MODULE 5

SYNTAX DIRECTED TRANSLATIONS(*Chapter 5*)

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - ✓ may generate intermediate codes
  - ✓ may put information into the symbol table
  - ✓ may perform type checking
  - ✓ may issue error messages
  - ✓ may perform some other activities
  - ✓ in fact, they may perform almost any activities.
- An attribute may hold almost anything.
  - ✓ a string, a number, a memory location, a complex record.

  Attributes for expressions:
  type of value: int, float, double, char, string,…
  type of construct: variable, constant, operations, …
  Attributes for constants: values
  Attributes for variables: name, scope
  Attributes for operations: operands, operator,…
- When we associate semantic rules with productions, we use two notations:
  - ✓ **Syntax-Directed Definitions**
  - ✓ **Translation Schemes**
- **Syntax-Directed Definitions:**
  - Gives high-level specification for translations
  - Hides many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - Indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, these schemes give a little information about implementation details.

5.1 Syntax directed definitions (SDD) :

- Syntax directed definition is used for specifying translations for programming language constructs.
- Syntax directed definition is a generalization of a context free grammar in which each grammar symbol is associated with a set of attributes and each production is associated with a set of semantic rules.
- 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*. If we implement the nodes of the parse tree by records or objects, then the attributes of *X* can be implemented by data fields in the records that represent the nodes for *X*. Attributes may be of any kind: numbers, types, table references, or strings, for instance.

- **Inherited and Synthesized Attributes:**
  There are two kinds of attributes for nonterminals:

1. ***synthesized attribute:***
   - An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the children of the node.
   - Suppose for a nonterminal *A*, at a parse-tree, node *N* is defined by a semantic rule associated with the production (with A as its head) at *N*. A synthesized attribute at node *N* is defined only in terms of attribute values at the children of *N* and at *N* itself.
   - Ex: Consider the following SDD:

Productions	Semantic Rules
E ->E_1+T	E.val = E_1.val+T.val
E -> T	E.val = T.val

     Here, *val* is a synthesized attribute associated with E and T
   - *Semantic rules* set up dependencies between attributes which can be represented by a ***dependency graph***.
   - An SDD that involves only synthesized attributes is called ***S-attributed***. 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 S-attributed SDD can be implemented naturally in conjunction with an LR parser.
   - 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.

2. ***Inherited Attribute:***
   - An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the parent and | or siblings of that node.
   - Suppose for a nonterminal B at a parse-tree, node *N* is defined by a semantic rule associated with 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.
   - Ex: Consider the following SDD:

Productions	Semantic Rules
A ->+BC	C.inh = B.val+A.inh
	C.syn = C.inh

     Here, *inh* is a inherited attribute associated with A and C

   NOTE: 1. Terminals can have synthesized attributes, but not inherited attributes.
   2. Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

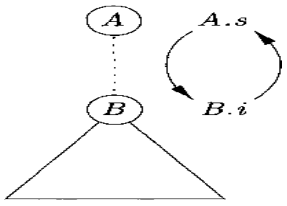
- **Evaluating an SDD at the Nodes of a Parse Tree:**

➢ The rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodesof the parse tree.
➢ A parse tree, showing the value(s) of its attribute(s) is called an ***annotated parse tree.***

- Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.
- If all attributes are synthesized, then we must evaluate the attributes at all of the children of a node before we can evaluate the attribute at the node itself.
- With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a post-order traversal of the 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.
  For instance, consider nonterminals *A* and *B,* with synthesized and inherited attributes *A.s* and *B.i*, respectively, along with the production and rules

Productions	Semantic Rules
A->B	*A.s = B.i;*
	B.i = A.s + l

it is impossible to evaluate either *A.s* at a node *N* or *B.i* at the child of *N* without first evaluating the other. This is called as circular dependency as shown below:



1. **Construct annotated parse tree for the string 3*5+4n for the expression grammar SDD using bottom-up approach.**

   SDD is given below:

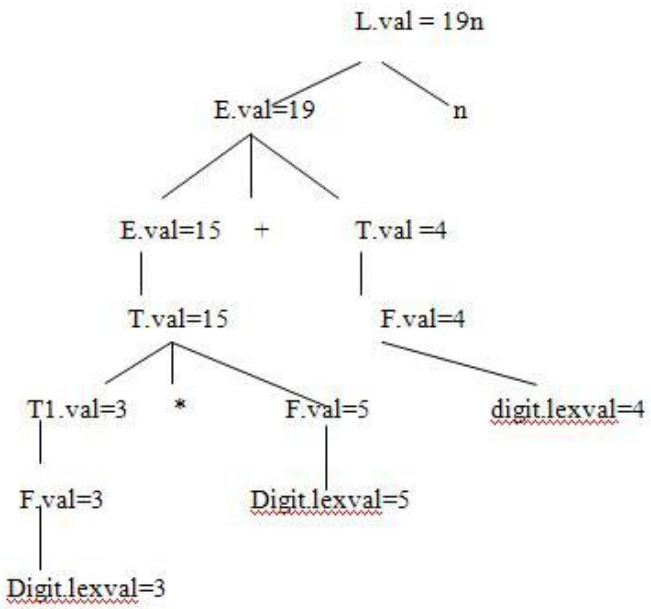
Production	**Semantic Rules**
L -> En	L.val = E.val
E -> E_1+T	E.val = E_1.val + T.val
E -> T	E.val = T.val
T -> T_1*F	T.val = T_1.val * F.val
T -> F	T.val = F.val
F -> (E)	F.val = E.val
F -> digit	F.val = digit.lexval

   Here, each of the nonterminals has a single synthesized attribute called, *val* .

   lexval represents an integer value returned by the lexical analyzer.

   In the first production, *n* can be ignored as it implies to new line.
   Here, E_1 and T_1 denotes E and T respectively, a subscript 1 is used just to show their usage in the body of the production.

   The annotated parse tree for the string 3*5+4n is shown below.



Exercise :

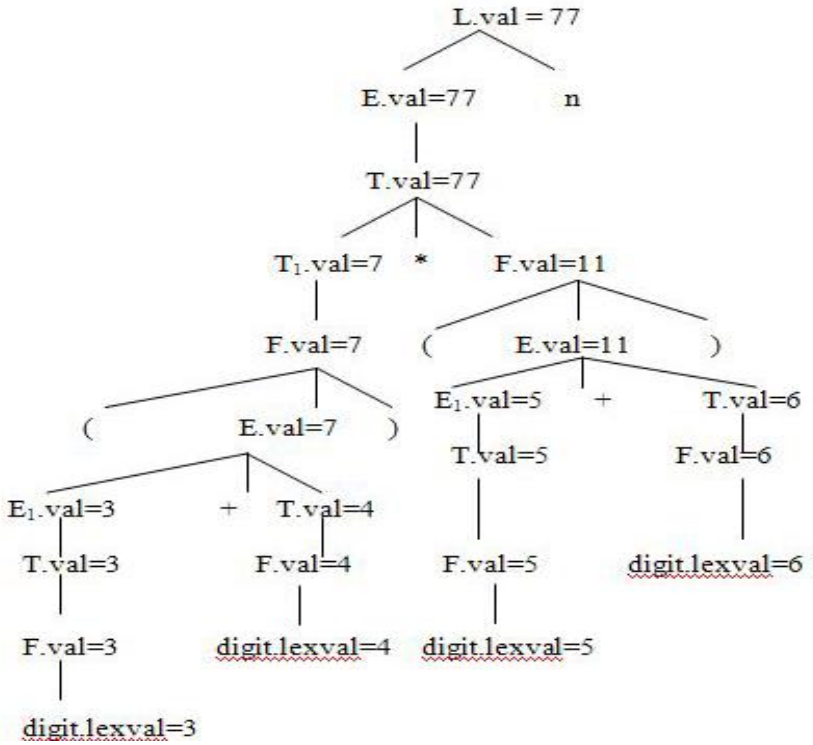
For the SDD of the previous problem, give annotated parse tree for the Following expressions:

a) (3+4)*(5+6)n

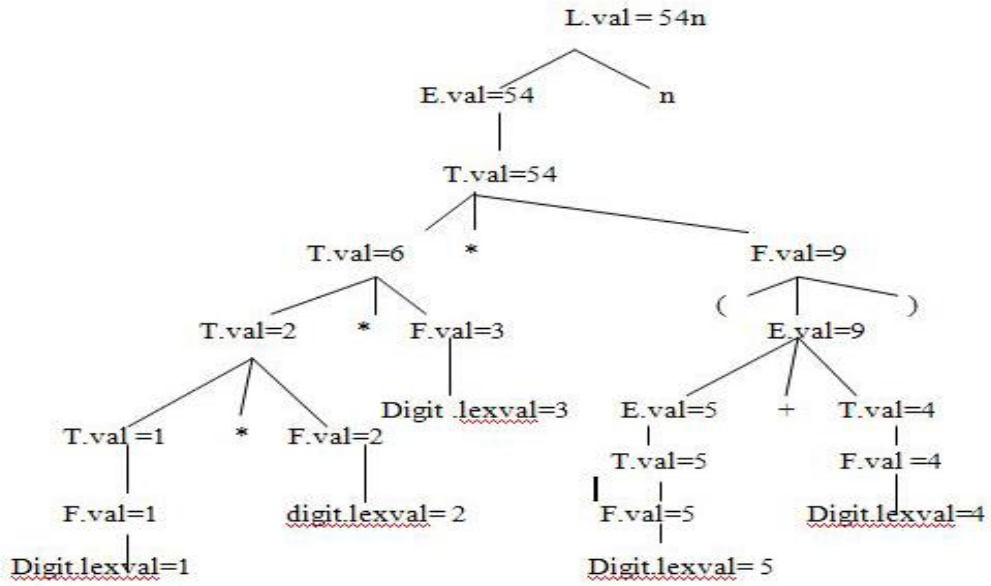
b) 1*2*3*(4+5)n

c) (9+8*(7+6)+5)*4n

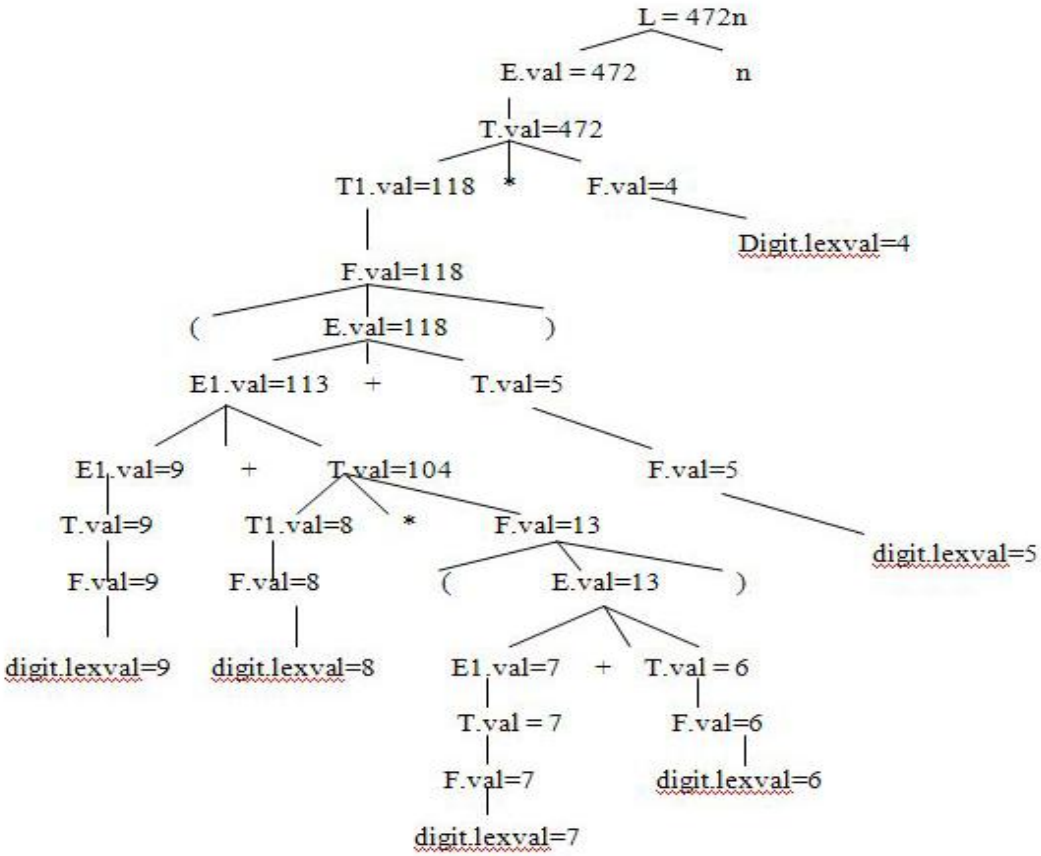
a)



b)


L.val = 54n
E.val=54
n
T.val=54
T.val=6
*
F.val=9
T.val=2
*
F.val=3
(
)
E.val=9
T.val =1
*
F.val=2
Digit .lexval=3
E.val=5
+
T.val=4
F.val=1
digit.lexval= 2
T.val=5
F.val =4
Digit.lexval=1
F.val=5
Digit.lexval=4
Digit.lexval= 5

c)


L = 472n
E.val = 472
n
T.val=472
T1.val=118
*
F.val=4
Digit.lexval=4
F.val=118
(
E.val=118
)
E1.val=113
+
T.val=5
E1.val=9
+
T.val=104
F.val=5
T.val=9
T1.val=8
*
F.val=13
digit.lexval=5
F.val=9
F.val=8
(
E.val=13
)
digit.lexval=9
digit.lexval=8
E1.val=7
+
T.val = 6
T.val = 7
F.val=6
F.val=7
digit.lexval=6
digit.lexval=7

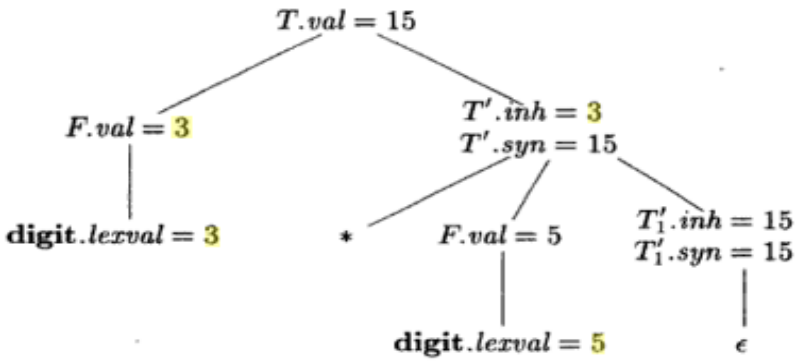
2. Construct annotated parse tree for the string 3*5 for the following grammar, suitable for top-down parsing.

T -> T*F | F
F -> digit

After removing left recursion from the above grammar, an SDD based on a grammar suitable for top-down parsing obtained as:

Production	Semantic Rules
1) *T-> FT'*	*T'.inh = F.val*
	T.val = T'.syn
2) *T'->*FT$_1$'*	*T$_1$'.inh = T'.inh x F.val*
	T'.syn = T$_1$'.syn
3) *T' -> e*	*T'.syn = T'.inh*
4) *F* **-> digit**	*F.val =* **digit.***lexval*

- Each of the nonterminals *T* and *F* has a synthesized attribute *val;*
- The terminal **digit** has a synthesized attribute *lexval.*
- 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'->*FT$_1$'* inherits the left operand of * in the production body.
- Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.
- the annotated parse tree for 3 * 5 is



5.2 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.

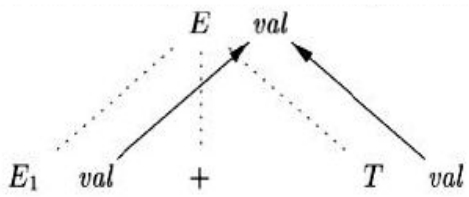
- **Dependency Graphs:**
  - A *dependency graph* depicts the flow of information among the attribute instances in a particular parse tree.
  - An edge from one attribute instance to another means that the value of the first is needed to compute the second.
  - Edges express constraints implied by the semantic rules. In more detail:
    1) 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.*
    2) 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.

       Ex: Consider the following production and rule:

Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

       a portion of the dependency graph for every parse tree in which this production is used looks like:



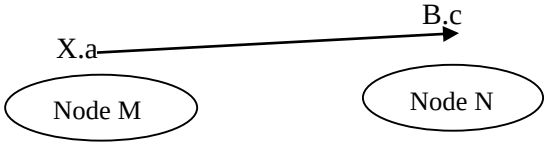
       Notations:

       Dotted lines: edges of parse tree

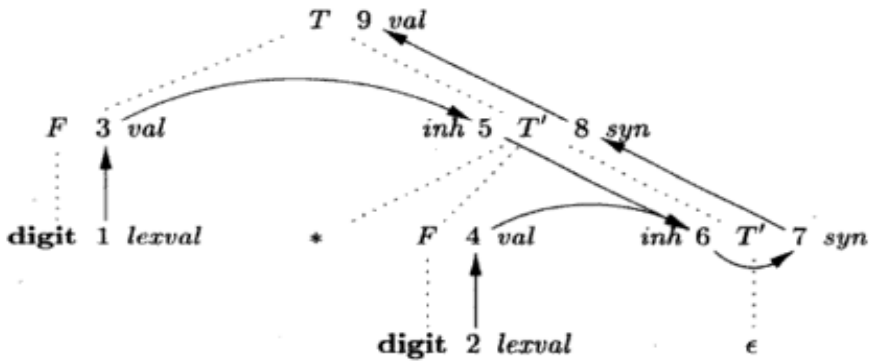
       Solid lines: edges of dependency graph

    3) 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. M* could be either the parent or a sibling of *N.*



Ex: A complete dependency graph for the annotated parse tree drawn for 3*5 is:



- The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes
- 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*
- 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 *T'*.
- The edge to node 7 from 6 is due to the semantic rule *T'.syn = T'.inh.*
- 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.*

- **<u>Ordering the Evaluation of Attributes:</u>**

- 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 N_1, N_2,... ,N_k 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:
  - Find a node with no edge entering.
  - Make this node the first in the topological order, remove it from the dependency graph
  - repeat the process on the remaining nodes.

- Ex: The dependency graph drawn for the string 3*5 has the following topological sequence:
  - 1,2,3,4,5,6,7,8,9
  - 1,3,5,2,4,6,7,8,9.

- **Classes of SDD:**

1) S-Attributed Definitions:

- An SDD is *S-attributed* if every attribute is synthesized.
- Ex:

Production	Semantic Rules
L -> En	L.val = E.val
E -> E_1+T	E.val = E_1.val + T.val
E -> T	E.val = T.val
T -> T_1*F	T.val = T_1.val * F.val
T -> F	T.val = F.val
F -> (E)	F.val = E.val
F -> digit	F.val = digit.lexval

- 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 simple to evaluate the attributes by performing a *post order 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;
}
```

2) 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 $A \rightarrow X_1 X_2 X_3 \bullet \bullet \bullet X_n$, and that there is an inherited attribute $Xi.a$ 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 $X_1, X_2, \ldots X_{i-1}$ 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_i .

- Ex: The following SDD is L-attributed.

Production	Semantic Rules
1) *T-> FT'*	***T'.inh = F.val***
	T.val = T'.syn
2) *T'->*FT_1'*	***T_1'.inh = T'.inh x F.val***
	T'.syn = T_1'.syn
3) *T' -> e*	*T'.syn = T'.inh*
4) *F* **-> digit**	*F.val =* **digit.***lexval*

The semantic rules written in bold uses information "from above or from the left," as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed.

- Ex: Consider the following SDD. Check whether it is S-attributed or L-attributed

Production	Semantic Rules
A -> BC	*A.s = B.b;*
	B.i = f(C.c,A.s)

  - The first rule, *A.s = B.b,* defines an S-attribute
  - 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.
  - While attributes at siblings in a parse tree may be used in L-attributed SDD's, they must be to the left of the symbol whose attribute is being defined.
  - Hence the above SDD is neither S-attributed nor L-attributed.

- **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.
  - We shall control side effects in SDD's in one of the following ways:
    1) Permit incidental side effects that do not constrain attribute evaluation. i.e., 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.

       Ex: Instead of the rule *L.val = E.val,* which saves the result in the synthesized attribute *L.val,* consider:

Production	Semantic Rules
1) *L-> E*n	*print(E.val)*

Semantic rules that are executed for their side effects, such as *print(E.val),* will be treated as the definitions of dummy synthesized attributes associated with the head of the production.

The modified SDD produces the same translation under any topological sort, since the print statement is executed at the end, after the result is computed into *E.val.*

2) 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.

Ex : The following SDD takes a simple declaration *D* consisting of a basic type *T* followed by a list *L* of identifiers. *T* can be int or float. For each identifier on the list, the type is entered into the symbol-table entry for the identifier. We assume that entering the type for one identifier does not affect the symbol-table entry for any other identifier. Thus, entries can be updated in any order.

Production	Semantic Rules
1) *D->TL*	*L.inh = T.type*
2) *T* -> int	*T.type* = integer
3) *T* -> float	*T.type* = float
4) *L* -> L_1 , id	L_1*.inh = L.inh*
	addType(id. entry, L.inh)
5) L -> id	*addType(id. entry, L.inh)*

D represents a declaration.

T has one attribute, *T.type,* which is the type in the declaration *D.*

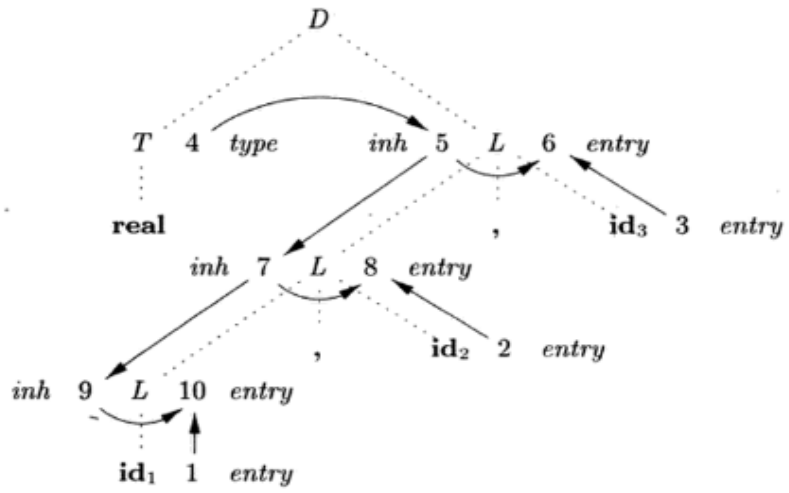
L also has one attribute, which we call *inh* to emphasize that it is an inherited attribute. The purpose of *L.inh* is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol-table entries.

addType is a function called with two arguments:

1. **id.***entry,* a lexical value that points to a symbol-table object, and
2. *L.inh,* the type being assigned to every identifier on the list.

function *addType* properly installs the type *L.inh* as the type of the represented identifier.

A dependency graph for the input string ***float id1 , id2 , id3*** appears below:



Numbers 1 through 10 represent the nodes of the dependency graph. Nodes 1, 2, and 3 represent the attribute *entry* associated with each of the leaves labeled **id.**
Nodes 6, 8, and 10 are the dummy attributes that represent the application of the function *addType* to a type and one of these *entry* values.
Node 4 represents the attribute *T.type*, and is actually where attribute evaluation begins.
This type is then passed to nodes 5, 7, and 9 representing *L.inh* associated with each of the occurrences of the nonterminal *L*.

5.3 Applications of Syntax-Directed Translation:

- The main application of syntax-directed translation techniques is in the construction of syntax trees.
- Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules

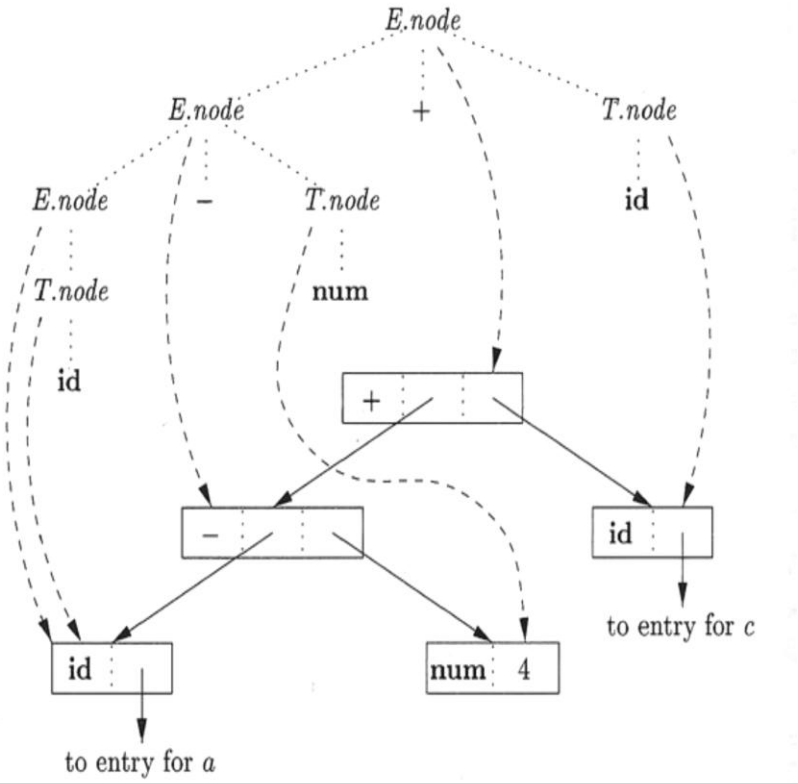
- **Construction of Syntax Trees:**
  - Each node in a syntax tree represents a construct
  - The children of the node represent the meaningful components of the construct.
  - The nodes of a syntax tree can be implemented by, objects with a suitable number of fields.
    - Each object will have an *op* 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.
        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*: ***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

- **Ex 1 :** Consider the following S-attributed definition ,which constructs syntax trees for a simple expression grammar involving the binary operators + and -. As usual, these operators are at the same precedence level and are left associative.

	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	E.node = **new** Node ('+', E_1 .node, T.node)
2)	$E \rightarrow E_1 - T$	E.node = **new** Node ('-', E_1 .node, T.node)
3)	$E \rightarrow T$	E. node= T.node
4)	$T \rightarrow (E)$	T.node = E.node
5)	$T \rightarrow$ **id**	T.node= **new** Leaf (**id**, **id**.entry)
6)	$T \rightarrow$ **num**	T.node= **new** Leaf (**num**, **num**.val)

(note: this grammar is also referred as desk calculator grammar)

  - All nonterminals have one synthesized attribute *node,* which represents a node of the syntax tree.
  - When 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 subexpressions.
  - The second production has a similar rule.
  - For production 3 & 4, no node is created, parentheses are used only for grouping;
  - The last two T-productions have a single terminal on the right.
  - Constructor *Leaf* is used to create a suitable node, which becomes the value of *T.node.*
  - The syntax tree for the input ***a* — 4 + *c*** using the above SDD is shown below:



Dotted Lines: Edges of parse tree

Dashed Lines: Represents the values of *E.node* and *T-node; E*ach line points to the appropriate syntax-tree node.

Solid Lines: Edges of Syntax-tree

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 followed are:

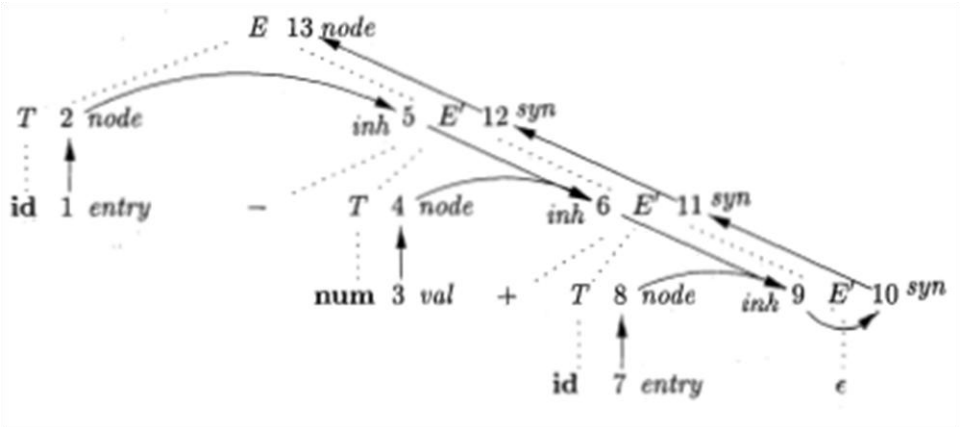
1) p_1 = new Leaf(id, entry-a);
2) p_2 = new Leaf (num, 4);

3) p_3 = new Node('-',p_1,p_2);
4) p_4 = new Leaf'(id, entry-c);
5) p_5 = new Node('+',p_3,p_4);

- **Ex 2:** Consider the following L-attributed definition suitable for top-down parsing. Draw dependency graph for the input: ***a-4+c*** .

Production	Semantic Rules
1) E->TE'	E.node = E'.syn E'.inh = T.node
2) E'->+TE_1'	E_1'.inh = new Node('+', E'.inh, T.node) E'.syn = E_1'.syn
3) E' ->-T E_1'	E_1'.inh = new Node('-', E'.inh, T.node) E'.syn = E_1'.syn
4) E' -> ε	E'.syn = E'.inh
5) T->(E)	T.node = E.node
6) T -> id	T.node = new Leaf (id, id. entry)
7) T -> num	T.node = new Leaf(num, num.val)

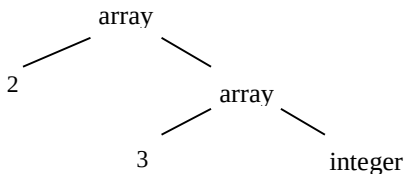
  - Here, *the* idea is to build a syntax tree for *x* + *y* by passing *x* as an inherited attribute, since *x* and + *y* appear in different subtrees. Nonterminal *E'* is the counterpart of nonterminal



- **<u>The Structure of a Type:</u>**

  - Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input; attributes can then be used to carry information from one part of the parse tree to another.

- The next example shows how a mismatch in structure can be due to the design of the language, and not due to constraints imposed by the parsing method.
- **Ex 3:**
  In C, the type **int** [2][3] can be read as, "array of 2 arrays of 3 integers."
  The corresponding type expression *array(2,* array(3, *integer))* is represented in the following tree.

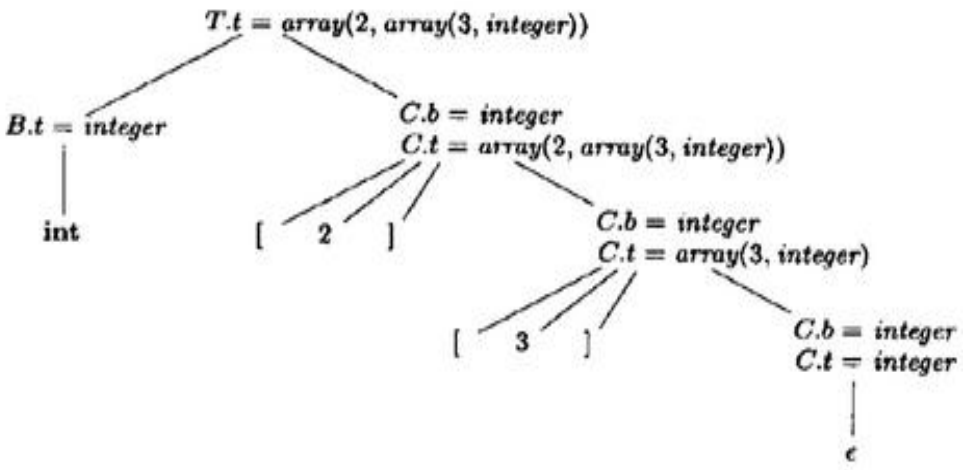


  The operator *array* takes two parameters, a number and a type. If types are represented by trees, then this operator returns a tree node labeled *array* with two children for a number and a type.

- The following SDD generates either a basic type or an array type.

Production	Semantic Rules
1)T -> BC	T.t = C.t C.b = B.t
2)B -> int	B.t = integer
3)B -> float	B.t = float
4)C -> [num] C_1	C.t = array(num.val, C_1.t) C_1.b = C.b
5)C -> ε	C.t = C.b

  - Nonterminal *B* generates one of the basic types **int** and **float.**
  - *T* generates a basic type when *T* derives *B C* and *C* derives e. Otherwise, *C* generates array components consisting of a sequence of integers, each integer surrounded by brackets.
  - The nonterminals *B* and *T* have a synthesized attribute *t* representing a type.
  - The nonterminal *C* has two attributes: an inherited attribute *b* and a synthesized attribute *t.*
  - The inherited *b* attributes pass a basic type down the tree, and the synthesized *t* attributes accumulate the result.
  - An annotated parse tree for the input string **int [** 2] [3] is shown here:



INTERMEDIATE CODE GENERATION *(Chapter 6)*

- In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. The organization is shown below:

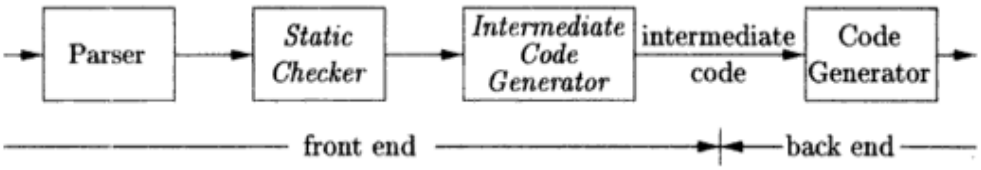
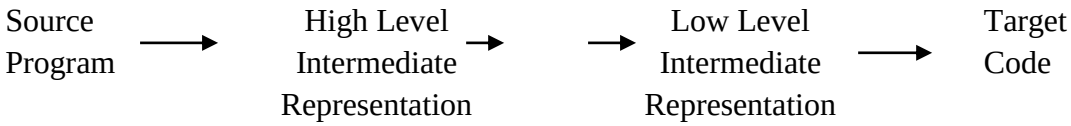


Fig: Logical structure of a compiler front end

- Static checking includes *type checking,* which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing.
  Ex: Static checking assures that break statements in C are enclosed within a while, for or switch statement; otherwise an error message is issued.
- In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations



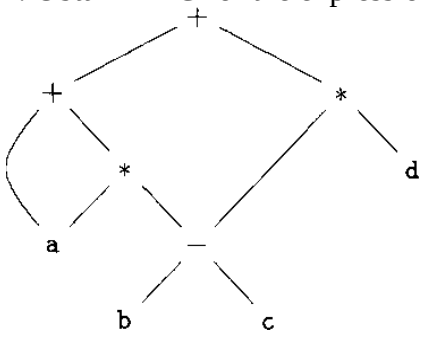
  - Some of the intermediate representations are: *syntax trees, three address code, postfix notations*
  - High-level representations are close to the source language and low-level representations are close to the target machine.
  - *Syntax trees* are high level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking.

- A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection.
- *Three-address code* can range from high- to low-level, depending on the choice of operators.

6.1 Variants of Syntax Trees:

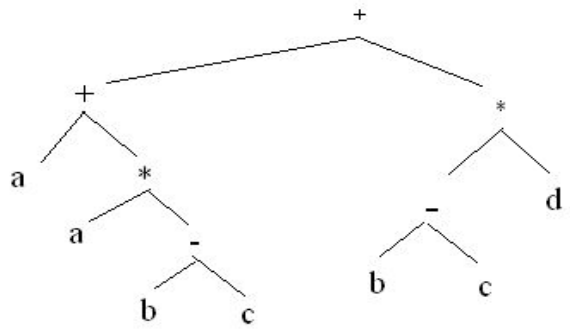
- Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct.
- A directed acyclic graph (*DAG)* for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression.

- **Directed Acyclic Graphs for Expressions:**
  - A DAG has leaves corresponding to atomic operands and interior codes corresponding to operators.
  - The difference is that a node *N* in a DAG has more than one parent if *N* represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.
  - Thus, a DAG not only represents expressions, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.
  - Ex 1: Obtain DAG for the expression ***a + a * (b - c) + (b - c) * d***

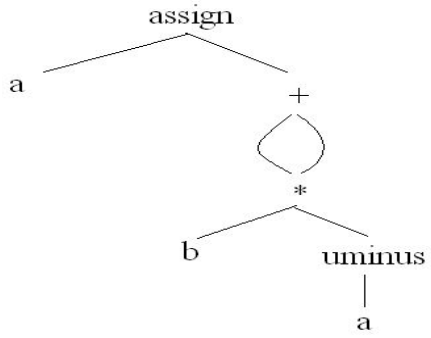


The leaf for *a* has two parents, because *a* appears twice in the expression.
The two occurrences of the common subexpression b-c are represented by one node, the node labeled -.

The Syntax tree for the same expression looks like:



- Ex 2: Obtain DAG for the expression: a=b*-c + b *-c



- SDD can be used to construct either syntax trees or DAG's.
- Ex: The following SDD constructs DAG:

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

fig: SDD1

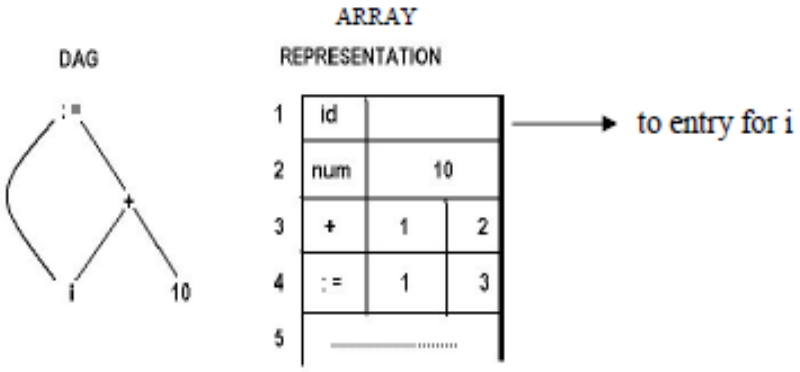
- Before creating a new node, the functions *Leaf* and *Node* first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. Otherwise, it creates a new node.

- The sequence of steps shown below constructs the DAG for the expression ***a + a * (b - c) + (b - c) * d***

1) $p_1 = Leaf($**id**, *entry-a*$)$
2) $p_2 = Leaf($**id**, *entry-a*$) = p_1$
3) $p_3 = Leaf($**id**, *entry-b*$)$
4) $p_4 = Leaf($**id**, *entry-c*$)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf($**id**, *entry-b*$) = p_3$
9) $p_9 = Leaf($**id**, *entry-c*$) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf($**id**, *entry-d*$)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

The Value-Number Method for Constructing DAG's:

- Often, the nodes of a syntax tree or DAG are stored in an array of records.
- 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.
- Leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant), and interior nodes have two additional fields indicating the left and right children.
- Ex: Nodes of a DAG for $i = i + 10$ allocated in an array is shown here:



- In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer called as the ***value number*** for the node or for the expression represented by the node.
- For instance, in the above figure, the node labeled **+** has value number 3, and its left and right children have value numbers 1 and 2, respectively.

- Suppose that nodes are stored in an array, and each node is referred to by its *value number*. Let the *signature* of an interior node be the triple <*op,l,r*>, where *op* is the label, *l* its left child's value number, and *r* its right child's value number. A unary operator may be assumed to have $r = 0$.

  Algorithm: The value-number method for constructing the nodes of a DAG.
  ***INPUT*:** Label *op*, node *l*, and node *r*.
  ***OUTPUT*:** The value number of a node in the array with signature <*op,l,r*>
  ***METHOD*:** Search the array for a node *M* with label *op*, left child *l*, and right child *r*. If there is such a node, return the value number of *M*. If not, create in the array a new node *N* with label *op*, left child *l*, and right child *r*, and return its value number.

- In the above algorithm, searching the entire array is expensive, A more efficient approach is to use a *hash table*, in which the nodes are put into "buckets," each of which typically will have only a few nodes.
- To construct a hash table for the nodes of a DAG, a *hash function h* is used that computes the index of the bucket for a signature *(op, l, r)*
- The bucket index *h(op,l, r)* is computed deterministically from *op, l,* and *r*, so that we may repeat the calculation and always get to the same bucket index for node *(op, l, r).*

- The buckets can be implemented as linked lists, as shown in the following figure.
- An array, indexed by hash value, holds the *bucket headers,* each of which points to the first cell of a list.
- Within the linked list for a bucket, each cell holds the value number of one of the nodes that hash to that bucket. That is, node *(op,l,r)* can be found on the list whose header is at index *h(op,l,r)* of the array.
- Thus, given the input node *op, l,* and r, we compute the bucket index *h(op,l,r)* and search the list of cells in this bucket for the given input node.
- For each value number *v* found in a cell, we must check whether the signature *(op,l,r)* of the input node matches the node with value number *v* in the list of cells.
- If match is found, return *v.* If match is not found, create a new cell, add it to the list of cells for bucket index *h(op, l, r),* and return the value number in that new cell.

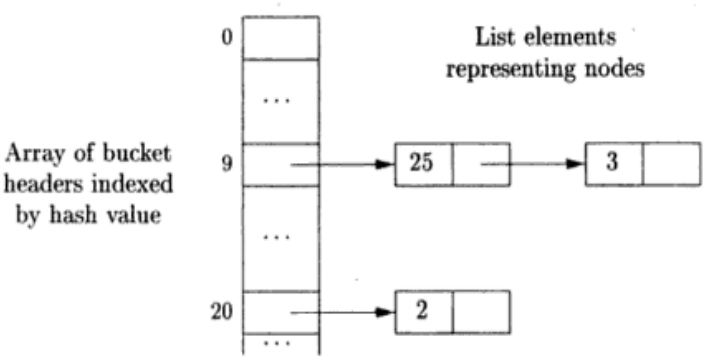
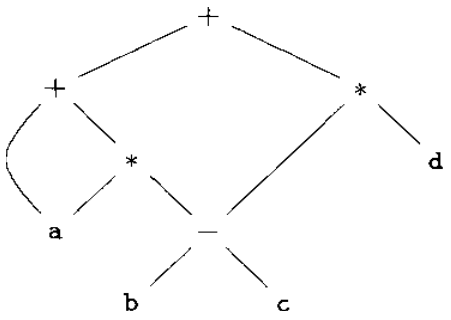


Fig: Data structure for searching buckets

6.2 Three Address Code(TAC):

- In three-address code, there is at most one operator on the right side of an Instruction.
- TAC can range from high- to low-level, depending on the choice of operators.
- In general, it is a statement containing at most 3 addresses or operands.
  The general form is ***x := y op z***, where "op" is an operator, x is the result, and y and z are operands.
  In particular, **x**, **y**, **z** are variables, constants, or "temporaries".
- Ex: x + y * z can be translated as
  t1 = y * z
  t2 = x + t1
  where t1 & t2 are compiler–generated temporary names.

- TAC is a linearized representation of a binary syntax tree. Explicit names correspond to interior nodes of the graph. E.g. for a looping statement, syntax tree represents components of the statement, whereas three-address code contains labels and jump instructions to represent the flow-of-control as in machine language.
- A DAG for the expression ***a + a * (b - c) + (b - c) * d*** and its corresponding three address code is shown below:



t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

- **<u>Addresses and Instructions:</u>**

o An address in a TAC can be one of the following:
1. *A **name***: Each name is a symbol table index. For convenience, we write the names as the identifier.
2. *A **constant**.*
3. *A **compiler-generated temporary***. Each time a temporary address is needed, the compiler generates temporary names.
   It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed.

o The common three-address instruction forms:
1. **Assignment instructions** :
   x = y op z, where *op* is a binary arithmetic or logical operation, and *x, y,* and *z* are addresses.

2. **Assignment**s :
   x = op y, where *op* is a unary operation.
   unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number.

3. **Copy instructions***:*
   x = y, where *x* is assigned the value of *y.*

4. An **unconditional jump** :
   goto *L*. , L is the symbolic label of an instruction.

5. **Conditional jumps**:
   - **if *x* goto *L*** : Execute the instruction with label ***L* next** if *x* is true
   - **if False *x* goto *L*** **:** Execute the instruction with label ***L* next** if *x* is false
     Otherwise, the following three-address instruction in sequence is executed next, as usual.
6. **Conditional jumps**:
   if *x relop y* goto *L*, which apply a relational operator (<, ==, >=, etc.) to *x* and *y*, Execute the instruction with label ***L* next** if *x* stands in relation *relop* to *y.*

If not, the three-address instruction following ***if x relop y goto L*** is executed next, in sequence.

7. **Procedure calls**:

   call p(x_1, …, x_n)

   where x_1, …, x_n are the parameters

   the above instruction generates the following code:

   param x_1
   param x_2
   …
   param x_n
   call p, n

   here *n* stands for no. of parameters

   Function calls :

   y= p(x_1, …, x_n)
   y = call p,n

   return instruction:

   return y

8. **Indexed copy instructions:**
   - **x = y[i]** sets *x* to the value in the location *i* memory units beyond location *y*.
   - **x[i] = y** sets the contents of the location *i* units beyond *x* to the value of *y*.

9. **Address and pointer assignments:**
   - **x = &y** sets the value of x to be the location (address) of y.
   - **x = *y** presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.
   - ***x = y** sets the value of the object pointed to by x to the value of y.

- Example: Given the statement **do i = i+1; while (a[i] < v); ,** the TAC can be written as below in two ways, using either symbolic labels or position number of instructions for labels.

```
L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L
```

(a) Symbolic labels.

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Position numbers.

Three Address Code representation in data structure:

- The three-address instructions does not specify the representation of the instructions in a data structure.
- In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands.
- Three such representations are "**quadruples**," "**triples**," and "**indirect triples**."

(1) **Quadruple** (or just *"quad!')*

- It has four fields, which we call *op, arg1, arg2,* and *result.*
- The *op* field contains an internal code for the operator with some exceptions to this rule:
  1. Instructions with unary operators like *x* = minus *y* or *x = y* do not use *arg2.* For a copy statement like *x = y, op* is =, while for most other operations, the assignment operator is implied.
  2. Operators like param use neither *arg2* nor *result.*
  3. Conditional and unconditional jumps put the target label in *result.*

- Ex:**:** Three-address code for the assignment a = b * - c + b * - c and its corresponding quadruples is :

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

(a) Three-address code

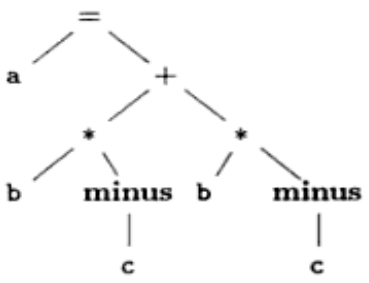
	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

(2) Triples:

- A *triple* has only three fields: *op, arg1,* and *arg2.*
- Using triples, we refer to the result of an operation *x op y* by its position, rather than by an explicit temporary name.
- Triples are equivalent to signatures in DAG and syntax trees.
- Triples and DAGs are equivalent representations only for expressions; they are not equivalent for control flow.
- Ternary operations like *x[i] = y* requires two entries in the triple structure, similarly for x = y[i].
- A 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 instructions that use *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.

- Ex: Syntax tree for the assignment a = b * - c + b * - c and its corresponding triple representation is :



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

(3) **Indirect triples**

- This consists of a listing of pointers to triples, rather than a listing of triples themselves. For example, let us use an array *instruction* to list pointers
- With indirect triples, an optimizing compiler can move an instruction by reordering the *instruction* list, without affecting the triples themselves.
- Ex: Indirect triples representation of three-address code :

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Static Single-Assignment Form(SSA):

- SSA is an intermediate representation that facilitates certain code optimizations.
- Two distinctive aspects distinguish SSA from three-address code.
  - ❖ All assignments in SSA are to variables with distinct names; hence the term *static single-assignment.*

Ex:

p = a + b	$p_1 = a + b$
q = p - c	$q_1 = p_1 - c$
p = q * d	$p_2 = q_1 * d$
p = e - p	$p_3 = e - p_2$
q = p + q	$q_2= p_3 + q_1$

(a) Three-address code (b) Static single-assignment form

- SSA uses a notational convention called the **φ-function** to combine the two definitions of the same variable.
  For example,
  if (flag) x = -1; else x = 1;
  y = x * a;
  Has two control-flow paths in which the variable x gets defined.

  Using φ-function it can be written as
  if (flag) x_1 = -1; else x_2 = 1;
  x_3 = φ(x_1,x_2);
  y = x_3 * a;

  The φ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment statement containing the φ-function.

CODE GENERATION(*Chapter 8*)

- The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in the following figure.

- Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the IR into IR from which more efficient code can be generated. In general, the code optimization and code-generation phases of a compiler, often referred to as the *back end*, may make multiple passes over the IR before generating the target program.

- A code generator has three primary tasks: instruction selection, register allocation and assignment, and instruction ordering.
  - Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements.
  - Register allocation and assignment involves deciding what values to keep in which registers.
  - Instruction ordering involves deciding in what order to schedule the execution of instructions.

8.1 Issues in the Design of a Code Generator:

1. **Input to the Code Generator**
   - The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

- The many choices for the IR include
  - Three-address representations such as quadruples, triples, indirect triples
  - Virtual machine representations such as byte codes and stack-machine code
  - Linear representations such as postfix notation
  - Graphical representations such as syntax trees and DAG's. Many

2. The Target Program

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator
- The most common target-machine architectures are
  RISC (reduced instruction set computer),
  CISC (complex instruction set computer), and
  Stack based.
- A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
- CISC machine typically has few registers, two-address instructions, and a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.
- In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers.
- However, stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM). The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers. The interpreter provides software compatibility across multiple platforms
- To overcome the high performance penalty of interpretation, *just-in-time* (JIT) Java compilers have been created. These J IT compilers translate bytecodes during run time to the native hardware instruction set of the target machine.
- Producing an **absolute machine-language program** as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.
- Producing a **relocatable machine-language program** (often called an *object module)* as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.
  - If we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module.
  - If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.
- **Producing an assembly-language program as output** makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code.

3. Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as
  - The level of the IR
  - The nature of the instruction-set architecture
  - The desired quality of the generated code.

- **Level of the IR:**
  - If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement by-statement code generation, often produces poor code that needs further optimization.
  - If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.

- **The nature of the instruction set**

  It has a strong effect on the difficulty of instruction selection.
  - **Uniformity and completeness of the instruction set** are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. On some machines, for example, floating-point operations are done using separate registers.
  - **Instruction speeds and machine idioms** are other important factors.
    - For example, every three-address statement of the form x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD R0, y        // R0 = y          (load y into register R0)
ADD R0, R0, z   // R0 = R0 + z     (add z to R0)
ST x, R0        // x = R0          (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

a = b + c
d = a + e

Would be translated into

```
LD R0, b        // R0 = b
ADD R0, R0, c   // R0 = R0 + c
ST a, R0        // a = R0
LD R0, a        // R0 = a
ADD R0, R0, e   // R0 = R0 + e
ST d, R0        // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if **a** is not subsequently used.

- **The quality of the generated code**: is usually determined by its **speed and size**.
  - On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations.

- For example, if the target machine has an "increment" instruction **(INC),** then the three-address statement **a = a +** 1 may be implemented more efficiently by the single instruction **INC a,** rather than by the following sequence

```
LD R0, a          // R0 = a
ADD R0, R0, #1    // R0 = R0 + 1
ST a, R0          // a = R0
```

4. Register Allocation

- The use of registers is often subdivided into two subproblems:
  1. ***Register allocation***, during which we select the set of variables that will reside in registers at each point in the program.
  2. ***Register assignment***, during which we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult
  - Example: Certain machines require ***register-pairs*** (an even and next odd numbered register) for some operands and results. For example, on some machines, integer multiplication and integer division involve register pairs.
  - The multiplication instruction is of the form ***M x, y***
    Where *x*, the multiplicand, is the even register of an even/odd register pair and *y*, the multiplier, is the odd register. The product occupies the entire even/odd register pair.
  - The division instruction is of the form ***D x, y***
    Where the dividend occupies an even/odd register pair whose even register is *x*; the divisor is *y*. After division, the even register holds the remainder and the odd register the quotient.
  - Now, consider the two three-address code sequences below:

```
t = a + b          t = a + b
t = t * c          t = t + c
t = t / d          t = t / d
   (a)                (b)
```

Two three-address code sequences:

```
L Rl,a             L R0, a
A Rl,b             A R0, b
M R0,c             A R0, c
D R0,d             SRDA R0, 32
ST Rl,t            D R0, d
                   ST Rl, t
   (a)                (b)
```

5. Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem.

8.2 The Target Language

- **A Simple Target Machine Model**

  - Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with *n* general-purpose registers, R0, R1,..., Rn - 1. We shall use a very limited set of instructions and assume that all operands are integers. Most instructions consists of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction.

  - We assume the following kinds of instructions are available:

    - ***Load*** operations: The instruction **LD *dst, addr*** loads the value in location *addr* into location *dst* This instruction denotes the assignment *dst = addr.*
      Ex: **LD *r, x*** which loads the value in location *x* into register r.
      LD *r1, r2* is a *register-to-register copy* in which the contents of register *r2* are copied into register *r1*.

    - ***Store*** operations: The instruction ***ST x, r*** stores the value in register *r* into the location *x.* This instruction denotes the assignment *x = r.*

    - ***Computation*** operations of the form ***OP dst, src1, src2***, where *OP* is an operator like ADD or SUB, and *dst, src1,* and *src2* are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP to* the values in locations *src1* and *src2,* and place the result of this operation in location *dst.*
      Ex: **SUB *r1, r2, r3*** computes *r1 = r2 - r3.*

    - ***Unconditional jumps:*** The instruction **BR *L*** causes control to branch to the machine instruction with label *L.* (BR stands for *branch.)*

    - ***Conditional jumps*** of the form ***Bcond* r, *L,*** where *r* is a register, *L* is a label, and *cond* stands for any of the common tests on values in the register r.
      Ex: **BLTZ r, *L*** causes a jump to label *L* if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

  - We assume our target machine has a variety of addressing modes:

    - In instructions, a location can be a variable name *x* referring to the memory location that is reserved for *x* (that is, the *l*-value of *x).*

    - A location can also be an indexed address of the form ***a(r),*** where *a* is a variable and r is a register. The memory location denoted by *a(r)* is computed by taking the *l*-value of *a* and adding to it the value in register *r*.
      Ex: The instruction **LD Rl, a(R2)** has the effect of setting
      Rl = *contents* (a + *contents* (R2))

This addressing mode is useful for accessing arrays, where *a* is the base address of the array and *r* holds the number of bytes past that address we wish to go to reach one of the elements of array *a*.

- A memory location can be an integer indexed by a register.
  Ex: LD Rl, 100(R2) // Rl = *contents(100 +conten*ts(R2))
  This feature is useful for pointers

- We also allow two indirect addressing modes:
  - **r* means the memory location found in the location represented by the contents of register r
  - *100(r) means the memory location found in the location obtained by adding 100 to the contents of r.
  - For example, **LD Rl, *100(R2)** //Rl = *contents(contents(100 + conten*ts(R2)))

- Immediate constant addressing mode: The constant is prefixed by #.
  Ex: **LD Rl, #100** // loads the integer 100 into register Rl,
  ADD Rl, Rl, #100 // adds the integer 100 into register Rl.

- **Examples :**

1. The three-address statement **x = y - z** can be implemented by the machine instructions:
   ```
   LD Rl, y        // Rl = y
   LD R2, z        // R2 = z
   SUB Rl, Rl, R2  // Rl = Rl - R2
   ST x, Rl        // x = Rl
   ```

2. Suppose *a* is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of *a* are indexed starting at 0. We may execute the three-address instruction **b = a[i]** by the machine instructions:
   ```
   LD Rl, i        // Rl = i
   MUL Rl, Rl, 8   // Rl = Rl * 8
   LD R2, a(Rl)    // R2 = contents(a + contents(Rl))
   ST b, R2        // b = R2
   ```

3. The assignment into the array *a* represented by three-address instruction **a[j] = c** is implemented by:
   ```
   LD Rl, c        // Rl = c
   LD R2, j        // R2 = j
   MUL R2, R2, 8   // R2 = R2 * 8
   ST a(R2), Rl    // contents(a + contents(R2)) = Rl
   ```

4. To implement a simple pointer indirection, such as **x = *p**, we can use :
   ```
   LD Rl, p        // Rl = p
   LD R2, 0(R1)    // R2 = contents(0 + contents(Rl))
   ST x, R2        // x = R2
   ```

5. The assignment through a pointer ***p = y** is implemented by:

```
LD Rl, p          // Rl = p
LD R2, y          // R2 = y
ST 0(R1), R2      // contents(0 + contents(Rl)) = R2
```

6. Conditional-jump three-address instruction like **if x < y goto L :**

```
LD R1, x          // Rl = x
LD R2, y          // R2 = y
SUB Rl, Rl , R2   // Rl = Rl - R2
BLTZ Rl , M       // if Rl < 0 jump to M
```

   Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L.

- **Program and Instruction Costs**

  - We often associate a cost with compiling and running a program.

  - Depending on what aspect of a program we are interested in optimizing, some common cost measures are
    - The length of compilation time and the size
    - Running time and Power consumption of the target program.

  - For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction.

  - Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.

  - Some examples:
    - The instruction LD R0, Rl copies the contents of register Rl into register R0. This instruction has a cost of one because no additional memory words are required.

    - The instruction LD R0, M loads the contents of memory location M into register R0. The cost is two since the address of memory location M is in the word following the instruction.

    - The instruction LD Rl, *100(R2) loads into register Rl the value given by *contents(contents(100 + contents(R2)))*. The cost is three because the constant 100 is stored in the word following the instruction.

  - Therefore, the cost of a target-language program on a given input is the sum of costs of the individual instructions executed when the program is run on that input.