MODULE-5

Syntax-Directed Definitions

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. 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. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

Inherited and synthesized attributes

There are two types of attributes for non-terminals namely:

1. Synthesized attribute
2. Inherited attribute

synthesized attribute: 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.

Inherited attribute: At a parse-tree node N is defined by a semantic rule associated with the production at the parent of N . An inherited attribute at node N is defined only in terms of attribute values at N 's parent, N itself, and N 's siblings.

Example1:An Syntax-directed definition of a simple desk calculator

Production	Semantic Rules
L→En	L.val=E.val
E→E1+T	E.val=E1.val+T.val
E→T	E.val=T.val
T→T*F	T.val=T1.val*F.val
T→F	T.val=F.val
F→(E)	F.val=E.val
F→digit	F.val=digit.lexval

An SDD that involves only synthesized attributes is called S-attributed; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the non-terminal 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.

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 nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.

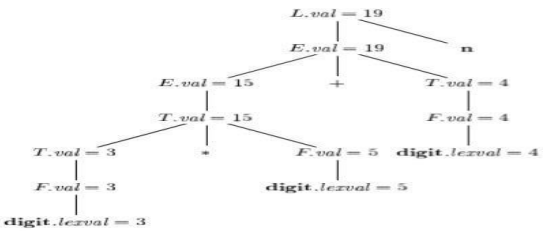


Fig1: An Annotated Parse tree for 3*5+4n

Example2: An SDD based on a grammar suitable for top-down parsing

Production	Semantic Rules
T→FT^1	T^1.inh=F.val T.val= T^1.syn
T^1→*F T^1	T^1.inh = T^1.inh *F.val T^1= T^1 .syn
T^1→€	T^1.syn= T^1.inh
F→digit	F.val=digit.lexval

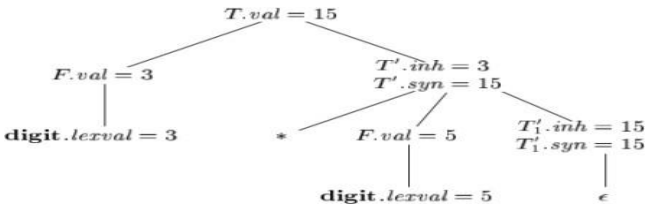


Fig2: Annotated Parse tree for 3*5

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

Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
E→E1 + T	E:val = E1: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. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid. 2

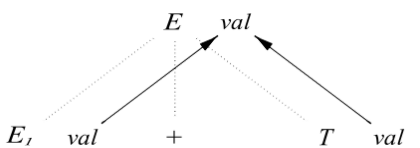
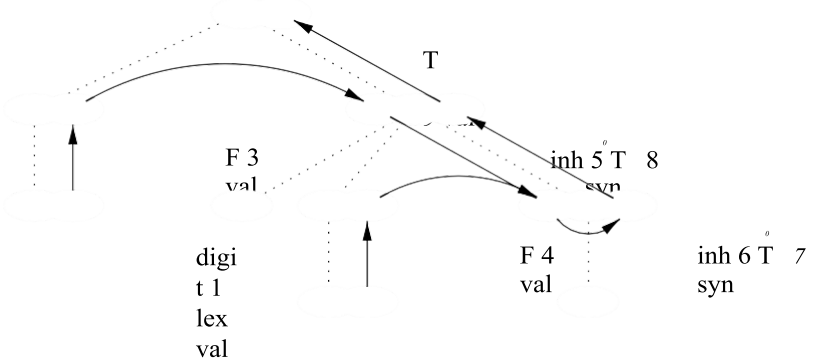


Fig3:E.val is synthesized from E1.val and T.val Example:
An example of a complete dependency graph appears in Fig.
The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 5.5.



digit 2 lexval

Fig4: Dependency graph for the annotated parse tree 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^1.inh associated with each of the occurrences of nonterminal T^1. The edge to 5 from 3 is due to the rule T^1.inh = F:val, which de nes T^1.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^1.inh and from node4 for F.val, because these values are multiplied to evaluate the attribute inh at node6.

Nodes 7 and 8 represent the synthesized attribute syn associated with the occurrences of T^1. The edge to node 7 from 6 is due to the semantic rule T^1.syn = T^1:inh associated with production 3 in Fig. The edge to node8 from 7 is due to a semantic rule associated with production2. Finally, node 9represents the attribute T.val. The edge to 9 from 8 is due to the semantic rule, T.val = T^1.syn, associated with production 1.

Ordering the Evaluation of Attributes

- The dependency graph characterizes the possible orders in which we can evalu- ate 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 N_i to N_j, then i < j. Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.
- If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree.
- If there are no cycles, however, then there is always at least one topological sort. To see why, since there are no cycles, we can surely add a node with no edge entering.
- For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yielding a cycle. Make this node the first in the topological order,

remove it from the dependency graph,and repeat the process on the remaining nodes.

S-Attributed Definitions

- An SDD is S-attributed if every attribute is synthesized.
- When an SDD is S-attributed, we can evaluate its attributes in any bottom- up order of the nodes of the parse tree.
- It is 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.
- Semantic rules will be declared only in the left side of the production.

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 Synthesized, or Inherited, but with the rules limited as follows.
- Suppose that there is a production $A \rightarrow X_1 X_2 \ldots X_n$, 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 occur-

rences of symbols X_1; X_2;: :: ; X_{i-1} located to the left of X

(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.

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

PRODUCTION	SEMANTIC RULE
T→F T^1	T^1.inh = F.val
T^1→F T^1	T^1:inh = T^1.inh F.val

- The first of these rules defines the inherited attribute T^1.inh using only F.val, and F appears to the left of T^1 in the production body, as required.
- The second rule defines T^1.inh using the inherited attribute T^1.inh associated with the head, and F.val, where F appears to the left of T^1 in the production body.
- In each of these cases, the rules use information from above or from the left," as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed

Example 5.9 : Any SDD containing the following production and rules cannot be L-attributed:

Production	Semantic Rules
A→B C	A.s = B.b;
	B.i = f(C.c; A.s)

- The first rule, A.s = B.b, is a legitimate rule in either an S-attributed or L-attributed SDD. It defines a synthesized attribute A.s in terms of an attribute at a child .
- 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.

Semantic Rules with Controlled Side Ef f ects

- 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 it allows any evaluation order consistent with the dependency graph.
- Translation schemes impose left- to-right evaluation and allow semantic actions to contain any program fragment; translation schemes are discussed in Section.
- We shall control side effects in SDD's in one of the following ways:
  a) Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a correct" translation, where correct" depends on the application.
  b) 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.

- As an example of an incidental side effect, let us modify the desk calculator of Example to print a result. Instead of the rule L.val = E.val, which saves the result in the synthesized attribute L.val, consider:

PRODUCTION	SEMANTIC RULE
L→En	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.

Example: The SDD in Fig. 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. This SDD does not check whether an identifier is declared more than once; it can be modified to do so.

Production	Semantic Rules
D→TL	L.inh = T.type
T→int	T.type = integer
T→float	T.type = float
L→L,id	L_1.inh = L.inh addType(id.entry; L.inh)
L→id	addType(id.entry; L.inh)

Fig5: Syntax-directed definition for simple type declarations

- A function addType is called with two arguments:

a) id.entry, a lexical value that points to a symbol-table object, and

b) L:inh, the type being assigned to every identifier on the list.

- A dependency graph for the input string float id_1,id_2,id_3 appears in Fig. 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.

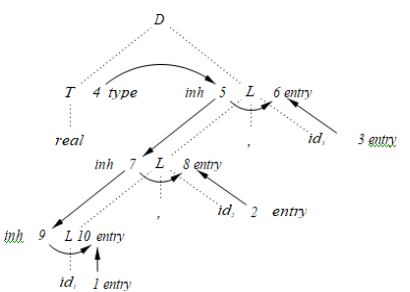


Fig6: Dependency graph for a declaration float id,id,id

Applications of Syntax-DirectedTranslation

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 $E_1 + E_2$ has label + and

two children representing the subexpressions E_1 and E_2.

- The nodes of a syntax tree is implemented as 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 elds 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_k.

Example: The S-attributed definition in Fig.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.

Every time the fi rst production E→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.

Production	Semantic Rules
E→E1+T	E.node = new Node("+"; E_1.node;T.node)
E→ E_1T	E:node = new Node("" ;E_1.node;T.node)
E→T	E.node=T.node
T→(E)	T.node=E.node
T→id	T.node = new Leaf (id; id.entry)
T→num	T.node = new Leaf (num; num.val)

Fig: Constructing syntax trees for simple expressions

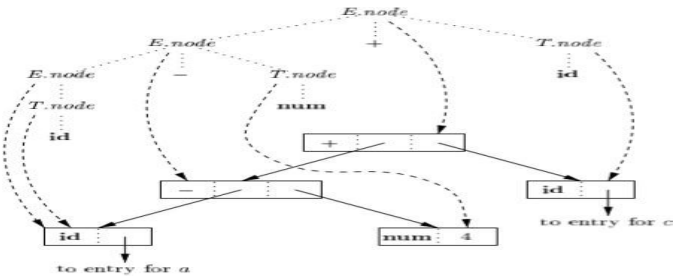


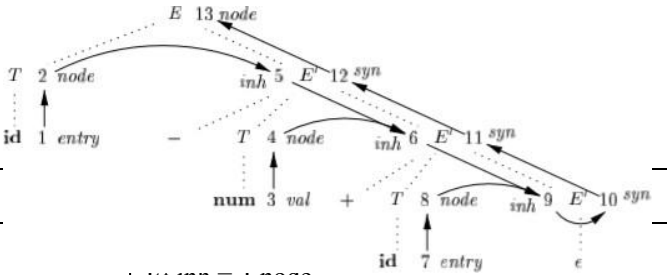
Fig7: Syntax tree for a 4+ c

1) p_1 = new Leaf (id; entry-a);
2) p_2 = new Leaf (num;
4); 3) p_3 = new Node(^{0} ^{0};
p_1; p_2);
4) p_4 = new Leaf (id; entry-c);
5) p_5 = new Node(^{0}+^{0}; p_3; p_4);

Fig8: Steps in the construction of the syntax tree for a 4+ c

1

Fig9: Constructing syntax trees during top-down parsing



Production	
E →TE	E'.inn = T.node
E^1 →+ T E^1_1	E^1 .inh = new Node("+"; E^1.inh;T.node) E^1.syn = E^1.syn
E^1 →-T E^1_1	E^1 .inh = new Node("-"; E^1.inh;T.node) E^1.syn = E^1.syn
E^1 →€	E^1.syn = E^1.inh
T→(E)	T.node = E.node
T→id	T.node = new Leaf (id; id.entry)
T →num	T.node = new Leaf (num; num.val)

Fig10: Dependency graph for a 4+ c, with the SDD of Fig. 5.13

2) Structure of the type

- 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

Example: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 by the tree in Fig.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.

Fig11: Type expression for int[2][3]

With the SDD in Fig. 5.16, nonterminal T generates either a basic type or an array type. Nonterminal B generates one of the basic types int and oat. T generates a basic type when T derives BC and C derives . Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.

Production	Semantic Rules
T→BC	T.t = C.t C.b = B.t
B→int	B.t=integer
B→Float	B.t=Float
C→[num]C1	C.t = array(num.val; C1.t)
C→€	C.t = C.b

Fig12: T generates either a basic type or an array type

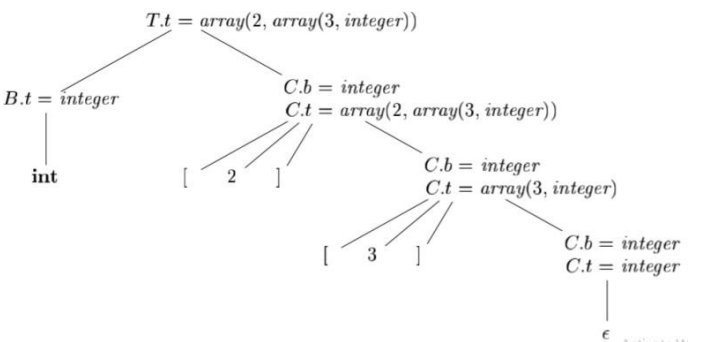


Fig13: Syntax-directed translation of arraytypes

INTERMEDIATE CODE GENERATION

VARIANTS OF SYNTAX TREE

- 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 (hereafter called a DAG) for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression.

Directed Acyclic Graphs for Expressions

- DAG construction is similar to syntax tree for an expression.
- A DAG has leaves corresponding to atomic operands and interior nodes 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.
- Example 6.1 : Figure 6.3 shows the 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 - . That node has two parents, representing its two uses in the subexpressions 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 subexpression b-c. 2

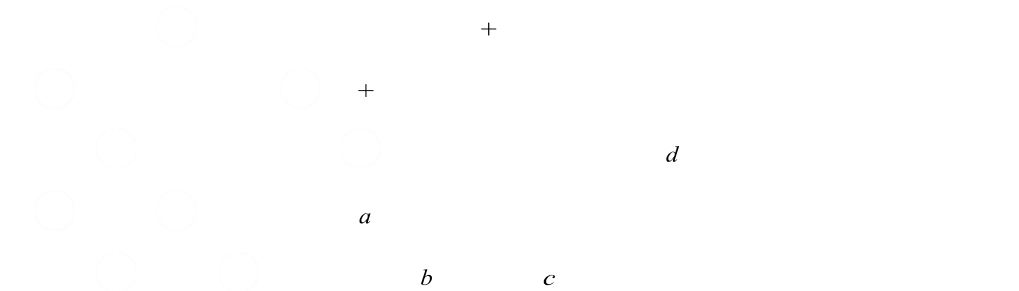


Figure 6.3: Dag for the expression a +a* (b- c) + (b - c)*d

PRODUCTION	SEMANTIC RULES
• E-> E_1 + T	E:node = new Node('+'; E_1:node;T:node)
• E->E_1- T	E:node = new Node('-';E_1:node;T:node)
• E-> T	E:node = T:node
• T->(E)	T:node = E:node
• T-> id	T:node = new Leaf (id; id:entry)
• T->num	T:node = new Leaf (num; num:val)

It will construct a DAG if, before creating a new node, these functions s ho u l d check whether an identical node already exists.

- If a previously created identical node exists, the existing node is returned.
- Before constructing a new node, Node(op; left; right), it is required to check whether there is already a node with label op, and children left and right, in that order. If so, Node returns the existing node; otherwise, it creates a new node.

Example 6.2

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

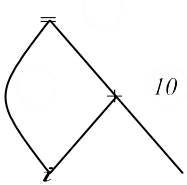
- p_1 = Leaf (id; entry-a)
- p_2 = Leaf (id; entry-a) = p_1
- p_3 = Leaf (id; entry-b)
- p4 = Leaf (id; entry-c)
- p5 = Node('-'; p3; p4)
- p6 = Node('*'; p1; p5)
- p7 = Node('+'; p1; p6)
- p8 = Leaf (id; entry-b) = p3
- p9 = Leaf (id; entry-c) = p4 10)
- p10 = Node('-'; p3; p4) = p5
- p11 = Leaf (id; entry-d)
- p12 = Node('*'; p5; p11)
- p13 = Node('+'; p7; p12)

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

- When the call to Leaf (id; entry-a) is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$.
- Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_9 = p_4$).
- Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_5$.

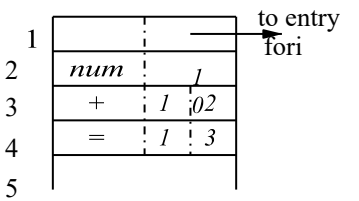
The Value-Number Method for Constructing DAG's

- The nodes of a syntax tree or DAG are stored in an array of records.
- Eachrow of the arrayrepresents one record, and therefore one node.
- In each record, the first field is an operation code, indicating the label of the node.
- In Fig. 6.6(b), leaves have one additional field, which holds the lexical value and interior nodes have two additional fields indicating the left and right



children.

id



(d) DAG *(b) Array*

Figure 6.6: Nodes of a DAG for i = i + 10 allocated in an array

- Here, nodes are referred by giving the integer index of the record for that node within the array. It is called the value number for the node or for the expression represented by the node.
- In Fig. 6.6, the node labeled + has value number 3, and its left and right children have value numbers 1 and 2, respectively.
- If stored in an appropriate data structure, value numbers help us construct expression DAG's efficiently;
- Suppose that nodes are stored in an array, as in Fig. 6.6, 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 6.3 : 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. 2

- An 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 required which computes the index of the bucket for a signature (op; l; r),

in a way that distributes the signatures across buckets, so that it is unlikely that any one bucket will get much more than a fair share of the nodes.

- 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 in Fig. 6.7. 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 hop; l; ri can be found on the list whose header is at index h(op; l; r) of the array.

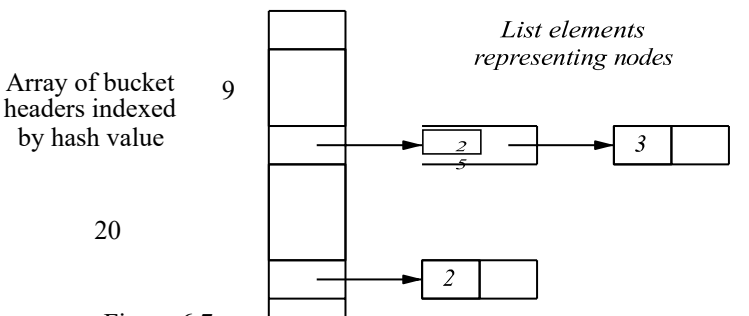


Figure 6.7: Data structure for searching buckets

- Thus, given the input node op, l, and r, the bucket index h(op; l; r) can be computed and searches the list of cells in this bucket for the given input node.
- Typically, there are enough buckets so that no list has more than a few cells.
- If all the cells within a bucket are observed, and for each value number v found in a cell, check whether the signature (op; l; r) of the input node matches the node with value number v in the list of cells (as in Fig. 6.7).
- If a match is found, we return v. If there is no match, then no such node can exist in any other bucket, hence 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.

Three Address Code

- In three-address code, there is at most one operator on the right side of an

instruction; that is, no built-up arithmetic expressions are permitted.

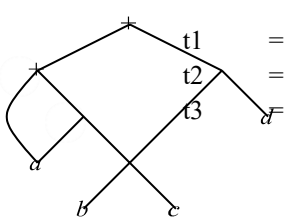
- Thus a source-language expression like x+y*z might be translated into the sequence of three-address instructions

$$t1 = y * z \quad t2 = x + t1$$

where t1 and t2 are compiler-generated temporary names.

- This unraveling of multi-operator arithmetic expressions and of nested ow-of-control statements makes three-address code desirable for target-code generationandoptimization.
- The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.

Example 6.4 : Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. The DAG in Fig. 6.3 is repeated in Fig. 6.8, together with a corresponding three-address code sequence.




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

(a) DAG (b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

Addresses and Instructions

- Three-address code is built from two concepts: addresses and instructions.

- Alternatively, three-address code can be implemented using records with fields for the addresses;
- An address can be one of the following:

    - **A name**. source-program names 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.** 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.

- A symbolic label represents the index of a three-address instruction in the sequence of instructions.
- Actual indexes can be substituted for the labels, either by making a separate pass or by \backpatching,"
- Here is a list of the common three-address instruction forms:

1. Assignment instructions of the form x = y op z, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form x = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a fl oating- point number.

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

4. An unconditional jump goto L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form if x goto L and ifFalse x goto L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as if x relop y goto L, which apply a relational operator (<, ==, >=, etc.) to x and y, and 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 and returns are implemented using the following instructions: param x for parameters; call p, n and y = call p, n for procedure and function calls, respectively; and return y, where y, representing a returned value, is optional. Their typical use is as the sequence of three- address instructions

```
param x1
param x2

param xn
call  p, n
```

   generated as part of a call of the procedure $p(x_1; x_2 ;::: ; x_n)$. The in-teger n, indicating the number of actual parameters in \call p, n," is not redundant because calls can be nested.

8. Indexed copy instructions of the form x = y[i] and x[i]= y. The instruc- tion x = y[i] sets x to the value in the location i memory units beyond location y. The instruction x[i]= y sets the contents of the location i units beyond x to the value of y.

9. Address and pointer assignments of the form x = & y, x = * y, and * x = y. The instruction x = & y sets the r-value of x to be the location (l- value) of y.[2] Presumably y is a name, perhaps a temporary, that denotes an expression with an l-value such as A[i][j], and x is a pointer name or temporary. In the instruction x = * y, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location. Finally, * x = y sets the r-value of the object pointed to by x to the r-value of y.

Example 6.5 : Consider the statement

```
do i = i+1; while (a[i] < v);
```

Two possible translations of this statement are shown in Fig. 6.9. The translation

in Fig. 6.9(a) uses a symbolic label L, attached to the first instruction.

Quadruples

- The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these 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 called \quadruples," \triples," and \indirect triples."
- A quadruple (or just \quad") has four fields: op, arg1, arg2, and result. The op field contains an internal code for the operator, i.e.,the three-address instruction x = y + z is represented by placing + in op, y in arg1, z in arg2, and x in result. The following are some exceptions to thisrule:
    - Instructions with unary operators like x = minus y or x = y do not use arg2. Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.
    - Operators like param use neither arg2 nor result.
    - Conditional and unconditional jumps put the target label in result.
- Example 6.6 : Three-address code for the assignment a = b* -c+b*-c; appears in Fig. 6.10(a).

The translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100. In both translations, the last instruction isa conditional jump to the fi rst instruction. The multiplication i* 8 is appropriate for an array of elements that each take 8 units of space.

```
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  goto100
```

(b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

- The choice of allowable operators is an important issue in the design of an intermediate form.
- The operator set clearly must be rich enough to implement the operations in the source language.
- Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine.
- Note that the unary-minus \three-address" statement has only two addresses, as does the copy statement a = t5.
  The quadruples in Fig. 6.10(b) implement the three-address code in (a).

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

        =
        a
        =
```

		arg2	result
			1
0			
1			
2			
3			
4			
5			

(a) Three-address code (b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

- Temporary names can either by entered into the symbol table like programmer-defined names, or they can be implemented as objects of a class Temp with its own methods.

Triples

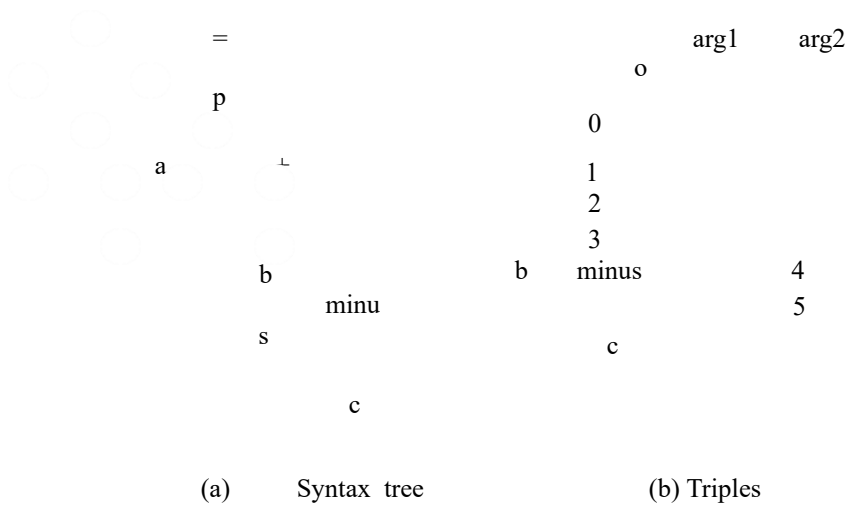
- A triple has only three fields: op, arg1, and arg2.
- Note that the result field is used primarily for temporary names.
- Using triples, we refer to the result of an operation x op y by its position, rather

than by an explicit temporary name.

- Thus, instead of the temporary t1 in Fig. 6.10(b), a triple representation would refer to position (0).
- Parenthesized numbers represent pointers into the triple structure itself. Triples are equivalent to signatures in Algorithm 6.3.
- Hence, the DAG and triple representations of expressions are equivalent. The equivalence ends with expressions, since syntax-tree variants and three-address code represent control ow quite differently.

Example 6.7 : The syntax tree and triples in Fig. 6.11 correspond to the three- address code and quadruples in Fig. 6.10. In the triple representation in Fig. 6.11(b), the copy statement a = t5 is encoded in the triple representation by placing a in the arg1 field and (4) in the arg2 field .

- A ternary operation like x[i] = y requires two entries in the triple structure; for example, we can put x and i in one triple and y in the next. Similarly, x = y[i] can implemented by treating it as if it were the two instructions t = y[i] and x = t, where t is a compiler-generated temporary.



(a) Syntax tree (b) Triples

Figure 6.11: Representations of a = b*-c +b*-c;

- Note that the temporary t does not actually appear in a triple, since temporary values are referred to by their position in the triple structure.
- A benefit of quadruples over triples is that we can 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.
- Indirect triples consist 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 to triples in the desired order. Then, the triples in Fig. 6.11(b) might be represented as in Fig. 6.12.

- With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves. When implemented in Java, an array of instruction objects is analogous to an indi- rect triple representation, since Java treats the array elements as references to objects.

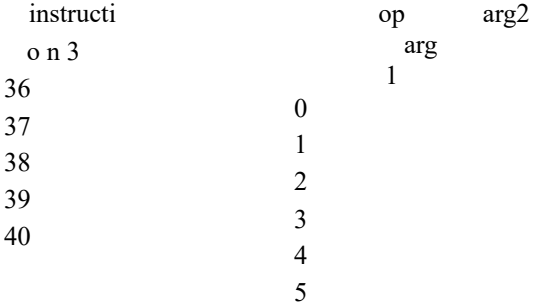


Figure 6.12: Indirect triples representation of three-address code

Static Single-Assignment form

- Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations.
- Two distinctive aspects distinguish SSA from three-address code.
    - The first is that all assignments in SSA are to variables with distinct names; hence the term static single-assignment form.

```
p = a + b          p1 = a + b
q = p - c          q1 = p1 - c
```

```
p = q * d          p2 = q1 * d
p = e - p          p3 = e - p2
q = p + q          q2 = p3 + q1
```

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

Figure 6.13: Intermediate program in three-address code and SSA

- The second distinctive aspect of SSA is that it uses a notational convention called the φ-function to combine the two definitions of x:

```
if ( flag ) x1 = -1; else x2
= 1; x3 = (x1; x2);
```

Here, (x1; x2) has the value x1 if the control flow passes through the true part of the conditional and the value x2 if the control flow passes through the false part.

Tthe 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

- 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.
- The requirements imposed on a code generator are severe.
- The target program must preserve the semantic meaning of the source program and be of high quality; .
- The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable;
- 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.

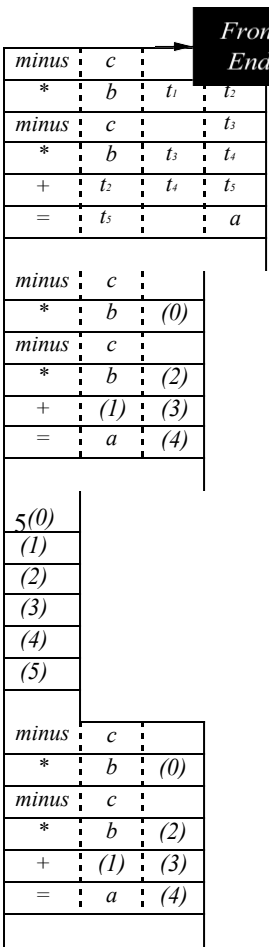


Figure 8.1: Position of code generator

- 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.

Issues in the design of a Code Generator

- The most important criterion for a code generator is that it produces correct code.

1. Input to the Code Generator

- o 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
  .
- o The intermediate representations can be-

- quadruples, triples, indirect triples;
- linear representations such as postfix notation;
- graphical representations such as syntax trees and DAG's.

target program

2. The Target Program

- The instruction-set architecture of the target machine has a significant impact on final code generated.
- The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.
- In RISC features are: has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
- In CISC features are: has few registers, two-address instructions, 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.
- Java Virtual Machine (JVM) uses the concept of stack based machine.
- Produces an absolute machine-language program as output, the advantage is that it can be placed in a fixed location in memory and immediately executed.
- Producing a relocatable machine-language program (often called an object module) as output, it allows subprograms to be compiled separately. A linking loader can then be used for combining these modules.
- The advantage is great flexibility since different modules can be combined separately.
- If the target machine does not handle relocation automatically, the compiler.
- Producing an assembly-language program as output makes the process of code generation easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code.
- For readability, assembly code is used as the target language.

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 factors such as
    - the level of the IR
    - the nature of the instruction-set architecture
    - the desired quality of the generated code.

- The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection.
- Instruction selection is dependent on the following factors:
  - the uniformity and com- pleteness of the instruction set .
  - Instruction speeds and machine idioms
- 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.

- The quality of the generated code is usually determined by its speed and size.
- 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 loading into several registers.
- Instruction cost is also known for designing a good code sequence.

4. Register Allocation

- The key problem in code generation is deciding what values to hold in what registers.
- Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.
- Values not held in registers need to reside in memory.
- Instructions involving register operands are invariably shorter and faster than those involving operands in memory.
- So efficient utilization of registers is important.
- 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 specifies register that a variable will reside in.

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.
- The problem of order of evaluation is solved by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

The Target Machine

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

A Simple Target Machine Model

- The 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
- A full- edged assembly language would have scores of instructions.
- Most instructions consists of an operator, followed by a target, followed by a list of source operands.
- A label may precede an instruction.
- The following are the kinds of instructions available:
    - Load operations: The instruction LD dst, addr loads the value in location addr into location dst. This instruction denotes the assignment dst = addr. An instruction of the form LD r_1; r_2 is a register-to-register copy in which the contents of register r_2 are copied into register r_1.

- 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; src_1; src_2, where OP is a operator like ADD or SUB, and dst, src_1, and src_2 are locations, not necessarily distinct. The e*ff*ect of this machine instruction is to apply the operation represented by OP to the values in locations src_1 and src_2, and place the result of this operation in location dst.
- 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. .

o The 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.
- A memory location can be an integer indexed by a register.. This feature is useful for following pointers.
- There are two other 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.
- Finally, an immediate constant addressing mode where the constant is prefixed by #. The instruction LD R1, #100 loads the integer 100 into register R1, and ADD R1, R1, #100 adds the integer 100 into register R1.

Comments at the end of instructions are preceded by //.

Example 8.2 : The three-address statement x = y- z can be implemented by the machine instructions:

```
LD R1, y            // R1  = y
LD R2, z            // R2  = z
SUB R1, R1, R2      // R1 = R1 - R2
ST   x, R1          // x = R1
```

- One of the goals of a good code-generation algorithm is to avoid using all four of these instructions, whenever possible.
- For example, y and/or z may have been computed in a register, and if so we can avoid the LD step(s).
- Similarly, it is possible to avoid ever storing x if its value is used within the register set and is not subsequently needed.
- 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  R1,  i              //  R1  = i MUL
R1, R1, 8               // R1 = R1 *8
LD  R2, a(R1)           // R2 = contents(a +
contents(R1)) ST        b, R2 // b = R2
```

- The second step computes 8i, and the third step places in register R2 the value in the ith element of a | the one found in the location that is 8i bytes past the base address of the array a.
- Similarly, the assignment into the array a represented by three-address instruction a[j] = c is implemented by:

```
LD R1, c            // R1  = c
LD R2, j            // R2  = j
MUL R2, R2, 8       // R2 = R2 * 8
ST a(R2), R1        // contents(a + contents(R2)) = R1
```

- To implement a simple pointer indirection, such as the three-address statement x = *p, we can use machine instructions like:

```
LD     R1, p // R1  = p
```

```
LD  R2, 0(R1)              // R2 = contents(0 +
contents(R1)) ST x, R2 // x = R2
```

- The assignment through a pointer *p = y is similarly implemented in machine code by:

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

- The machine-code equivalent would be something

  like:

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

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

Program and Instruction Costs

- 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.
- Determining the actual cost of compiling and running a program is a com- plex problem.
- Finding an optimal target program for a given source program is an undecidable problem in general, and many of the subproblems involved are NP-hard.
- The cost of an instruction is taken 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, R1 copies the contents of register R1 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.