
RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Compiler Design**

Subject Code: **IT-603**

Semester: **6th**

**Department of Information Technology**
**Subject Notes**
**IT603 (A) – Compiler Design**
**B.Tech, IT-6th Semester**

## Unit III

**Syllabus:** Syntax Directed Translation: Definitions, Inherited Attributes, L-attributed definitions, Sattributed definitions, Dependency graph, Construction of syntax trees, Top down translation, postfix notation, bottom up evaluation.

**Unit Outcome:** Write a scanner, parser, and semantic analyser without the aid of automatic generators.

### UNIT- 3: Syntax Directed Translation

### Syntax Directed Definition (SDD)

During the syntax analysis of the language we use syntax-directed definitions. The set of attributes are associated with each terminal and non-terminal symbols. The attribute can be a string, a number, a type, a memory location or anything else.

The syntax directed definition is a kind of abstract specification. The conceptual view of syntax directed translations is:


Figure 3.1: Syntax Directed Translation

Definition: Syntax-directed definition is a generalization of context free grammar in which each grammar production X → α is associated with it a set of semantic rules of the form a: f(b1, b2,……, bk), where a is an attribute obtained from the function f.

Attribute: The attribute can be a string, a number, a type, a memory location or anything else.

Consider X → α be a context free grammar and a: f(b1, b2,……, bk) where a is the attribute. Then there are two types of attributes:

1. Synthesized attribute: The attribute 'a' is called synthesized attribute of X and b1, b2, ….bk are attributes belonging to the production symbols. The value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.
2. Inherited attribute: The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of the production (i.e. α ) and b1, b2, …bk are belonging to either X or α.

The inherited attribute can be computed from the values of the attributes at the siblings and parent of that node.

**S Attributed Definition:** If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).

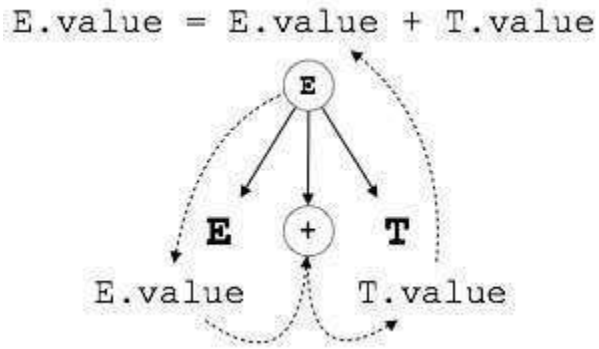


Figure 3.2: S Attributed Definition

As shown in figure, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
Following steps are followed to compute S-attributed definition

1. Write the syntax-directed definition using the appropriate semantic actions for corresponding production rule of the given grammar.
2. The annotated parse tree is generated and attribute values are computed. The computation is done in bottom up manner.
3. The value obtained at the root node is supposed to be the final output.

**L Attributed Definition:** This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

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

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

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

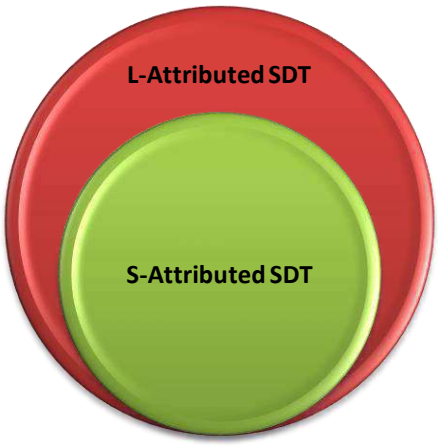


Figure 3.3: S and L Attribute

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

**Dependency Graph:** The directed graph that represents the interdependencies between synthesized and inherited attribute at nodes in the parse tree is called dependency graph.
For the rule X → YZ the semantic action is given by X.x = f(Y.y, Z.z) then synthesized attribute X.x depends on attributes Y.y and Z.z.

*Algorithm to construct Dependency graph*

```
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=f(c1,c2,…..,ck)
Also, associated with the products used at n do
for i=1 to k do
Moreover, construct an edge from the node for Ci to the node for b;
```

Example: Design the dependency graph for the following grammar
E → E1 + E2
E → E1 * E2
Solution: The semantic rules for above grammar is as:

| Production Rule | Semantic Rule |
|---|---|
| E → E1 + E2 | E.val: = E1.val + E2.val |
| E → E1 * E2 | E.val: = E1.val x E2.val |

The dependency graph is as follows:

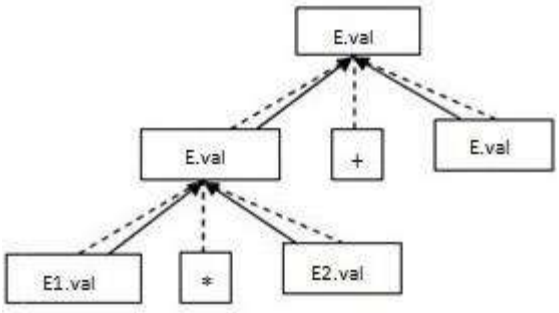


Figure 3.4: Dependency Graph

The synthesized attributes can be represented by .val. Hence the synthesized attributes are given by E.val and E2.val. The dependencies among the nodes is given by solid arrows. The arrows from E1 and E2 show the value of E depends upon E1 and E2. The parse tree is represented by using dotted lines.

**Construction of Syntax Tree:**
The syntax tree is an abstract representation of the language constructs. The syntax tree are used to write the translation routines using syntax-directed definitions.

Constructing syntax tree for an expression means translation of expression into postfix form. The nodes can be implemented as a record with multiple fields.
The following functions are used for making syntax tree

1. mknode (op, left, right): Creates a node with the field operator having operator as label, and the two pointers to left and right.
2. mkleaf(id, entry): Creates an identifier node with label id and a pointer to symbol table is given by 'entry'.
3. mkleaf(num, val): Creates node for number with label num and val is for value of that number.

Example: Construct the syntax tree for the expression a-4+c

- Step 1: Convert the expression from infix to postfix a4-c+
- Step 2: Make use of the functions mknode(), mkleaf(id, ptr) and mkleaf(num, val)
- Step 3: The sequence of function call is given

P1: mkleaf (id, entry for a);
P2: mkleaf (num, 4);
P3: mknode ('-', p1, p2);
P4: mkleaf (id, entry for c);
P5: mknode ('+', p3, p4);

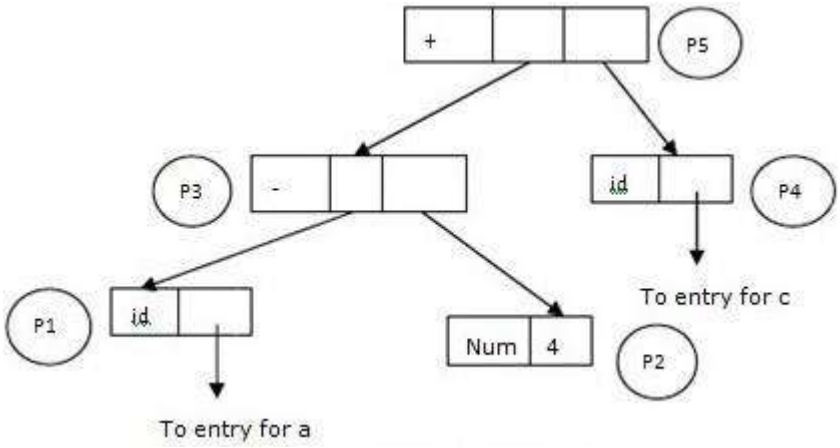


Figure 3.5: Syntax Tree

**Top-Down Translation:**
In top-down translation we will look at the implementation of L-attributed definitions during predictive parsing. Instead of the syntax-directed translations, we will work with translation schemes. The top down translation will see how to evaluate inherited attributes (in L-attributed definitions) during recursive predictive parsing. This will also look at what happens to attributes during the left-recursion elimination in the left-recursive grammars.
D → T id {addtype (id.entry, T.type), L.in = T.type}
LT → int {T.type = integer}
T → real {T.type = real}
L → id {addtype (id.entry, L.in), L1.in = L.in}
L → ε
This is a translation scheme for an L-attributed definitions.

**Postfix Notation:**

Postfix notation is a notation for writing arithmetic expressions in which the operands appear before their operators. There are no precedence rules to learn, and parentheses are never needed. Because of this simplicity, some popular hand-held calculators use postfix notation to avoid the complications of the multiple parentheses required in nontrivial infix expressions.

Most programming languages uses Postfix Notation, because it clearly shows the order in which operations are performed, and because it disambiguates operator groupings.

For example, look at this postfix expression:

"2 3 1 * + 9 -". We scan all elements one by one.

1) Scan '2', it's a number, so push it to stack. Stack contains '2'

2) Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)

3) Scan '1', again a number, push it to stack, stack now contains '2 3 1'

4) Scan '*', it's an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result '3' to stack. Stack now becomes '2 3'.

5) Scan '+', it's an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now becomes '5'.

6) Scan '9', it's a number, we push it to the stack. Stack now becomes '5 9'.

7) Scan '-', it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result '-4' to stack. Stack now becomes '-4'.

8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

**Bottom-Up Evaluation of Inherited Attributes:**

Inherited attributed can be handled by L-attributed definition. The bottom-up parser reduces the right side of the production X → ABC by removing C, B and A from the parser stack.

The parser is implemented as combination of state and value. The state[i] is for grammar symbol A and value[i] holds the synthesized attribute A.a.

Consider the translation scheme for the example given below:

S → T List [List.in := T.type]

T → int [T.type := int]

T → float [T.type := float]

List → List1, id [List1 := List.in] {Enter_type(id.entry, List.in)}

List → id {Enter_type(id.entry, List.in)}

Consider the input int a,b,c for bottom-up evaluation of inherited attributes

| Input String | State | Production Used |
|---|---|---|
| int a,b,c | _ | |
| a,b,c | int | |
| a,b,c | T | T → int |
| ,b,c | Ta | |
| ,b,c | T List | List → id |
| b,c | T List , | |
| ,c | T List , b | |
| ,c | T List | List → List1,b |

| | | |
|---|---|---|
| C | T List , | |
| | T List , c | |
| | T List | List → List1 , id |
| | S | S → T List |

Table 3.1: Bottom-Up Evaluation of Inherited Attributes

Following are the observation about the above implementation:

- The value of **List.in** is always decided by T.type
- The 'T' will always be below the 'list' in the parser stack and each time to decide what the value of list should be; value of T will be consulted.
- The 'T' will always be at the known position in the value array of parser stack.
- The function Enter-type is used to copy the corresponding type to the identifier.
  The first parameter to function Entre-type will be entry for id and second parameter will be type for that id. Hence it can be written as Enter-type(id.entry, T,type).

Basically the function **Enter_type** performs the task of copying **T.type** to **List.in**. In this way the bottom up implementation of inherited attributes can be done.

**Bottom-Up Evaluation of S-Attributed Definitions:**

S-attributed definition is one such class of syntax-directed definition with synthesized attributes only. Synthesized attributes can be evaluated using the bottom-up parser. The purpose of stack is to keep track of values of the synthesized attributes associated with the grammar symbol on tis stack. This stack is commonly known as parser stack.

Synthesized Attributes on the Parser Stack

1. A translator for S-attributes definition is implemented using LR parser generator.
2. A bottom up method is used to hold the values of synthesized attributes.
3. A parser stack is used to hold eh values of synthesized attribute.

The stack is implemented as a pair of state and value. Each state entry is the pointer to the LR(1) parsing table. There is no need to store the grammar symbol implicitly in the parser stack at the state entry.

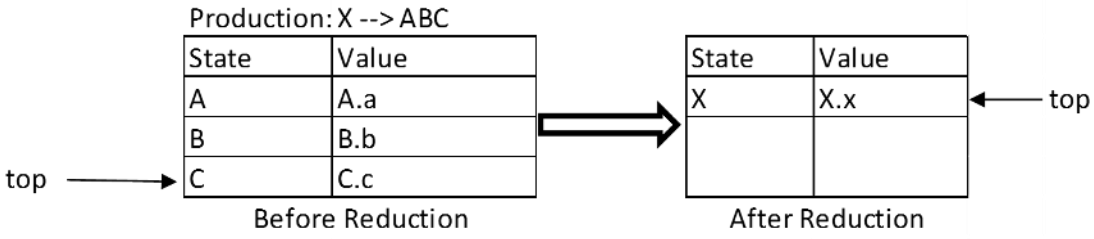


Figure 3.6: Bottom-Up Evaluation of S-Attributed Definitions

Before reduction the state A, B and C can be inserted in the stack along with the values. A.a, B.b and C.c. The top pointer of value[top] will point the value C.c, similarly B.b is in value[top-1] and A.a is in value[top-2]. After reduction the left hand side symbol of the production i.e. X will be placed in the stack along with the value X.x at the top. Hence after reduction value[top] = X.x.

4. After reduction top is decremented by 2 the state covering X is placed at the top of state[top] and value of synthesized attribute X.x is put in value[top].
5. If the symbol has no attribute then the corresponding entry in the value array will be kept undefined.


RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in