

# **Run Time System and Intermediate Code Generation**

**Content :**

1. One pass Assembler
2. Multi pass Assembler
3. Difference between one pass and two pass
4. Role of semantic analyzer
5. Implementation of S-attribute
6. Translation Schemes
7. Implementation of L-attribute
8. Type of Analysis & type Checker
9. Type system

**One pass Assembler**

Go through the source code once and creates equivalent binary program. The assembler substitute all of the symbolic instruction with machine code in one pass.

- **Advantage:** Every source statement needs to be processed once.
- **Disadvantages:** We cannot use any forward reference in our program. Forward reference means a reference to an instruction which has not yet been encountered by the assembler. In order to handle forward reference, the program needs to be scanned twice. In other words, a two pass assembler is needed.

**Multi pass Assembler**

- It means more than one pass is used by assembler.
- It is used to eliminate forward reference in symbol definition.
- It creates a number of passes that is necessary to process the definition of symbols.
- A multi pass assembler
- Does the work in two pass
- Resolves the forward references.
- Scans the code
- Validates the tokens

- Creates a symbol table  
.Solves forward references
- Converts the code to the machine code

### **Difference between one pass and two pass assemblers**

- The basic difference between both is basically in the name.
- One pass assembler passes over the source file exactly once in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references and assemble code in one pass.
- A two pass assembler does two passes over the source file. In the first pass, all it does looks for label definitions and introduces them in the symbol table. In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations and so on.

**Role of semantic analyzer:** The job of semantic analyzer is to ensure that a program conforms to certain extra-syntactic rules and is, therefore, correct. This analysis may be carried out by examining the text and is, therefore, static in nature. It analysis properties that cannot be captured by context free grammars.

**Syntax Directed Definitions:** It is an augmented context free grammar in which each grammar symbol, terminal or non-terminal, has an associated set of attributes. An attribute can be represent anything depending on the type of analysis we are doing. All tokens have at least one attribute which is the associated lexeme.

It is a generalization of a context free grammar where each production  $A \rightarrow \alpha$  is associated with a set of semantic rules of the form

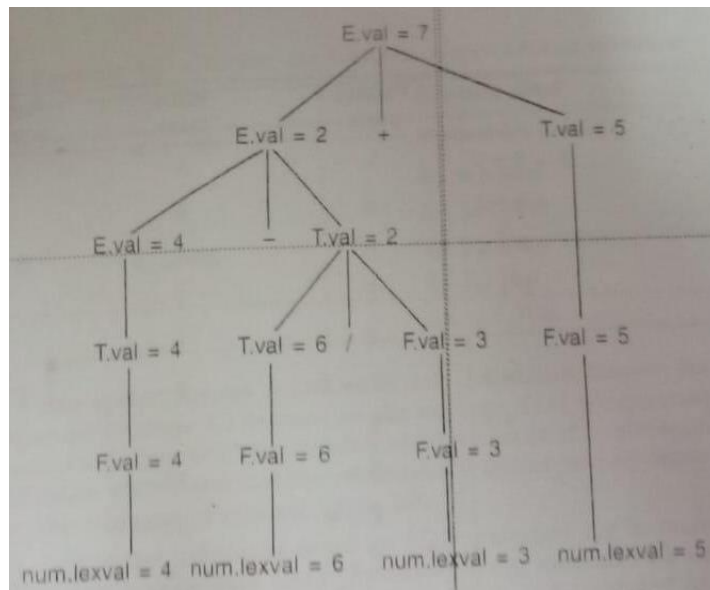
$X : f(a_1, a_2, \dots, a_k)$

Where  $f$  is a function and  $x$  is an attribute. There are two possibilities.

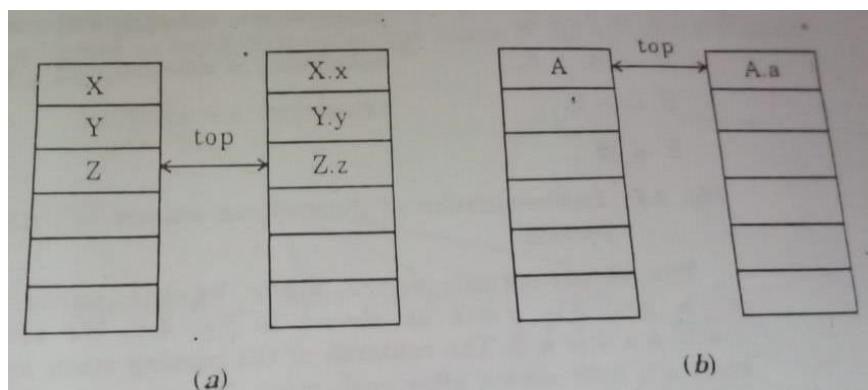
1.  $X$  is a synthesized attribute of  $A$  and  $a_1, a_2, \dots, a_k$  are attributes belonging to the grammar symbols of  $\alpha$ .
2.  $X$  is an inherited attribute of one of the grammar symbols in  $\alpha$  and  $a_1, a_2, \dots, a_k$  are attributes belonging to  $A$  or  $\alpha$ .



**Example:** let us annotate the parse tree for the S - attributed definition for the input 4-6/3+5



**Implementation of S-Attributed Definitions:** It may be implemented with a bottom up parsing strategy. It have only synthesized attributes which can be evaluated by a bottom up parser as the input is being parsed. We augment the parsing stack with a second stack called value stack value so that the parser can store the values of the synthesized attributes associated with the grammar symbols.



The values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right hand side of the reducing production whenever a reduction is made. If there is a semantic rule.

$A \rightarrow XYZ \quad A.a = f(X.x, Y.y, Z.z)$

L-attributed definitions S-attributed schemes do not permit inherited attributes and are too restrictive. Therefore, we now introduce another class of syntax directed definitions known as L-attributed definitions. The attributes of this class can always be evaluated in depth-first order.

A syntax directed definition is said to be L-attributed if for every production rule  $A \rightarrow X_1 X_2 \dots X_n$  of the grammar, each inherited attribute of  $X_j$ ,  $1 \leq j \leq n$ , depends only on

1. The attributes of the symbols  $X_i$ ,  $1 \leq i \leq j - 1$ , i.e. the attributes of the symbols on the left of  $X_j$  in the production.
2. The inherited attributes of  $A$

According to this definition every S-attributed definition is also a L-attributed definition.

	Semantic rule
$A \rightarrow MN$	$M.i := m(A.i)$ $N.i := n(M.i)$ $A.s := f(N.s)$
$A \rightarrow RS$	$S.i := s(A.i)$ $R.i := r(S.i)$ $A.s := f(R.s)$

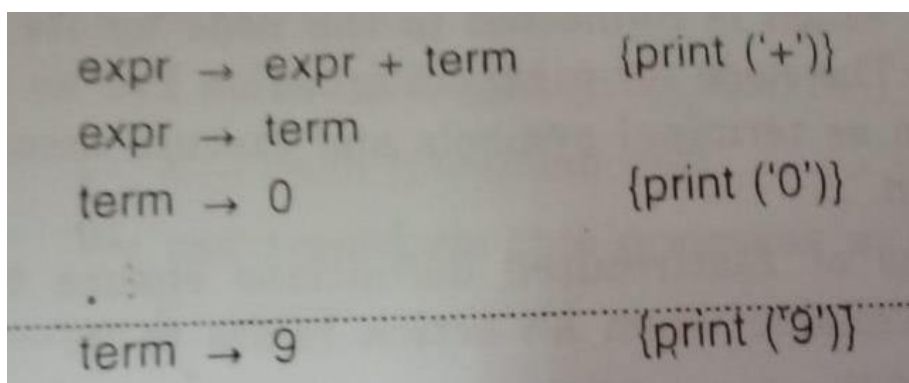
**Example-** consider the following syntax directed definition

This syntax directed definition is not L attributed because the inherited  $R.i$  depends on the attribute  $S.i$  of the grammar symbol to its right which is a violation of the restriction of L-attributed definition which states that the attributes of a symbol can only depend on the attributes of symbols on its left.



**Translation Schemes:** it is a refinement of a syntax directed definition in which semantic actions enclosed between braces{} are inserted within the right sides of production to specify the exact point in time when the actions are to be executed.

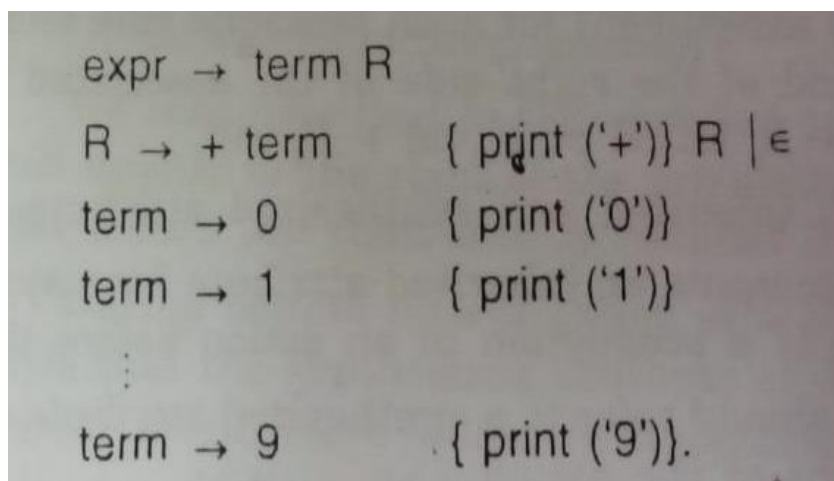
It generates an output for each sentence  $x$  generated by the underlying grammar by executing the actions in the order they appear during a depth-first traversal of a parse tree for  $x$ . For example in the translation scheme  $A \quad X_1 \dots X_i \{action\} X_{i+1} \dots X_n$ , action will be performed after the sub tree for  $X_i$  is traversed but before the beginning of  $X_{i+1}$



```
expr → expr + term    {print ('+' )}
expr → term
term → 0               {print ('0' )}
⋮
term → 9               {print ('9' )}
```

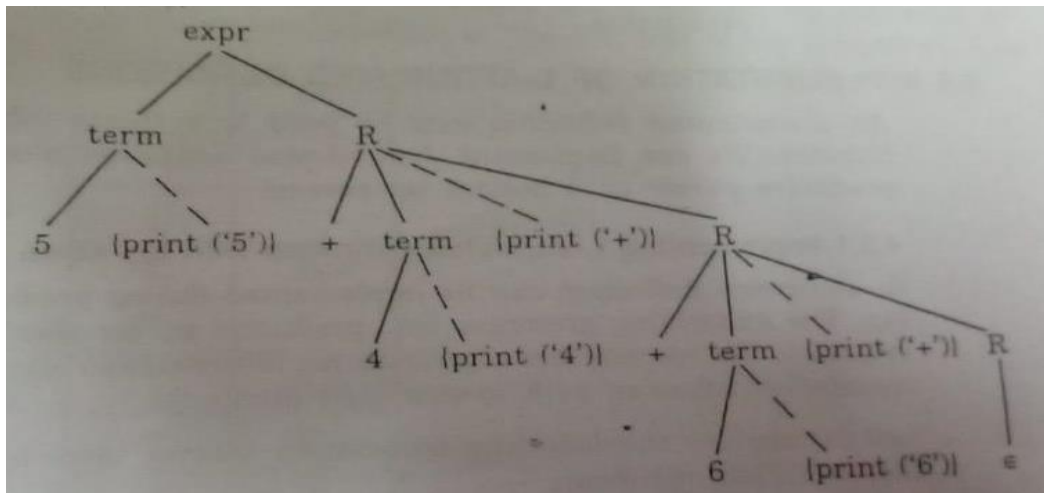
**Example:** Let us consider the grammar to generate simple expressions in infix form and a translator which converts it into postfix form.

We can remove left recursion from this grammar. The transformed translation scheme is as follows:



```
expr → term R
R → + term    { print ('+' )} R | ε
term → 0      { print ('0' )}
term → 1      { print ('1' )}
⋮
term → 9      { print ('9' )}.
```

We can show the parse tree for the input 5+4+6 with each semantic action attached as the appropriate child of the node corresponding to the left side of their production. When perform in depth-first order, the action print the output 5 4 + 6 +



**Implementation of L- attributed definitions:** it may be used to evaluate inherited attributes. We can implement L- attributed definition along with a predictive parser or a bottom up parser.

**Implementation of L- attributed definitions with top down parser:** L -attributed definition can be implemented during predictive parsing. The underlying grammar for predictive parser should be non- recursive.

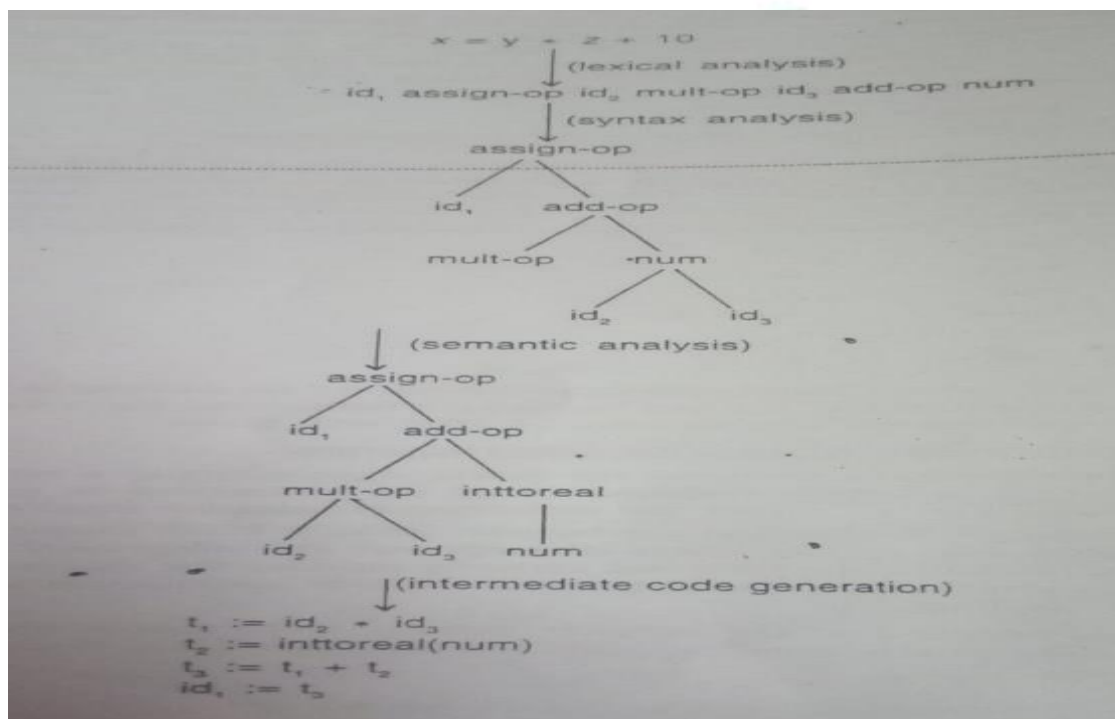
Implementation of L- attributed scheme with bottom up parsing: in addition to L-attributed definition based on an LL(1) grammar, this method is capable of handling many L- attributed definitions based on LR(1) grammars.

**Type Analyses and type checking:** semantic analyzer checks that the source program is semantically correct. Static type checking which is an important part of semantic analysis ensures that certain type of checks like uniqueness check, flow of control checks etc. must be performed and if there is any error, it must be reported. This type checker is required to detect and report error if an operator is applied to an incompatible operand.

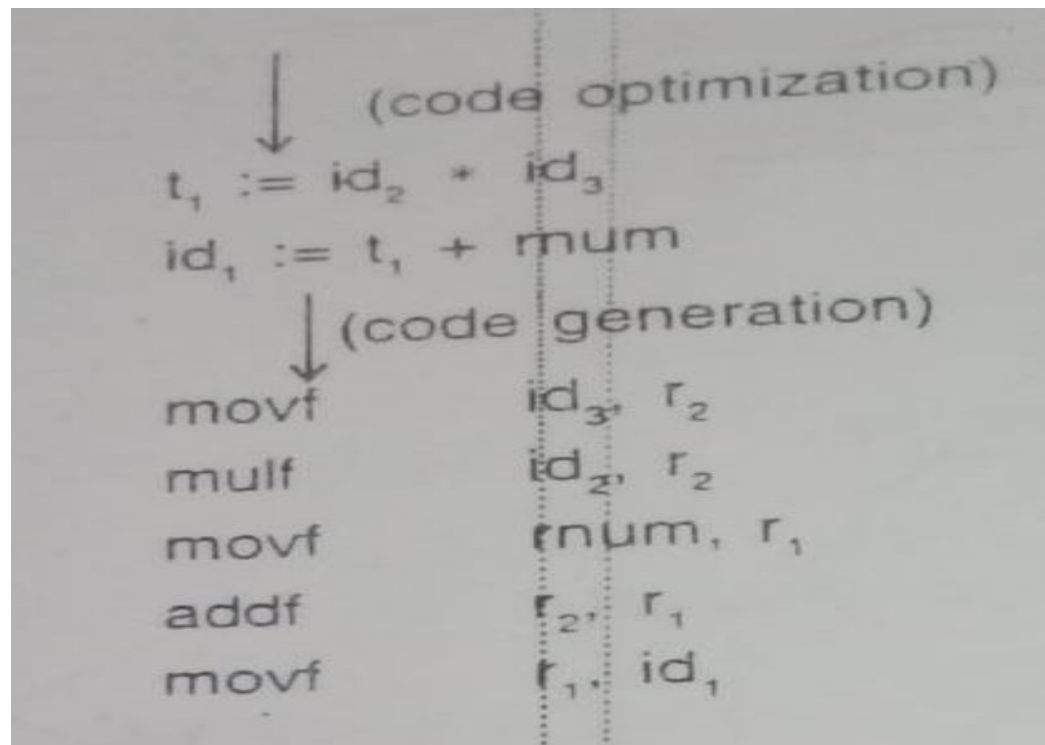
**Type system:** It is a component of a language which keeps track of the types of variables and in general, of the types of all expressions in a program.

There are two kind of execution errors-trapped errors and un-trapped errors

1. Trapped errors are the ones that cause the computation to stop immediately. For example division by zero and accessing an illegal address.
2. Un-trapped errors are the ones that go un-noticed for a while and later cause arbitrary behavior. For example, accessing data past the end of an array in absence of run time bounds checks.







gradeup





# Gradeup UGC NET Super Superscription

## Features:

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

---

Gradeup Super Subscription, Enroll Now