Syntax Directed Definition

A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If *X* is a symbol and *a* is one of its attributes, then we write *X.a* to denote the value of a at a particular parse-tree node labeled *X.* For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \mid\mid T.code \mid\mid$ '+'

This production has two non terminals, *E* and T; the subscript in E_1 distinguishes the occurrence of *E* in the production body from the occurrence of *E* as the head. Both *E* and *T* have a string-valued attribute *code.* The semantic rule specifies that the string *E.code* is formed by concatenating *E_1.code, T.code,* and the character '+'.

1.1 Inherited and Synthesized Attributes

Two kinds of attributes for nonterminals:

1. A *synthesized attribute* for a nonterminal *A* at a parse-tree node *N* is defined by a semantic rule associated with the production at *N.* Note that the production must have *A* as its head. A synthesized attribute at node *N* is defined only in terms of attribute values at the children of *N* and at *N* itself.
2. An *inherited attribute* for a nonterminal B at a parse-tree node *N* is defined by a semantic rule associated with the production at the parent of *N.* Note that the production must have *B* as a symbol in its body. An inherited attribute at node *N* is defined only in terms of attribute values at N's parent, *N* itself, and *N's* siblings.

Example 3.3.1: The SDD in Fig. 3.3.1 is based on our familiar grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an end marker n. In the SDD, each of the nonterminals has a single synthesized attribute, called *val.* We also suppose that the terminal digit has a synthesized attribute *lexval,* which is an integer value returned by the lexical analyzer.

Production	Semantic Rules
$L \rightarrow E\ \mathbf{n}$	$print\ (E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val + F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \mathbf{digit}$	$F.val := \mathbf{digit}.lexval$

Figure 3.3.1 : Syntax-directed definition of a simple desk calculator

- The rule for production 1, *L* →*E* n, sets *L.val* to *E.val,* which we shall see is the numerical value of the entire expression.
- Production 2, *E* →*E_1+ T,* also has one rule, which computes the *val* attribute for the head *E* as the sum of the values at *E_1*and *T.* At any parse tree node *N* labeled *E,* the value of *val* for *E* is the sum of the values of *val* at the children of node *N* labeled *E* and *T.*

- Production 3, *E* →*T,* has a single rule that defines the value of *val* for *E* to be the same as the value of *val* at the child for *T.*
- Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them.
- The rules for productions 5 and 6 copy values at a child, like that for the third production. Production 7 gives *F.val* the value of a digit, that is, the numerical value of the token digit that the lexical analyzer returned.

An SDD that involves only synthesized attributes is called *S-attributed;* the SDD in Fig. 3.3.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.
An SDD without side effects is sometimes called an *attribute grammar.* The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

1.2 Evaluating an SDD at the Nodes of a Parse Tree
A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree.* For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes.

Example 3.3.2: Figure 3.3.2 shows an annotated parse tree for the input string3 * 5 + 4 n, constructed using the grammar and rules of Fig. 3.3.1. The values of *lexval* are presumed supplied by the lexical analyzer. Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node. For instance, at the node with a child labeled *, after computing *T.val =* 3 and *F.val*= 5 at its first and third children, we apply the rule that says *T.val* is the product of these two values,or 15.

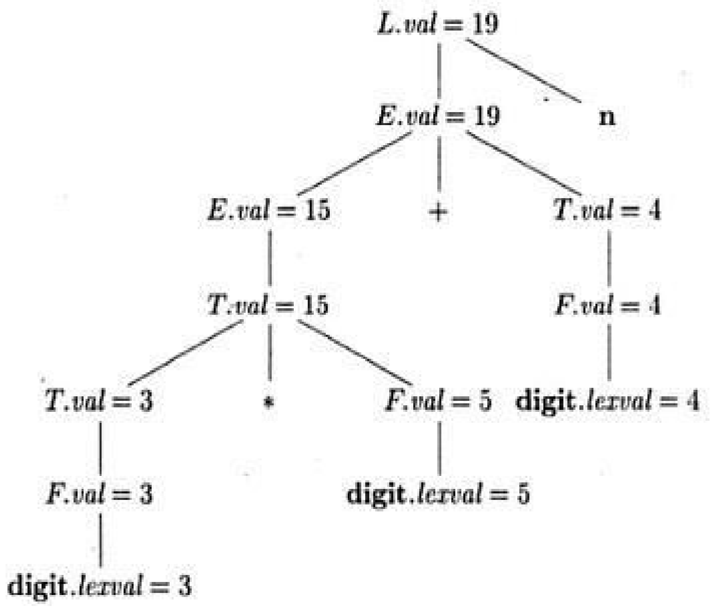
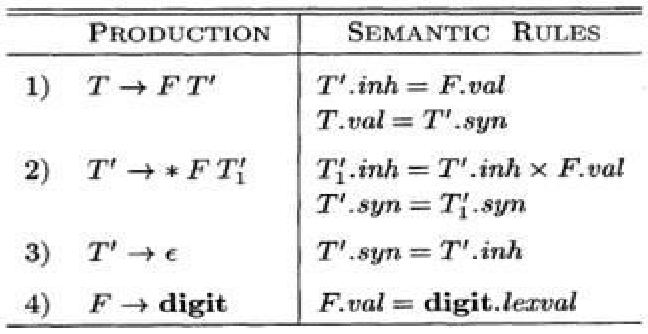


Figure 3.3.2: An SDD based on a grammar suitable for top-down parsing

Example 3.3.3: The SDD in Fig. 3.3.3 computes terms like 3 * 5 and 3 * 5 * 7.The top-down parse of input 3*5 begins with the production *T* →*FT'.* Here, *F* generates the digit

3, but the operator * is generated by T'. Thus, the left operand 3 appears in a different sub tree of the parse tree from *. An inherited attribute will therefore be used to pass the operand to the operator.



PRODUCTION	SEMANTIC RULES
1) $T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to * \,F\,T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Figure 3.3.3: An SDD based on a grammar suitable for top-down parsing

Each of the nonterminals *T* and *F* has a synthesized attribute *val;* the terminal digit has a synthesized attribute *lex val.* The nonterminal *T* has two attributes: an inherited attribute *inh* and a synthesized attribute *syn.*

The semantic rules are based on the idea that the left operand of the operator* is inherited. More precisely, the head T' of the production *T'*→ * *F* T_1' inherits the left operand of * in the production body. Given a term *x * y * z,* the root of the subtree for * *y* * *z* inherits *x.* Then, the root of the subtree for * *z* inherits the value of *x * y,* and so on, if there are more factors in the term. Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

To see how the semantic rules are used, consider the annotated parse tree for 3 * 5 in Fig. 3.3.5. The leftmost leaf in the parse tree, labeled digit, has attribute value *lexval*= 3, where the 3 is supplied by the lexical analyzer. Its parent is for production 4, *F* -> digit. The only semantic rule associated with this production defines *F.val*= digit.*lexval,* which equals 3.

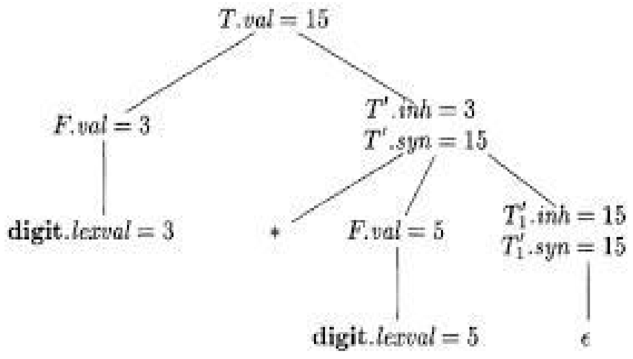


*Figure 3.3.5: Annotated parse tree for 3 * 5*

At the second child of the root, the inherited attribute *T'.inh* is defined by the semantic rule *T'.inh = F.val* associated with production 1. Thus, the left operand, 3, for the * operator is passed from left to right across the children of the root.

The production at the node for *T* is T' → * FT_1'.(We retain the subscript1 in the annotated parse tree to distinguish between the two nodes for *T'.)* The inherited attribute $T_1'.inh$ is defined by the semantic rule $T_1'.inh = T'.inh * F.val$ associated with production 2.

With *T'.inh* = 3 and *F.val* = 5, we get *T_1'.inh* = 15 . At the lower node for *T_1',* the production is T'-> ε. The semantic rule *T'.syn= T'.inh*defines*T_1'.syn* =15. The *syn* attributes at the nodes for T' pass the value 15 up the tree to the node for T, where *T.val* = 15.

Evaluation Orders for SDD's

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.

- 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.* 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.
- 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.*

Example 3.4.1: Consider the following production and rule:

PRODUCTION	SEMANTIC RU LE
E ->E_1+ T	*E.val= E_1.val + T.val*

At every node *N* labeled *E,* with children corresponding to the body of this production, the synthesized attribute *val* at *N* is computed using the values of *val* at the two children, labeled *E* and *T.* Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig. 3.4.1. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

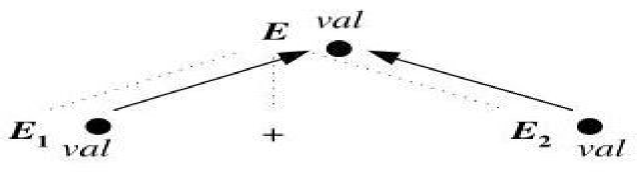


Figure 3.4.1: E.val is synthesized from Ei.val and T.val

Example 3.4.2: An example of a complete dependency graph appears in Fig. 3.4.2. The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 3.3.5

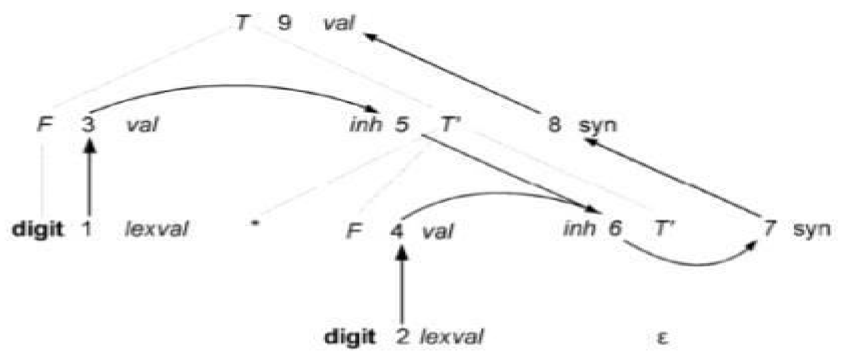


Figure 3.4.2: Dependency graph for the annotated parse tree of Fig. 3.3.5

- Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled digit.
- Nodes 3 and 4 represent the attribute *val* associated with the two nodes labeled *F.* The edges to node 3 from 1 and to node 4 from 2 result from the semantic rule that defines *F.val* in terms of *digit.lexval* In fact, *F.val* equals digit *.lexval,* but the edge represents dependence, not equality.
- Nodes 5 and 6 represent the inherited attribute *T'.inh* associated with each of the occurrences of nonterminal *T'.* The edge to 5 from 3 is due to the rule *T'.inh = F.val,* which defines *T'.inh* at the right child of the root from *F.val* at the left child. We see edges to 6 from node 5 for *T'.inh* and from node 4 for *F.val,* because these values are multiplied to evaluate the attribute *inh* at node 6.
- Nodes 7 and 8 represent the synthesized attribute *syn* associated with the occurrences of X". The edge to node 7 from 6 is due to the semantic rule *T'.syn = T'.inh* associated with production 3 in Fig. 3.3.4. The edge to node 8 from 7 is due to a semantic rule associated with production 2.
- Finally, node 9 represents the attribute *T.val.* The edge to 9 from 8 is due to the semantic rule, *T.val = T'.syn,* associated with production 1.

The dependency graph of Fig. 3.4.2 has no cycles. One topological sort is the order in which the nodes have already been numbered: 1,2,... ,9. Notice that every edge of the graph goes from a node to a higher-numbered node, so this order is surely a topological sort. There are other topological sorts as well, such as 1,3,5,2,4,6,7,8,9.

2. S-Attributed Definitions
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 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.

Example 3.4.2: The SDD of Fig 3.3.1 is an example of an S-attributed definition. Each attribute, *L.val, E.val, T.val,* and *F.val* 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

```
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.

3. 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 $A \rightarrow X_1X_2 \ldots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:
   (a) Inherited attributes associated with the head *A.*
   (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1X_2 \ldots X_n$ located to the left of X_i.
   (c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_j .

Example 3.4.2: The SDD in Fig. 3.3.4 is L-attributed. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

PRODUCTION	SEMANTIC RULE
$T \rightarrow FT'$	$T'.inh = F.val$
$T' \rightarrow *F\, T_1'$	$T_1'.Inh = T'.inh \times F.val$

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[.inh* using the inherited attribute *T'.inh* associated with the head, and *F.val,* where *F* appears to the left of *T[* 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.

Applications of Syntax-Directed Translation

The main application in the construction of syntax trees is some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing.

3.1 Construction of Syntax Trees

Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression *E1 + E2* has label + and two children representing the sub expressions *E1*and *E2.*

We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an op field that is the label of the node. The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function *Leaf (op, val)* creates a leaf object. Alternatively, if nodes are viewed as records, then *Leaf* returns a pointer to a new record for a leaf.
- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function *Node* takes two or more arguments: *Node(op,c_1,c_2,... ,c_k)* creates an object with first field *op* and *k* additional fields for the *k* children c_1, c_2,... ,c_k.

Example 3.5.1: The S-attributed definition in Fig. 3.5.1 constructs syntax trees for a simple expression grammar involving only the binary operators +and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node,* which represents a node of the syntax tree.

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

Figure 3.5.1 : Constructing syntax trees for simple expressions

Every time the first production $E \rightarrow E_1 + T$ is used, its rule creates a node with ' + ' for *op* and two children, E_1*.node* and *T.node,* for the sub expressions. The second production has a similar rule.

For production 3, $E \rightarrow T$, no node is created, since *E.node* is the same as *T.node.* Similarly, no node is created for production 4, T → (*E*) . The value of *T.node* is the same as *E.node,* since parentheses are used only for grouping; they influence the structure of the parse tree and the syntax tree, but once their job is done, there is no further need to retain them in the syntax tree.

The last two T-productions have a single terminal on the right. We use the constructor *Leaf* to create a suitable node, which becomes the value of *T.node.*

Figure 3.5.2 shows the construction of a syntax tree for the input *a* — 4 + *c.*

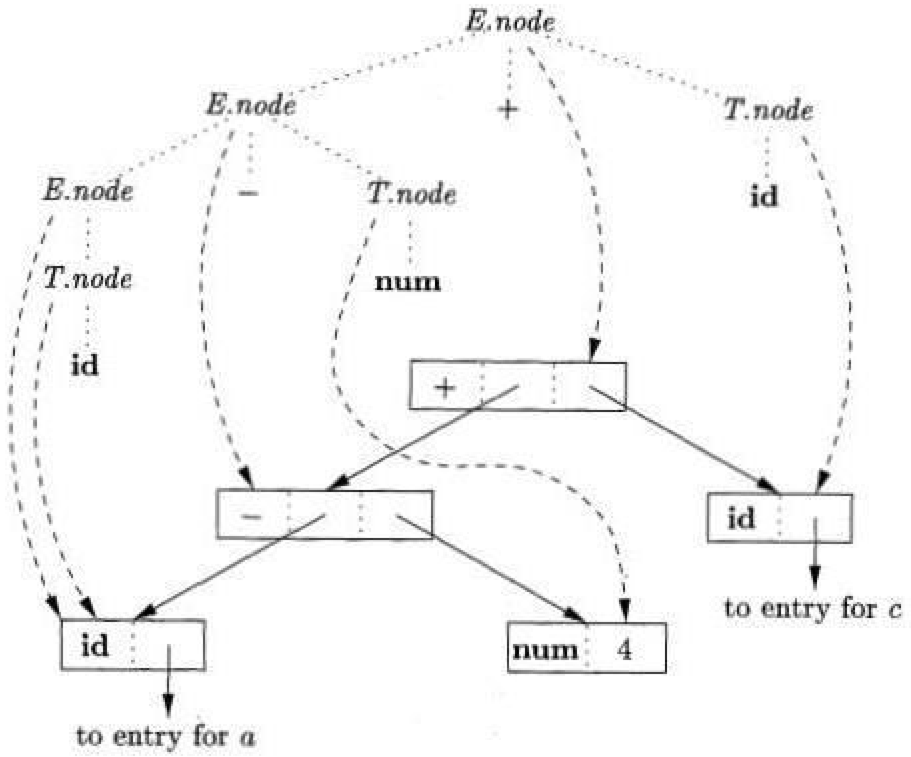


Figure 3.5.2: Syntax tree for a — 4 + c

The nodes of the syntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The third type of line, shown dashed, represents the values of *E.node* and *T-node;* each line points to the appropriate syntax-tree node.

At the bottom we see leaves for *a,* 4 and c, constructed by *Leaf.* We suppose that the lexical value id. *entry* points into the symbol table, and the lexical value num.val is the numerical value of a constant. These leaves, or pointers to them, become the value of *T.node* at the three parse-tree nodes labeled *T,* according to rules 5 and 6. Note that by rule 3, the pointer to the leaf for *a* is also the value of *E.node* for the leftmost *E* in the parse tree.

Rule 2 causes us to create a node with *op* equal to the minus sign and pointers to the first two leaves. Then, rule 1 produces the root node of the syntax tree by combining the node for — with the third leaf.

If the rules are evaluated during a postorder traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown in Fig. 3.5.3 ends with p5 pointing to the root of the constructed syntax tree.

1) $p_1 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new}\ Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ Node('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new}\ Node('+', p_3, p_4);$

Figure 3.5.3: Steps in the construction of the syntax tree for a — 4 + c

Example 3.5.2: The L-attributed definition in Fig. 3.5.4 performs the same translation as the S-attributed definition in Fig. 3.5.1. The attributes for the grammar symbols *E, T,* id, and num are as discussed in Example 3.5.2.

	PRODUCTION	SEMANTIC RULES
1)	$E \to T\,E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \to +\,T\,E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \to -\,T\,E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \to \epsilon$	$E'.syn = E'.inh$
5)	$T \to (\ E\ )$	$T.node = E.node$
6)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 3.5.4 : Constructing syntax trees during top-down parsing

The rules for building syntax trees in this example are similar to the rules for the desk calculator. In the desk-calculator example, a term *x * y* was evaluated by passing *x* as an inherited attribute, since *x* and * *y* appeared in different portions of the parse tree. Here, *the* idea is to build a syntax tree for *x + y* by passing *x* as an inherited attribute, since *x* and + *y* appear in different sub trees. Nonterminal *E'* is the counterpart of nonterminal T'.

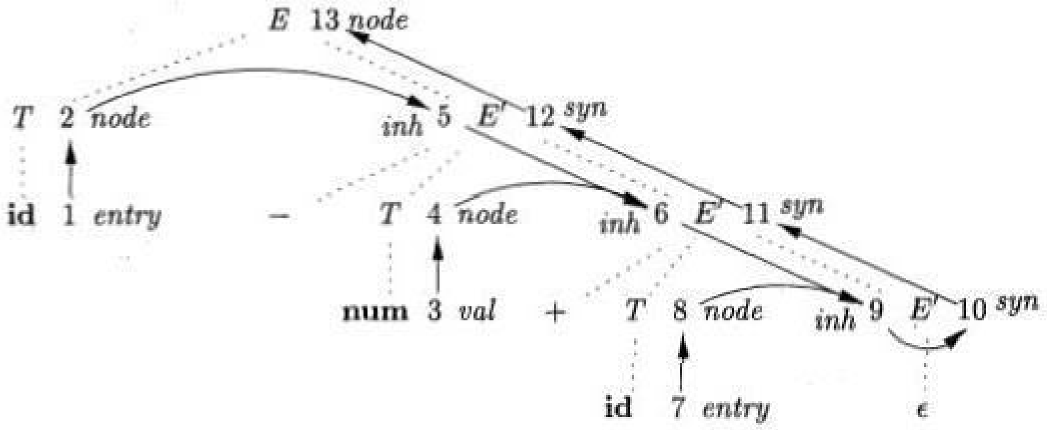


Figure 3.5.5 : Dependency graph for a - 4 + c, with the SDD of Fig. 3.5.2

Nonterminal *E'* has an inherited attribute *inh* and a synthesized attribute *syn.* Attribute *E'.inh* represents the partial syntax tree constructed so far. Specifically, it represents the root of the tree for the prefix of the input string that is to the left of the subtree for *E'.* At node 5 in the dependency graph in Fig. 3.5.5, *E'.inh* denotes the root of the partial syntax tree for the identifier a; that is, the leaf for *a.* At node 6, *E'.inh* denotes the root for the partial syntax tree for the input *a* — 4. At node 9, *E'.inh* denotes the syntax tree for *a* — 4 + c.

Since there is no more input, at node 9, *E'.inh* points to the root of the entire syntax tree. The *syn* attributes pass this value back up the parse tree until it becomes the value of *E.node.* Specifically, the attribute value at node 10 is defined by the rule *E'.syn = E'.inh* associated with the production *E' —> e.* The attribute value at node 11 is defined by the rule *E'.syn* = E_1*'.syn* associated with production 2 in Fig.3.5.4. Similar rules define the attribute values at nodes 12 and 13.

Syntax-Directed Translation Schemes

Syntax-directed translation schemes are a complementary notation to syntax directed definitions. A *syntax-directed translation scheme* (SDT) is a context free grammar with program fragments embedded within production bodies. The program fragments are called *semantic actions* and can appear at any position within a production body.

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.

Typically, SDT's are implemented during parsing, without building a parse tree. To implement SDT's, two important classes of SDD's:

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.
2. The underlying grammar is LL-parsable, and the SDD is L-attributed.

1. Postfix Translation Schemes

The simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called *postfix SDT's.*

Example 3.6.1: The postfix SDT in Fig. 3.6.1 implements the desk calculator SDD of Fig. 3.3.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser.

$$\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \mathrm{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}$$

Figure 3.6.1: Postfix SDT implementing the desk calculator

2. Parser-Stack Implementation of Postfix SDT's

Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur. The attribute(s) of each grammar symbol can be put on the stack in a place where they can be found during the reduction. The best plan is to place the attributes along with the grammar symbols (or the LR states that represent these symbols) in records on the stack itself.

In Fig. 3.6.2, the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols *X YZ* are on top of the stack; perhaps they are about to be reduced according to a production like *A —> X YZ.* Here, we show *X.x*as the one attribute of *X,* and so on. In general, we can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack. With small attributes, it may be simpler to make the records large enough, even if some fields go unused some of the time. However, if one or more attributes are of unbounded size — say, they are character strings — then it would be better to put a pointer to the attribute's value in the stack record and store the actual value in some larger, shared storage area that is not part of the stack.

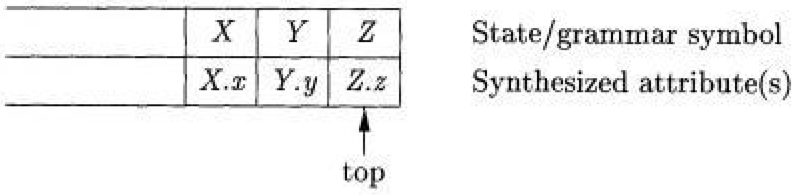


Figure 3.6.2: Parser stack with a field for synthesized attributes

If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head. If we reduce by a production such *as A →X Y Z, then* we have all the attributes of *X, Y,* and *Z* available, at known positions on the stack, as in Fig.3.6.2. After the action, *A* and its attributes are at the top of the stack, in the position of the record for *X.*

Example 3.6.2: Let us rewrite the actions of the desk-calculator SDT of Example 3.6.1 so that they manipulate the parser stack explicitly. Such stack manipulation is usually done automatically by the parser.

PRODUCTION	ACTIONS
$L \rightarrow E\ \mathbf{n}$	$\{\ \text{print}(stack[top-1].val);$ $top = top - 1;\ \}$
$E \rightarrow E_1 + T$	$\{\ stack[top-2].val = stack[top-2].val + stack[top].val;$ $top = top - 2;\ \}$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;$ $top = top - 2;\ \}$
$T \rightarrow F$	
$F \rightarrow (\ E\ )$	$\{\ stack[top-2].val = stack[top-1].val;$ $top = top - 2;\ \}$
$F \rightarrow \mathbf{digit}$	

Figure 3.6.3: Implementing the desk calculator on a bottom-up parsing stack

Suppose that the stack is kept in an array of records called *stack,* with *top* a cursor to the top of the stack. Thus, *stack [top]* refers to the top record on the stack, *stack [top* - 1] to the record below that, and so on. Also, we assume that each record has a field called *val,* which holds the attribute of whatever grammar symbol is represented in that record.

Thus, we may refer to the attribute *E.val* that appears at the third position on the stack as *stack[top -2].val.* The entire SDT is shown in Fig. 3.6.3.

For instance, in the second production, $E \rightarrow E_1 + T$, we go two positions below the top to get the value of E_1, and we find the value of *T* at the top. The resulting sum is placed where the head *E* will appear after the reduction, that is, two positions below the current top. The reason is that after the reduction, the three topmost stack symbols are replaced by one. After computing *E.val,* we pop two symbols off the top of the stack, so the record where we placed *E.val* will now be at the top of the stack.

In the third production, $E \rightarrow T$, no action is necessary, because the length of the stack does not change, and the value of *T.val* at the stack top will simply become the value of *E.val.* The same observation applies to the productions $\rightarrow F$ and $F \rightarrow$digit. Production $F \rightarrow ($ *E*) is slightly different. Although the value does not change, two positions are removed from the stack during the reduction, so the value has to move to the position after the reduction.

3. SDT's With Actions inside Productions

An action may be placed at any position within the body of a production. It is performed immediately after all symbols to its left are processed. Thus, if we have a production $\rightarrow X$ *{a} Y,* the action *a* is done after we have recognized *X* (if *X* is a terminal) or all the terminals derived from *X* (if *X* is a nonterminal). More precisely,

- If the parse is bottom-up, then we perform action *a* as soon as this occurrence of *X* appears on the top of the parsing stack.
- If the parse is top-down, we perform *a* just before we attempt to expand this occurrence of *Y* (if *Y* a nonterminal) or check for *Y* on the input (if *Y* is a terminal).

SDT's that can be implemented during parsing include postfix SDT's and a class of SDT's that implements L-attributed definitions.

Example 3.6.3: As an extreme example of a problematic SDT, suppose that we turn our desk-calculator running example into an SDT that prints the prefix form of an expression, rather than evaluating the expression. The productions and actions are shown in Fig. 3.6.4.

$$\begin{array}{lll l}
1) & L & \rightarrow & E\ \mathbf{n} \\
2) & E & \rightarrow & \{\ \mathrm{print}('+');\ \}\ \ E_1 + T \\
3) & E & \rightarrow & T \\
4) & T & \rightarrow & \{\ \mathrm{print}('*');\ \}\ \ T_1 * F \\
5) & T & \rightarrow & F \\
6) & F & \rightarrow & (\ E\ ) \\
7) & F & \rightarrow & \mathbf{digit}\ \ \{\ \mathrm{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}$$

Figure 3.6.4: Problematic SDT for infix-to-prefix translation during parsing

Unfortunately, it is impossible to implement this SDT during either topdown or bottom-up parsing, because the parser would have to perform critical actions, like printing instances of * or +, long before it knows whether these symbols will appear in its input.

Using marker nonterminals M2 and M4 for the actions in productions 2and 4, respectively, on input 3, a shift-reduce parser has conflicts between reducing by M2 $\rightarrow \varepsilon$, reducing by M4 $\rightarrow$ ε, and shifting the digit.

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node *N,* say one for production *A* $\rightarrow$*α.* Add additional children to *N* for the actions in *α,* so the children of *N* from left to right have exactly the symbols and actions of *α.*
3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

For instance, Fig. 3.6.5 shows the parse tree for expression 3 * 5 + 4 with actions inserted. If we visit the nodes in preorder, we get the prefix form of the expression: + * 3 5 4.

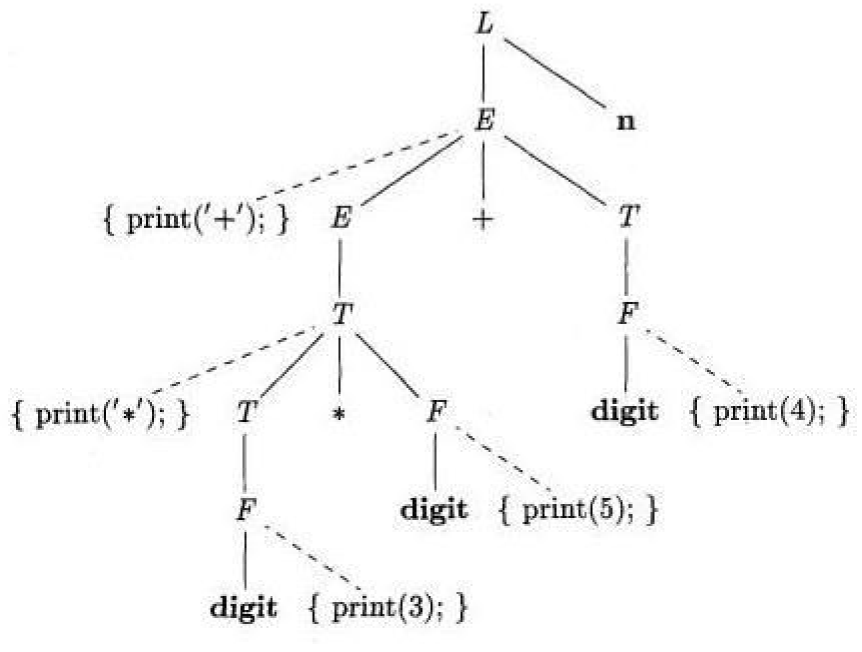


Figure 3.6.5: Parse tree with actions embedded

4. Eliminating Left Recursion from SDT's

Since no grammar with left recursion can be parsed deterministically top-down, we examined left-recursion elimination. When the grammar is part of an SDT, we also need to worry about how the actions are handled.

First, consider the simple case, in which the only thing we care about is the order in which the actions in an SDT are performed. For example, if each action simply prints a string, we care only about the order in which the strings are printed. In this case, the following principle can guide us:

• When transforming the grammar, treat the actions as if they were terminal symbols.

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string. The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up. The "trick" for eliminating left recursion is to take two productions

$$A \rightarrow A\alpha \mid \beta$$

that generate strings consisting of a β and any number of α's, and replace them by productions that generate the same strings using a new nonterminal *R* (for “remainder") of the first production:

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \varepsilon$$

If βdoes not begin with *A,* then *A* no longer has a left-recursive production. In regular-definition terms, with both sets of productions, *A* is defined by **β***(α)*.*

Example 3.6.4: Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

$$E \rightarrow E_1 + T \text{ \{ print('+'); \}}$$
$$E \rightarrow T$$

If we apply the standard transformation to *E,* the remainder of the left-recursive production is $\alpha = + T \text{ \{ print('+'); \}}$

and the body of the other production is *T.* If we introduce *R* for the remainder of *E,* we get the set of productions:

$$E \rightarrow T\ R$$
$$R \rightarrow + T \text{ \{ print('+'); \} } R$$
$$R \rightarrow \varepsilon$$

When the actions of an SDD compute attributes rather than merely printing output, we must be more careful about how we eliminate left recursion from a grammar. However, if the SDD is S-attributed, then we can always construct an SDT by placing attribute-computing actions at appropriate positions in the new productions.

5. SDT's for L-Attributed Definitions

We converted S-attributed SDD's into postfix SDT's, with actions at the right ends of productions.

The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal *A* immediately before that occurrence of *A* in the body of the production. If several inherited attributes for *A* depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.
2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

INTERMEDIATE CODE GENERATION

4.1. INTRODUCTION

What is Intermediate Code?
Intermediate Code is a modified input source program which is stored in some data structure. The front end translates a source program into an intermediate representation from which the back end generates target code.

Why Intermediate Code Generation is required?
Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimization can be applied to the intermediate representation.

Position of intermediate code generator

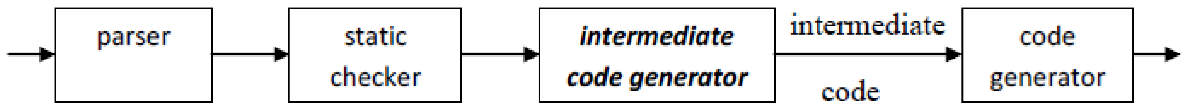


Fig. 4.1: Position of Intermediate Code Generator

Intermediate code can be represented by using the following:

i) Three Address Code
ii) Syntax Tree

4.1.1 Three-Address Code:
Three-address code is a sequence of statements of the general form
x = y *op* z
Where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean valued data. Thus a source language expression like x+ y*z might be translated into a sequence
t1= y * z
t2= x + t1
Where t1 and t2 are compiler-generated temporary names.
The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.
An address can be one of the following:
• *A name.* For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
• *A constant.* In practice, a compiler must deal with many different types of constants and variables.
• *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.
Types of Three-Address Statements:
The common three-address statements are:

1. Assignment statements of the form x = y *op* z, where *op* is a binary arithmetic or logical operation.
2. Assignment instructions of the form x = *op* y, where *op* is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. *Copy statements* of the form x = y where the value of *y* is assigned to *x*.
4. The unconditional jump goto L. The three-address statement with label L is the next to

5. Conditional jumps such as if *x relop y* goto L. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation *relop to y*. If not, the three-address statement following if *x relop y* goto L is executed next, as in the usual sequence.
6. *param x* and *call p, n* for procedure calls and *return y*, where y representing a returned valueis optional. For example,
   param x1
   param x2
   . . .
   paramxn
   callp,n
   generated as part of a call of the procedure p(x1, x2, …. ,x_n).
7. Indexed assignments of the form x:= y[i] and x[i] = y.
8. Address and pointer assignments of the form x = &y, x= *y, and *x= y.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands.

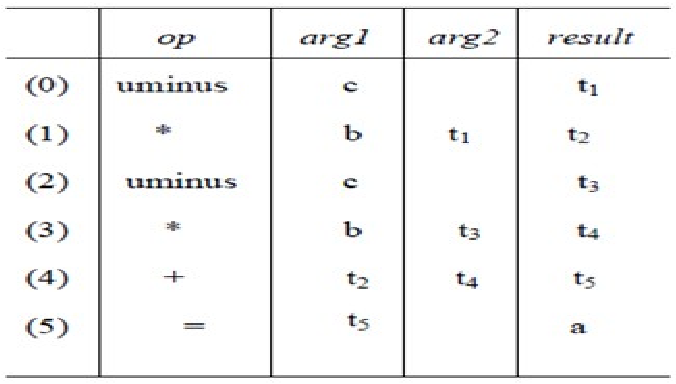
Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

Three-Address Code does not specify the internal representation of 3-Address instructions. This limitation is overcome by Quadruple.

- A quadruple is a record structure with four fields, which are, *op, arg1, arg2* and *result.*
- The *op* field contains an internal code for the operator. The three-address statement *x =y op z*is represented by placing *y* in *arg1*, *z* in *arg2* and *x* in *result.*
- The contents of field's arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.
  Example*: a: =b*-c+b*-c* represent the expression using Quadruple, Triples and Indirect Triples.



	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

(a) Quadruples

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields:
  op, arg1 and *arg2.*
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as *triples*.

	op	arg1	arg2
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

**Note: The benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With Quadruples if we move an instruction that computes a temporary t, then the instruction that uses t require no change. With Triples the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur in Indirect Triples.

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order.

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	minus	c	
(15)	*	b	(14)
(16)	minus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

4.1.2 ABSTRACT SYNTAX TREES:

Abstract Syntax tree is a condensed version of a Syntax Tree eliminating all syntactic elements of the language. Abstract Syntax Tree is also used to represent the Intermediate Code. The procedure for constructing abstract syntax tree is same as the procedure that we used to convert an expression into a postfix notation. The operators act as the parent nodes and variables, constants, identifiers as leaf nodes. Abstract Syntax tree is constructed form bottom to top.

- Every node in a syntax tree is a record with many fields. For example an operator will have two operands.
- The three functions makeleaf(identifier,entry), makeleaf(number,value) and makenode(operator,operand1,operand2) are used while constructing the abstract syntax trees.

1. Makeleaf(identifier,entry)
   This function creates an identifier node with the name or label "identifier" and a pointer to symbol table entry given by "entry".
2. Makeleaf(number,value)
   This function creates a leaf node "number" and "value".
3. Makenode(Operator,Operand1,Operand2)
   This function creates an operator node with a name "operator" and a pointer to the left child (Operand1) and a pointer to the right child (Operand 2).The left and right child can be again an operator node.

Example: Construct Abstract Syntax Tree for the Expression *a*b-(c+d).*

1. Pointer1=Makeleaf(identifier, entry a);
2. Pointer2= Makeleaf(identifier, entry b);

3. Pointer3= Makenode('*', Pointer1,Pointer2);
4. Pointer4= Makeleaf(identifier, entry c);
5. Pointer5= Makeleaf(identifier, entry d);
6. Pointer6= Makenode('+', Pointer4,Pointer5);
7. Pointer7= Makenode('-', Pointer3,Pointer6);

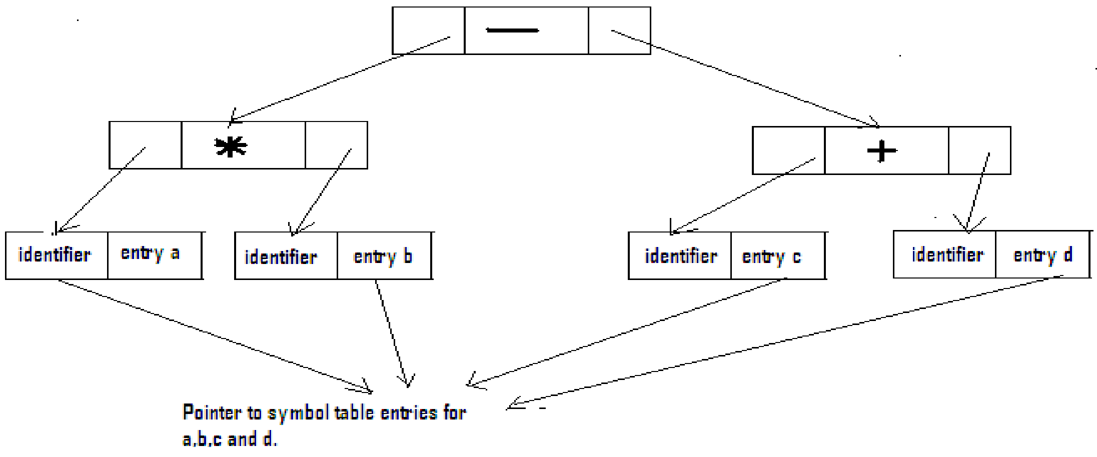


Fig.4.2: Abstract Syntax Tree

4.1.3. Directed Acyclic Graph:

An important derivative of abstract syntax tree is known as Directed Acyclic Graph. It is used to reduce the amount of memory used for storing the Abstract Syntax Tree data structure.

Consider an expression:

k=k-7;

The AST and DAG is shown in the fig below

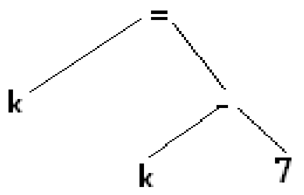


Fig. 4.3: AST

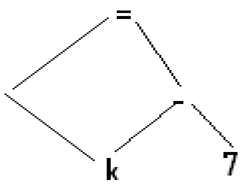


Fig.4.4: DAG

Note: There are 2 nodes for the identifier 'k' in fig 1, one representing k on the LHS of the expression and the other representing the k on the RHS. The DAG identifies such common nodes and eliminates their duplication in the AST. The DAG for the above expression is shown in fig2.
In DAG, a node may have multiple parents. In fig2 node 'k' has two parents (- node and = node).
The creation of DAG is identical to the AST except for the extra check to determine whether a node with identical properties already exists. In the event of the node already created before, it is chained to the existing node avoiding a duplicate node.

Example 2: a + a * (b - c) + (b - c) * d
The leaf for a has two parents, because a appears twice in the expression. More interestingly, the two occurrences of the common sub expression b - c are represented by one node, the node labeled —. That node has two parents, representing its two uses in the sub expressions a* (b - c) and (b-c)*d. Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common sub expression b - c.

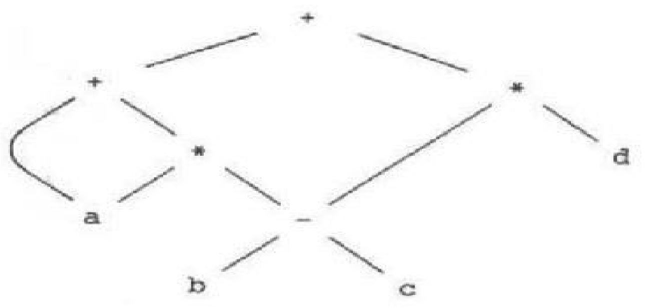
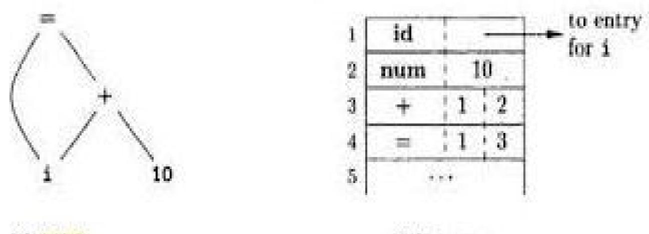


Fig 4.5: DAG for a + a * (b - c) + (b - c) * d

The Value-Number Method for Constructing DAG's
Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 4.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 4.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and interior nodes have two additional fields indicating the left and right children.



(a) DAG (b) Array.

Fig. 4.6: Steps for constructing the DAG of Fig. 4.3

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer historically has been called the *value number* for the node or for the expression represented by the node. For instance, in Fig. 4.6, the node labeled +has value number 3, and it's left and right children have value numbers 1 and 2, respectively. In practice, we could use pointers to records or references to objects instead of integer indexes, but we shall still refer to the reference to a node as its "value number." If stored in an appropriate data structure, value numbers help us construct expression DAG's efficiently.

4.2 Types and Declarations
The applications of types can be grouped under checking and translation:

- *Type checking* uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator.
- *Translation Applications.* From the type of a name, a compiler can determine the storage that will be needed for that name at run time.

4.2.1 Type Expressions
Types have structure, which we shall represent using *type expressions:* a type expression is either a basic type or is formed by applying an operator called a *type constructor* to a type expression. The sets of basic types and constructors depend on the language to be checked.
Example: The array type int [2] [3] can be read as "array of 2 arrays of 3 integers each" and written as a type expression *array (2, array (3, integer)).* This type is represented by the tree in Fig. 4.12. The operator *array* takes two parameters, a number and a type.

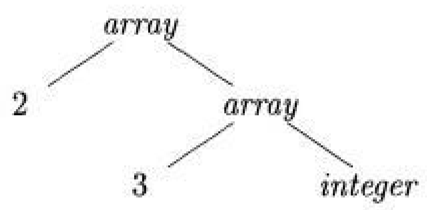


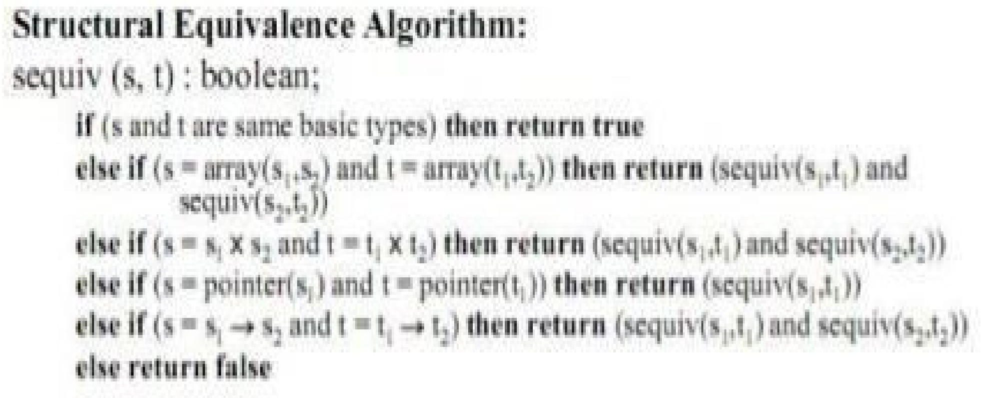
Figure 4.7: Type expression for int [2][3]

We shall use the following definition of type expressions:

- A *basic type* is a type expression. Typical basic types for a language include *boolean, char, integer, float,* and *void.*
- A *type name* is a type expression.
- A *type expression* can be formed by applying the array type constructor to a number and a type expression.
- A *record* is a data structure with named fields. A type expression can be formed by applying the *record* type constructor to the field names and their types.
- A type expression can be formed by using the type constructor → for function types. We write *s* →*t* for "function from type *s* to type t."
- If *s* and *t* are type expressions, then their Cartesian product *s Xt* is a type expression.
- Type expressions may contain variables whose values are type expressions.

4.2.2 Type Equivalence

- The basic question is "when are two type expressions equivalent?"
- Two expressions are structurally equivalent if there are two expressions of same basic type or are formed by applying same constructor.



Structural Equivalence Algorithm:

```
sequiv (s, t) : boolean;
    if (s and t are same basic types) then return true
    else if (s = array(s_1,s_2) and t = array(t_1,t_2)) then return (sequiv(s_1,t_1) and
            sequiv(s_2,t_2))
    else if (s = s_1 X s_2 and t = t_1 X t_2) then return (sequiv(s_1,t_1) and sequiv(s_2,t_2))
    else if (s = pointer(s_1) and t = pointer(t_1)) then return (sequiv(s_1,t_1))
    else if (s = s_1 → s_2 and t = t_1 → t_2) then return (sequiv(s_1,t_1) and sequiv(s_2,t_2))
    else return false
```

- Example: int a, b
  Here a and b are structurally equivalent

4.2.3 Declarations

Types and declarations using a simplified grammar that declares just one name at a time; declarations with lists of names can also be handled. The grammar is

D →**T id ; D | ε**
T → B C | record '{' D '}'
B →int | float
C → **ε | [num] C**

- Non terminal *D* generates a sequence of declarations.
- Non terminal *T* generates basic, array, or record types.
- Non terminal *B* generates one of the basic types int and float.
- Non terminal 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 non terminal *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.

4.2.4 Storage Layout for Local Names

From the type of a name, we can determine the amount of storage that will be needed for the name at run time. At compile time, we can use these amounts to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name. Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known fixed amount of storage for a pointer to the data.

The *width* of a type is the number of storage units needed for objects of that type. A basic type, such as a character, integer, or float, requires an integral number of bytes. For easy access, storage for aggregates such as arrays and classes is allocated in one contiguous block of bytes.

4.3 Type Checking

Type checking has the potential for catching errors in programs. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element. A *sound* type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs. An implementation of a language is *strongly typed* if a compiler guarantees that the programs it accepts will run without type errors.

4.3.1 Rules for Type Checking

Type checking can take on two forms: synthesis and inference.

1. *Type synthesis* builds up the type of an expression from the types of its sub expressions. It requires names to be declared before they are used. The type of $E_1 + E_2$ is defined in terms of the types of E_1 *and* E_2.
2. *Type inference* determines the type of a language construct from the way it is used. Let *null* be a function that tests whether a list is empty. Then, from the usage *null(x),* we can tell that *x* must be a list. The type of the elements of *x* is not known; all we know is that *x* must be a list of elements of some type that is presently unknown.

4.3.2 Type Conversions

Consider expressions like $x + i$, where x is of type float and i is of type integer. Since the expression has two different types of operands, the compiler may need to convert one of the operands of + to ensure that both operands are of the same type when the addition occurs. Suppose that integers are converted to floats when necessary, using a unary operator (float). For example, the integer 2 is converted to a float in the code for the expression 2*3.14:

$$t_1 = (float)\ 2$$
$$t_2 = t_1 * 3.14$$

Type synthesis will be illustrated by extending the scheme for translating expressions. We introduce another attribute *E.type,* whose value is either *integer* or *float.* The rule associated with $E \rightarrow E1+E2$ builds on the pseudo code

```
if ( $E_1$.type = integer and $E_2$.type = integer ) E.type = integer:
else if ( $E_1$.type = float and $E_2$.type = integer ) . . .
```

Type conversion rules vary from language to language. The rules for Java in Fig. 4.16 distinguish between *widening* conversions, which are intended to preserve information, and *narrowing* conversions, which can lose information.

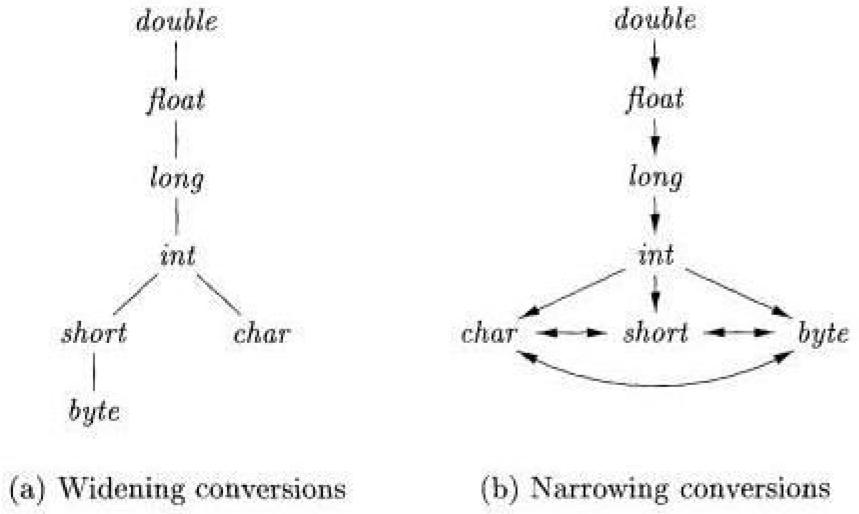


Figure 4.8: Conversions between primitive types in Java

The widening rules are given by the hierarchy in Fig. 4.8(a): any type lower in the hierarchy can be widened to a higher type. Thus, a *char* can be widened to an *int* or to a *float,* but a *char* cannot be widened to a *short.* The narrowing rules are illustrated by the graph in Fig. 4.8(b): a type *s*can be narrowed to a type *t* if there is a path from *s* to *t.* Note that *char, short,* and *byte* are pair wise convertible to each other.

Conversion from one type to another is said to be *implicit* if it is done automatically by the compiler. Implicit type conversions, also called *coercions,* are limited in many languages to widening conversions. Conversion is said to be *explicit* if the programmer must write something to cause the conversion. Explicit conversions are also called *casts.*

The semantic action for checking $E \rightarrow E_1 + E_2$ uses two functions:

1. *max*(t_1,t_2) takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.
2. *widen(a, t, w)* generates type conversions if needed to widen an address *a* of type t into a value of type *w.* It returns *a* itself if *t* and *w* are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary *t,* which is returned as the result.

4.4 Control Flow

4.4.1 Boolean Expressions

Boolean expressions are composed of the boolean operators (which we denote &&, II, 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. We consider boolean expressions generated by the following grammar:

B →B ||B | B && B | ! B |(B) | E relE | true | false

We use the attribute rel*op* to indicate which of the six comparison operators <, <= , =, ! =, >, or >= is represented by rel. As is customary, we assume that II and && are left-associative, and that II has lowest precedence, then &&, then !.

Given the expression *B1* || *B2,* if we determine that *B1* is true, and 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.

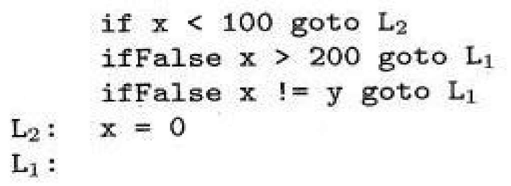
4.4.2 Short-Circuit Code

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: The statement

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

might be translated into the code.



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

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.

4.4.3. 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:

$S \rightarrow$ if (*B*) S_1

$S \rightarrow$ if (*B*) S_1 else S_2

$S \rightarrow$ while (B) S_1

In these productions, non terminal *B* represents a boolean expression and non terminal S represents a statement.

The translation of if *(B)* S_1 consists of *B.code* followed by S_1*.code,* as illustrated in Fig. 6.35(a). Within *B.code* are jumps based on the value of *B.* If *B* is true, control flows to the first instruction of S_1*.code,* and if *B* is false, control flows to the instruction immediately following S_1*.code.*

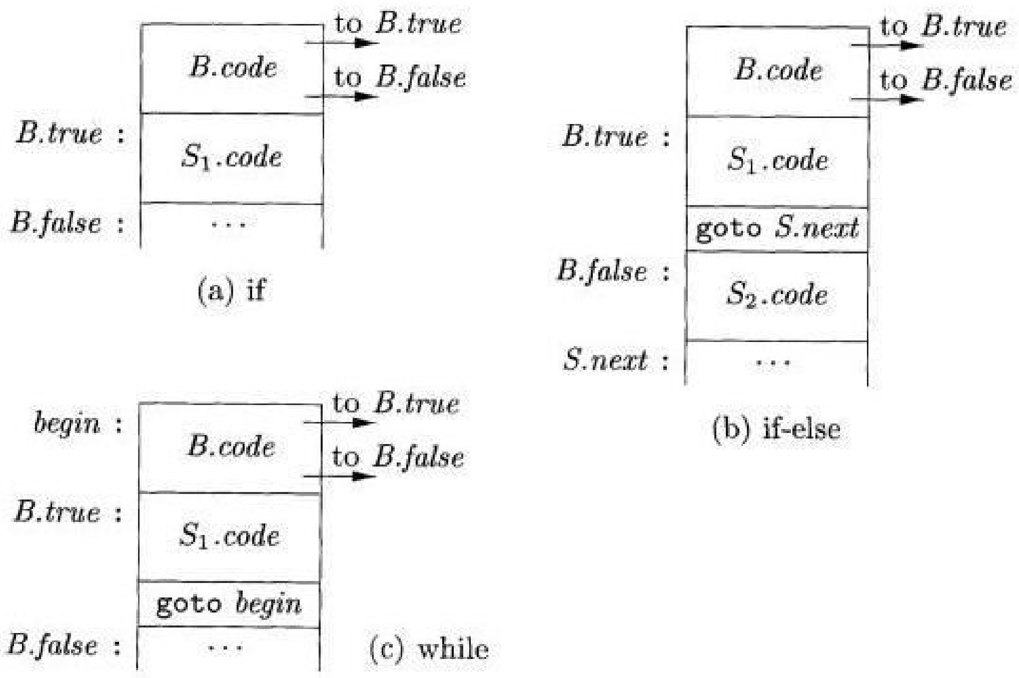


Figure 4.9: Code for if-, if-else-, and while-statements

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.*

4.5 Backpatching:

It is the process of filling up the unspecified labels.

The following functions are required for backpatching:

1. *makelist(i)* creates a new list containing only *i,* an index into the array of instructions; *makelist* returns a pointer to the newly created list.
2. *merge(*p_1*,*p_2*)* concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.
3. *backpatch(p,I)*inserts *i* as the target label for each of the instructions on the list pointed to by *p.*

We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing. A marker non terminal *M* in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

$B \rightarrow B_1 \;||\; M\; B_2 \mid B_1 \;\&\&\; M\; B_2 \mid\; !\, B_1 \mid (\, B_1 \,) \;||\; E1\ rel\ E2 \mid true \mid false$

$M \rightarrow \varepsilon$

The translation scheme is in below figure 4.10.

1) $B \rightarrow B_1 \mid\mid M\ B_2$ { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

2) $B \rightarrow B_1\ \&\&\ M\ B_2$ { $backpatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$; }

3) $B \rightarrow\ !\ B_1$ { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

4) $B \rightarrow (\ B_1\ )$ { $B.truelist = B_1.truelist$;
$B.falselist = B_1.falselist$; }

5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$emit($'`if`' $E_1.addr$ **rel**$.op$ $E_2.addr$ '`goto _`'$)$;
$emit($'`goto _`'$)$; }

6) $B \rightarrow$ **true** { $B.truelist = makelist(nextinstr)$;
$emit($'`goto _`'$)$; }

7) $B \rightarrow$ **false** { $B.falselist = makelist(nextinstr)$;
$emit($'`goto _`'$)$; }

8) $M \rightarrow \epsilon$ { $M.instr = nextinstr$; }

Figure 4.10: Translation scheme for boolean expressions

Consider semantic action (1) for the production B → B_1 || M B_2. If *B_1* is true, then *B* is also true, so the jumps on *B_1.truelist* become part of *B.truelist.* If *B_1* is false, however, we must next test *B_2,* so the target for the jumps *B_1.falselist* must be the beginning of the code generated for *B_2*. This target is obtained using the marker nonterminal *M.* That nonterminal produces, as a synthesized attribute *M.instr,* the index of the next instruction, just before *B_2* code starts being generated.
To obtain that instruction index, we associate with the production M→ ε the semantic action

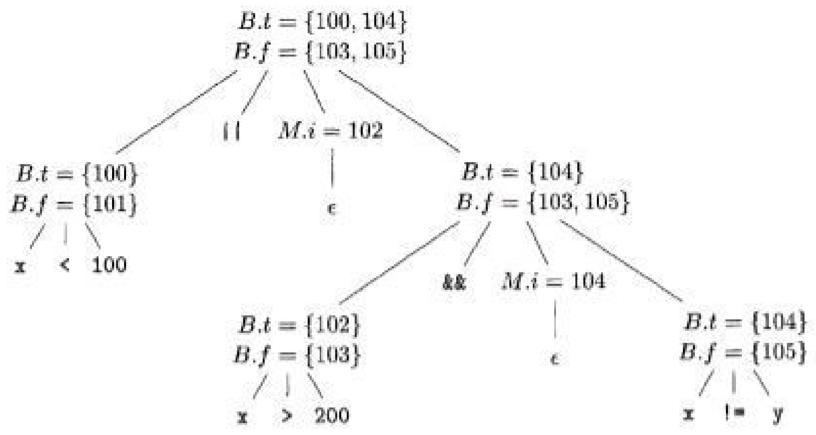
{ M.instr = nextinstr; }

The variable *nextinstr* holds the index of the next instruction to follow. This value will be backpatched onto the *B_1.falselist* (i.e., each instruction on the list *B_1.falselist* will receive *M.instr* as its target label) when we have seen the remainder of the production B→ B_1 || M B_2.

Semantic action (2) for B → B_1 && M B_2 is similar to (1). Action (3) for *B* →B_1 swaps the true and false lists. Action (4) ignores parentheses. For simplicity, semantic action (5) generates two instructions, a conditional goto and an unconditional one. Neither has its target filled in. These instructions are put on new lists, pointed to by *B.truelist* and *B.falselist,* respectively.

Example: x<100 || x>200 && x!=y
Solution: Generate TAC for the given expression

```
100     if x<100 goto ______
101     goto
102     if x>200 goto ______
103     goto
104     if x!=y goto ________
105     goto
```



```
100     if x<100 goto___
101     goto 102
102     if x>200 goto 104
103     goto ___
104     if x!=y goto ___
105     goto ____
```

4.6. Intermediate Code for Procedures

Suppose that a is an array of integers, and that f is a function. Then the assighmet statement
 n = f(a[i]);
can be translated into three-address code as follows:

1) t1 = i * 4
2) t2 = a [t1]
3) param t2
4) t3 = call f, 1
5) n = t3

The procedure are important and frequently used programming construct so it is necessary for a compiler to generate good code for procedure calls and returns.