Unit 3
Chapter 3
Syntax Directed Translation

Structure of the Unit

1. Syntax-Directed definitions
2. Constructions of Syntax Trees
3. Bottom-up evaluation of S-attributed definitions
4. L-attributed definitions
5. Top-down translation

3.1 Course Outcomes

After the successful completion of this unit, the student will be able to:

1. Define Syntax-Directed Translation.
2. Compare and contrast S-attributed and L-attributed definitions in Syntax-Directed Translation.
3. Create Syntax Trees for given code snippets.
4. Evaluate the advantages and disadvantages of Top-down vs. Bottom-up translation approaches.
5. Design a program that utilizes Syntax-Directed Translation for a specific task.

3.2 Syntax Directed Definitions

This chapter deals with the translation of languages guided by context-free-grammars. Information is associated with the programming language constructs by attaching attributes with the grammar symbols representing the construct. The values of these attributes are computed by the "semantic rules" associated with the productions.

There are two notations for associating the semantic rules with the productions, *syntax directed definitions* and *translation schemes*. *Syntax directed definitions* are high level specifications. Many implementation details are hidden and the user need not specify explicitly the order in which the translation takes place. *Translation schemes* indicate the order in which the semantic rules are to be evaluated. Hence, some implementation details are to be shown. The techniques discussed here will be applied to type checking and intermediate code generation. Conceptually in both the schemes we parse the input token stream, build the parse tree and traverse it to evaluate the attributes(Fig 3.1).

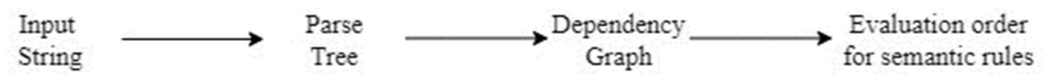

Fig 3.1 Conceptual view of Syntax-Directed Translation

Evaluation of semantic rules may

- Generate code,
- Save information in the symbol Table,
- Issue Error Messages or
- Perform any other activities.

A **Syntax-Directed Definition (SDD)** is a formal specification used in compiler design. It combines the syntax of a programming language with semantic actions, providing a way to associate specific computations or transformations with language constructs.

In essence, an SDD augments the context-free grammar of a programming language by associating attributes with grammar symbols(terminals or non-terminals) and actions with productions. These attributes are evaluated by executing the actions during parsing.

Grammar and Semantics Integration: SDDs bring together the syntax (structure) of a language and its semantics (meaning). They describe how the program's structure is related to its behavior.

Attributes: SDDs use attributes to associate data with grammar symbols, such as terminals and non-terminals. These attributes can represent various types of information, like type information, values, or memory locations.

Semantic Actions: Semantic actions are code snippets or functions associated with grammar productions. They are executed when parsing reaches specific parts of the input code and can perform computations, modify attributes, or generate intermediate code. Semantic actions are enclosed within braces '{' and '}'. Its position in the production body determines the order in which the action is performed.

Traversal: SDDs often specify the order in which grammar productions are processed, defining how the parsing process traverses the syntax tree or input code.
SDDs play a critical role in compiler construction by providing a formal framework for defining the semantics of a programming language. They are used to generate intermediate representations of code, perform type checking, optimize programs, and produce target code. In summary, Syntax-Directed Definitions bridge the gap between the syntax and semantics of a programming language, enabling the development of compilers and translators that can understand and process code correctly.

3.3 Syntax-Directed-Definitions

A *syntax directed definition* (SDD) is an extension of the context free grammar where grammar symbols are associated with attributes and productions with semantic actions. If X is a grammar symbol and a is one of its attributes, it is denoted by $X.a$.

3.3.1 Inherited and Synthesized Attributes

The attributes associated with the non-terminals are classified as *Synthesized* and *Inherited* attributes.
Synthesized Attributes: The synthesized attribute for a non-terminal A at the parse tree node N is defined by the semantic rule associated with the production at N. The non-terminal A must be the head of the production. The value of the synthesized attribute at N is computed as function of the values of the attributes at the children of N and N itself.
Inherited Attributes: An inherited attribute for a non-terminal B at the node N in the parse tree is computed by the semantic rule associated with production at the parent of N. B must be a symbol in the production body. The value of the inherited attribute at N is computed as function of the values of the attributes at the parent of N, N itself and N's siblings.

Terminals may have synthesized attributes but not inherited attributes. Attributes for terminals(leaf nodes in the parse tree) have lexical values supplied by the lexical analyzer.

Example 1: SDD to evaluate an arithmetic expression with $+$ and $*$ as operators and $number$ as an operand.

Solution: We extend the unambiguous grammar for arithmetic expressions. The expression is terminated by some symbol. We assume the symbol **n** terminates the expression. Each non-terminal has one synthesized attribute val. The operand $number$(a terminal) is associated with the synthesized attribute $lexval$, an integer whose value is returned by the lexical analyzer.

PRODUCTION	SEMANTIC ACTION
1) $S \rightarrow E\ \boldsymbol{n}$	$S.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow digit$	$F.val = digit.lexval$

Table 3.1 SDD for desktop calculator

The above SDD involves only synthesized attributes. The SDD that involves only synthesized attributes is called $S - attributed$ definition. The $S - attributed$ definition is normally implemented with a LR parser.

3.3.2 Evaluating the SDD

The order in which the semantic actions are performed(translation) by the SDD is visualized better with a parse tree. The translator need not actually build the parse tree. Assume the parse tree is constructed first and then the rules are applied to evaluate the attributes at each of the nodes. The parse tree showing the values of the attributes is called an *annotated parse tree*.

It is difficult to determine the order in which the values of the attributes are evaluated. If all the attributes are synthesized($S - attributed$ definition) then any bottom-up order, such as post order traversal is used to evaluate the attributes. If the SDD has both synthesized and inherited attributes there is no particular order in which the semantic actions are performed.

Example 2: Show the annotated parse tree for the expression $4 * 5 + 7$ **n** using the SDD in table 3.1
Solution: The annotated parse tree is shown in Fig 3.2.

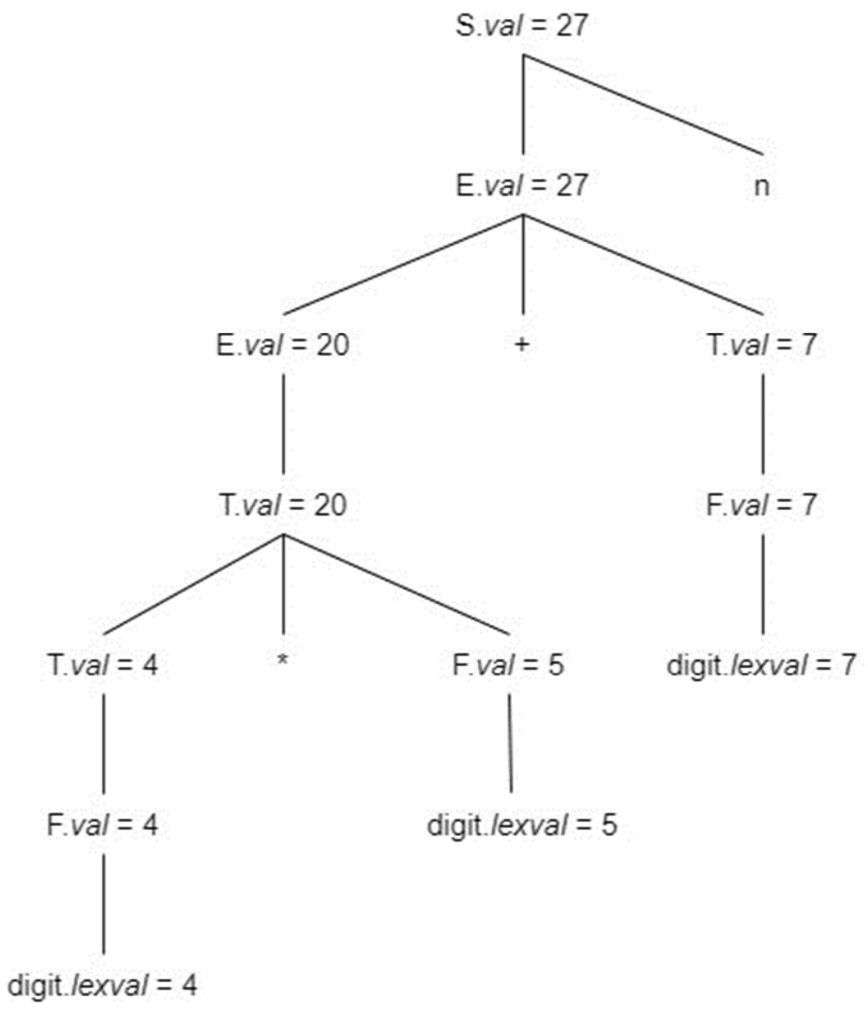

Fig 3.2 Annotated Parse Tree

Example 3: Write the Syntax Directed Definition to associate data types for the identifiers in a variable declaration statement.

Solution:

The context free grammar for variable declaration statement is shown below.

$$D \rightarrow T\ L$$
$$T \rightarrow \boldsymbol{int}$$
$$T \rightarrow \boldsymbol{real}$$
$$L \rightarrow L, \boldsymbol{id}$$
$$L \rightarrow \boldsymbol{id}$$

In the above grammar the non-terminal

- D represents a declaration statement,
- T represents the data type and
- L represents the list of one or more identifiers separated by comma.

Table 3.2 shows the SDD. The non-terminal T is associated with a synthesized attribute $type$ and the non-terminal L is associated with an inherited attribute in. The function $addtype()$ locates the identifier in the symbol table and associates the data type with it. The attribute $\boldsymbol{id}.entry$ is the pointer to the identifier in the symbol table and is supplied by the lexical analyzer.

PRODUCTION	SEMANTIC RULE
$D \rightarrow T\ L$	$L.in := T.type$
$T \rightarrow \boldsymbol{int}$	$T.type := integer$
$T \rightarrow \boldsymbol{real}$	$T.type := real$
$L \rightarrow L_1, \boldsymbol{id}$	$L_1.in \coloneqq L.in$ $addtype(\boldsymbol{id}.entry, L.in)$
$L \rightarrow \boldsymbol{id}$	$addtype(\boldsymbol{id}.entry, L.in)$

Table 3.2 SDD for variable declaration statement

The annotated parse tree for the declaration $real\ id_1, id_2, id_3$ is shown in Fig 3.3

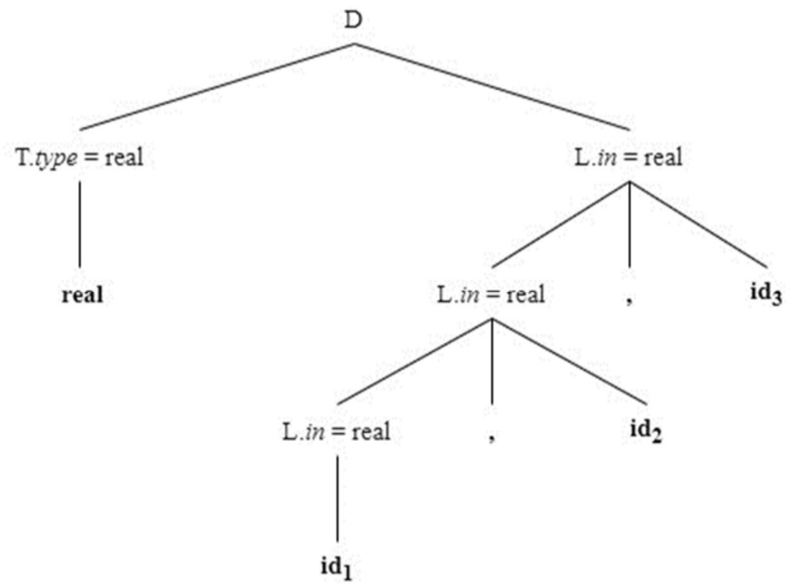

Fig 3.3 Annotated parse tree for the declaration $real\ id_1, id_2, id_3$

3.3.3 Dependency Graphs

Dependency graphs are used to illustrate the interdependencies among the synthesized and inherited attributes at the parse tree nodes. Dependency graph is a directed graph. If a semantic rule consists of a procedure call, then a semantic rule of the form $b \coloneqq f(c_1, c_2, \cdots, c_k)$ is introduced. A dummy synthesized attribute b is used. Each attribute a associated with a parse tree node is represented by a vertex(node) in the dependency graph. The dependency graph has an edge from node c to node b if the attribute b depends on attribute c.

```
for each node $n$ in the parse tree do
 for each attribute $a$ of the grammar symbol at $n$ do
     construct a node in the dependency graph for $a$;
for each node $n$ in the parse tree do
 for each semantic rule $b \coloneqq f(c_1, c_2, \cdots, c_k)$ associated with the production at $n$ do
   for $i \coloneqq 1$ to $k$ do
     construct an edge from the node for $c_i$ to the node $b$;
```
Algorithm 3.1 Constructing dependency graph

Let $A.a \coloneqq f(X.x, Y.y)$ be the semantic rule associated with the production $A \to XY$, the dependency graph in Fig 3.4 shows the interdependencies among the attributes.

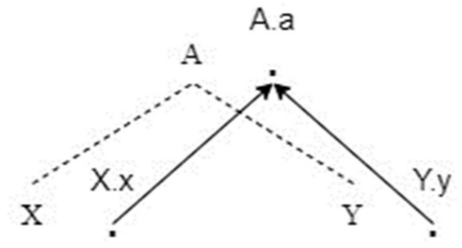

Fig 3.5 dependency graph

The vertices of the graph(attributes) are represented by **.**, the dotted lines represent the edges of the parse tree whereas the solid lines represent the dependency graph edges(interdependencies).

Evaluation order:

The evaluation order of the semantic rules in a SDD is obtained by performing the *topological sorting* of the dependency graph. The topological sort of a directed acyclic graph is an oredering $m_1, m_2, \cdots, m_k$ of the graph nodes such that edges are from the nodes earlier in the order to the nodes late in the order. If $m_i \to m_j$ is an edge in the graph then m_i appears before m_j in the *topological order*.

The context free grammar is used to construct the parse tree for the input string. The dependency graph is constructed using the Algorithm 3.1. Topological sorting of the dependency graph gives the evaluation order of the semantic rules.

3.4 Construction of Syntax Trees

Syntax tree is an intermediate representation that allows translation to be decoupled from parsing. Syntax trees provide a visual representation of the hierarchical structure of a sentence(statement), making it easier to analyze and understand the syntactic relationships between its components. Parsing algorithms are used to generate syntax trees from sentences(statements), which can then be used for various purposes, such as machine translation, grammar checking, and syntactic analysis.

Syntax tree is a condensed form of parse tree. In a syntax tree operators and keywords appear as internal nodes whereas the operands and other tokens appear as leaves. Unlike the parse tree in a syntax tree the chains of single productions are collapsed.

Syntax trees are composed of nodes and edges, where nodes represent tokens, and edges represent the syntactic relationships between them. Here's a basic overview of the components of a syntax tree:

Root Node: The topmost node of the tree, which represents the entire sentence(statement).

Terminal Nodes (Leaves): Nodes at the bottom of the tree that represent individual tokens or lexical items. These nodes do not have any children.

Non-terminal Nodes: Nodes that represent phrases or constituents. These nodes have one or more child nodes and may be further subdivided into smaller constituents.

Branches/Edges: The lines connecting nodes in the tree, which indicate the hierarchical relationships between words and phrases. These edges show how constituents are combined to form larger constituents.

Constructing Syntax Trees for Expressions:

The least precedence operator is the root of the tree. Subtrees are constructed for subexpressions by creating a node for each operator and operand. The children of an operator node are roots of the subtrees representing subexpressions.

Each node is represented by a record with several fields.

- For an operator, one field is the operator itself and other fields are pointers to the roots of the subtrees or operands.
- For an operand one field is the token and the other field is the attribute of the token(as returned by the lexical analyzer).

We use the following functions to create nodes of the syntax tree. Each function returns a pointer to the newly created node.

1. $mknode(op, left, right)$ creates an operator node with op as the label and two fields containing pointers $left$ and $right$.
2. $mkleaf(\boldsymbol{id}, entry)$ creates a node for the identifier with the label $\boldsymbol{id}$ and the field containing $entry$, a pointer to the symbol entry for the identifier.
3. $mkleaf(\boldsymbol{num}, val)$ creates a node for the number with the label $\boldsymbol{num}$ and a field containing val, the value of the number.

Example 1: Write the sequence of function calls to create the syntax tree for the expression $a + 6 * b$ and show the syntax tree.

Solution: Following is the sequence of function calls;

1) $p_1 = mkleaf(\boldsymbol{num}, 6);$
2) $p_2 = mkleaf(\boldsymbol{id}, entryb);$
3) $p_3 = mknode('*', p_1, p_2);$
4) $p_4 = mkleaf(\boldsymbol{id}, entrya);$
5) $p_5 = mknode('+', p_4, p_3);$

The tree is shown in Fig 3.6.

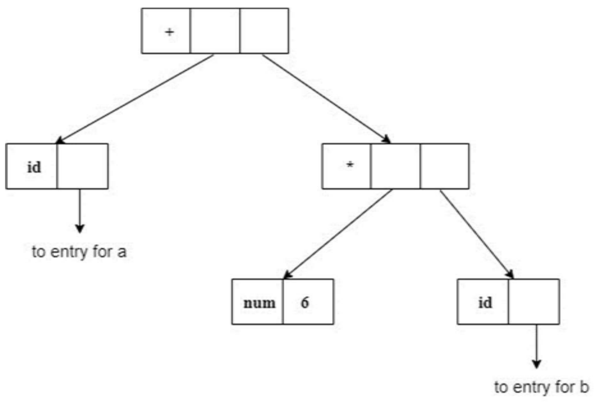

Fig 3.5 Syntax tree for the expression $a + 6 * b$

Syntax Directed Definition to Construct the Syntax Tree:

PRODUCTION	Semantic Rule
$E \rightarrow E_1 + T$	$E.ptr = mknode('+', E_1.ptr, T.ptr)$
$E \rightarrow T$	$E.ptr = T.ptr$
$T \rightarrow T_1 * F$	$T.ptr = mknode('*', T_1.ptr, F.ptr)$
$T \rightarrow F$	$T.ptr = F.ptr$
$F \rightarrow (E)$	$F.ptr = E.ptr$
$F \rightarrow \boldsymbol{id}$	$F.ptr = mkleaf(\boldsymbol{id}, \boldsymbol{id}.entry)$
$F \rightarrow \boldsymbol{num}$	$F.ptr = mkleaf(\boldsymbol{num}, \boldsymbol{num}.val)$

Table 3.3 SDD to construct syntax tree

Directed Acyclic Graphs(DAG) for expressions:

Consider the expression $(a - b) * c + d/(a - b)$. The subexpression $a - b$ is repeated. The syntax tree for the expression is shown in fig 3.6. It is evident from the figure that there are two nodes representing the subexpression $a - b$. When this intermediate representation is converted to the machine code, the code to evaluate the subexpression $a - b$ will be generated twice and also executed twice even though the result of the subexpression is same at both the places. This adds to the execution time and also the size of the object code produced is large.

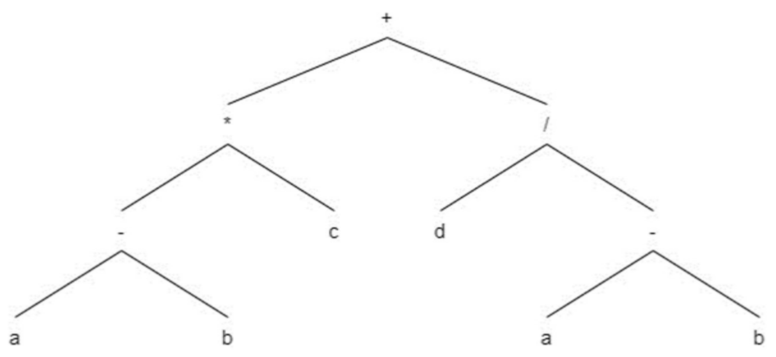

Fig 3.6 Syntax tree for the expression $(a - b) * c + d/(a - b)$

A directed acyclic graph is similar to a syntax tree, but the common nodes are represented once. For example, in the syntax tree of figure 3.6 the subexpression $a - b$ has two nodes, whereas the *dag* for will have only one node. Hence the nodes representing common subexpressions have more than one "parent". The *dag* is shown in Fig 3.7. The node for the operator – has two parents.

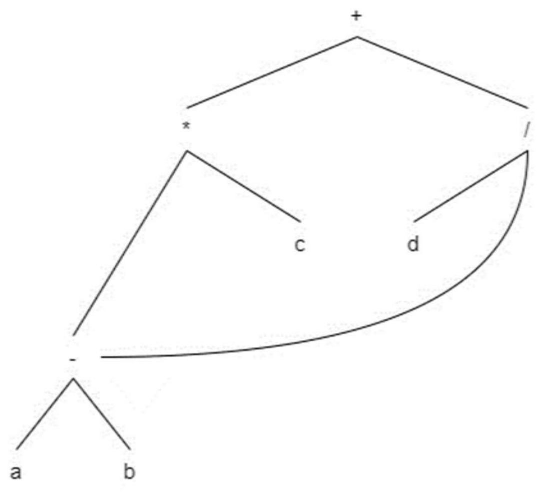

Fig 3.7 DAG for the expression $(a - b) * c + d/(a - b)$

The SDD in table 3.3 may be used with little modifications to create a *dag*. The functions for creating the nodes are modified. The function creating a node, first checks to see if the node already exists. If so, the function returns a pointer to the already existing node, otherwise it will create a new node and return the pointer to the new node. The sequence of function calls to create the *dag* follows;

1) $p_1 = mkleaf(\boldsymbol{id}, entrya)$;

2) $p_2 = mkleaf(\boldsymbol{id}, entryb)$;

3) $p_3 = mknode('-', p_1, p_2)$;

4) $p_4 = mkleaf(\boldsymbol{id}, entryc);$

5) $p_5 = mknode('*', p_3, p_4)$;

6) $p_6 = mkleaf(\boldsymbol{id}, entryd);$

7) $p_7 = mkleaf(\boldsymbol{id}, entrya) = p_1$; The node for the identifier a already exists, so returns p_1

8) $p_8 = mkleaf(\boldsymbol{id}, entryb) = p_2$; The node for the identifier b already exists, so returns p_2

9) $p_9 = mknode('-', p_1, p_2) = p_3$; The node for the expression $a - b$ already exists, so returns p_3
OR $p_9 = mknode('-', p_7, p_8);$

10) $p_{10} = mknode('/', p_6, p_9)$ **OR** $p_{10} = mknode('/', p_6, p_3);$

11) $p_{11} = mknode('+', p_5, p_{10});$

The call $mkleaf(\boldsymbol{id}, entrya)$ is made on line 1 and is repeated on line 7. Hence the node constructed on line 1 is returned i.e., $p_7 = p_1$. Similarly, the function call $mkleaf(\boldsymbol{id}, entryb)$ is made on line 2 and is repeated on line 8. Hence the node constructed on line 2 is returned i.e., $p_8 = p_2$. Also, the calls on line 9 and 3 are same. Hence $p_9 = p_3$.
This *dag* can be represented as an array of nodes(records). The structure of the record depends on the type of the node.

Operand nodes:

- *Label field*: It is the token type. For identifiers it is **id** and for numbers it is **num**.
- *Attribute field*: For identifiers it is pointer to the symbol table entry and for numbers it is the number itself.

Operator nodes:

- *Label field*: The operator itself.
- *Pointers*: Pointers pointing to the operands on which the operator is applied.

The *dag* for the expression $a = a * 20$ is depicted in Fig 3.8 and the array representation of the *dag* is shown in table 3.4.

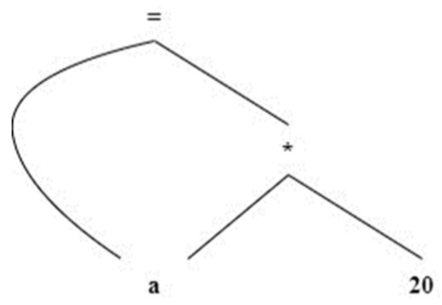

Fig 3.4 Directed Acyclic Graph for the expression $a = a * 20$

1	**id**	→		Pointer to entry for **a**
2	**num**	20		
3	*	1	2	
4	=	1	3	

Table 3.4 Array representation of the *dag* in fig 3.4

The index of the node in the array is called the *value number*. The method for constructing the *dag* is known as the *value number method*.

3.5 Bottom-Up Evaluation of S-attributed Definition

In the earlier sections we have discussed about the usage of SDDs to specify translations. In this section we begin to study how to implement them. S-attributed definition refers to SDD where all the attributes are synthesized attributes only. Bottom-up evaluation means the values of these attributes are computed as the parsing process progresses from the leaves of the syntax tree to its root.

Synthesized attributes can be evaluated by the bottom-up parser while scanning the input. The parser keeps the values of synthesized attributes on its stack. Whenever the parser makes a reduction, the semantic rule(action) associated with the reducing production is executed. The values of the attributes are computed using the values of the attributes appearing on the stack(these attributes correspond to the grammar symbols on the right-hand side or body of the production. We shall see how parser stack can be extended to hold the values of the attributes.

Synthesized Attributes on the Parser Stack

The bottom-up parser uses a stack to hold the information about the subtrees that have been parsed. We can use extra fields in the parser stack to hold the values of the attributes. Table 3.5 shows the parser stack containing the grammar symbols along with their attributes. X, Y, Z are the grammar symbols and $X.x, Y.y, Z.z$ are their attributes.

	$grammar\ symbol$	$attribute\ value$
	$\cdots$	$\cdots$
	X	$X.x$
$top\ \rightarrow$	Y	$Y.y$
	Z	$Z.z$
	$\cdots$	$\cdots$

Table 3.5 Parser Stack with attribute values

The top of stack is indicated by the pointer top. Suppose $A \rightarrow XYZ$ is a production in the grammar and $A.a = f(X.x, Y.y, Z.z)$ is the semantic rule associated with it. When the parser action is to reduce by the production $A \rightarrow XYZ$, the semantic rule $A.a = f(X.x, Y.y, Z.z)$ is executed. The attributes $X.x, Y.y, Z.z$ are popped along with the grammar symbols X, Y, Z, the action is performed and the newly computed attribute $A.a$ is along with the grammar symbol A is shifted on the stack. The stack contents are shown in table 3.6.

	$grammar\ symbol$	$attribute\ value$
	$\cdots$	$\cdots$
$top\ \rightarrow$	A	$A.a$
	$\cdots$	$\cdots$

Table 3.6 Parser Stack after the execution of the semantic rule

Consider the SDD for the desktop calculator in section 3.3.1(Table 3.1). We shall see the usage of stack to implement the desktop calculator. The array val is used to implement the stack that holds the attributes. The variable top points to the current top of the stack. The implementation is shown in table 3.7.Since the SDD is s-attributed all the attributes are evaluated in the bottom-up order and hence bottom-up parsing may be used.

PRODUCTION	SEMANTIC ACTION
1) $S \rightarrow E\ \boldsymbol{n}$	print($val[top]$)
2) $E \rightarrow E_1 + T$	$val[ntop] = val[top-2] + val[top]$
3) $E \rightarrow T$	
4) $T \rightarrow T_1 * F$	$val[ntop] = val[top-2] * val[top]$
5) $T \rightarrow F$	
6) $F \rightarrow (E)$	$val[ntop] = val[top-1]$
7) $F \rightarrow digit$	Lexer places the value of digit in $val[top]$

Table 3.7 Implementation of the desktop calculator with LR parser

The SDD does not show how the variables top and $ntop$ are managed. When the parser is reducing by a production $A \rightarrow \beta$ and $r = |\beta|$ i.e., r is the length of the right-hand side of the production then r symbols are popped of the stack and the variable $ntop$ is set to $n - r + 1$. After the execution of each code segment top is set to $ntop$.

3.6 L-Attributed Definitions

When the SDD has both synthesized and inherited attributes, the natural order to execute the semantic rules is visiting the parse tree nodes in the *depth first order*. Even though the parse tree is not actually constructed, it is useful to study the *depth-first* evaluation of attributes to understand the translation process. The *depth-first* evaluation order of attributes is represented in algorithm 3.2.

```
procedure dfsvisit(n: node) {
    for each child $m$ of $n$ do {
        evaluate inherited attributes of $m$;
        dfsvisit($m$);
    }
    evaluate the synthesized attributes of $n$;
}
```

Algorithm 3.2 *depth-first* evaluation order of attributes

The attributes of a new class of syntax directed definitions known as **L-attributed definitions**, can always be evaluated in the *depth-first* order. (The **L** stands for “left”, as the attribute information flows from left to right)

L-attributed Definitions

Consider the production $A \rightarrow X_1 X_2 \cdots X_k$. A syntax-directed definition is *L-attributed* if each inherited attribute of X_j $1 \leq j \leq k$, on the right-hand side of the production depends only on

1. The attributes of symbols $X_1, X_2, \cdots, X_{j-1}$ to the left of X_j in the production and
2. The inherited attributes of A.

Every L-attributed definition is also S-attributed, since the restrictions (1) and (2) apply only to inherited attributes.

Production	Semantic Rule
$A \rightarrow LM$	$L.i = l(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$

Table 3.8 L-attributed definition

Consider the syntax directed definition in table 3.8. The attributes $A.s, L.s, M.s$ are synthesized attributes, whereas $A.i, L.i, M.i$ are inherited attributes. It is L-attributed definition as all the inherited attributes are computed according to the rules (1) and (2) above.

Production	Semantic Rule
$A \rightarrow QR$	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

Table 3.9 non-L-attributed definition

Consider the syntax directed definition in table 3.9. The attributes $A.s, Q.s, R.s$ are synthesized attributes, whereas $A.i, Q.i, R.i$ are inherited attributes. It is not an L-attributed definition as in the semantic rule $Q.i = q(R.s)$, the inherited attribute $Q.i$ depends on the attribute of the grammar symbol R, which is to the right of Q in the production.

Translation Schemes

Translation scheme is a context free grammar in which

- Attributes are associated with the grammar symbols and
- The semantic actions enclosed between braces are inserted within the right-hand side of the production.

The translation schemes may have both synthesized and inherited attributes. Below is a translation scheme that maps infix expression to its postfix equivalent. The infix expression uses the operators $+, -$ and number is the operand.

$$E \rightarrow T\ R$$
$$R \rightarrow +T\ \{print("+")\}\ R_1 \mid \varepsilon$$
$$R \rightarrow -T\ \{print("-")\}\ R_1 \mid \varepsilon$$
$$T \rightarrow \boldsymbol{num}\ \{print(\boldsymbol{num}.val)\}$$

Fig 3.5 depicts the parse tree for the expression $7 + 3 - 8$ using the above translation scheme.

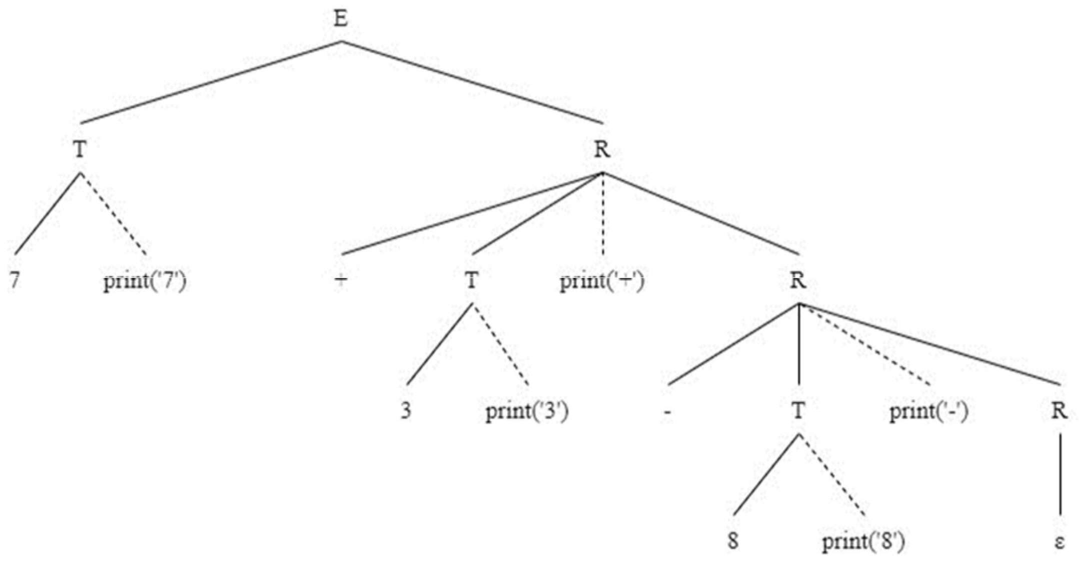

Fig 3.5 parse tree for the expression $7 + 3 - 8$

3.7 Top-Down Translation

In this section we try to implement L-attributed definitions during predictive parsing. Since most arithmetic operators are evaluated from left, we prefer to use the left recursive grammars for expressions. But top-down parsers cannot use left recursive grammars. Left recursion needs to be eliminated. We extend the left recursion elimination algorithm of section 2.4.4 to transform the translation scheme. Table 3.10 shows the translation scheme for desktop calculator.

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val = E_1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow (E)$	$T.val = E.val$
$T \rightarrow \boldsymbol{num}$	$T.val = \boldsymbol{num}.val$

Table 3.10 translation scheme for desktop calculator

The grammar in table 3.10 is left recursive. The translation scheme in table 3.11 is after eliminating left recursion from the grammar.

Production	Semantic Rule
$E \rightarrow T$ R	$\{R.i = T.val\}$ $\{E.val = R.s\}$
$R \rightarrow +$ T R_1	 $\{R_1.i = R.i + T.val\}$ $\{R.s = R_1.s\}$
$R \rightarrow -$ T R_1	 $\{R_1.i = R.i - T.val\}$ $\{R.s = R_1.s\}$
$R \rightarrow \epsilon$	$R.s = R.i$
$T \rightarrow ($ E $)$	 $T.val = E.val$
$T \rightarrow \boldsymbol{num}$	$\{T.val = \boldsymbol{num}.val\}$

Table 3.11 transformed translation scheme for desktop calculator

3.8 Multiple Choice Questions

Q1. What is a Syntax-Directed Definition (SDD)?

a. A definition that associates attributes with grammar symbols and specifies how the values of these attributes are computed.
b. A definition that specifies the syntax of a programming language.
c. A definition that only deals with lexical analysis.
d. A definition that describes the structure of parse trees.

Q2. What is the purpose of synthesized attributes in an SDD?

a. To provide information about the context in which a grammar symbol appears.
b. To compute values based on the attributes of its children in the parse tree.
c. To specify the syntax of the language.
d. To handle lexical analysis.

Q3. In an SDD, an inherited attribute is computed:

a. Bottom-up
b. Top-down
c. Both bottom-up and top-down
d. Neither bottom-up nor top-down

Q4. Which phase of a compiler is most closely associated with SDDs?

a. Lexical analysis
b. Syntax analysis
c. Semantic analysis
d. Code generation

Q5. Which type of attributes are commonly used in syntax-directed definitions to guide the construction of syntax trees?

a. Synthesized attributes
b. Inherited attributes
c. Semantic attributes
d. Contextual attributes

3.9 Summary

This topic enriches students' comprehension of the intricacies involved in developing efficient translation systems for programming languages. Students will receive a concise introduction to syntax-directed definitions, connecting semantic actions with grammar productions. They will learn how to employ attributes to guide and optimize the translation process. Additionally, learners will be introduced to methods for constructing syntax trees based on programming language grammar, providing them with insight into the critical role of syntax trees in subsequent compilation phases.

The course covers S-Attributed Definitions, focusing on the evaluation of semantic attributes in a bottom-up manner. Students will grasp the significance of a stack-based evaluation approach within S-Attributed Definitions, along with practical applications of bottom-up evaluation in the compilation process.

A brief overview of L-Attributed Definitions is provided, addressing the limitations of S-Attributed Definitions and showcasing how L-Attributed Definitions overcome these constraints. The course also presents an overview of top-down translation as a strategy that initiates the translation process from the root of the syntax tree.

3.10 Key Words

- Syntax Directed Definition
- Translation Scheme
- Synthesized Attributes
- Inherited Attributes
- Attribute Grammar
- Syntax Trees
- Dependency Graphs
- L-Attributed Definitions
- S-Attributed Definitions