reduce *T* → *F*
$ *E* + *T*	*15* _ *4*	**n$**	reduce *E* → *E* + *T*
$ *E*	*19*	**n$**	shift
$ *E* **n**	*19* _	**$**	reduce *L* → *E* **n**
$ *L*	*19*	**$**	accept

2.3.12 L-Attributed Definitions

If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner. Semantic actions are placed anywhere in RHS.

Example : S->ABC, Here attribute B can only obtain its value either from the parent – S or its left sibling A but It can’t inherit from its right sibling C. Same goes for A & C – A can only get its value from its parent & C can get its value from S, A, & B as well because C is the rightmost attribute in the given production.

Unit-3

3.1 Symbol Table Design

Symbol table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable. It is built in lexical and syntax analysis phase. It is used by various phases of compiler.

1. **Lexical Analysis:** Create a new table entry in the table, example like entries about token.
2. **Syntax analysis:** adds information regarding attribute type, scope dimension, line of reference use etc.
3. **Semantic analysis:** use available information in the table to check for semantics.
4. **Intermediate code generation:** helps in adding temporary varibale information.
5. **Code optimization:** use information present in symbol table for machine dependent optimization.
6. **Target code generation**: generate code by using address information of identifier present in the table.

3.2 Function of a symbol Table

- To store the names of all entities in a structured form at one place
- To verify if a variable has been declared or not
- To determine the scope of a name (scope resolution)
- To access information associated with a given name.
- To add new information associated with a given name.
- To add new information with a given name.
- To delete a name or group of names from the table.

3.3 Information used by compiler from symbol table

- **Name:** Name of Identifier may be stored directly or as a pointer to another character string in an associated string table names can be arbitararily long.
- **Type:** Type of identifier: variable, label, procedure name etc. For variable, it’s type: basic types, derived types etc.
- **Location :** Offset within the program where the current definition is valid.
- **Other Attributes:** Array limit, fields of records, parameters, return values etc.
- **Scope:** Region of the program where the current definition is valid.

3.4 Basic Oprations on a symbol table

- **Allocate:** To allocate a new empty symbol table.
- **Free:** To remove all entry and free the storage for symbol table.
- **Insert:** To insert a name in a symbol table and return a pointer
- **Lookup:** To search for a name and return a pointer to its entry.
- **Set_attribute:** To associate an attribute with a given entry.
- **Get_attribute:** To get an attribute associated with a given entry.

3.5 Storing Names in Symbol Table

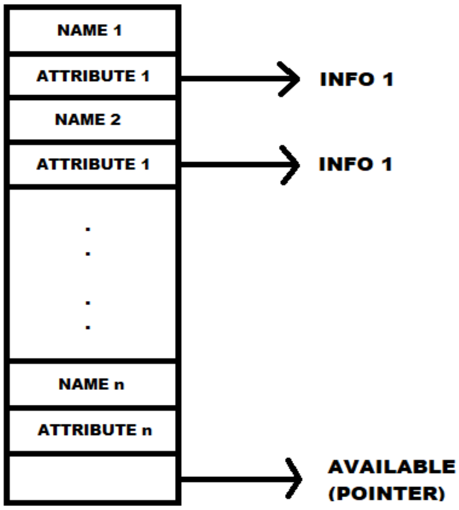
There are two type of name representation.

1. **Fixed length name:**
    - A fixed space for each name is allocated in symbol table.
    - In this type of storage if name is too small then there is wastage of space.
2. **Variable Length Name:**
    - The amount of space required by string is used to store the name.
    - The name can be stored with the name of starting index and length of each name.

3.6 Data structures for symbol table

1. **List Data Structure**

- In this method, an array is used to store names and associated information.
- A pointer "available" is maintained at the end of all stored records and new name are added in the order as they arrive.



2. Self organizing list data structure

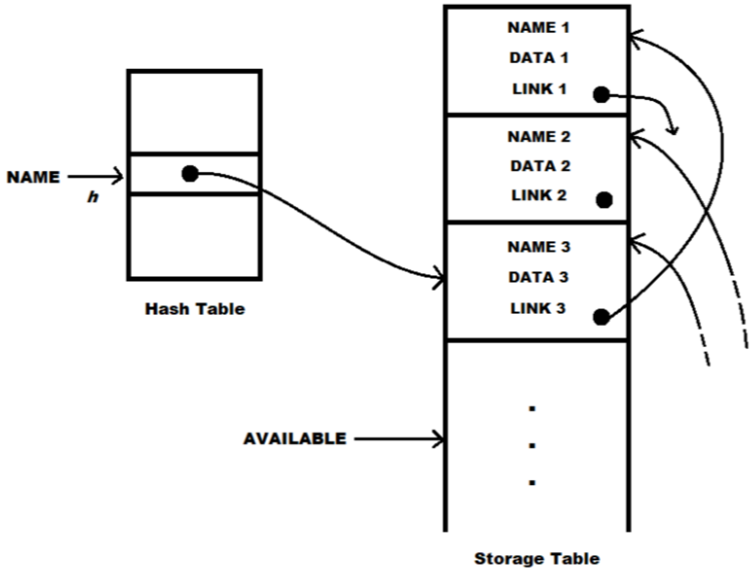
- This implementation is using linked list.
- A pointer "first" is maintained to point of first record of symbol table.
- Searching of name is done in order pointed by link of link field.

3. Binary Trees

- Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.
- All names are created as child of the root node that always follows the property of the binary search tree.

4. Hash Table

- In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables.
- A hash table is an array with an index range: 0 to table size-1. These entries are pointers pointing to the names of the symbol table.
- To search for a name, we use a hash function that will result in an integer between 0 to table size-1.
- The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.

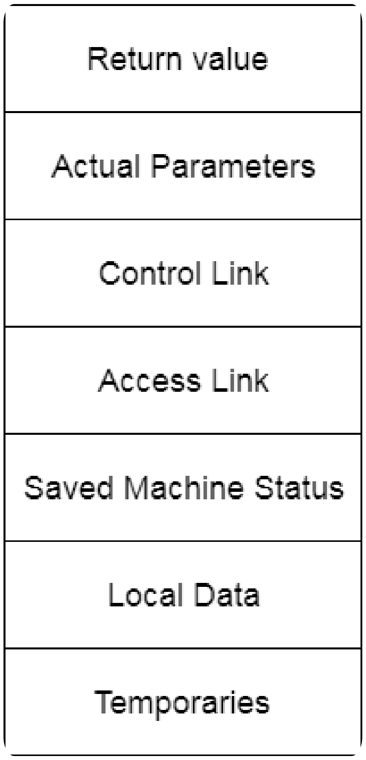


3.7 Runtime Storage Management

Activation Record: Activation record is a block of memory used for managing information needed by a single execution of a procedure. The following three-address statements are associated with the run-time allocation and de-allocation of activation records:

1. Call
2. Return
3. Halt
4. Action, a placeholder for other statements.

The execution of a procedure is called its **activation**. An **activation record** contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used). In any function call, the routine that initiates the call is called **caller** and the routine that is being called is known as **callee**.



Return value

Actual Parameters

Control Link

Access Link

Saved Machine Status

Local Data

Temporaries

Activities performed by caller and callee during procedure call and return

On caller's side	On the callee's side
Save registers for later use	Control is transformed from the caller to the starting address.

Arrange for the procedure arguments to be found by the procedure.	Save register for later use.
Call the procedure.	Allocate storage for temporaries.
Copy the return value.	Allocate storage for automatic local variables.
Throw array procedure arguments.	Put the return value in appropriate place.
Restore save registers	Throw away local variable and temporaries
	Restore save registers

On the caller's side:

- save registers for later use
  - stack push
  - caller-save registers
- arrange for the procedure arguments to be found by the procedure
  - copy some arguments to registers
  - push additional arguments onto stack
- call the procedure
  - save return address in register or on stack
  - jump to procedure starting address
- later, the return value (if any) is in a pre-determined place, and control is transferred back to the return address
- copy the return value
  - move to another register or store to memory (pop the stack if necessary)
- throw away the procedure arguments
  - adjust stack pointer
- restore saved registers
  - stack pop
- continue

On the callee's side:

- control is transferred from the caller to the starting address
    - return address is in register or on stack
    - arguments are in registers and on stack
- save registers for later use
    - stack push
    - arguments and return address
    - caller's frame pointer
    - other callee-save registers
- allocate storage for automatic local variables
    - set the frame pointer, adjust the stack pointer
    - assign initial values
- allocate storage for temporaries
    - adjust the stack pointer
- put return value in appropriate place
    - register or stack
- throw away the local variables and temporaries
    - adjust the stack pointer
- restore saved registers
    - stack pop
    - arguments and return address
- jump through the return address register

3.8 Different storage allocation strategies are

1. **Static storage allocation:**

- In static allocation, if memory is created at compile time, memory will be created in the static area and only once.
- It doesnot support dynamic data structure i.e., memory is created at compile time and de-allocated after program completion.
- The drawback with static storage allocation is recursion is not supported and size of data should be known at compile time.
- Its advantage is , it is faster than stack storage allocation.

2. Stack Storage Allocation:

- A stack is last in first out storage device where new storage is allocated and de-allocated at only one end, called the top of the stack.
- Storage is organized as a stack and activation records are pushed and popped as activations being and end respectively.
- Storage for the local in each call of procedure is contained in the activation record for that call.
- The values of local are deleted when the activation ends.
- At run time, an activation record can be allocated and de-allocated by incrementing and decrementing top of the stack respectively.

3. Heap storage allocation

- The de-allocation of activation records need not occur in a last-in first-out fashion, so storage cann not be organized as a stack.
- Heap allocation parcels out piecs of contiguous storage, as needed for activation records or other objects.
- Pieces may be de-allocated in any order. So over time the heap consist of alternate areas that are free and in use.
- Heap is an alternate for stack.