## MODULE III:

## Syntax-Directed Definitions, Evaluation Orders for SDD's, Applications of Syntax-Directed Translation, Syntax-Directed Translation Schemes, and Implementing L-Attributed SDD's.

## Syntax Directed Translation

- **Background :** Parser uses a CFG(Context-free-Grammer) to validate the input string and produce output for next phase of the compiler.
- Output could be either a parse tree or abstract syntax tree.
- Now to interleave semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation.

- **Definition:** Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis.
- SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes.
- Syntax directed translation rules use
  1) lexical values of nodes,
  2) constants &
  3) attributes associated to the non-terminals in their definitions.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order.

- In many cases, translation can be done during parsing without building an explicit tree.

Example

E -> E+T | T
T -> T*F | F
F -> INTLIT

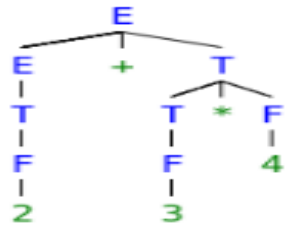
- This is a grammar to syntactically validate an expression having additions and multiplications in it.
- Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any.
- In this example we will focus on evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

| | | |
|---|---|---|
| E -> E+T | { E.val = E.val + T.val } | PR#1 |
| E -> T | { E.val = T.val } | PR#2 |
| T -> T*F | { T.val = T.val * F.val } | PR#3 |
| T -> F | { T.val = F.val } | PR#4 |
| F -> INTLIT | { F.val = INTLIT.lexval } | PR#5 |

- For understanding translation rules further, we take the first SDT augmented to [ E -> E+T ] production rule.
- The translation rule in consideration has val as attribute for both the non-terminals – E & T.
- Right hand side of the translation rule corresponds to attribute values of right side nodes of the production rule and vice-versa.
- Generalizing, SDT are augmented rules to a CFG that associate

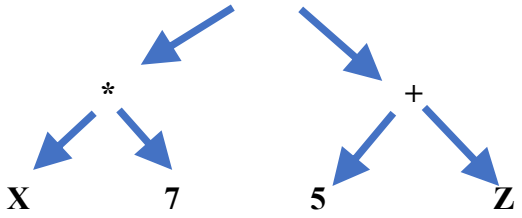
1. set of attributes to every node of the grammar and
2. set of translation rules to every production rule using attributes, constants and lexical values.

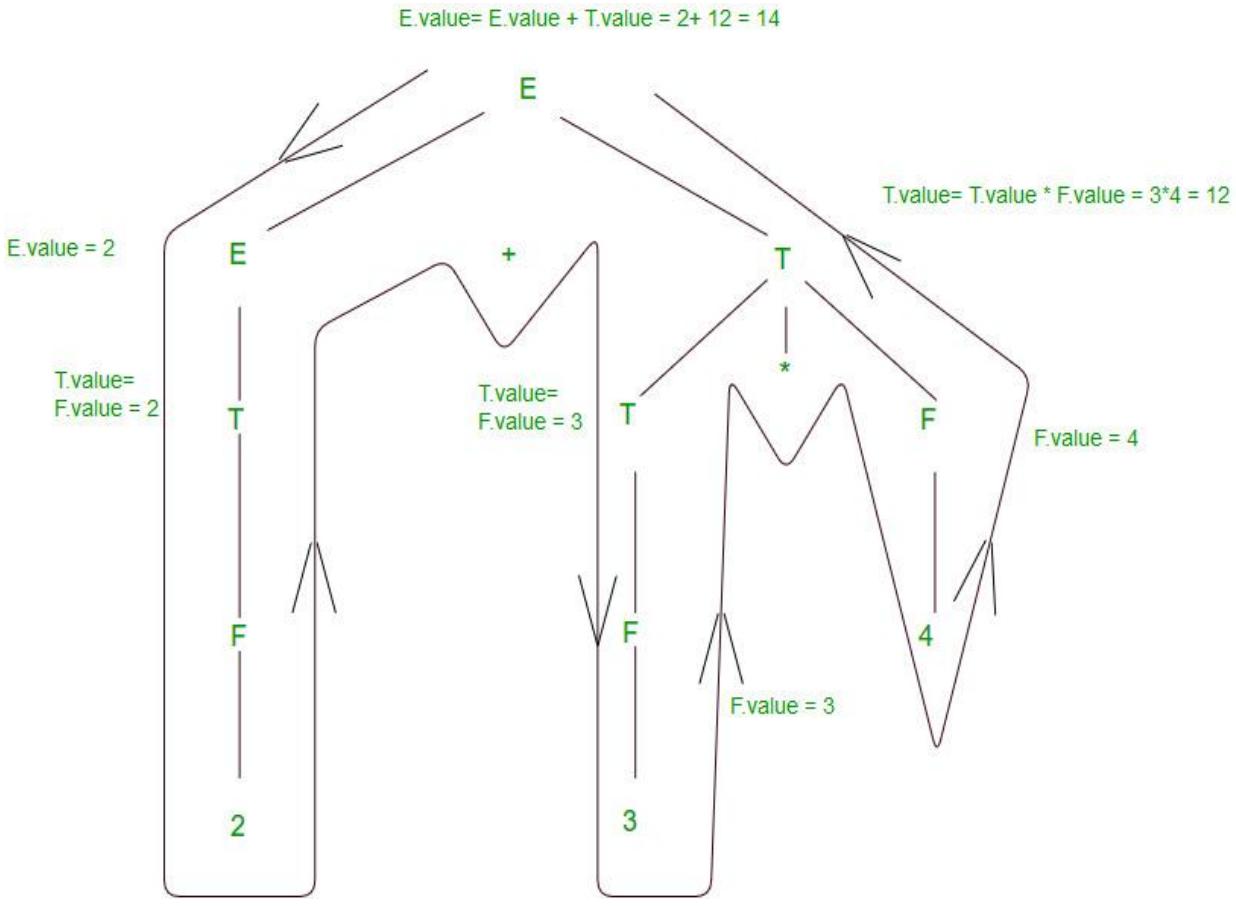
- Let's take a string to see how semantic analysis happens – S = 2+3*4.
- Parse tree corresponding to S would be




- To evaluate translation rules, we can employ one depth first search traversal on the parse tree.
- This is possible only because SDT rules don't impose any specific order on evaluation until children attributes are computed before parents for a grammar having all synthesized attributes.
- Otherwise, we would have to figure out the best suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals.
- For better understanding, we will move bottom up in left to right fashion for computing translation rules of our example.

**Construct the syntax tree for the expression X * 7 – 5 + Z**






P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

- Above diagram shows how semantic analysis could happen.
- The flow of information happens bottom-up and all the children attributes are computed before parents, as discussed above.
- Right hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parent.

## Evaluation Orders for SDD's

Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given **parse tree**. While an annotated parse tree shows the values of attributes, a **dependency graph** helps us determine how those values can be computed.

In this section, in addition to dependency graphs, we define two important classes of SDD's: the "S-attributed" and the more general "L-attributed" SDD's. The translations specified by these two classes fit well with the parsing methods we have studied, and most translations encountered in practice can be written to conform to the requirements of at least one of these classes.

### 1 Dependency Graphs

A *dependency graph* depicts the flow of information among the attribute instances in a particular **parse tree**; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the **semantic rules**. In more detail:

- For each parse-tree node, say a node labeled by grammar symbol X , the dependency graph has a node for each attribute associated with X .
- Suppose that a semantic rule associated with a production p defines the value of **synthesized attribute** A.b in terms of the value of X.c (the rule may define A.b in terms of other attributes in addition to X.c). Then, the dependency graph has an edge from X.c to A.b. More precisely, at every node N labeled A where production p is applied, create an edge to attribute b at N , from the attribute c at the child of N corresponding to this instance of the symbol X in the body of the production.

Since a node N can have several children labeled X , we again assume that subscripts distinguish among uses of the same symbol at different places in the production.

- Suppose that a semantic rule associated with a production p defines the value of inherited attribute B.c in terms of the value of X.a. Then, the dependency graph has an edge from X.a to B.c. For each node N labeled B that corresponds to an occurrence of this B in the body of production p, create an edge to attribute c at N from the attribute a at the node M that corresponds to this occurrence of X . Note that M could be either the parent or a sibling of N .

NOTE: What is described above is an algorithm for constructing a dependency graph.

### 2 Ordering the Evaluation of Attributes

The *dependency graph* characterizes the possible orders in which we can evaluate the attributes at the various nodes of a **parse tree**. If the dependency graph has an edge from node M to node N , then the attribute corresponding to M must be evaluated before the attribute of N . Thus, the only allowable orders of evaluation are those sequences of nodes N1,N2,…,Nk such that if there is an edge of the dependency graph from Ni to Nj, then $i < j$ . Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

If there is any **cycle** in the graph, then there are no **topological sorts**; that is, there is no way to evaluate the SDD on this **parse tree**. If there are no **cycles**, however, then there is always at least one topological sort. To see why, since there are no cycles, we can surely find a node with no edge entering. For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yielding a cycle. Make this node the first in the topological order, remove it from the dependency graph, and repeat the process on the remaining nodes.

## 3 S-Attributed Definitions

As mentioned earlier, given an SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles. In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order, since they do not permit dependency graphs with cycles. Moreover, the two classes introduced in this section can be implemented efficiently in connection with **top-down** or **bottom-up** parsing.

The first class is defined as follows:

- An SDD is S-attributed if every attribute is synthesized.

When an SDD is **S-attributed**, we can evaluate its attributes in any bottom-up order of the nodes of the **parse tree**. It is often especially simple to evaluate the attributes by performing a **postorder traversal** of the **parse tree** and evaluating the attributes at a node N when the traversal leaves N for the last time. That is, we apply the function postorder, defined below, to the root of the parse tree (see also the box "Preorder and Postorder Traversals" in Section 2.3.4):

```
postorder(N) {
        for ( each child C of N, from the left ) postorder(C);
        evaluate the attributes associated with node N;
}
```

**S-attributed definitions** can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a **postorder traversal**. Specifically, postorder corresponds exactly to the order in which an **LR parser** reduces a production body to its head. This fact will b e used in Section 5.4.2 to evaluate **synthesized attributes** and store them on the **stack** during **LR parsing**, without creating the tree nodes explicitly.

## 4 L-Attributed Definitions

The second class of SDD's is called *L-attributed definitions*. The idea behind this class is that, between the attributes associated with a production body, **dependency-graph edges** can go from **left** to **right**, but not from **right** to **left** (hence "L-attributed"). More precisely, each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production A→X1,X2,…,Xn, and that there is an **inherited attribute** Xi.a computed by a rule associated with this production. Then the rule may use only:
3. Inherited attributes associated with the head A.
4. Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,…,Xi−1 located to the left of Xi.
5. Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this Xi.

**Ex:** The SDD in Fig. 5.4 is **L-attributed**. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $T \rightarrow F\ T'$ | $T'.inh = F.val$ |
| $T' \rightarrow *\ F\ T'_1$ | $T'_1.inh = T'.inh \times F.val$ |

The first of these rules defines the inherited attribute T'.inh using only F.val , and F appears to the left of T' in the production body, as required. The second rule defines T′1.inh using the inherited attribute T'inh associated with the **head**, and F.val , where F appears to the left of T′1 in the production body.

In each of these cases, the rules use information "from above or from the left," as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed.

**Ex:** Any SDD containing the following production and rules cannot be L-attributed:

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $A \rightarrow B\ C$ | $A.s = B.b;$ |
| | $B.i = f(C.c, A.s)$ |

The second rule defines an inherited attribute B.i, so the entire SDD cannot be S-attributed. Further, although the rule is legal, the SDD cannot be L-attributed, because the attribute C.c is used to help define B.i, and C is to the right of B in the production body.

### 5. Semantic Rules with Controlled Side Effects

In practice, translations involve side effects: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table. With SDD's, we strike a balance between **attribute grammars** and **translation schemes**. Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph. Translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment;

We shall control side effcts in SDD's in one of the following ways:

- Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a "correct" translation, where "correct" depends on the application.
- Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph.

## Applications of Syntax-Directed Translation

The main application of *Syntax-Directed Translation* is in the construction of syntax trees. Compilers use syntax trees as an intermediate representation, using a common form of *Syntax-Directed Definitions*, the input string is converted into a tree. The compiler then traverses the tree using rules that are in effect an SDD on the syntax tree rather than the parse tree.

- Executing Arithmetic Expressions
- Conversion from infix to postfix expression
- Conversion from infix to prefix expression
- For Binary to decimal conversion

- Counting the number of Reductions
- Creating a Syntax tree
- Generating intermediate code
- Storing information into the symbol table
- Type checking

**Example :**

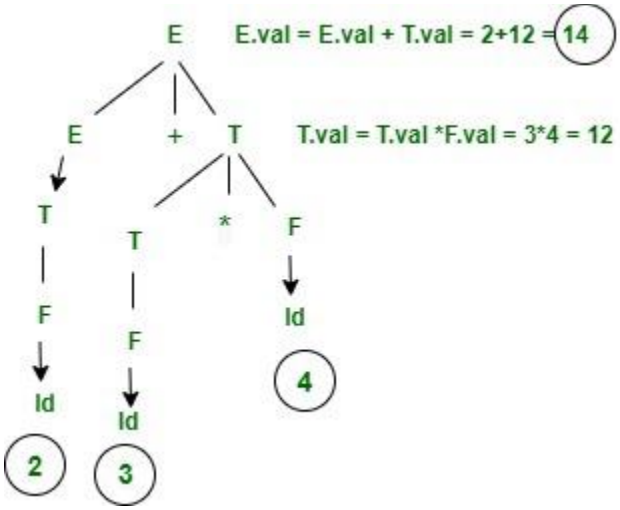
Here, we are going to cover an example of application of SDT for better understanding the SDT application uses. let's consider an example of arithmetic expression and then you will see how SDT will be constructed.

Let's consider Arithmetic Expression is given.

Input : 2+3*4

output: 14

SDT for the above example.




*SDT for 2+3*4*

**Semantic Action is given as following.**

E -> E+T { E.val = E.val + T.val then print (E.val)}
|T { E.val = T.val}
T -> T*F { T.val = T.val * F.val}
|F { T.val = F.val}
F -> Id {F.val = id}

## Syntax-Directed Translation Schemes

**DECLARATIONS:**

Let assume declarations using a simplified grammar that declares just one name at a time;

The grammar is

D → T id ; D | ε

T → B C | record ' {' D '} '

B → int | float

C → ε | [ num ] C

Nonterminal D generates a sequence of declarations.

Nonterminal T generates basic, array, or record types.

Nonterminal B generates one of the basic types int and float.
Nonterminal C, for \component, generates strings of zero or more integers, each integer surrounded by brackets.
An array type consists of a basic type specified by B, followed by array components specified by nonterminal C.
A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

- $T \rightarrow B$ { t = B.type; w = B.width; }
  C { T.type = C.type; T.width = C.width; }
- $B \rightarrow$ int { B.type = integer; B.width = 8; }
- $B \rightarrow$ float { B.type = float; B.width = 16; }
- $C \rightarrow \varepsilon$ { C.type = t; C.width = w; }
- $C \rightarrow$ [ num ] C1 { C.type = array(num.value, C1:type);
  C.width = num.value x C1.width; }

Figure: Computing types and their widths

The translation scheme (SDT) in Fig. computes types and their widths for basic and array types;
The SDT uses synthesized attributes type and width for each nonterminal and two variables t and w to pass type and width information from a B node in a parse tree to the node for the production $C \rightarrow \varepsilon$ .
In a syntax-directed definition, t and w would be inherited attributes for C.

- The body of the T-production consists of nonterminal B, an action, and nonterminal C, which appears on the next line.
- The action between B and C sets t to B.type and w to B.width.
- If $B \rightarrow$ int then B.type is set to integer and B.width is set to 8, the width of an integer.
- Similarly, if $B \rightarrow$ float then B.type is float and B.width is 16, the width of a float.
- The productions for C determine whether T generates a basic type or an array type.
- If $C \rightarrow \varepsilon$, then t becomes C.type and w becomes C.width.
- Otherwise, C specifies an array component.
- The action for $C \rightarrow$ [ num ] C1 forms C.type by applying the type constructor array to the operands num.value and C1.type.

**Control Flow:**

- The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions.
- In programming languages, boolean expressions are often used to

1. Alter the flow of control: Boolean expressions are used as conditional expressions in statements that alter the flow of control.

The value of such boolean expressions is implicit in a position reached in a program.
For example, in if (E) S, the expression E must be true if statement S is reached.
2. Compute logical values: A boolean expression can represent true or false as values.
Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

- The intended use of boolean expressions is determined by its syntactic context.
- For example, an expression following the keyword if is used to alter the flow of control, while an expression on the right side of an assignment is used to denote a logical value.
- Such syntactic contexts can be specified in a number of ways:
- we may use two different nonterminals, use inherited attributes, or set a flag during parsing.
- Alternatively we may build a syntax tree and invoke different procedures for the two different uses of boolean expressions.

**ASSIGNMENTS:**

- The syntax-directed definition in Fig. builds up the three-address code for an assignment statement S using attribute code for S and attributes addr and code for an expression E.

| PRODUCTION | SEMANTIC RULES |
|---|---|
| S → id = E ; | S.code = E.code \|\|<br>gen(top.get(id.lexeme) '=' E.addr) |
| E → E1 + E2 | E.addr = new Temp ()<br>E.code = E1.code \|\| E2.code \|\|<br>gen(E.addr '=' E1.addr '+' E2.addr) |
| \| - E1 | E.addr = new Temp ()<br>E.code = E1.code \|\|<br>gen(E.addr '=' 'minus' E1.addr) |
| \| ( E1 ) | E.addr = E1.addr<br>E.code = E1.code |
| \| id | E.addr = top.get(id.lexeme)<br>E.code = ' ' |

- Attributes S.code and E.code denote the three-address code for S and E, respectively.
- Attribute E.addr denotes the address that will hold the value of E.
- Consider the last production, E → id, in the syntax-directed definition in Fig. above. When an expression is a single identifier, say x, then x itself holds the value of the expression.
- The semantic rules for this production define E.addr to point to the symbol-table entry for this instance of id.
- Let top denote the current symbol table.
- Function top.get retrieves the entry when it is applied to the string representation id.lexeme of this instance of id.
- E.code is set to the empty string.
- When E → (E1), the translation of E is the same as that of the subexpression E1.
- Hence, E.addr equals E1.addr, and E.code equals E1.code.

- The operators + and unary - in Fig. are representative of the operators in a typical language.
- The semantic rules for E → E1 +E2, generate code to compute the value of E from the values of E1 and E2.
- Values are computed into newly generated temporary names.
- If E1 is computed into E1.addr and E2 into E2.addr, then E1 +E2 translates into t = E1.addr+E2.addr, where t is a new temporary name.

- E.addr is set to t.
- A sequence of distinct temporary names t1, t2, … is created by successively executing **new** Temp().
- For convenience, we use the notation gen(x '=' y '+' z ') to represent the three-address instruction x = y+z.
- Expressions appearing in place of variables like x, y, and z are evaluated when passed to gen, and quoted strings like '=' are taken literally.
- Other three-address instructions will be built up similarly by applying gen to a combination of expressions and strings.
- When we translate the production E → E1 +E2, the semantic rules in
- Fig. build up E.code by concatenating E1.code, E2.code, and an instruction that adds the values of E1 and E2.
- The instruction puts the result of the addition into a new temporary name for E, denoted by E.addr.
- The translation of E → -E1 is similar.
- The rules create a new temporary for E and generate an instruction to perform the unary minus operation.
- Finally, the production S → id =E; generates instructions that assign the value of expression E to the identifier id.
- The semantic rule for this production uses function top.get to determine the address of the identifier represented by id, as in the rules for E → id.
- S.code consists of the instructions to compute the value of E into an address given by E.addr, followed by an assignment to the address top.get(id.lexeme) for this instance of id.
- Example: The syntax-directed definition in Fig. translates the assignment statement a = b +-c; into the three-address code sequence

    t1 = minus c
    t2 = b + t1
    a = t2

**BOOLEAN EXPRESSIONS:**

- Boolean expressions are composed of the boolean operators (which we denote &&, ||, and !, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions.
- Relational expressions are of the form E1 rel E2, where E1 and E2 are arithmetic expressions.
- Consider boolean expressions generated by the following grammar:
  B → B || B | B && B | ! B | ( B ) | E rel E | true | false
- We use the attribute rel.op to indicate which of the six comparison operators <, <=, =, ! =, >, or >= is represented by rel.
- Let we assume that || and && are left-associative, and that || has lowest precedence, then &&, then !.
- Given the expression B1 || B2, if we determine that B1 is true, then we can conclude that the entire expression is true without having to evaluate B2.
- Similarly, given B1&&B2, if B1 is false, then the entire expression is false.
- The semantic definition of the programming language determines whether all parts of a boolean expression must be evaluated.

- If the language definition permits (or requires) portions of a boolean expression to go unevaluated, then the compiler can optimize the evaluation of boolean expressions by computing only enough of an expression to determine its value.
- Thus, in an expression such as B1 || B2, neither B1 nor B2 is necessarily evaluated fully.
- If either B1 or B2 is an expression with side effects (e.g., it contains a function that changes a global variable), then an unexpected answer may be obtained.

Ex: The translation for a || b **&& !** c is the three-address sequence

$t_1$ : = **!** c
$t_2$ : = b **&&** $t_1$
$t_3$ : = a || $t_2$

**Methods of Translating Boolean Expressions:**

There are two principal methods of representing the value of a boolean expression. They are :

– To encode true and false *numerically* and to evaluate a boolean expression analogously to an arithmetic expression.
– Often, 1 is used to denote true and 0 to denote false.
– To implement boolean expressions by *flow of control*, that is, representing the value of a boolean expression by a position reached in a program

Numerical Representation

– Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

A relational expression such as a < b is equivalent to the conditional statement if a < b then 1 else 0 which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100)

```
100 :  if a < b goto 103
101 :  t : = 0
102 :  goto 104
103 :  t : = 1
```

**Short-Circuit Code:**

- In short-circuit (or jumping) code, the boolean operators &&, ||, and ! translate into jumps.
- The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Example 6.21 : The statement if ( x < 100 || x > 200 && x != y ) x = 0;
might be translated into the code of Fig. 6.34. In this translation, the Boolean expression is true if control reaches label L2.
If the expression is false, control goes immediately to L1, skipping L2 and the assignment x = 0.

```
        if x < 100 goto L2
        if False x > 200 goto L1
        if False x != y goto L1
L2:     x = 0
L1:
```

Figure : Jumping code

## Switch-Statements:

- The \switch" or \case" statement is available in a variety of languages.
- Our switch-statement syntax is shown in Fig.
- There is a selector expression E, which is to be evaluated, followed by n constant values V1, V2, … ,Vn that the expression might take, perhaps including a default \value," which always matches the expression if no other value does.

```
switch ( E ) {
        case V1: S1
        case V2: S2
                ””
        case Vn-1: Sn-1
default: Sn
}
```

Figure 6.48: Switch-statement syntax

## Syntax-Directed Translation of Switch-Statements:

- The intermediate code in Fig. 6.49 is a convenient translation of the switch-statement in Fig.
- The tests all appear at the end so that a simple code generator can recognize the multiway branch and generate efficient code for it, using the most appropriate implementation.

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        …
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        …
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

Figure 6.49: Translation of a switch-statement

## Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false ***numerically*** and to evaluate a boolean expression analogously to an arithmetic expression.
- Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by ***flow of control***, that is, representing the value of a boolean expression by a position reached in a program

**Numerical Representation**

- Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

A relational expression such as a < b is equivalent to the conditional statement if a < b then 1 else 0 which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```
100 :  if a < b goto 103
101 :  t : = 0
102 :  goto 104
103 :  t : = 1
```

**Short-Circuit Code:**

- In short-circuit (or jumping) code, the boolean operators &&, ||, and ! translate into jumps.
- The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Example 6.21 : The statement  if ( x < 100 || x > 200 && x != y ) x = 0;
might be translated into the code of Fig. In this translation, the Boolean expression is true if control reaches label L2.
If the expression is false, control goes immediately to L1, skipping L2 and the assignment x = 0.

```
            if x < 100 goto L2
            if False x > 200 goto L1
            if False x != y goto L1
L2:         x = 0
L1:
```

Figure: Jumping code

**Switch-Statements:**

- The \switch" or \case" statement is available in a variety of languages.
- Our switch-statement syntax is shown in Fig.
- There is a selector expression E, which is to be evaluated, followed by n constant values V1, V2, … ,Vn that the expression might take, perhaps including a default \value," which always matches the expression if no other value does.

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ,,,,
        case Vn-1: Sn-1
default: Sn
}
```

Figure : Switch-statement syntax

**Syntax-Directed Translation of Switch-Statements:**

- The intermediate code in Fig. is a convenient translation of the switch-statement in Fig.
- The tests all appear at the end so that a simple code generator can recognize the multiway branch and generate efficient code for it, using the most appropriate implementation.

```
        code to evaluate E into t          Ln:    code for Sn
        goto test                                 goto next
L1:     code for S1                        test:  if t = V1 goto L1
        goto next                                 if t = V2 goto L2
L2:     code for S2                               …
        goto next                                 if t = Vn-1 goto Ln-1
        …                                         goto Ln
Ln-1:   code for Sn-1                      next:
        goto next
```

**Figure: Translation of a switch-statement**

**Flow-of-Control Statements:**

- We now consider the translation of boolean expressions into three-address code in the context of statements such as those generated by the following grammar:
  S → if ( B ) S1
  S → if ( B ) S1 else S2
  S → while ( B ) S1
- In these productions, nonterminal B represents a boolean expression and nonterminal S represents a statement.
- As in that example, both B and S have a synthesized attribute code, which gives the translation into three-address instructions.
- For simplicity, we build up the translations B.code and S.code as strings, using syntax-directed definitions
- The semantic rules defining the code attributes could be implemented instead by building up syntax trees and then emitting code during a tree traversal.
- The translation of if (B) S1 consists of B.code followed by S1.code, as illustrated in Fig(a). Within B.code are jumps based on the value of B.
- If B is true, control flows to the first instruction of S1.code, and if B is false, control flows to the instruction immediately following S1.code.

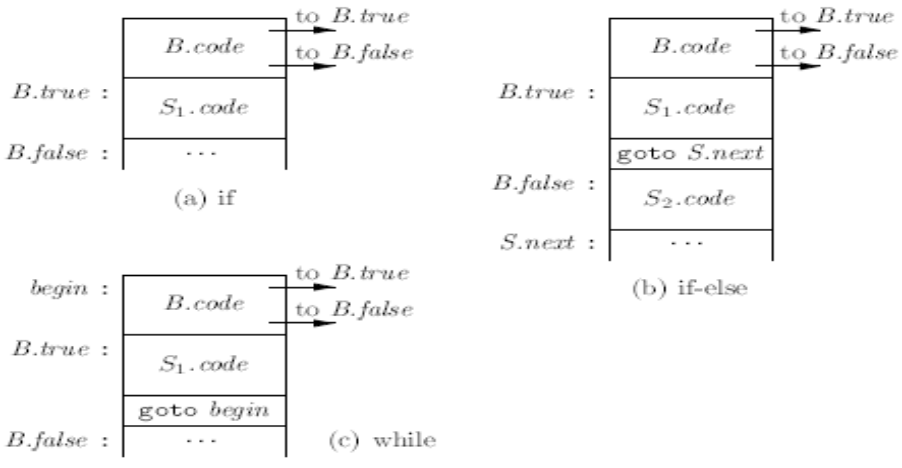



Figure 6.35: Code for if-, if-else-, and while-statements

- The labels for the jumps in B.code and S.code are managed using inherited attributes.
- With a boolean expression B, we associate two labels: B.true, the label to which control flows if B is true, and B.false, the label to which control flows if B is false.
- With a statement S, we associate an inherited attribute S.next denoting a label for the instruction immediately after the code for S.

- In some cases, the instruction immediately following S.code is a jump to some label L.
- A jump to a jump to L from within S.code is avoided using S.next.
- The syntax-directed definition in Fig. produces three-address code for boolean expressions in the context of if-, if-else-, and while-statements.
- We assume that newlabel() creates a new label each time it is called, and that label(L) attaches label L to the next three-address instruction to be generated.

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $P \rightarrow S$ | S.next = newlabel()<br>P.code = S.code \|\| label(S.next) |
| $S \rightarrow$ assign | S.code = assign.code |
| $S \rightarrow$ if ( B ) $S_1$ | B.true = newlabel()<br>B.false = $S_1$.next = S.next<br>S.code = B.code \|\| label(B.true) \|\| $S_1$.code |
| $S \rightarrow$ if ( B ) $S_1$ else $S_2$ | B.true = newlabel()<br>B.false = newlabel()<br>$S_1$.next = $S_2$.next = S.next<br>S.code = B.code \|\| label(B.true) \|\| $S_1$.code<br>\|\| gen('goto' S.next) \|\| label(B.false) \|\| $S_2$.code |
| $S \rightarrow$ while ( B ) $S_1$ | begin = newlabel()<br>B.true = newlabel()<br>B.false = S.next<br>S1.next = begin<br>S.code = label(begin) \|\| B.code \|\| label(B.true) \|\| $S_1$.code<br>\|\| gen('goto' begin) |
| $S \rightarrow S_1 S_2$ | $S_1$.next = newlabel()<br>$S_2$.next = S.next<br>S.code = $S_1$.code \|\| label($S_1$.next) \|\| $S_2$.code |

Figure: Syntax-directed definition for flow-of-control statements.

- A program consists of a statement generated by $P \rightarrow S$.
- The semantic rules associated with this production initialize S.next to a new label.
- P.code consists of S:code followed by the new label S.next.
- Token **assign** in the production $S \rightarrow$ **assign** is a placeholder for assignment statements.
- Let for ex: control flow, S:code is simply **assign**.code.
- In translating $S \rightarrow$ **if** (B) S1, the semantic rules in Fig create a new label B.true and attach it to the first three-address instruction generated for the statement S1, as illustrated in Fig(a).
- Thus, jumps to B.true within the code for B will go to the code for S1.
- Further, by setting B.false to S.next, we ensure that control will skip the code for S1 if B evaluates to false.
- In translating the if-else-statement $S \rightarrow$ **if** (B) S1 else S2, the code for the boolean expression B has jumps out of it to the first instruction of the code for S1 if B is true, and to the first instruction of the code for S2 if B is false, as illustrated in Fig(b).
- Further, control flows from both S1 and S2 to the three-address instruction immediately following the code for S - its label is given by the inherited attribute S.next.
- An explicit goto S.next appears after the code for S1 to skip over the code for S2.
- No goto is needed after S2, since S2.next is the same as S:next.
- The code for $S \rightarrow$ **while** (B) S1 is formed from B:code and S1:code as shown in Fig. (c).
- We use a local variable begin to hold a new label attached to the first instruction for this while-statement, which is also the first instruction for B.

- We use a variable rather than an attribute, because begin is local to the semantic rules for this production.
- The inherited label S.next marks the instruction that control must flow to if B is false;
- hence, B.false is set to be S.next.
- A new label B.true is attached to the first instruction for S1; the code for B generates a jump to this label if B is true.
- After the code for S1 we place the instruction goto begin, which causes a jump back to the beginning of the code for the boolean expression.
- Note that S1.next is set to this label begin, so jumps from within S1.code can go directly to begin.
- The code for S → S1 S2 consists of the code for S1 followed by the code for S2.
- The semantic rules manage the labels; the first instruction after the code for S1 is the beginning of the code for S2; and the instruction after the code for S2 is also the instruction after the code for S.
- There we shall see an alternative method, called \backpatching," which emits code for statements in one pass.

**Control-Flow Translation of Boolean Expressions:**

- The semantic rules for boolean expressions in Fig below complement the semantic rules for statements in Fig above.
- As in the code layout of Fig, a Boolean expression B is translated into three-address instructions that evaluate B using conditional and unconditional jumps to one of two labels: B.true if B is true, and B.false if B is false.
- The fourth production in Fig., B → E1 rel E2, is translated directly into a comparison three-address instruction with jumps to the appropriate places.
- For instance, B of the form a < b translates into:
  if a < b goto B.true
  goto B.false

| PRODUCTION | SEMANTIC RULES |
|---|---|
| B → B1 \|\| B2 | B1.true = B.true<br>B1.false = newlabel()<br>B2.true = B.true<br>B2.false = B.false<br>B.code = B1.code \|\| label(B1.false) \|\| B2.code |
| B → B1 && B2 | B1.true = newlabel()<br>B1.false = B.false<br>B2.true = B.true<br>B2.false = B.false<br>B.code = B1.code \|\| label(B1.true) \|\| B2.code |
| B → ! B1 | B1.true = B.false<br>B1.false = B.true<br>B.code = B1.code |
| B → E1 rel E2 | B.code = E1.code \|\| E2.code<br>\|\| gen('if' E1.addr rel.op E2.addr 'goto' B.true)<br>\|\| gen('goto' B.false) |
| B → true | B.code = gen('goto' B.true) |
| B → false | B.code = gen('goto' B.false) |

**Figure: Generating three-address code for booleans**

The remaining productions for B are translated as follows:

1. Suppose B is of the form B1 || B2. If B1 is true, then we immediately know that B itself is true, so B1:true is the same as B:true. If B1 is false, then B2 must be evaluated, so we make B1:false be the label of the first instruction in the code for B2. The true and false exits of B2 are the same as the true and false exits of B, respectively.
2. The translation of B1 &&B2 is similar.
3. No code is needed for an expression B of the form !B1: just interchange the true and false exits of B to get the true and false exits of B1.
4. The constants true and false translate into jumps to B:true and B:false, respectively.

Example: Consider again the following statement:

if( x < 100 || x > 200 && x != y ) x = 0;

Using the previous syntax-directed definitions we would obtain the code below

```
    if x < 100 goto L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
L2:  x = 0
L1:
```

Figure: Control-flow translation of a simple if-statement

## Attributed grammars

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

**Example:**

- E → E + T { E.value = E.value + T.value }

- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted.
- Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.
- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.
- Based on the way the attributes get their values, they can be broadly divided into two categories:
  a. Synthesized attributes and
  b. Inherited attributes.

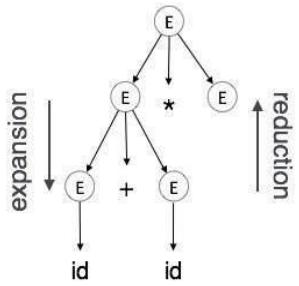
**Synthesized attributes:**

- A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production.
- Synthesized attributes represent information that is being passed up the parse tree.
- These attributes get values only from the attribute values of their child nodes. (Variables in the RHS of the production).
- For eg. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.
- Synthesized attributes never take values from their parent nodes or any sibling nodes.

**Inherited attributes:**

- An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute.
- The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production).
- For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute.

**Expansion**: When a non-terminal is expanded to terminals as per a grammatical rule

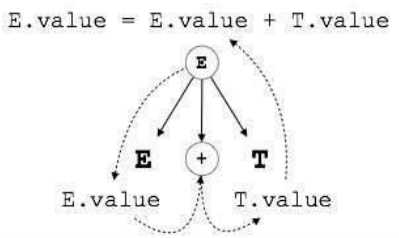



Reduction: When a terminal is reduced to its corresponding non-terminal according to grammar rules.

Syntax trees are parsed top-down and left to right.
Whenever reduction occurs, we apply its corresponding semantic rules (actions).
Semantic analysis uses Syntax Directed Translations to perform the above tasks.
Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).
Semantic analyzer attaches attribute information with AST, which are called Attributed AST.
Attributes are two tuple value, <attribute name, attribute value>
For example:
int value = 5;
<type, "integer">
<presentvalue, "5">
For every production, we attach a semantic rule.

## S-attributed SDT

- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- Semantic actions are placed in rightmost place of RHS.



As depicted above, 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.

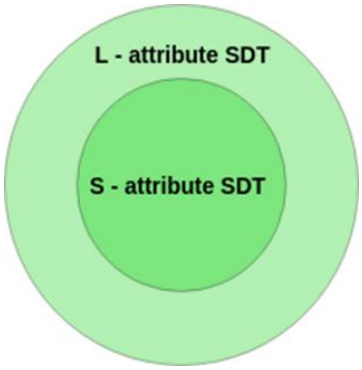
## L-attributed SDT:

If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute, a non-terminal can inherit values from its parent and left siblings only, it is called as L-attributed SDT.

As in the following production

$S \rightarrow ABC$

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.
Semantic actions are placed anywhere in RHS.




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

Example – Consider the given below SDT.
P1: S -> MN {S.val= M.val + N.val}
P2: M -> PQ {M.val = P.val * Q.val and P.val =Q.val}

Select the correct option.
A. Both P1 and P2 are S attributed.
B. P1 is S attributed and P2 is L-attributed.
C. P1 is L attributed but P2 is not L-attributed.
D. None of the above

Explanation: - –
The correct answer is option C as, In P1, S is a synthesized attribute and in L-attribute definition synthesized is allowed.
So P1 follows the L-attributed definition.
But P2 doesn't follow L-attributed definition as P is depending on Q which is RHS to it.

# Implementing L-Attributed SDD's.



## 1. Translation During Recursive-Descent Parsing

A recursive-descent parser has a function A for each nonterminal A, can extend the parser into a translator as follows:

a) The arguments of function A are the inherited attributes of nonterminal A.

b) The return-value of function A is the collection of synthesized attributes of nonterminal A.

In the body of function A, we need to both parse and handle attributes:

Decide upon the production used to expand A.

Check that each terminal appears on the input when it is required. We shall assume that no backtracking is needed, but the extension to recur-sive-descent parsing with backtracking can be done by restoring the input position upon failure.

## 2. On-The-Fly Code Generation

The construction of long strings of code that are attribute values, is undesirable for several reasons, including the time it could take to copy or move long strings. In common cases such as our running code-generation example, we can instead incrementally generate pieces of the code into an array or output file by executing actions in an SDT.

## 3. L-Attributed SDD's and LL Parsing

Suppose that an L-attributed SDD is based on an LL-grammar and that we have converted it to an SDT with actions embedded in the productions, as described in Section 5.4.5. We can then perform the translation during LL parsing by extending the parser stack to hold actions and certain data items needed for attribute evaluation. Typically, the data items are copies of attributes.

In addition to records representing terminals and nonterminals, the parser stack will hold *action-records* representing actions to be executed and *synth-esize-records* to hold the synthesized attributes for nonterminals. We use the following two principles to manage attributes on the stack:

The inherited attributes of a nonterminal *A* are placed in the stack record that represents that nonterminal. The code to evaluate these attributes will usually be represented by an action-record immediately above the stack record for *A;* in fact, the conversion of L-attributed SDD's to SDT's ensures that the action-record will be immediately above *A.*

The synthesized attributes for a nonterminal *A* are placed in a separate synthesize-record that is immediately below the record for *A* on the stack.

This strategy places records of several types on the parsing stack, trusting that these variant record types can be managed properly as subclasses of a "stack-record" class. In practice, we might combine several records into one, but the ideas are perhaps best explained by separating data used for different purposes into different records.

Action-records contain pointers to code to be executed. Actions may also appear in synthesize-records; these actions typically place copies of the synthe-sized attribute(s) in other records further down the stack, where the value of that attribute will be needed after the synthesize-record and its attributes are popped off the stack.

**4. Bottom-Up Parsing of L-Attributed SDD's**

We can do bottom-up every translation that we can do top-down. More pre-cisely, given an L-attributed SDD on an LL grammar, we can adapt the gram-mar to compute the same SDD on the new grammar during an LR parse. The "trick" has three parts:

Start with the SDT, which places embed-ded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized at-tributes.