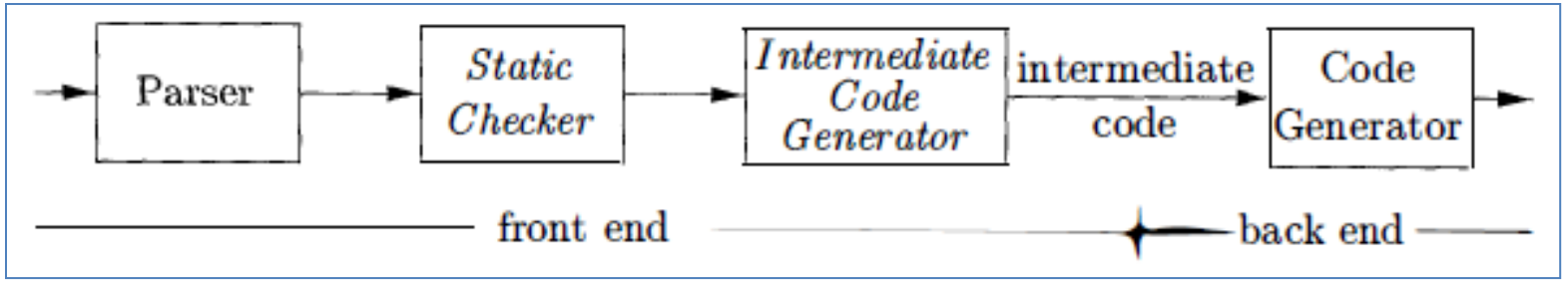
Intermediate Code Generation

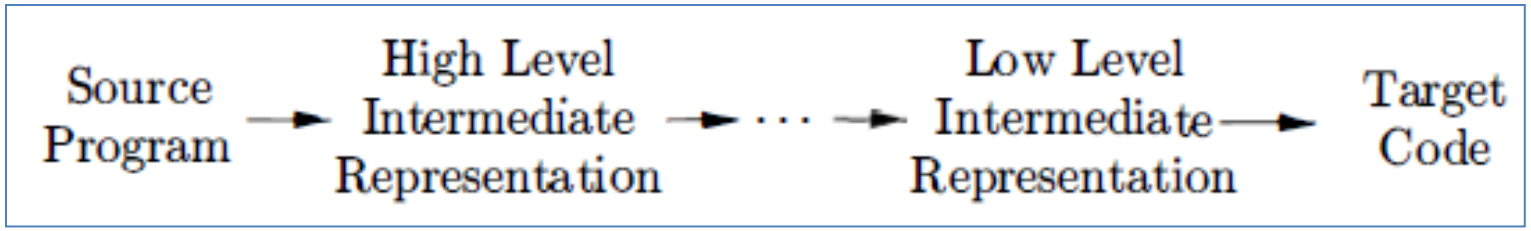
Md. Khorshed Alam

CSE, NDUB

Logical Structure



- A compiler might use a sequence of intermediate representations



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

DAG for Expressions

- A DAG has leaves corresponding to atomic operands & interior codes corresponding to operators.

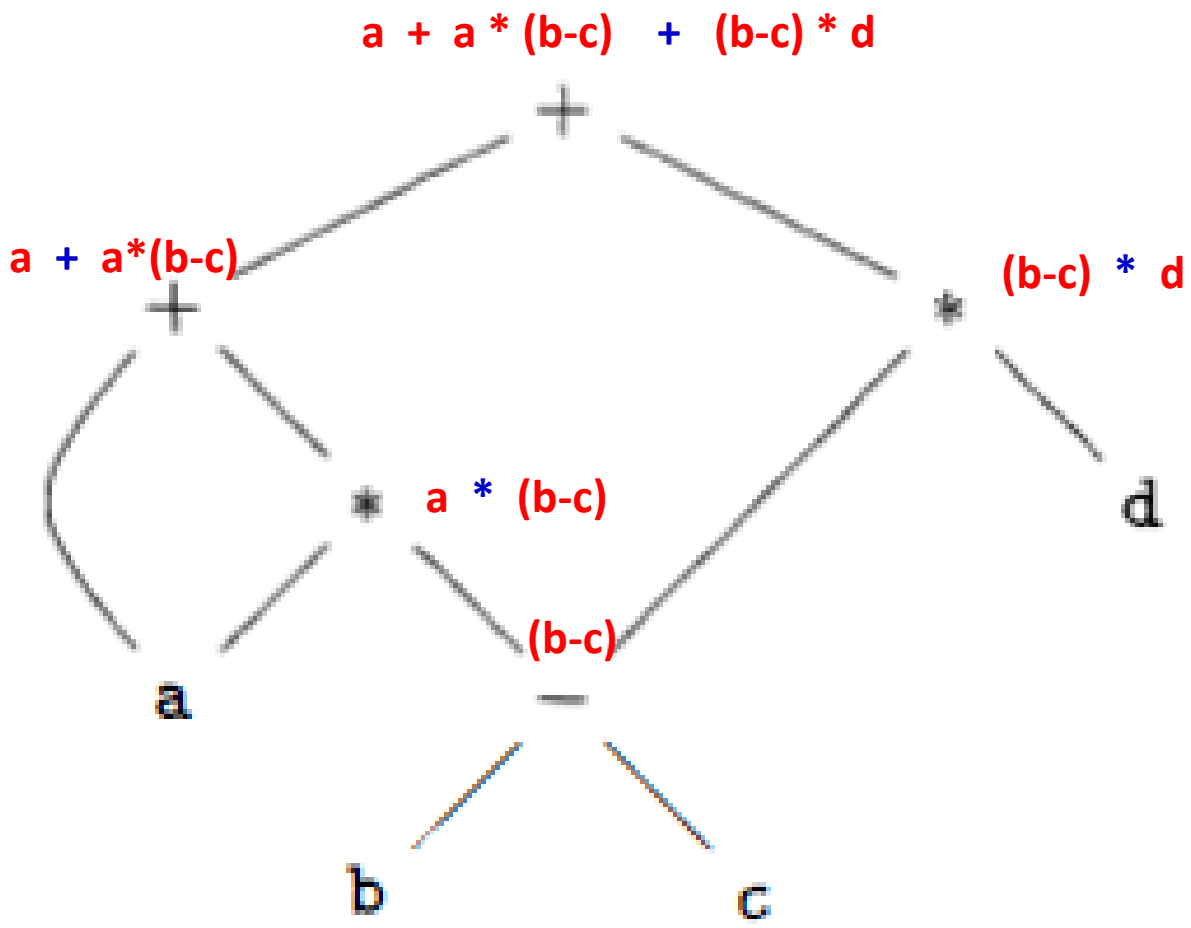
❑ **Difference between DAG & Syntax Tree**

- a node N in a DAG has more than one parent if N represents a common sub-expression
- in a syntax tree, the tree for the common sub expression would be replicated as many times as the sub expression appears in the original expression.

➢ Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

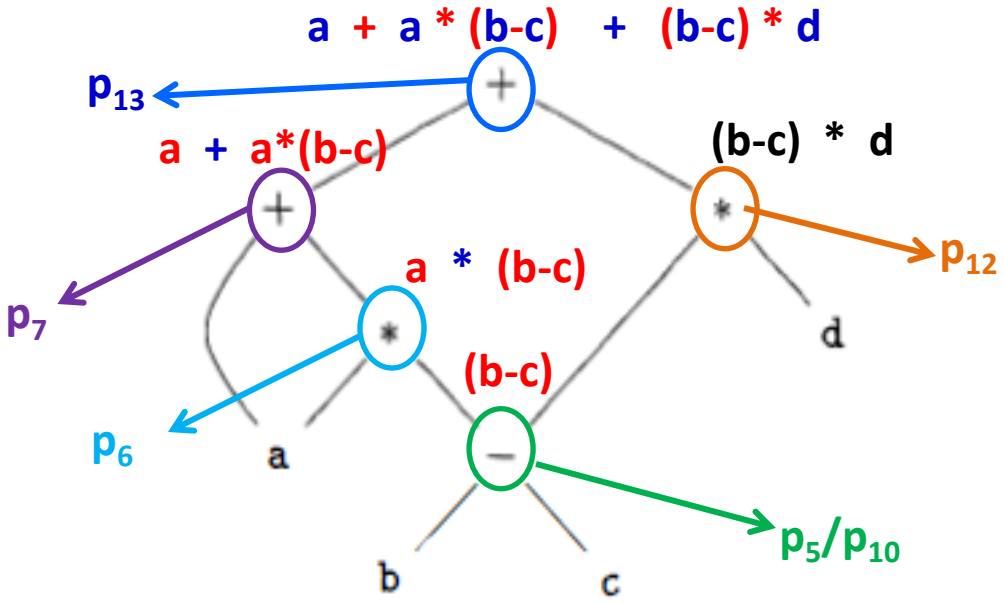
Example 6.1: Figure 6.3 shows the DAG for the expression

```
a + a * (b - c) + (b - c) * d
```



Example 6.2: The sequence of steps shown in Fig. 6.5 constructs the DAG in Fig. 6.3, provided *Node* and *Leaf* return an existing node, if possible, as discussed above. We assume that *entry-a* points to the symbol-table entry for **a**, and similarly for the other identifiers.

	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**.$val)$



Steps for DAG

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

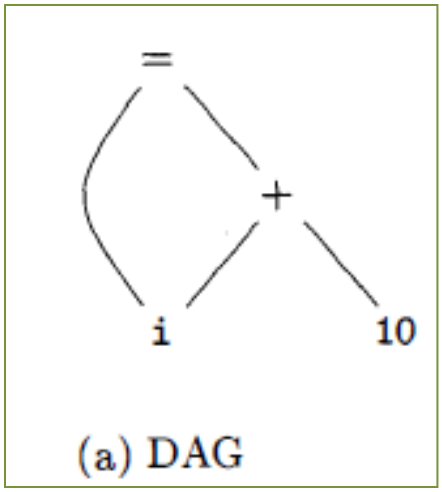
Value Number Method for Constructing DAG's

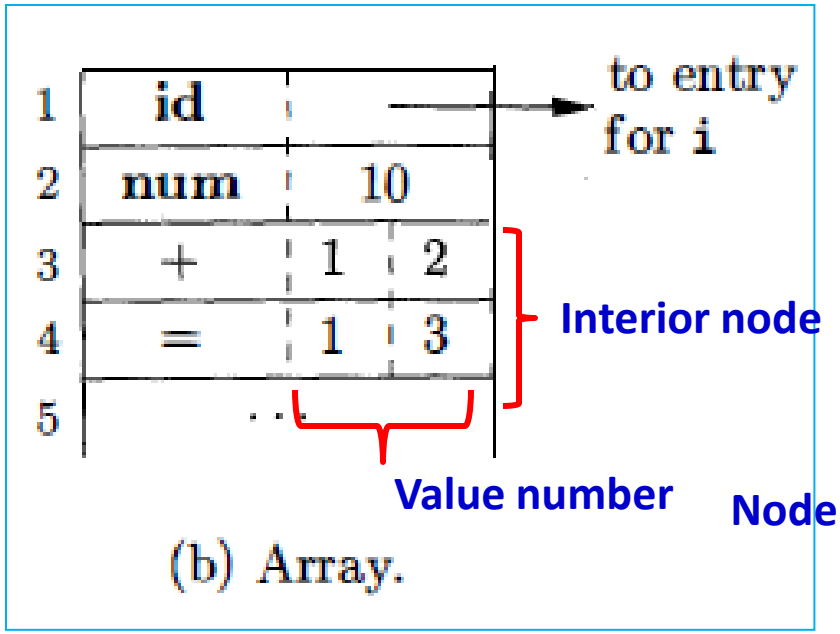
- Often, the nodes of a syntax tree or DAG are stored in **an array of records**.
- Each row of the array represents *one record*, and therefore one node.
- In each record, the first field is an operation code, indicating the label of the node.
- **Array:**

(i) leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case),

(ii) interior nodes have two additional fields indicating the left and right children

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





Array:

(i) leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case),

(ii) interior nodes have two additional fields indicating the left and right children

Node: < op, l, r >

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 $\langle op, l, r \rangle$.

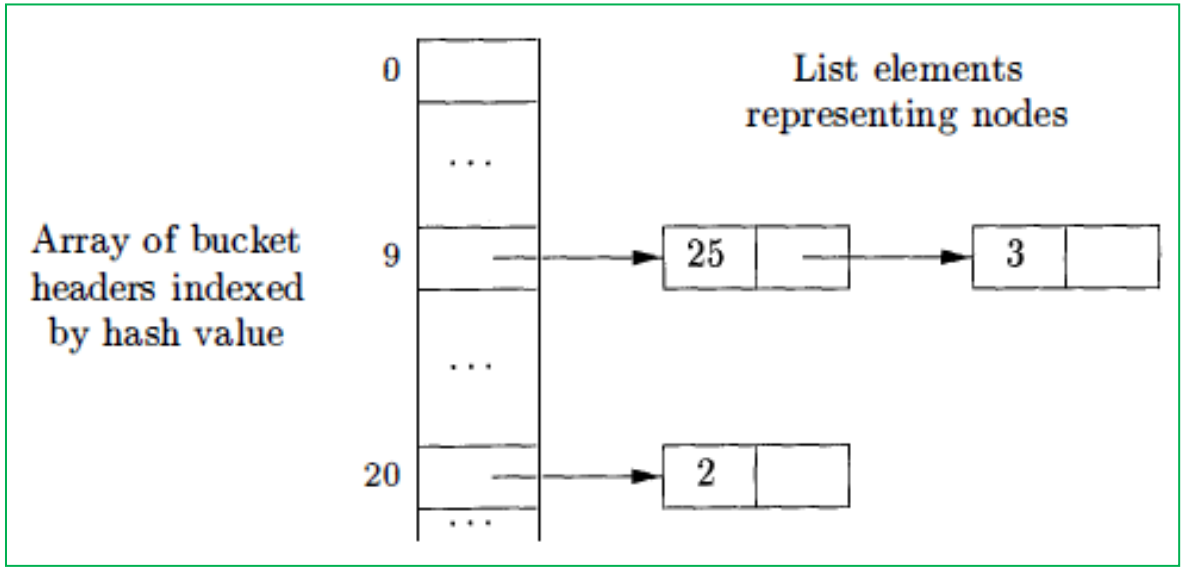
METHOD: Search the array for a node M with label op, left child l, and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label op, left child l, and right child r, and return its value number. □

While Algorithm yields the desired output, searching the entire array every time is expensive, especially if the array holds expressions from an entire program.

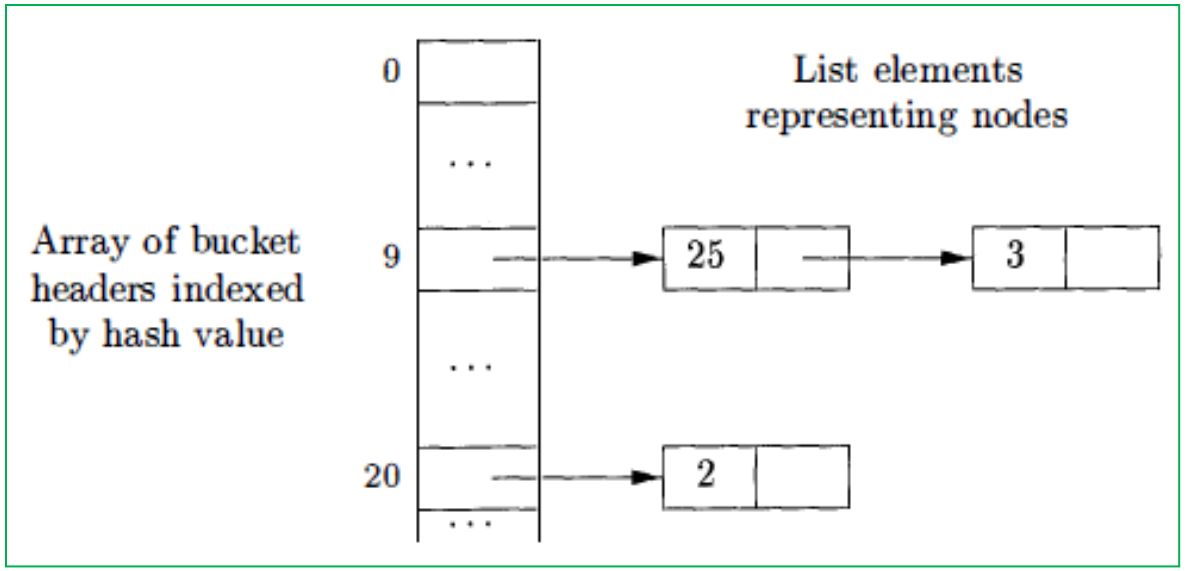
- **More efficient**
  - **hash table**
  - in which the nodes are put into "**buckets**" each of which typically will have only a few nodes.
  - The hash table is one of several data structures that support dictionaries efficiently.
    - A dictionary is an abstract data type that allows us to insert & delete elements of a set, & to determine whether a given element is currently in the set.
    - A good data structure for dictionaries, such as a hash table, performs each of these operations in time that is constant or close to constant, independent of the size of the set.

How Constructs?

To construct a hash table for the nodes of a DAG, we need a *hash function* h that computes the index of the bucket for a signature $\langle op, l, r \rangle$, 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 $\langle op, l, r \rangle$.



The buckets can be implemented as linked lists



❑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,** & **r**, we compute the bucket index **h(op, l, r) &** search 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.
- We may need to look at all the cells within a bucket, however, and 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 we find a match, we return **v**
- If we find no match, no such node can exist in any other bucket
- so we create a new cell, add it to the list of cells for bucket index **h(op, l, r)**, & return the value number in that new cell.

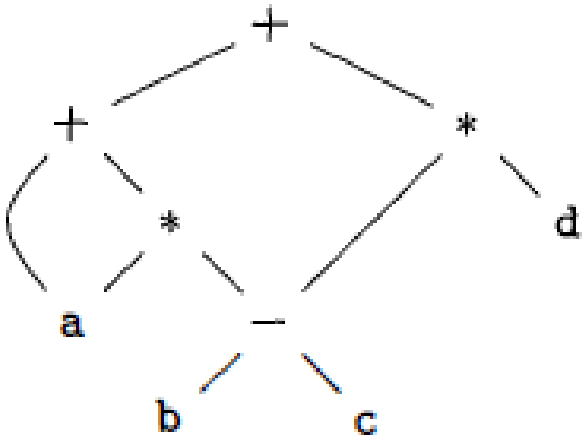
Three Address Code

- 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.
- source-language expression: **x + y * z**

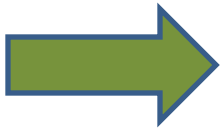
$$t_1 = y * z$$
$$t_2 = x + t_1$$

where t_1 & t_2 are compiler-generated temporary names.

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. □



(a) DAG



$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(b) Three-address code

Addresses and Instructions

- Three-address code is built from two concepts: addresses & instructions
- In object-oriented terms, these concepts correspond to classes, & various kinds of addresses & instructions correspond to appropriate subclasses.
- Three-address code can be implemented using records with fields for the addresses;
- **Records** called quadruples and triples

An address can be one of the following:

- A ***name***. For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
- A ***constant***. In practice, a compiler must deal with many different types of constants and variables.
- A ***compiler-generated temporary***. It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables .

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, shift operators, and conversion operators that, for example, convert an integer to a floating-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 $p(x_1, x_2, \ldots, x_n)$. The integer n, indicating the number of actual parameters in "`call` p, n," is not redundant because calls can be nested. That is, some of the first `param` statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call. The implementation of procedure calls is outlined in Section 6.9.

8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ 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
- The translation uses a symbolic label L, attached to the first instruction.
- The translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100.
- In both translations, the last instruction is a conditional jump to the first 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 goto 100
```

(b) Position numbers.

Three-address Implementation Technique

- 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.
  - **quadruples**
  - **triples**
  - **indirect triples**

Quadruples

- A **quadruple** (or "**quad**") has *four fields:* ***op,*** ***arg_1, arg_2*** & ***result***.
- The **op** field contains an internal code for the operator.

- x = y + z is represented by placing **+** in **op**, **y** in **arg_1**, **z** in **arg_2** , and **x** in **result**.

Few exceptions

1. Instructions with unary operators like x = `minus` y or x = y do not use arg_2. Note that 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 arg_2 nor *result*.

3. 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 special operator `minus` is used to distinguish the unary minus operator, as in `- c`, from the binary minus operator, as in `b - c`. Note that the unary-minus "three-address" statement has only two addresses, as does the copy statement `a = t5`.

---

a = b * - c + b * - c

Three-address code

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

Quadruples

	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
		...		

Triples

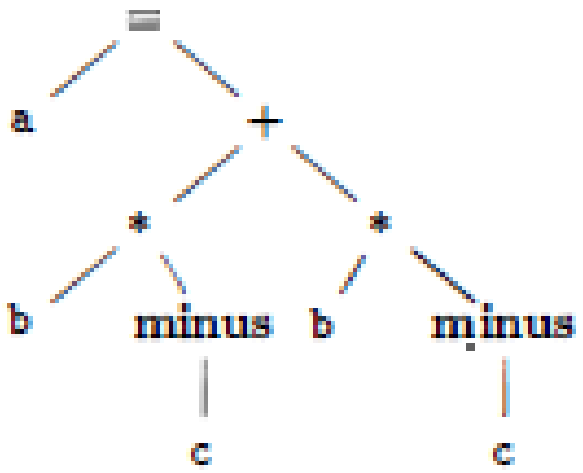
- A triple has only three fields: **op**, **arg$_1$** , & **arg$_2$**
- Using triples, we refer to the result of an operation ***x op y*** by its *position*, rather than by an explicit temporary name.
- Thus, instead of the temporary t_1, a triple representation would refer to position (0).
- Parenthesized numbers represent pointers into the triple structure itself (*value numbers*)

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 = t_5 is encoded in the triple representation by placing a in the arg_1 field and (4) in the arg_2 field. □

a = b * - c + b * - c

Three-address code

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

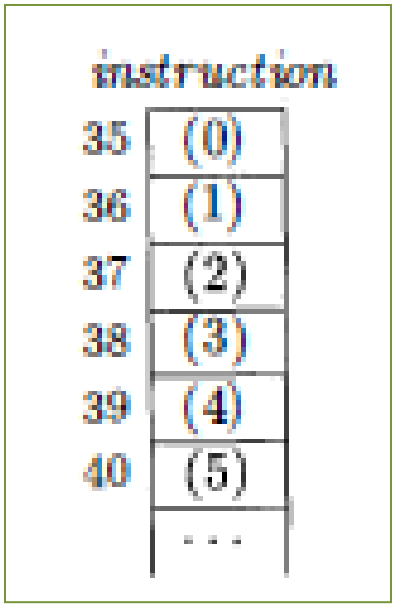


(a) Syntax tree

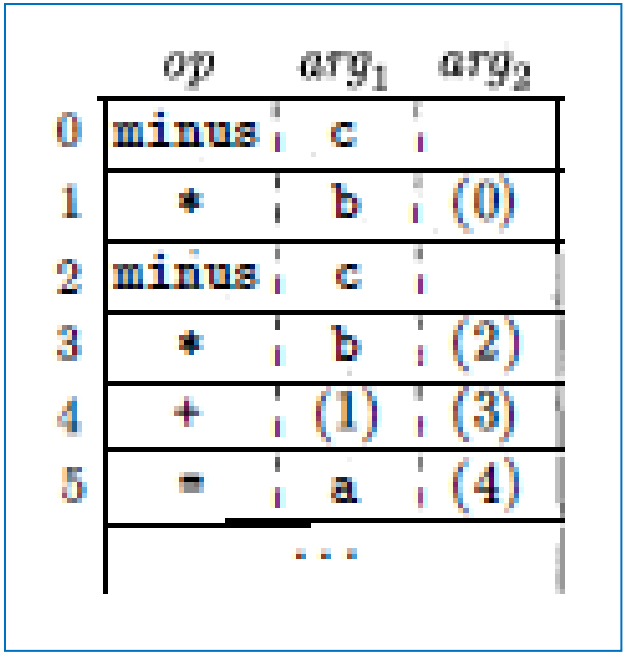
	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Indirect triples



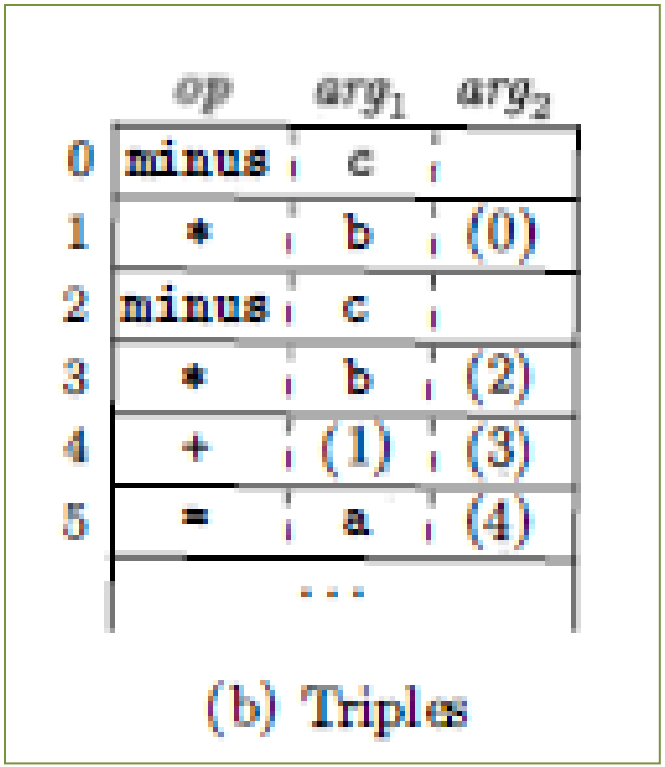
- Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.
- For example, let us use an array instruction to list pointers to triples in the desired order.



Array to pointers

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

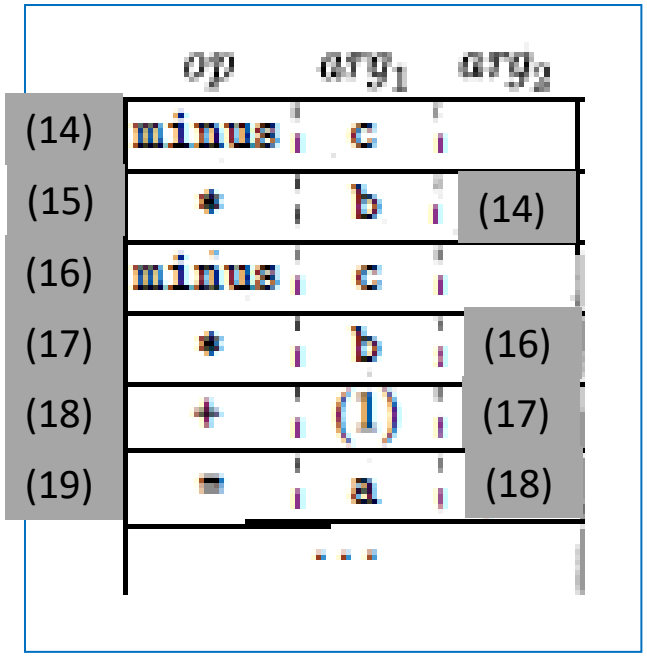
Triples



	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

Indirect Triples



	op	arg_1	arg_2
(14)	minus	c	
(15)	*	b	(14)
(16)	minus	c	
(17)	*	b	(16)
(18)	+	(1)	(17)
(19)	=	a	(18)
	...		

Comparisons

- 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
- This problem makes triples difficult to use in an optimizing compiler

- This problem does not occur with indirect triples, which we consider next.
- With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.
- Save some space compared with quadruples
- When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.