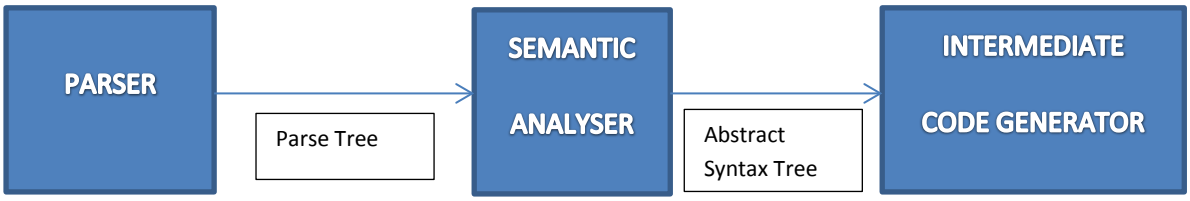
21nd OCTOBER 2013

COMPILER CONSTRUCTION

SEMANTIC ANALYSIS

- **Semantic Analysis:**



Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and generates the symbol table. Semantic checks are also performed which may include type checking. Parse Tree is passed as the input for Semantic Analyser. Semantic Analyser generates the Abstract Syntax Tree which further passes to the Intermediate Code Generator. Using this Abstract Syntax Tree the Intermediate Code Generator generates the Three Address Codes. This completes the Front End of the Compiler.

- **Attributes:**
  We associate additional information to every grammar symbol (Terminals and Non Terminals) of the Context Free Grammar (CFG). This additional information is called attribute. Attribute can be of any type for example string, integer, table, references or even a fragment of code. There are two types of attributes Synthesized and Inherited.

- **Synthesized Attributes:**
  The attributes of a Nonterminal which can be computed only from the attribute values of children and the node itself from the generated parse tree are called Synthesized Attributes.

- **Inherited Attributes:**
  The attributes of a Nonterminal which can be computed only from the attribute values of parents, siblings and the node itself are called Inherited Attributes.

  Thus from definitions above Synthesized attribute computation cannot involve the attributes of parent or sibling while in the computation of an Inherited attribute the attribute of the children is not allowed.

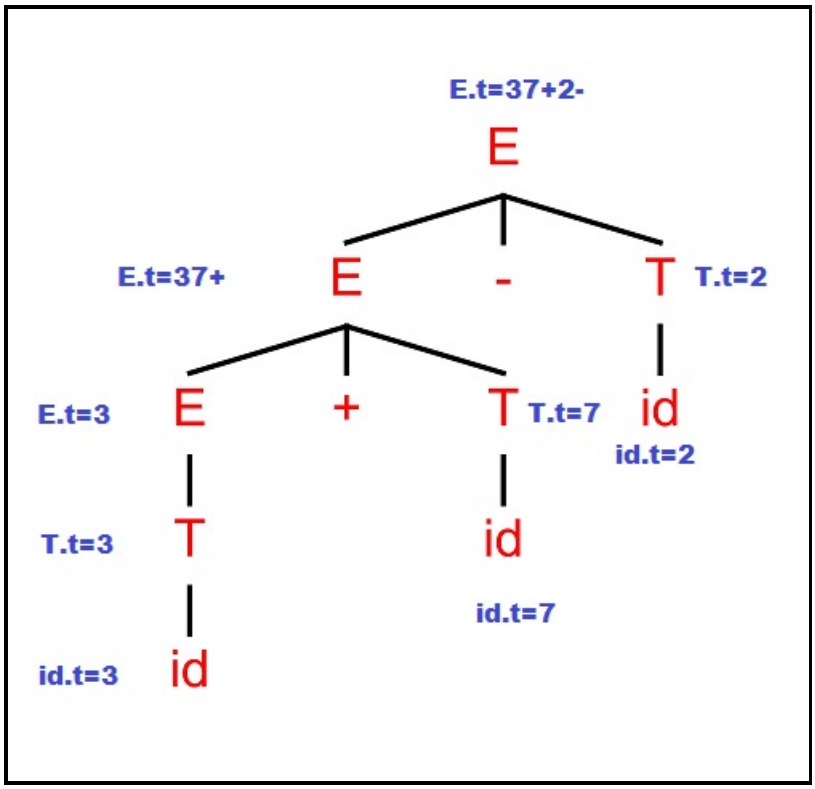
Terminals can only have synthesized attributes. They are generated by the Lexical Analyser. While Non Terminals can be associated with both Inherited and synthesized attributes. Semantic Rules are associated with every production of the CFG. Semantic Rules are nothing but rules involving computation of the attributes.

- **Syntax Directed Definition(SDD):**
  A Syntax Directed Definition (SDD) is a Context Free Grammar associated with attribute for every symbol and semantic rules.

- **Example 1: INFIX to POSTFIX Conversion**
  Now we illustrated the power of SDD by showing how to convert from Infix to Postfix. Let us consider the following Expression Grammar:

$E \rightarrow E + T$	$E.t = E.t \,\|\|\, T.t \,\|\|\, +$
$E \rightarrow E - T$	$E.t = E.t \,\|\|\, T.t \,\|\|\, -$
$E \rightarrow T$	$E.t = T.t$
$E \rightarrow id$	$E.t = id.t$

The semantic rule associated with each production is shown on the right side. The attributes of the Grammar symbol are denoted by '.t' (Any symbol can be used after '.'). Here our attributes are of the type sting. In the semantic rules '||' stands for concatenation operator. Any symbol of the grammar can be associated with one or more nodes in the parse tree. The attribute of id (terminal) comes from the Lexical Analyser and is the value of the integer in this case.



Fig(2):Annotated Parse Tree along with the Attributes of the Grammar Symbols denoted in Blue for Infix to Postfix Conversion

Let us perform the infix to postfix conversion of the expression 3+7-2 using the SDD. First we parse this expression.

$E \rightarrow E+T$

$\rightarrow E+T-T$

$\rightarrow T+T-T$

$\rightarrow id(3)+id(7)-id(2)$

Note that we have used synthesized attributes in the SDD.

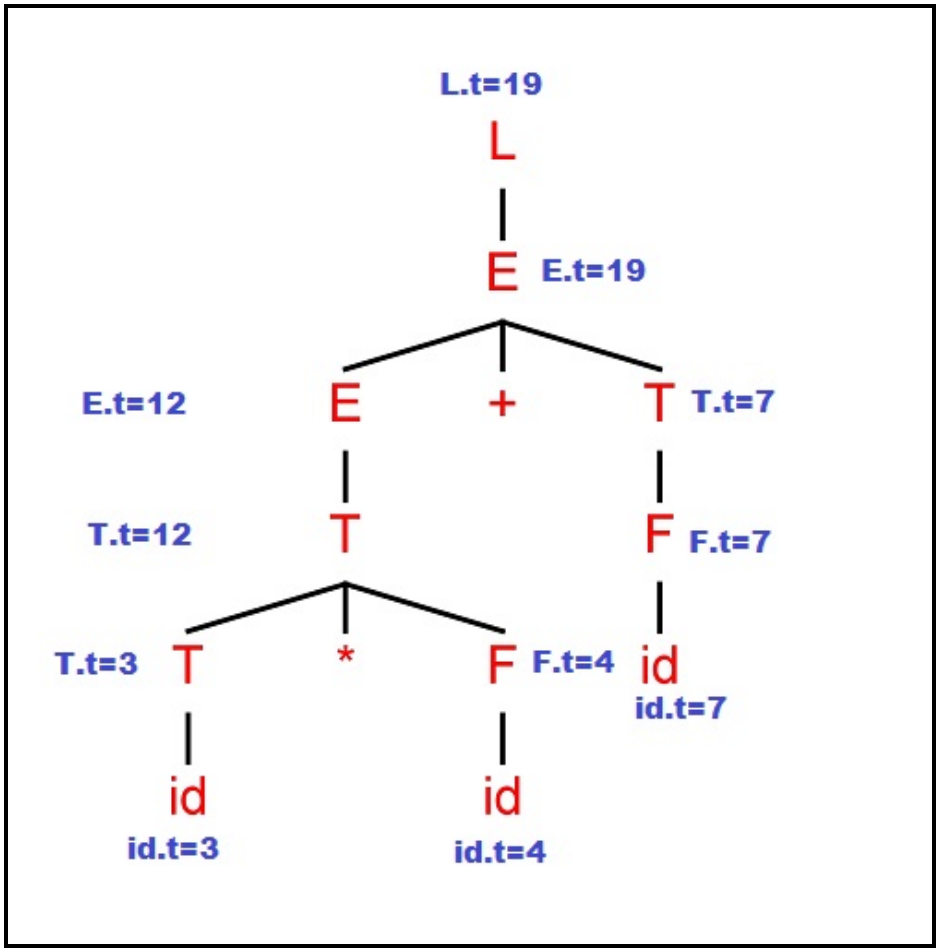
The figure shows how the Postfix Expression is obtained at the parent node.

- **Example 2: EXPRESSION Evaluation**

Let us consider one more example of classic expression evaluator using SDD.

Consider the following CFG:

$L \rightarrow E$	$L.t = E.t$
$E \rightarrow E+T$	$E.t = E.t + T.t$
$E \rightarrow T$	$E.t = T.t$
$T \rightarrow T*F$	$T.t = T.t * F.t$
$T \rightarrow F$	$T.t = F.t$
$F \rightarrow id$	$F.t = id.lexval$



Fig(3):Annotated Parse Tree along with the Attributes of the Grammar Symbols denoted in Blue for Expression Evaluation.

Here our attributes are of the type integer.
Suppose we want to evaluate 3*4+7.

The derivation will be as follows:

$$L \rightarrow E$$
$$\rightarrow E+T$$
$$\rightarrow T+T$$
$$\rightarrow T*F+F$$
$$\rightarrow F*F+F$$
$$\rightarrow id(3)*id(4)+id(7)$$

The final value of this expression (19) is obtained in the attribute of L (Parent Node).

- **Abstract Syntax Tree:**

The abstract syntax tree is generated by the Semantic Analyser. All the internal nodes of AST are operators while the external nodes are operands.
Parse tree depends on our Context Free Grammar (CFG) while Syntax Tree Depends on the way we choose to evaluation the expression and also on the design of the Syntax Directed Definition(SDD) . For example consider the follow grammar generated expressions for multiply of two or more operands:

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

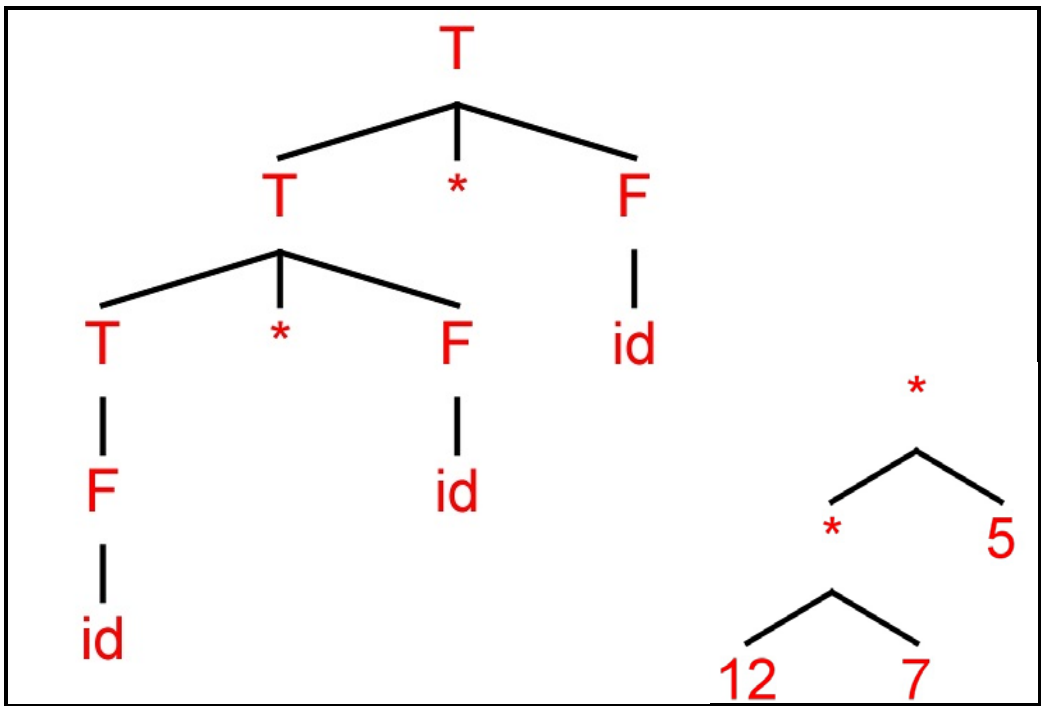
Grammar 1

This CFG is suitable for LR Parsing. We need to remove the Left Recursion if we want to use the CFG for LL(1) Parsing. The modified Grammar becomes:

$$T \rightarrow FT'$$
$$T' \rightarrow *FT'$$
$$T' \rightarrow \in$$
$$F \rightarrow id$$

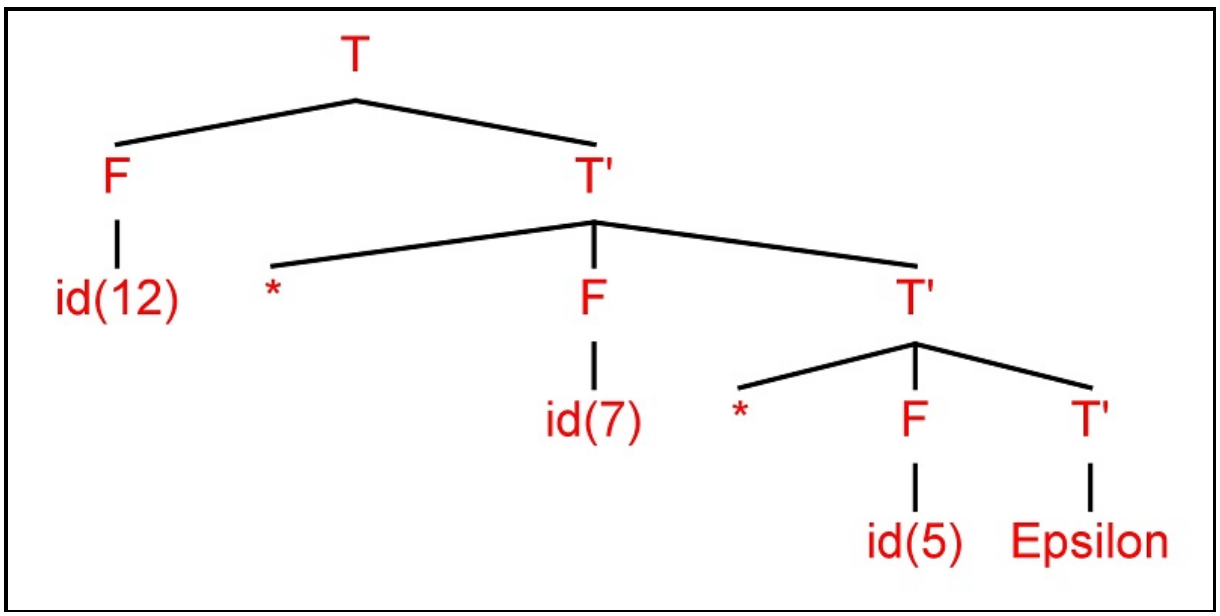
Grammar 2

We want to demonstrate for this expression 12*7*5.



Fig(4)Parse tree using Grammar 1 on Right and Syntax Tree on Left

We observe similarity between the parse tree of Grammar 1 and the Syntax Tree. But no such similarity is observed for the parse tree of Grammar 2. It suggests that that evaluation of S Attributed SDD (will be covered in next lecture) can be done easily using Bottom Parsing and post order traversal of the parse tree. For the sake of completeness we mention that L Attributed SDD (yet to be covered) can be integrated very easily with the LL(1) or the Top Down Parsers.



Fig(5)Parse tree using Grammar 2