

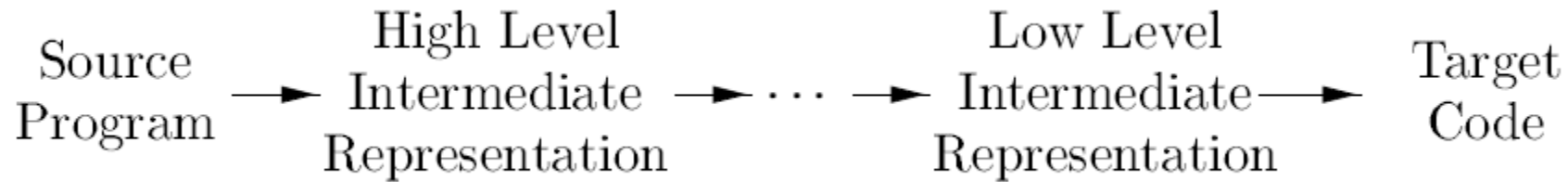
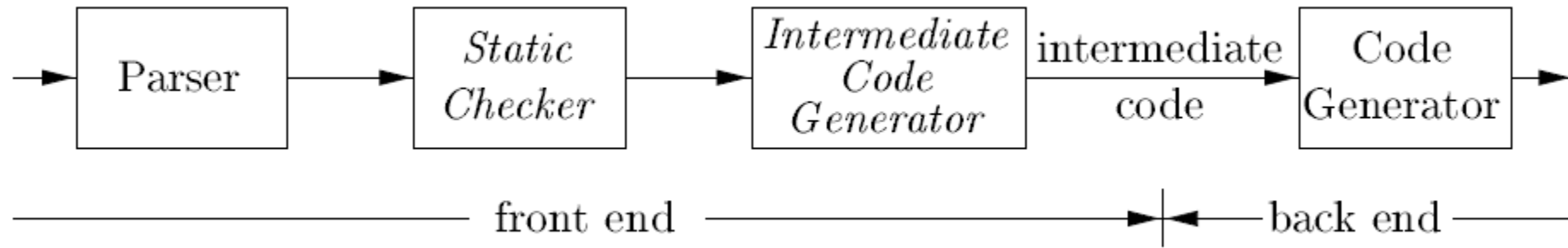
# Chapter-06

## Intermediate-Code Generation

# Outline

- ❑ [Variants of Syntax Trees](#)
  - ❑ Directed Acyclic Graphs for Expressions
  - ❑ The Value-Number Method for Constructing DAG's
- ❑ [Three-Address Code](#)
  - ❑ Addresses and Instructions
  - ❑ Quadruples
  - ❑ Triples
  - ❑ Static Single-Assignment Form
- ❑ [Types and Declarations](#)
  - ❑ Type Expressions
  - ❑ Type Equivalence
  - ❑ Declarations
  - ❑ Storage Layout for Local Names
  - ❑ Sequences of Declarations
- ❑ [Translation of Expressions](#)

# Introduction



# Variants of Syntax Trees

# Variants of Syntax Trees

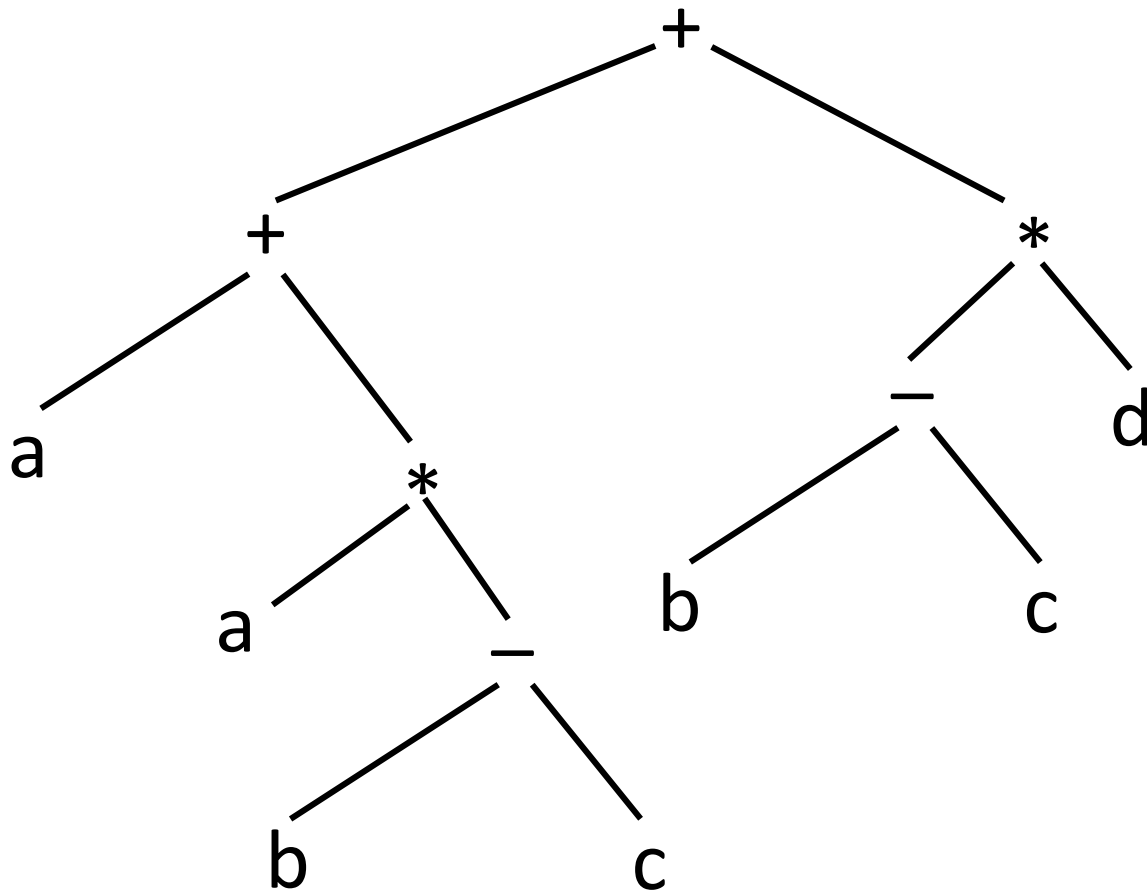
- Nodes in a syntax tree represent constructs in the source program
- The children of a node represent the meaningful components of a construct
- A **directed acyclic graph (hereafter called a DAG)** for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression
- As we shall see, DAG's can be constructed by using the same techniques that construct syntax trees

# Directed Acyclic Graphs for Expressions

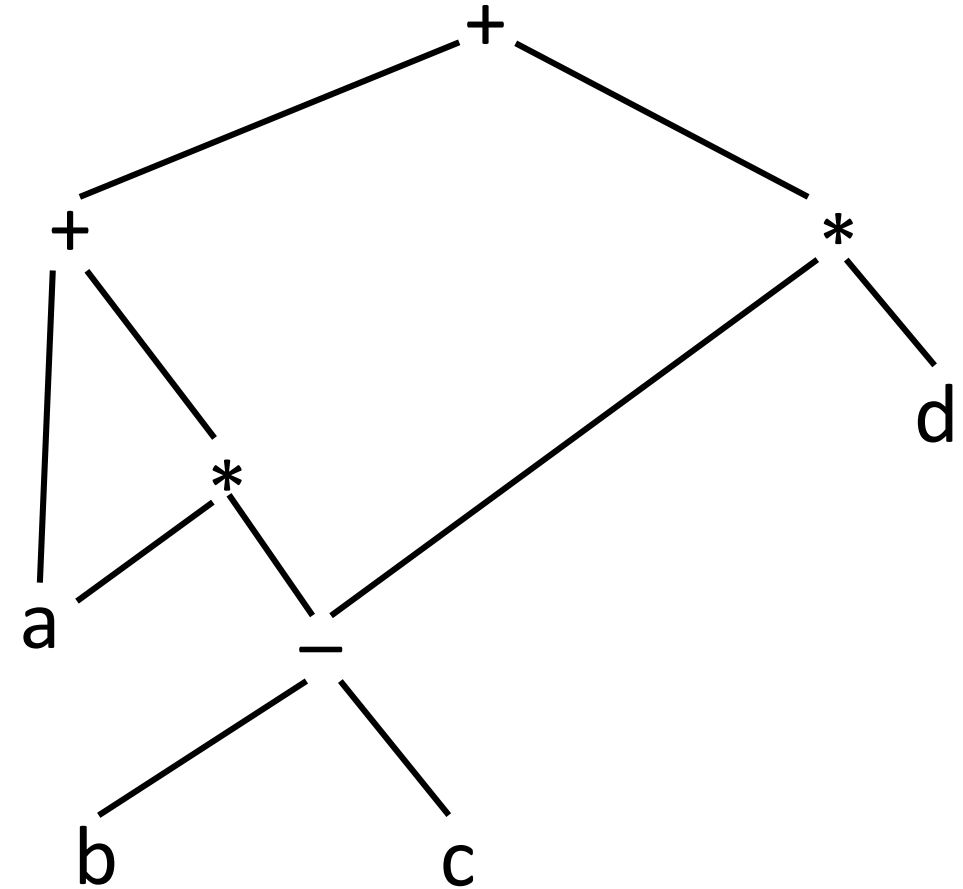
- Like the syntax tree for an expression, a DAG has **leaves** corresponding to **atomic operands** and **interior** nodes corresponding to **operators**
- The difference is that a node N in a DAG has more than one parent if N represents a common subexpression
- In a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression
- 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 shows the **Syntax Tree** and **DAG** for the expression,  $a + a*(b-c) + (b-c)*d$



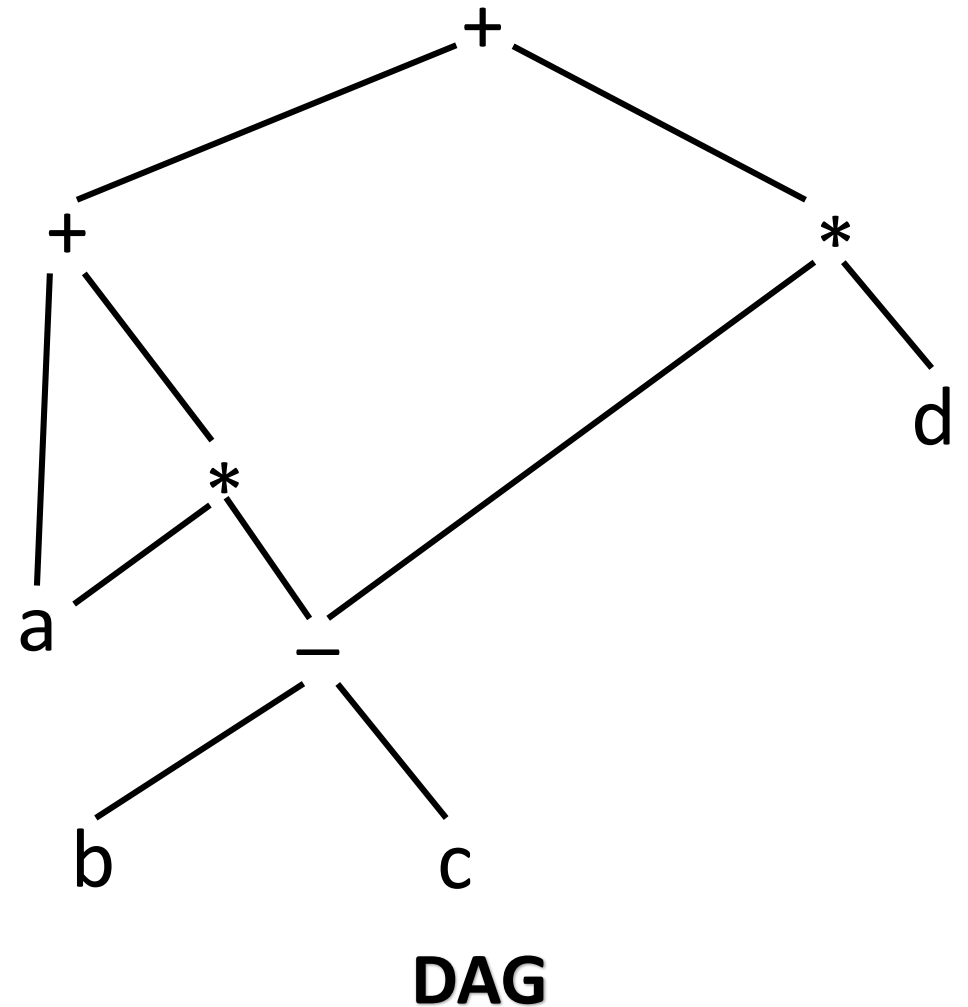
**Syntax Tree**



**DAG**

## Example 6.1 (Cont...)

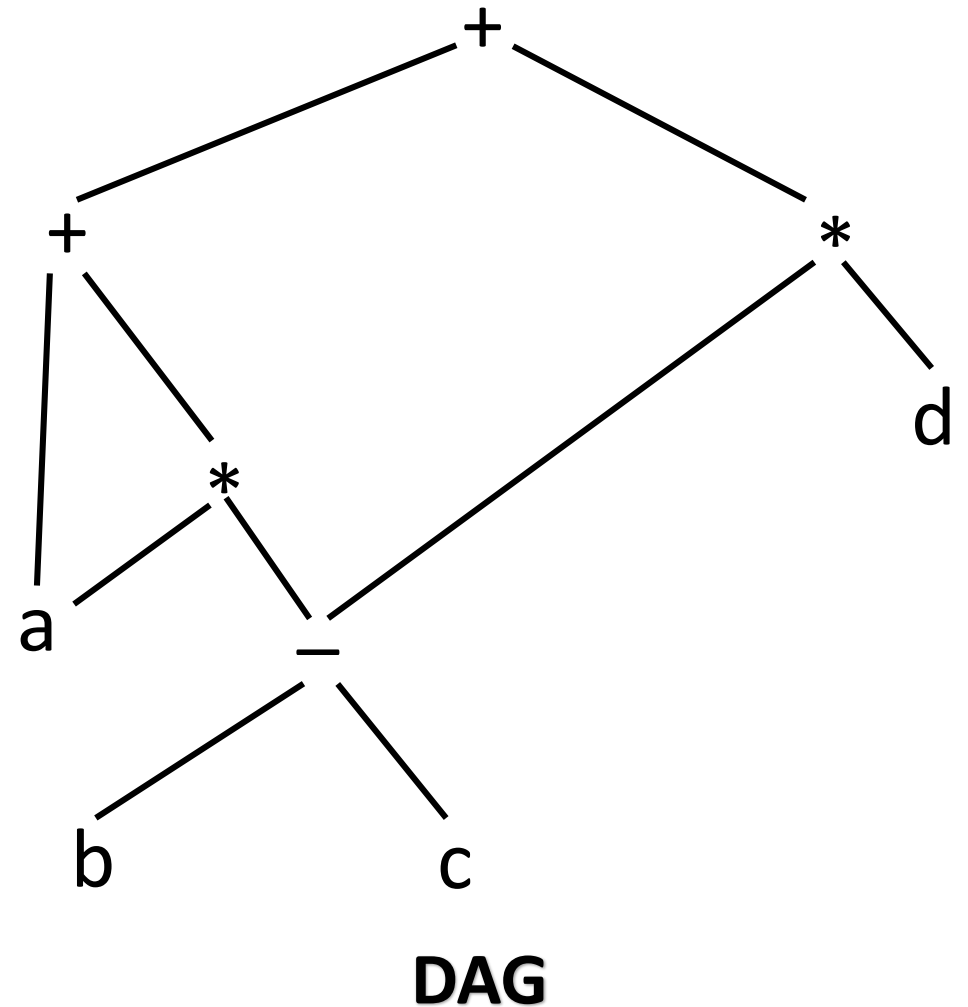
- The leaf for 'a' has two parents, because 'a' appears twice in the expression
- More interestingly, the two occurrences of the common subexpression 'b-c' are represented by one node, the node labeled '-'





## Example 6.1 (Cont...)

- 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'



## Example 6.1(Cont...)

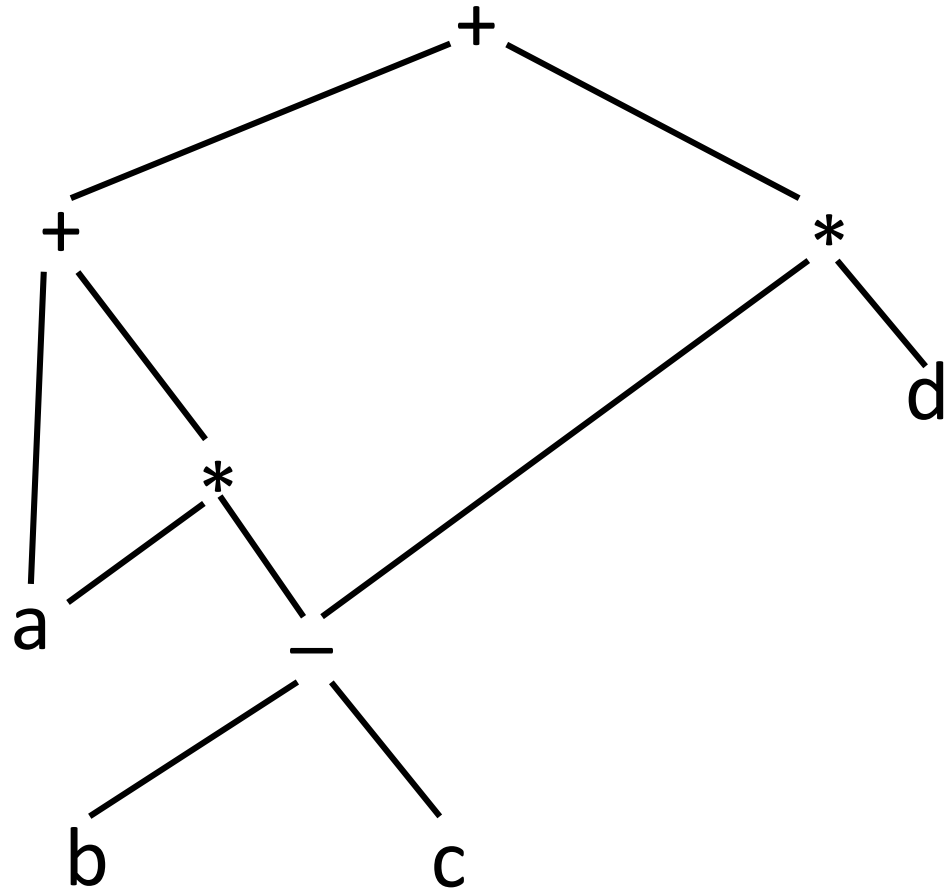
The SDD of figure can construct either syntax trees or DAG's

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \mathbf{new} \text{ Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \mathbf{new} \text{ Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow T_1 * F$	$T.node = \mathbf{new} \text{ Node}('*', T_1.node, F.node)$
5) $T \rightarrow T_1 / F$	$T.node = \mathbf{new} \text{ Node}('/', T_1.node, F.node)$
6) $T \rightarrow F$	$T.node = F.node$
7) $F \rightarrow (E)$	$F.node = E.node$
8) $F \rightarrow \mathbf{id}$	$F.node = \mathbf{new} \text{ Leaf}(\mathbf{id}, \mathbf{id}.entry)$
9) $F \rightarrow \mathbf{num}$	$F.node = \mathbf{new} \text{ Leaf}(\mathbf{num}, \mathbf{num}.val)$

## Example 6.2

The sequence of steps shown in figure constructs the DAG

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



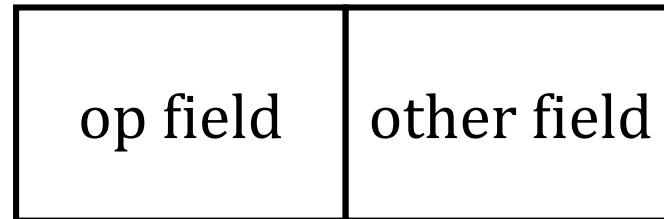
**DAG**

- 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 ('+',  $p_7$ ,  $p_{12}$ )

**Steps for Constructing DAG**

# The Value-Number Method for Constructing DAG's

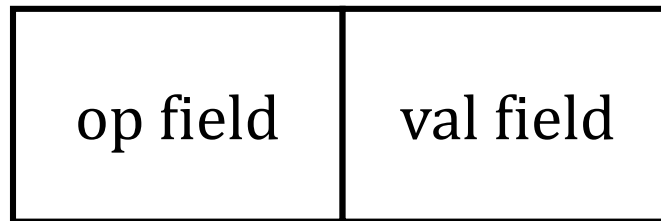
- Often, the nodes of a syntax tree or DAG are stored in an array of records
- Each row of the array represents one record, and therefore one node
- In each record, the first field is an operation code, indicating the label of the node



**Syntax Tree or DAG Node Representation as a Record**

# The Value-Number Method for Constructing DAG's (Cont...)

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



**Syntax Tree or DAG Node for a Leaf Node**

# The Value-Number Method for Constructing DAG's (Cont...)

- Interior nodes have two additional fields indicating the left and right children



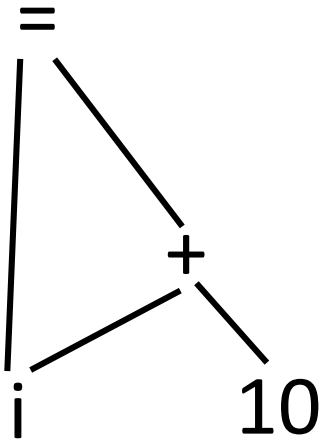
**Syntax Tree or DAG Node for an Interior Node with two children**

# **The Value-Number Method** for Constructing DAG's (Cont...)


- In the array, we refer to nodes by giving the integer index of the record for that node within the array
- This integer historically has been called the value number for the node or for the expression represented by the node

# The Value-Number Method for Constructing DAG's (Cont...)

DAG for  $i = i + 10$



(a) DAG

1	id		
2	num	10	
3	+	1	2
4	=	1	3
5	...		

(b) Array

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



## Algorithm 6.3

The value-number method for constructing the nodes of a DAG.

**INPUT:** Label  $op$ , node  $\ell$ , and node  $r$ .

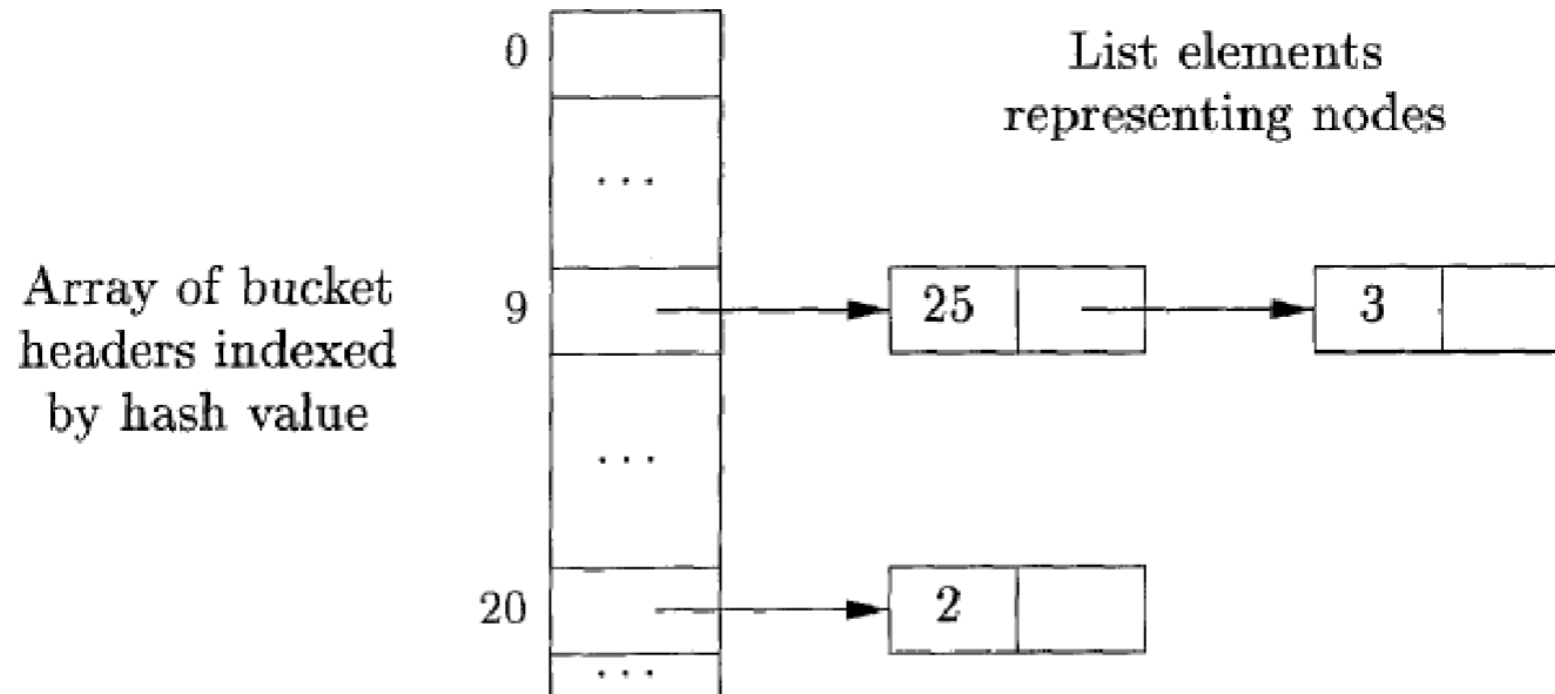
**OUTPUT:** The value number of a node in the array with signature  $\langle op, \ell, r \rangle$ .

**METHOD:** Search the array for a node  $M$  with label  $op$ , left child  $\ell$ , 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  $\ell$ , and right child  $r$ , and return its value number.

# The Value-Number Method for Constructing DAG's (Cont...)



Data structure for searching buckets

# Three-Address Code

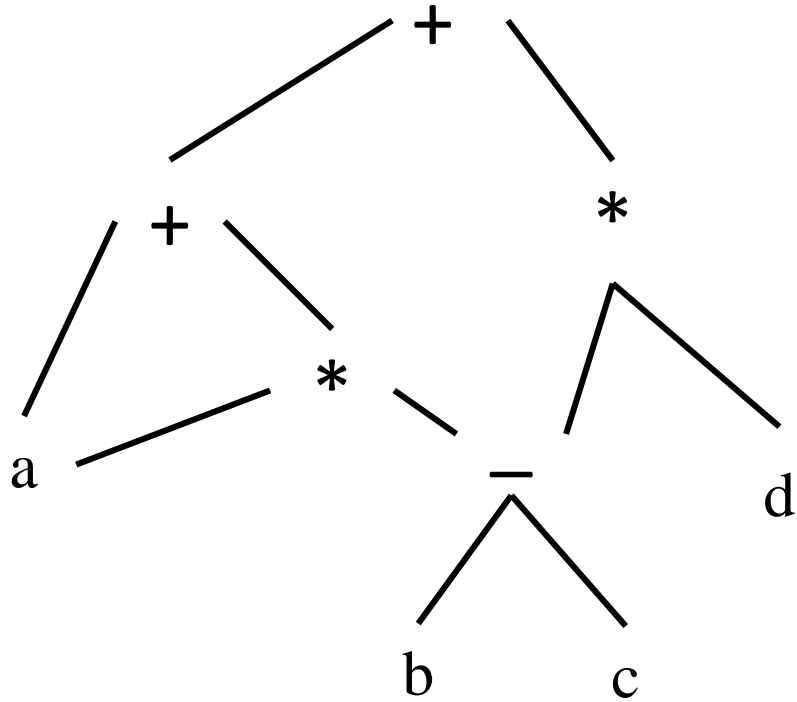
# Three-Address Code

- In three-address code, there is at most one operator on the right side of an instruction
- Thus a source-language expression like 'x + y \* z' might be translated into the sequence of three-address instructions
  - $t1 = y * z$
  - $t2 = x + t1$where t1 and t2 are compiler-generated temporary names
- 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

## Example 6.4

Figure shows the **DAG** and **Three-Address Code** for the expression,

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



**(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**

**A DAG and its Corresponding Three-Address Code**

# Addresses and Instructions

- Three-address code is built from two concepts: **addresses** and **instructions**
- An address can be one of the following:
  - Name
    - **$a = b + c$**  where **a**, **b** and **c** are source program names used as addresses
  - Constant
    - **$a = 10 + 20$**  where **10** and **20** are constants used as addresses
  - Compiler-generated temporary
    - **$t_1 = t_2 + t_3$**  where  **$t_1$** ,  **$t_2$**  and  **$t_3$**  are compiler-generated temporary used as addresses

# Addresses and Instructions (Cont...)

A list of the common three-address instruction forms:

- ❑ Assignment instructions of the form  $x = y \text{ op } z$ , where op is a binary arithmetic or logical operation, and x, y, and z are addresses  
For example:  $a = b + c$
- ❑ Assignments of the form  $x = \text{op } y$ , where op is a unary operation including unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number  
For example:  $a = -c$  ;  $a = (\text{float}) c$

# Addresses and Instructions (Cont...)

- ❑ Copy instructions of the form `x = y`, where x is assigned the value of y  
For example, `a = b`
- ❑ An unconditional jump `goto L` where the three-address instruction with label L is the next to be executed
- ❑ 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



# Addresses and Instructions (Cont...)

- ❑ 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
- ❑ Procedure calls and returns are implemented using the following instructions:
  - `param x` for parameters
  - `call p,n` for procedure calls
  - `y = call p,n` for function calls
  - `return y`, where y representing a returned value is optional

# Addresses and Instructions (Cont...)

Example: call of the procedure

$p(X_1, X_2, \dots, X_n)$

The sequence of three-address instructions will be,

```
param X1  
param X2  
...  
param Xn  
call p,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.

# Example 1 of Procedure Calls:

- Suppose that **a** is an array of integers
- **f** is a function from integers to integers
- Then, the **n = f(a[i])** assignment translate into the following three-address code:

1)  $t_1 = i * 4$

2)  $t_2 = a[t_1]$

3) param  $t_2$

4)  $t_3 = \text{call } f, 1$

5)  $n = t_3$

## Example 2 of Procedure Calls:

- Now let us suppose we need to execute the nested procedure call
- Then, the  **$n = f(a, g(a))$**  assignment translate into the following three-address code:

```
param a
t1 = call g, 1
param t1
t2 = call f, 2
n = t2
```

# Addresses and Instructions (Cont...)

- ❑ 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  $y$

# Addresses and Instructions (Cont...)

- 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  $y$
  - Presumably  $y$  is a name and  $x$  is a pointer name
  - In the instruction  $x = *y$ , presumably  $y$  is a pointer 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 figure

**L:      $t_1 = i + 1$   
       $i = t_1$   
       $t_2 = i * 8$   
       $t_3 = a [ t_2 ]$   
      if  $t_3 < v$  goto L**

**(a) Symbolic Labels**

**100:    $t_1 = i + 1$   
101:    $i = t_1$   
102:    $t_2 = i * 8$   
103:    $t_3 = a [ t_2 ]$   
104:   if  $t_3 < v$  goto 100**

**(b) Position Numbers**

**Two Ways of Assigning Labels to Three-Address Statements**

# 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”



# Quadruples (Cont...)

- A quadruple (or just “quad”) has four fields, which we call **op**, **arg<sub>1</sub>**, **arg<sub>2</sub>**, and **result**
- The op field contains an internal code for the operator
- For instance, the three-address instruction **x = y + z** is represented by placing + in op, y in arg<sub>1</sub>, z in arg<sub>2</sub>, and x in result

op	arg <sub>1</sub>	arg <sub>2</sub>	result
+	y	z	x

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Instructions with unary operators like  $x = \text{minus } y$  or  $x = y$  do not use  $\text{arg}_2$

op	arg <sub>1</sub>	arg <sub>2</sub>	result
minus	y		x

# Quadruples (Cont...)

The following are some exceptions to this rule:

- For a copy statement like  $x = y$ , **op is =**, while for most other operations, the assignment operator is implied

op	arg <sub>1</sub>	arg <sub>2</sub>	result
=	y		x

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Operators like param use neither  $\text{arg}_2$  nor result. For example, [param x](#)

op	$\text{arg}_1$	$\text{arg}_2$	result
param	x		

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Conditional and unconditional jumps put the target label in result
- For example, `goto L`

op	arg <sub>1</sub>	arg <sub>2</sub>	result
goto			L

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Conditional and unconditional jumps put the target label in result
- For example, `if x goto L`

op	arg <sub>1</sub>	arg <sub>2</sub>	result
goto	x		L

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Conditional and unconditional jumps put the target label in result
- For example, `if x < y goto L`

op	arg <sub>1</sub>	arg <sub>2</sub>	result
<	x	y	t <sub>1</sub>
goto	t <sub>1</sub>		L

# Example 6.6

Three-Address Code and Quadruples for the expression,  $a = b * -c + b * -c$

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

**(a) Three-Address Code**

	op	arg <sub>1</sub>	arg <sub>2</sub>	result
0	minus	c		t <sub>1</sub>
1	*	b	t <sub>1</sub>	t <sub>2</sub>
2	minus	c		t <sub>3</sub>
3	*	b	t <sub>3</sub>	t <sub>4</sub>
4	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
5	=	t <sub>5</sub>		a
...				

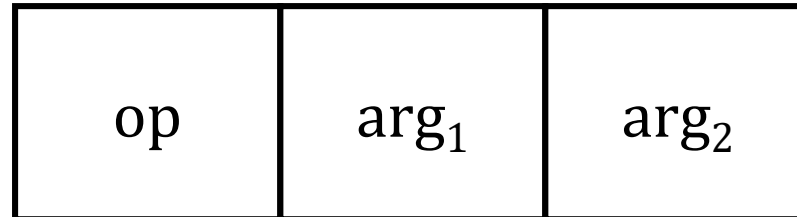
**(b) Quadruples**

**A Three-Address Code and its Corresponding Quadruples**



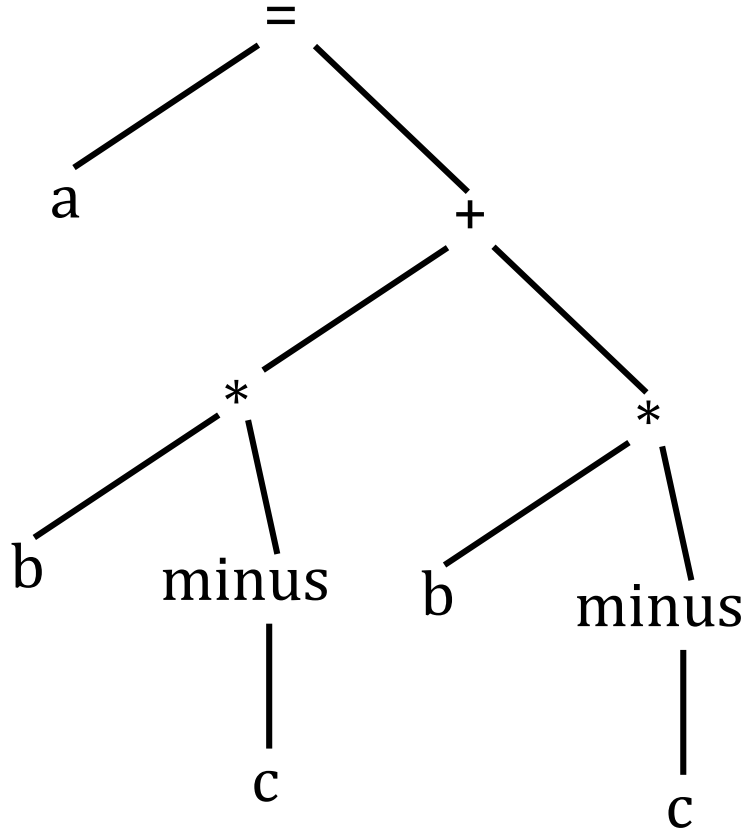
# Triples

- A triple has only **three fields**, which we call **op**, **arg<sub>1</sub>**, **arg<sub>2</sub>**
- 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



# Example 6.7

**Syntax Tree and Triples** for the expression,  $a = b * -c + b * -c$



**(a) Syntax Tree**

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
...			

**(b) Triples**

**A Syntax Tree and its Corresponding Triples**

## Triples (Cont...)

- In the triple representation in (b), the copy statement  $a = t5$  is encoded in the triple representation by placing  $a$  in the  $arg_1$  field and  $(4)$  in the  $arg_2$  field

5	op	arg <sub>1</sub>	arg <sub>2</sub>
	=	a	(4)

# Triples (Cont...)

- 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

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	*	i	4
1	+	x	(0)
2	=	(1)	y

# Triples (Cont...)

- Similarly,  $x = y[i]$  can be implemented by treating it as if it were the two instructions  $t = y[i]$  and  $x = t$ , where  $t$  is a compiler-generated temporary
- 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

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	*	i	4
1	+	y	(0)
2	=	x	(1)

# Benefit of Quadruples over Triples

- 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 does not occur with [indirect triples](#), which we consider next.

# Benefit of Quadruples over Triples (Cont...)

Before Optimizing,

$t_1 = \text{itof}(60)$

$t_2 = a * t_1$

$t_3 = t_2 + 40$

	op	arg <sub>1</sub>	arg <sub>2</sub>	result
0	itof	60		$t_1$
1	*	a	$t_1$	$t_2$
2	+	$t_2$	40	$t_3$

**Quadruples**

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	itof	60	
1	*	a	(0)
2	+	(1)	40

**Triples**

# Benefit of Quadruples over Triples (Cont...)

After Optimizing,

$$t_2 = a * 60.0$$

$$t_3 = t_2 + 40$$

	op	arg <sub>1</sub>	arg <sub>2</sub>	result
0	*	a	60.0	t <sub>2</sub>
1	+	t <sub>2</sub>	40	t <sub>3</sub>

**Quadruples**

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	*	a	60.0
1	+	(0)	40

**Triples**

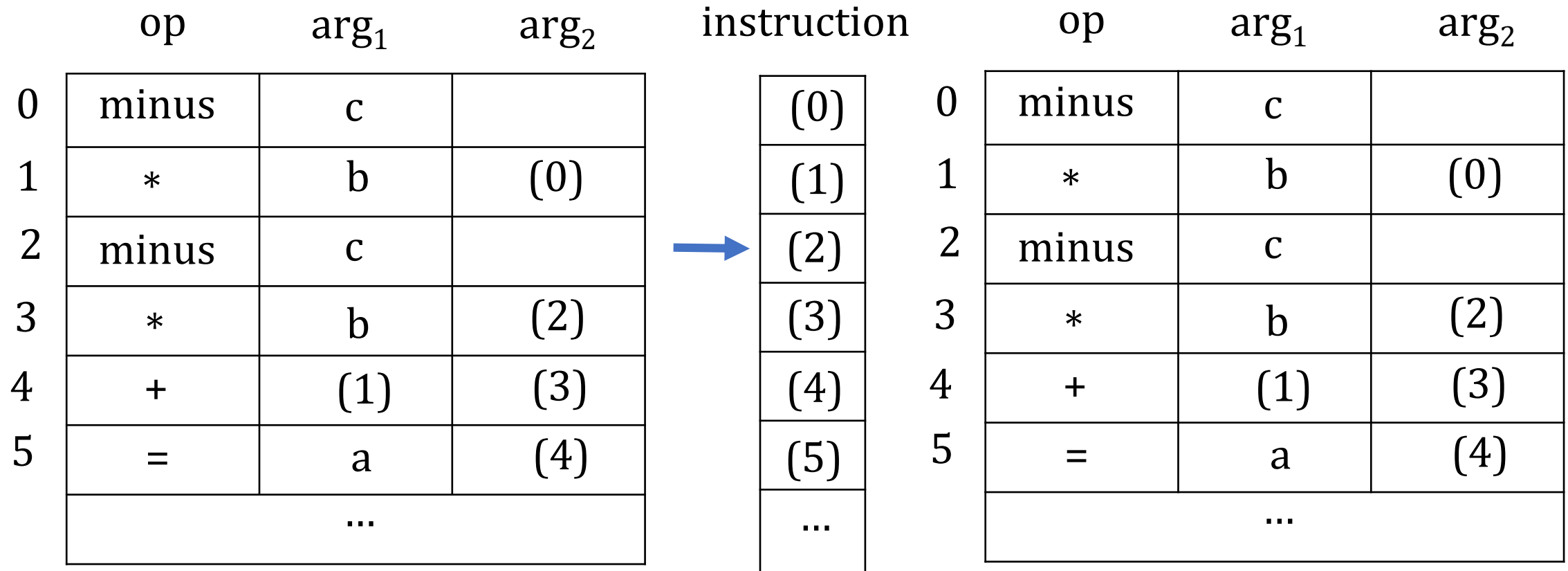


# Indirect Triples

- Indirect triples overcomes the limitation of triples
- It 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 to triples in the desired order
- With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves

# Indirect Triples(Cont...)

The triples in (b) might be represented as in this figure



**(b) Triples**

**Indirect Triples Representation**

# Use of Indirect Triples by an Optimized Compiler

Before Optimizing,

$t_1 = \text{itof}(60)$

$t_2 = a * t_1$

$t_3 = t_2 + 40$

After Optimizing,

$t_2 = a * 60.0$

$t_3 = t_2 + 40$

(0)		op	arg <sub>1</sub>	arg <sub>2</sub>
(1)	0	itof	60	
(2)	1	*	a	(0)
...	2	+	(1)	40

**Indirect Triples**



(1)		op	arg <sub>1</sub>	arg <sub>2</sub>
(2)	1	*	a	60.0
...	2	+	(1)	40
	3	...		

**Indirect Triples**

# Static Single-Assignment Form (SSA)

- 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**

# Static Single-Assignment Form (Cont...)

- Following shows the same intermediate program in three-address code and in static single assignment form
- Note that subscripts distinguish each definition of variables p and q in the SSA representation

**$p = a + b$**

**$q = p - c$**

**$p = q * d$**

**$p = e - p$**

**$q = p + q$**

**$p_1 = a + b$**

**$q_1 = p_1 - c$**

**$p_2 = q_1 * d$**

**$p_3 = e - p_2$**

**$q_2 = p_3 + q_1$**

**(a) Three-Address Code**

**(b) Static Single-Assignment Form**

**Intermediate Program in Three-Address Code and SSA**

# Static Single-Assignment Form (Cont...)

- The same variable may be defined in two different control-flow paths in a program

- For example, the source program

```
if ( flag ) x = -1; else x = 1;  
y = x * a;
```

has two control-flow paths in which the variable **x** gets defined

- If we use different names for **x** in the true part and the false part of the conditional statement, then which name should we use in the assignment **y = x \* a**?

# Static Single-Assignment Form (Cont...)

- Here is where the second distinctive aspect of SSA comes into play
- SSA uses a notational convention called the  $\phi$ -function to combine the two definitions of  $x$ :

```
if ( flag )  $x_1 = -1$ ; else  $x_2 = 1$ ;  
 $x_3 = \phi(x_1, x_2)$   
 $y = x_3 * a$ ;
```

- Here,  $\phi(x_1, x_2)$  has the value  $x_1$  if the control flow passes through the true part of the conditional and the value  $x_2$  if the control flow passes through the false part.
- That is to say, the  $\phi$ -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  $\phi$ -function

# Practice Problem-1

Expression:  $((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$

- Construct for the above expression by maintaining precedence
  - Syntax Tree
  - DAG
  - Allocate the Nodes of the DAG in an Array by using Value-Number Method
  - Three-Address Code
  - Quadruples
  - Triples
  - Indirect Triples
  - SSA



# Types and Declarations

# Types and Declarations

- The applications of types can be grouped under checking and translation
  - **Type checking** uses logical rules to reason about the behavior of a program at run time
  - Specifically, it ensures that the types of the operands match the type expected by an operator
  - For example, the && operator in Java expects its two operands to be booleans and the result is also of type Boolean

# Types and Declarations(Cont...)

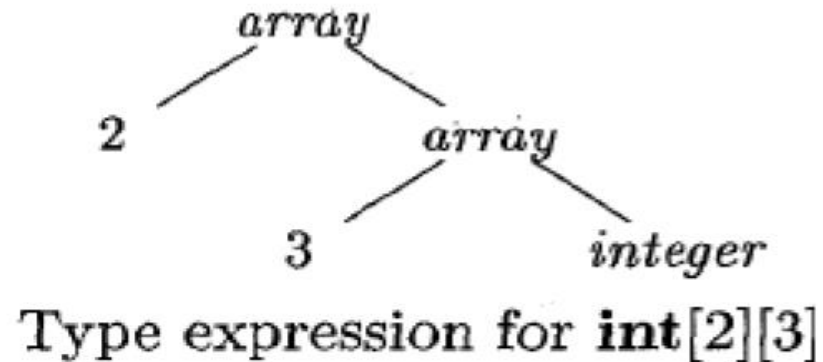
- The applications of types can be grouped under checking and translation
  - **Translation Applications:** From the type of a name, a compiler can determine the storage that will be needed for that name at run time
  - Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions etc.

# Type Expressions

- Types have structure, which we shall represent using type expressions
- A type expression is either a basic type or is formed by applying an operator called a type constructor to a type expression
- The sets of basic types and constructors depend on the language to be checked

## Example 6.8

- 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 figure



- 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

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - **A basic type is a type expression**
  - Typical basic types for a language include boolean, char, integer, float, and void
  - “void” denotes “the absence of a value”
  - All these are examples of basic types

float x;                      **//Type Expression: float**

int i;                        **//Type Expression: int**

float m;                    **//Type Expression: float**

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - **A type name is a type expression**
  - **A type expression can be formed by applying the array type constructor to a number and a type expression**
  - All these are examples of array type constructor applied to a number and a type expression

array int[2];                   //**Type Expression: array(2,int)**

array float[2][3];           //**Type Expression: array(2,array(3,float))**

array float[2][3][4];       //**Type Expression: array(2,array(3,array(4,float)))**

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - **A record is a data structure with named fields**
  - **A type expression can be formed by applying the record type constructor to the field names and their types**
  - The use of a name x for a field within a record does not conflict with other uses of the name outside the record
  - Thus, the three uses of x in the following declarations are distinct and do not conflict with each other:

<code>float x;</code>	<code>//Type Expression: float</code>
<code>struct {float x; float y;} p;</code>	<code>//Type Expression: record(x:float,y:float)</code>
<code>struct {int t; float x; float y;} q;</code>	<code>//Type Expression: record(t:int,x:float,y:float)</code>



# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - A type expression can be formed by using the type constructor  $\rightarrow$  for function types
  - We write  $s \rightarrow t$  for “function from type  $s$  to type  $t$ ”
  - Here the functions map one type expression to the other

function int f(float x); Type Expression: float  $\rightarrow$  int

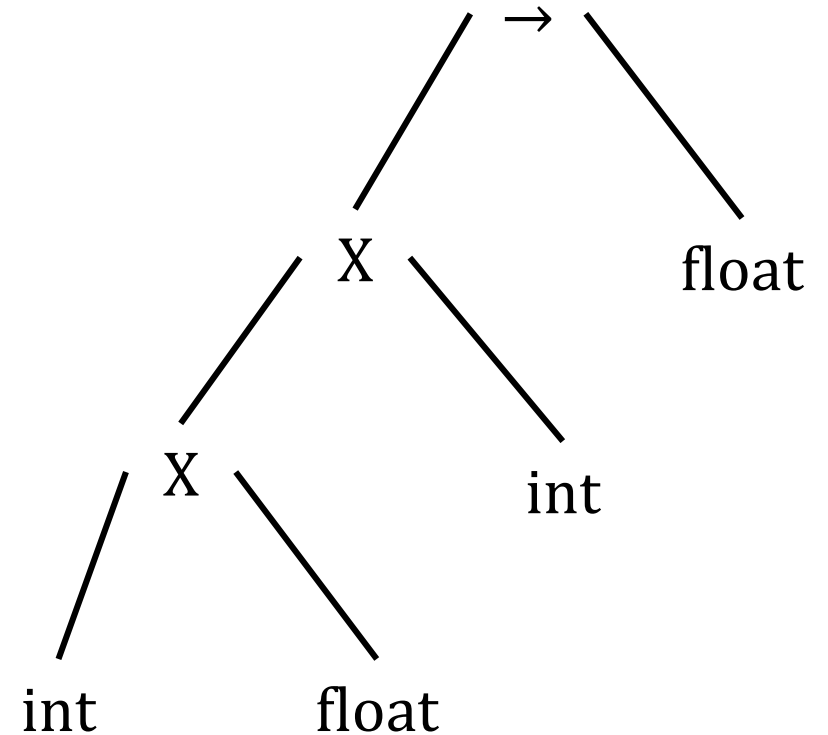
function float g(int a, float b, int c);  
Type Expression: int  $\times$  float  $\times$  int  $\rightarrow$  float

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - If  $s$  and  $t$  are type expressions, then their Cartesian product  $\mathbf{s \times t}$  is a type expression
  - They can be used to represent function parameters
  - We assume that  $\times$  associates to the left and that it has higher precedence than  $\rightarrow$
  - **Type expressions may contain variables whose values are type expressions**

# Type Expressions (Cont...)

- A convenient way to represent a type expression is to use a graph
- Example: Type expression for  
**function float g(int a, float b, int c);**  
is **int × float × int → float**
- It can be represented by generating **expression tree** like below
- **Interior Nodes:** type constructor
- **Leaves:** basic types, types names and type variables



# Type Equivalence

- When are two type expressions **equivalent**?
- Many type-checking rules have the form, “**if** two type expressions are equal **then** return a certain type **else** error.”
- Two types can be either
  - Structurally Equivalent or
  - Name Equivalent

# Type Equivalence (Cont...)

- When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:
  - They are the same basic type
  - They are formed by applying the same constructor to structurally equivalent types
  - One is a type name that denotes the other
- If type names are treated as standing for themselves, then the first two conditions in the above definition lead to name equivalence of type expressions

# Name Equivalence

- **Name Equivalence:** two types are equal if and only if they have the same name
- Thus, for example in the code (using C syntax)

```
struct ST{  
    int a;  
    float b;  
}X,Y;
```

```
struct T{  
    int a;  
    float b;  
}M,N;
```

# Name Equivalence (Cont...)

```
struct ST{  
    int a;  
    float b;  
}X,Y;
```

```
struct T{  
    int a;  
    float b;  
}M,N;
```

- If name equivalence is used in the language then
  - **X and Y** would be of the same type and
  - **M and N** would be of the same type but
  - The type of **X or Y** would not be equivalent to the type of **M or N**
- This means that statements such as:
  - **X = Y** and **M = N** would be valid but
  - **X = M** would not be valid (would not be accepted by a translator)

# Structural Equivalence

- **Structural Equivalence:** two types are equal if and only if, they have the same structure which can be interpreted in different ways
  - A strict interpretation would be that the names and types of each component of the two types must be the same and must be listed in the same order in the type definition
  - A less stringent requirement would be that the component types must be the same and in the same order in the two types, but the names of the components could be different



# Structural Equivalence (Cont...)

```
struct ST{  
    int a;  
    float b;  
}X,Y;
```

```
struct T{  
    int a;  
    float b;  
}M,N;
```

- Again looking at the example above using structural equivalence the two types **ST** and **T** would be considered equivalent which means that a translator would accept statements such as **X = M**
- Note that C doesn't support structural equivalence for struct and classes and will give error for above assignment

# Declarations

- Following grammar consisting of types and declarations that declares just one name at a time

$$D \rightarrow T \text{ id} ; D \mid \epsilon$$

$$T \rightarrow B C \mid \text{record } \{ D \}$$

$$B \rightarrow \text{int} \mid \text{float}$$

$$C \rightarrow \epsilon \mid [ \text{num} ] C$$

## Declarations (Cont...)

$$D \rightarrow T \textbf{id} ; D \mid \epsilon$$

$$T \rightarrow B C \mid \textbf{record} \{ D \}$$

$$B \rightarrow \textbf{int} \mid \textbf{float}$$

$$C \rightarrow \epsilon \mid [ \textbf{num} ] C$$

- Nonterminal D generates a sequence of declarations
- Nonterminal T generates basic, array or record types
- Nonterminal B generates one of the basic types int and float

# Declarations (Cont...)

$$D \rightarrow T \text{ id} ; D \mid \epsilon$$
$$T \rightarrow B C \mid \text{record } \{ D \}$$
$$B \rightarrow \text{int} \mid \text{float}$$
$$C \rightarrow \epsilon \mid [ \text{num} ] C$$

- Nonterminal  $C$  for “component” generates strings of zero or more integers and each integer surrounded by brackets
- An array type consists of a basic type specified by  $B$  followed by array components specified by nonterminal  $C$
- A record type (the second production for  $T$ ) is a sequence of declarations for the fields of the record and all surrounded by curly braces

## Practice Problem-2

$$D \rightarrow T \text{ id} ; D \mid \epsilon$$

$$T \rightarrow B C \mid \text{record } \{ D \}$$

$$B \rightarrow \text{int} \mid \text{float}$$

$$C \rightarrow \epsilon \mid [ \text{num} ] C$$

- For the grammar given above, construct parse tree for the following input strings:
  - **int a; float b;**
  - **int [3] [4] a;**
  - **record { int a; float b; int [3] [4] a; }**

## Practice Problem-3

$$D \rightarrow T \text{ id} ; D \mid \epsilon$$

$$T \rightarrow B C \mid \text{record } \{ D \}$$

$$B \rightarrow \text{int} \mid \text{float}$$

$$C \rightarrow \epsilon \mid [ \text{num} ] C$$

- Modify the grammar given above so that it can construct parse tree for declarations with list of names and then construct parse tree for the following input
  - **record { int a, b; int [3] [4] a; }**

**[Hint: Use Example 5.10]**

# Storage Layout for Local Names

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

# Storage Layout for Local Names (Cont...)

- Suppose storage comes in blocks of contiguous bytes where byte is the smallest unit of addressable memory
- Typically a byte is eight bits, and some number of bytes form a machine word
- Multibyte objects are stored in consecutive bytes and given the address of the first byte
- The width of a type is the number of storage units needed for objects of that type



# Storage Layout for Local Names (Cont...)

- The syntax directed translation scheme (SDT) in figure computes types and their widths for basic and array types

- |                                     |  |
|-------------------------------------|--|
| 1) $T \rightarrow B$                | $\{ t = B.type; w = B.width; \}$   |
| $C$                                 | $\{ T.type = C.type; T.width = C.width; \}$  |
| 2) $B \rightarrow \text{int}$       | $\{ B.type = \text{integer}; B.width = 4; \}$  |
| 3) $B \rightarrow \text{float}$     | $\{ B.type = \text{float}; B.width = 8; \}$  |
| 4) $C \rightarrow \epsilon$         | $\{ C.type = t; C.width = w; \}$   |
| 5) $C \rightarrow [\text{num}] C_1$ | $\{ C.type = \text{array}(\text{num.value}, C_1.type);$<br>$C.width = \text{num.value} \times C_1.width; \}$ |

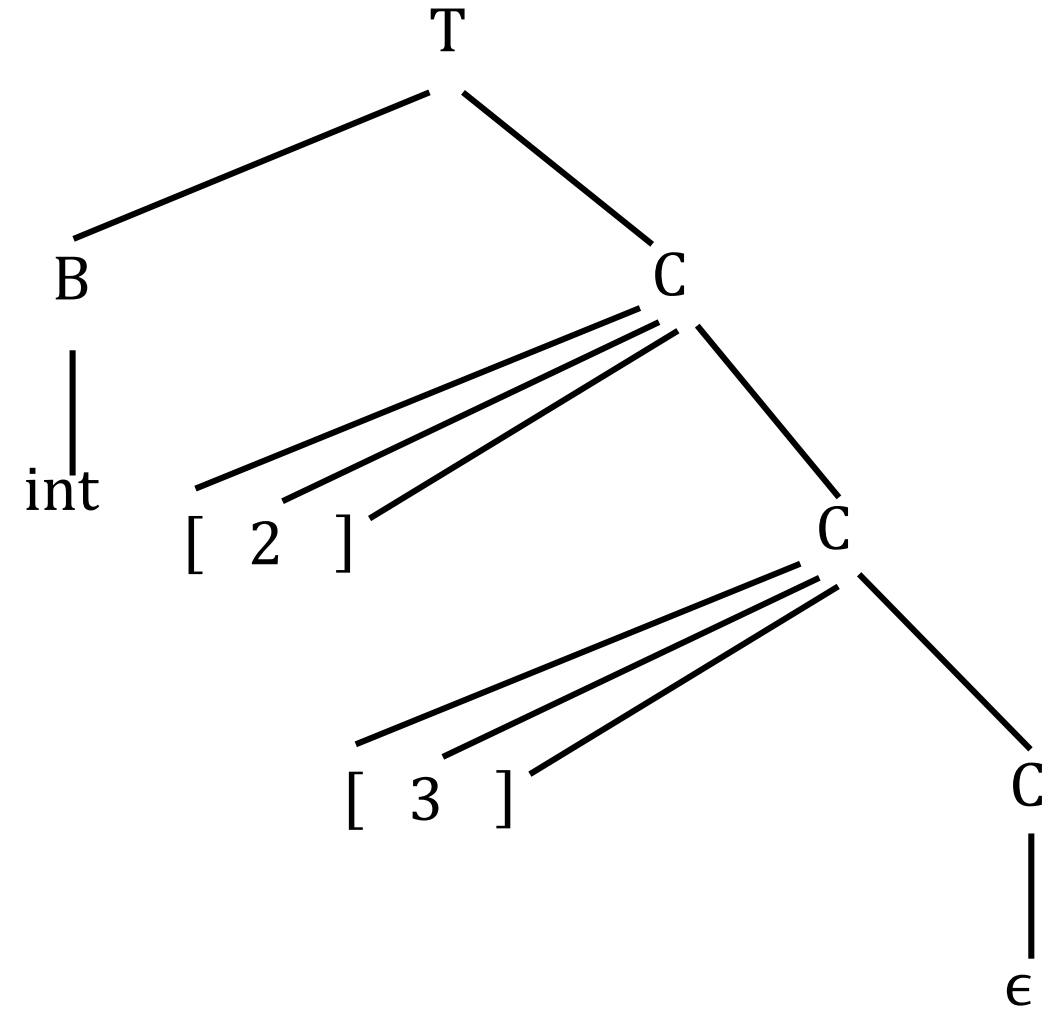
**Computing types and their width**

## Example 6.9 (Cont...)

### SDT of Array Types for int [2] [3]

$T \rightarrow B$       {  $t = B.type$ ;  $w = B.width$ ; }  
           $C$       {  $T.type = C.type$ ;  $T.width = C.width$ ; }  
 $B \rightarrow \text{int}$       {  $B.type = \text{integer}$ ;  $B.width = 4$ ; }  
 $B \rightarrow \text{float}$      {  $B.type = \text{float}$ ;  $B.width = 8$ ; }  
 $C \rightarrow \epsilon$         {  $C.type = t$ ;  $C.width = w$ ; }  
 $C \rightarrow [\text{num}] C_1$  {  $C.type = \text{array}(\text{num.value}, C_1.type)$ ;  
                           $C.width = \text{num.value} \times C_1.width$ ; }

**Computing types and their width**



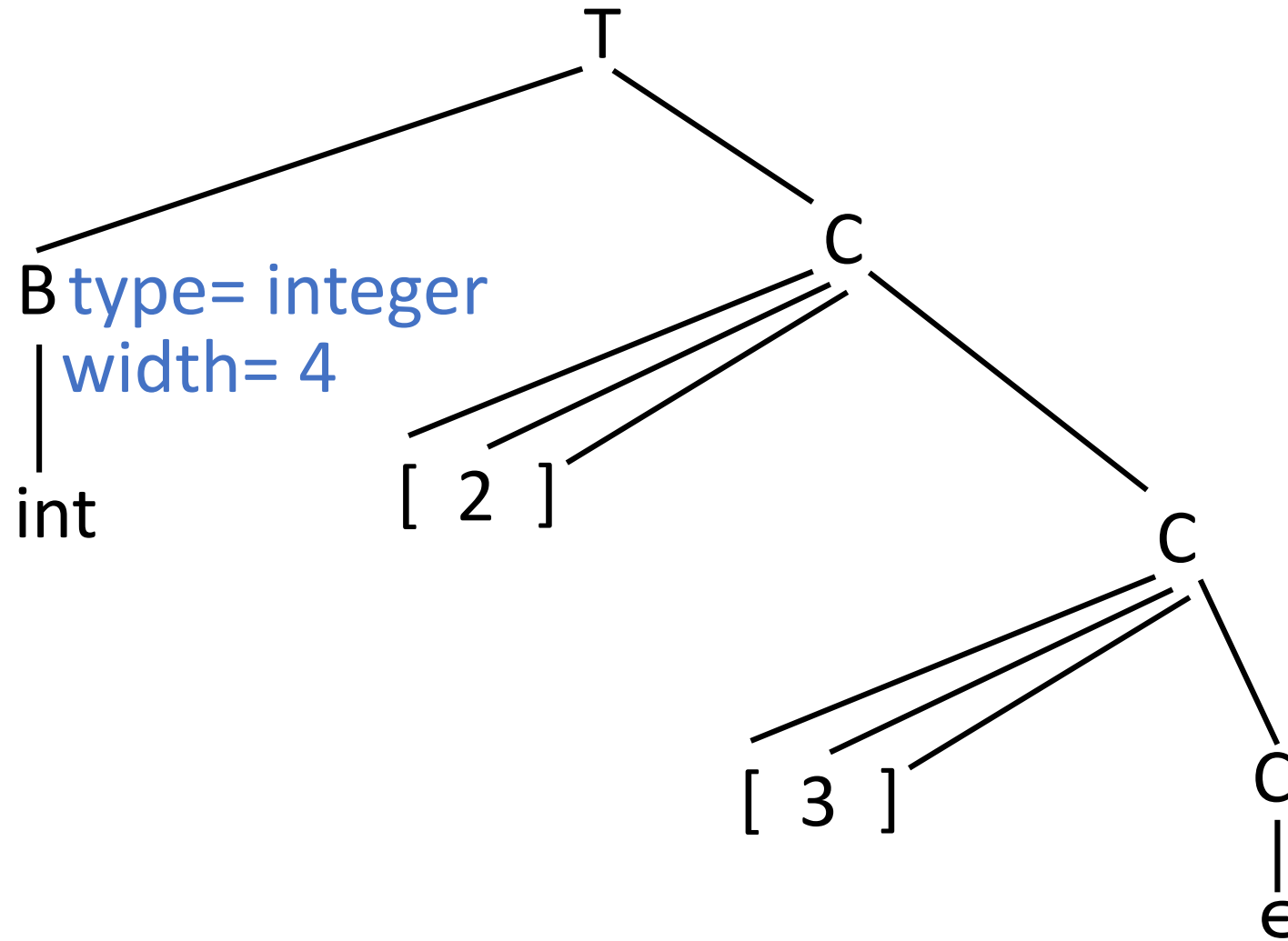
**Parse Tree for int [2] [3]**

# Example 6.9 (Cont...)

## SDT of Array Types for int [2] [3]

$T \rightarrow B$       {  $t = B.type$ ;  $w = B.width$ ; }  
           $C$       {  $T.type = C.type$ ;  $T.width = C.width$ ; }  
 $B \rightarrow \text{int}$     {  $B.type = \text{integer}$ ;  $B.width = 4$ ; }  
 $B \rightarrow \text{float}$    {  $B.type = \text{float}$ ;  $B.width = 8$ ; }  
 $C \rightarrow \epsilon$         {  $C.type = t$ ;  $C.width = w$ ; }  
 $C \rightarrow [\text{num}] C_1$  {  $C.type = \text{array}(\text{num.value}, C_1.type)$ ;  
                           $C.width = \text{num.value} \times C_1.width$ ; }

**Computing types and their width**

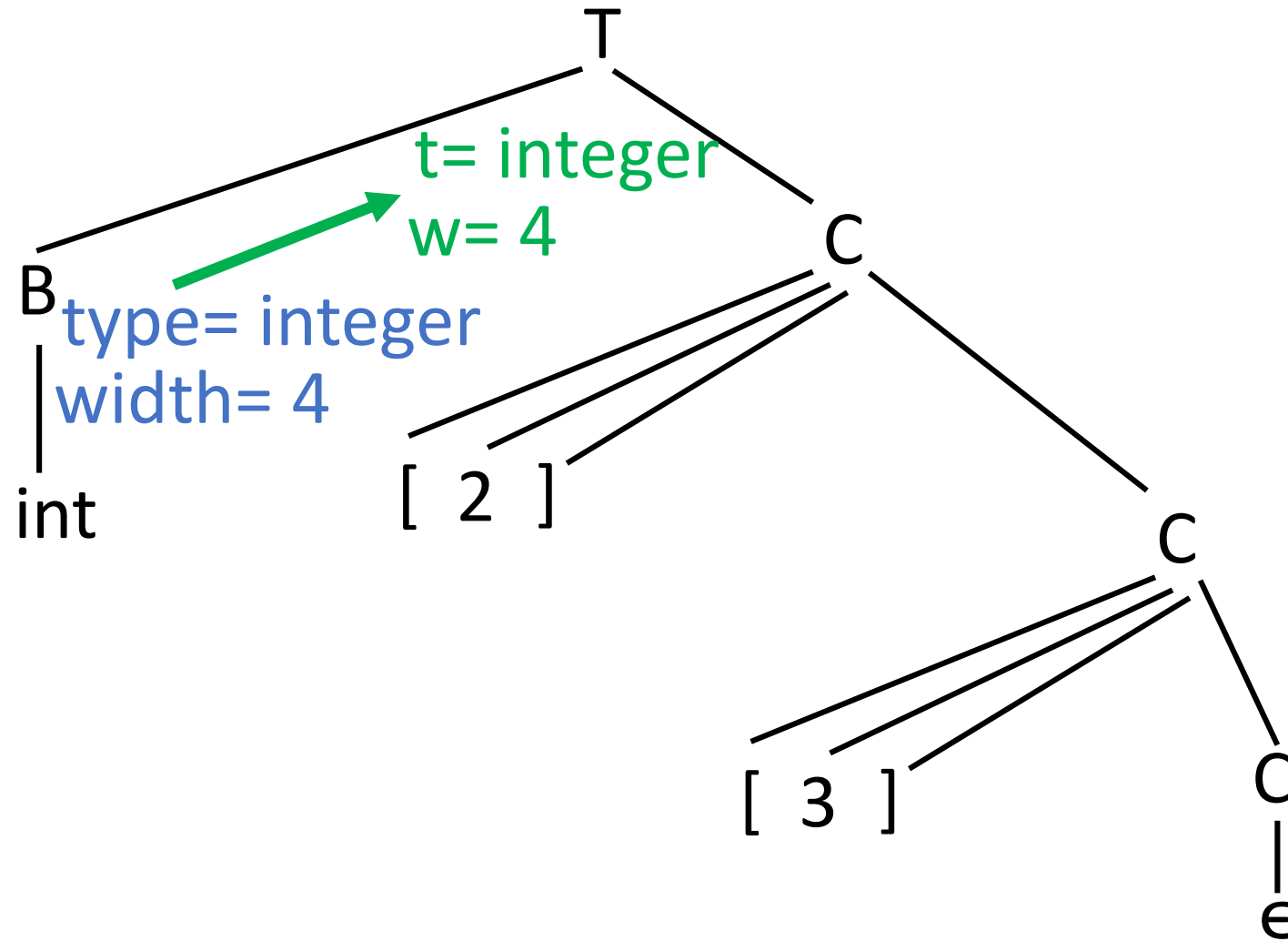


# Example 6.9 (Cont...)

## SDT of Array Types for int [2] [3]

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
$C$	$\{ T.type = C.type; T.width = C.width; \}$
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$
$C \rightarrow [\text{num}] C_1$	$\{ C.type = \text{array}(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width; \}$

Computing types and their width

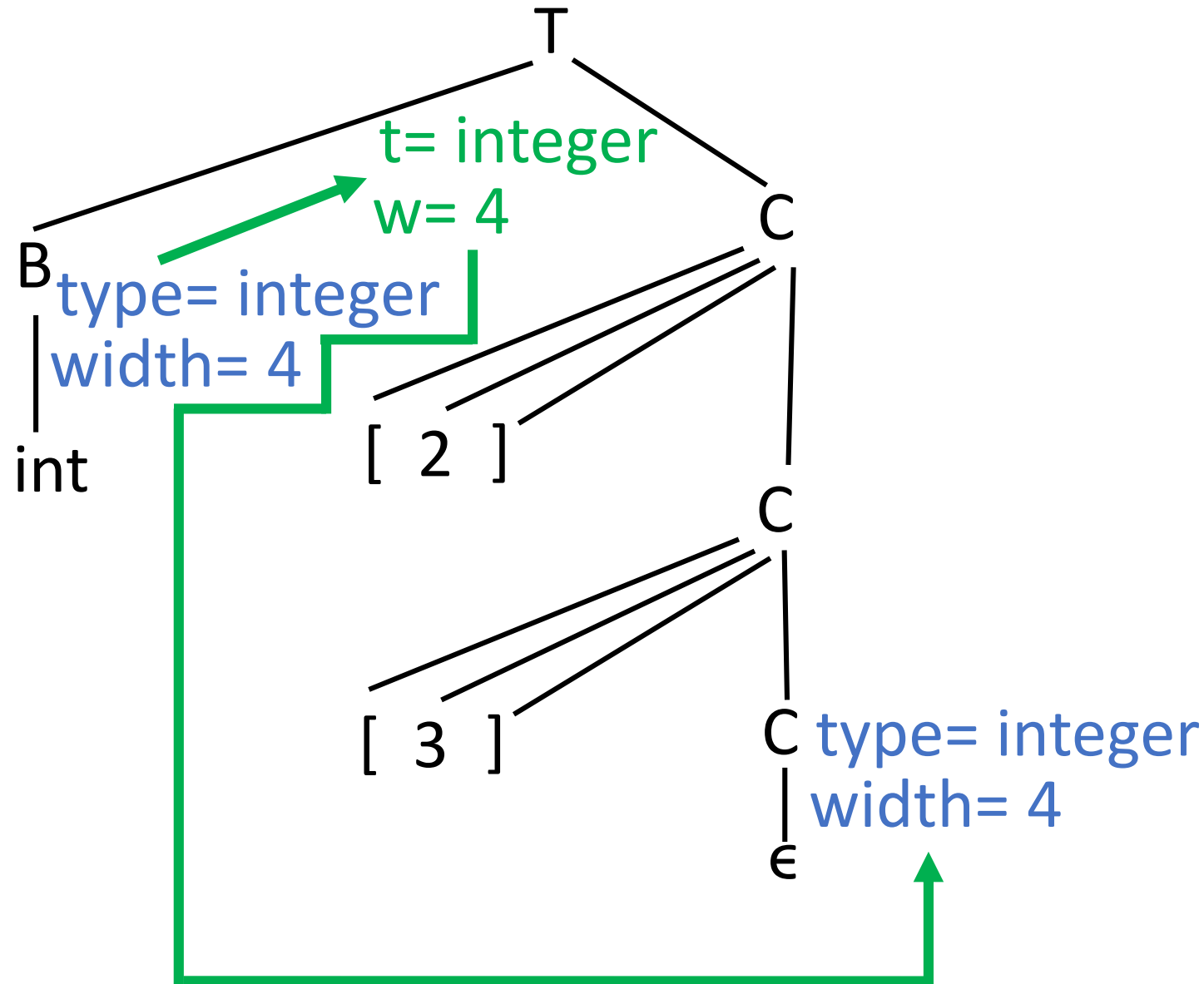


# Example 6.9 (Cont...)

## SDT of Array Types for int [2] [3]

$T \rightarrow B$       {  $t = B.type$ ;  $w = B.width$ ; }  
           $C$       {  $T.type = C.type$ ;  $T.width = C.width$ ; }  
 $B \rightarrow \text{int}$       {  $B.type = \text{integer}$ ;  $B.width = 4$ ; }  
 $B \rightarrow \text{float}$      {  $B.type = \text{float}$ ;  $B.width = 8$ ; }  
 $C \rightarrow \epsilon$         {  $C.type = t$ ;  $C.width = w$ ; }  
 $C \rightarrow [\text{num}] C_1$  {  $C.type = \text{array}(\text{num.value}, C_1.type)$ ;  
                           $C.width = \text{num.value} \times C_1.width$ ; }

**Computing types and their width**

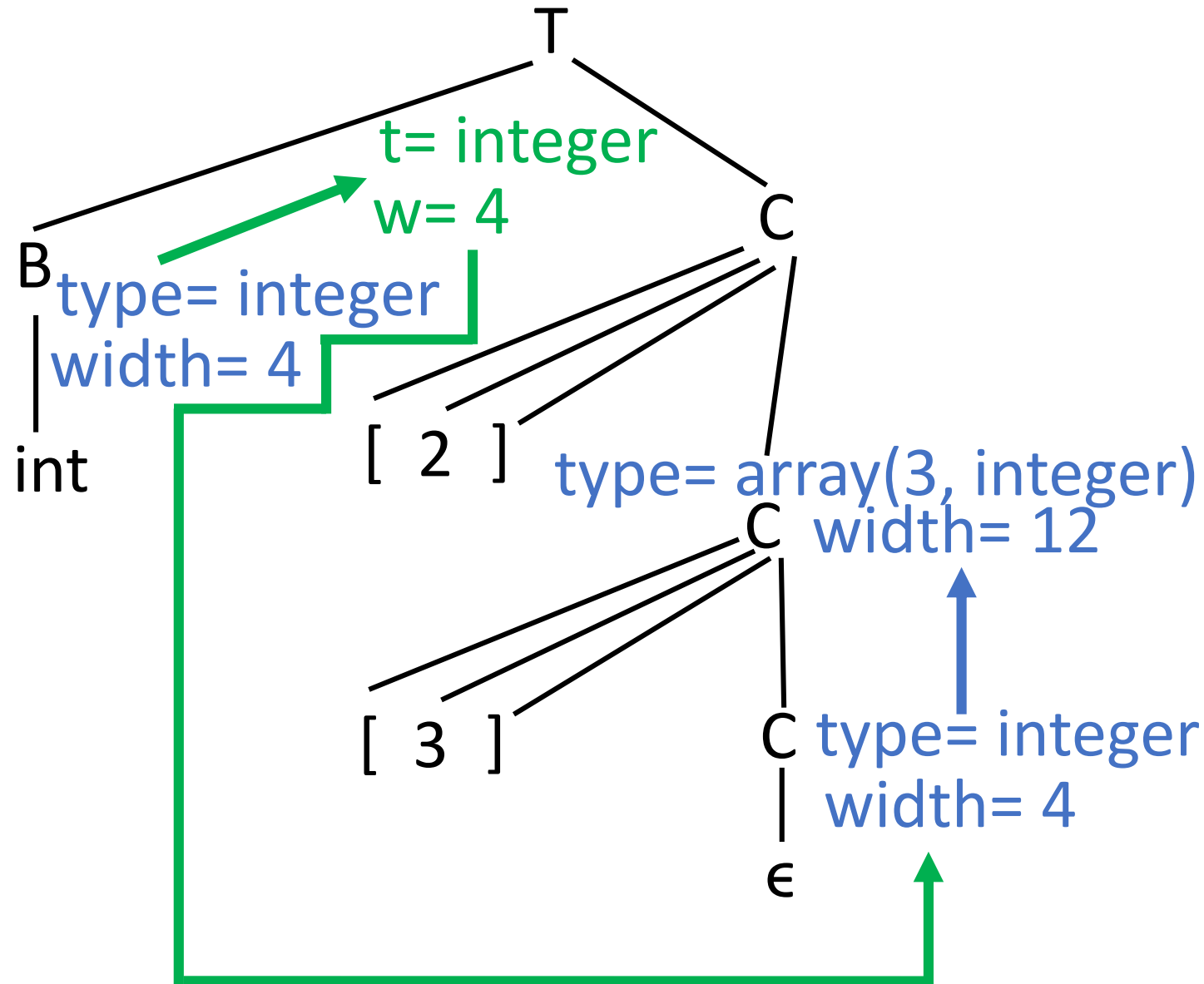


# Example 6.9 (Cont...)

## SDT of Array Types for int [2] [3]

$T \rightarrow B$       {  $t = B.type$ ;  $w = B.width$ ; }  
           $C$       {  $T.type = C.type$ ;  $T.width = C.width$ ; }  
 $B \rightarrow \text{int}$       {  $B.type = \text{integer}$ ;  $B.width = 4$ ; }  
 $B \rightarrow \text{float}$      {  $B.type = \text{float}$ ;  $B.width = 8$ ; }  
 $C \rightarrow \epsilon$         {  $C.type = t$ ;  $C.width = w$ ; }  
 $C \rightarrow [\text{num}] C_1$  {  $C.type = \text{array}(\text{num.value}, C_1.type)$ ;  
                           $C.width = \text{num.value} \times C_1.width$ ; }

**Computing types and their width**

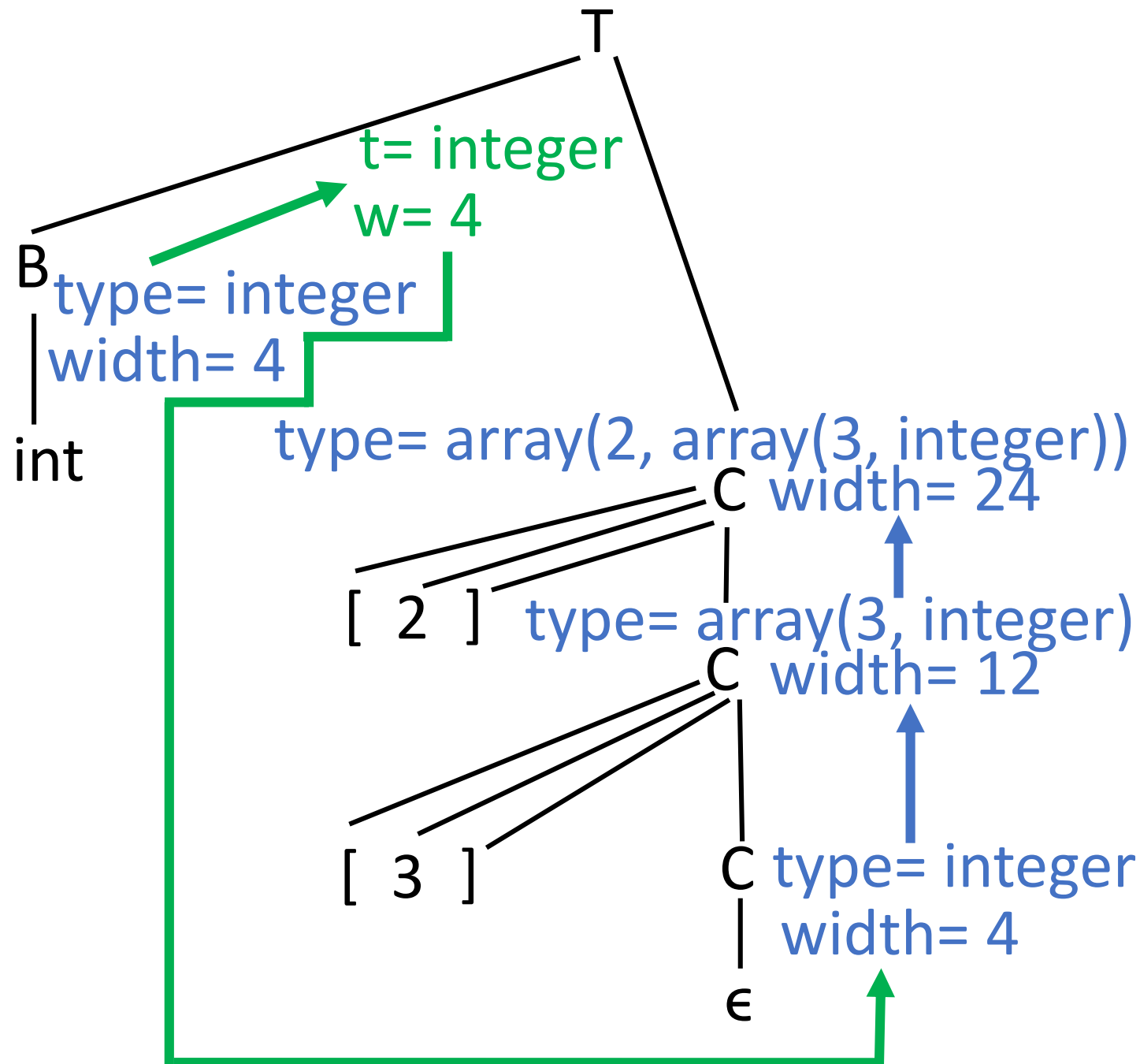


## Example 6.9 (Cont...)

### SDT of Array Types for int [2] [3]

$T \rightarrow B$       {  $t = B.type$ ;  $w = B.width$ ; }  
           $C$       {  $T.type = C.type$ ;  $T.width = C.width$ ; }  
 $B \rightarrow \text{int}$       {  $B.type = \text{integer}$ ;  $B.width = 4$ ; }  
 $B \rightarrow \text{float}$      {  $B.type = \text{float}$ ;  $B.width = 8$ ; }  
 $C \rightarrow \epsilon$         {  $C.type = t$ ;  $C.width = w$ ; }  
 $C \rightarrow [\text{num}] C_1$  {  $C.type = \text{array}(\text{num.value}, C_1.type)$ ;  
                           $C.width = \text{num.value} \times C_1.width$ ; }

**Computing types and their width**

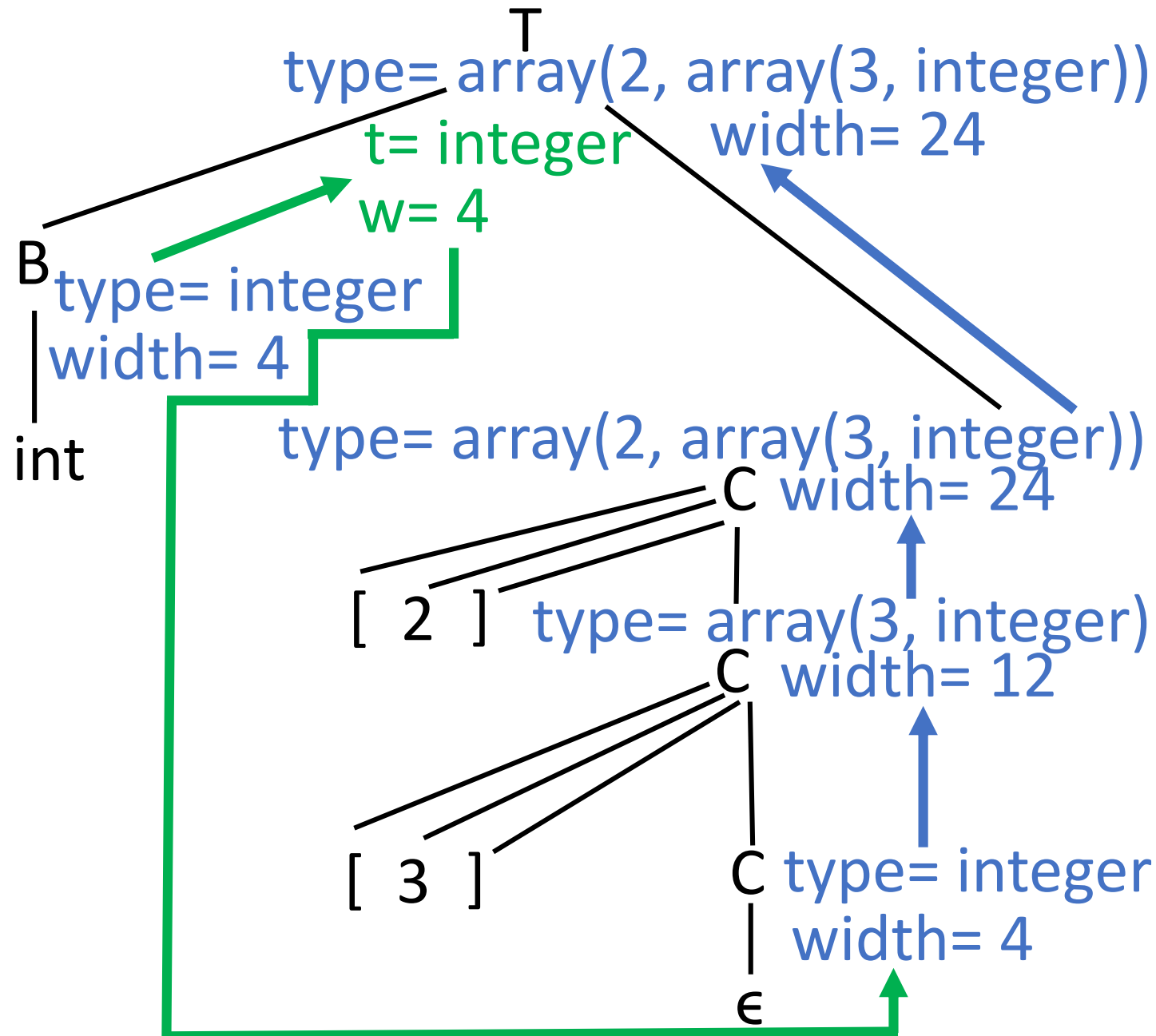


## Example 6.9 (Cont...)

### SDT of Array Types for int [2] [3]

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
$C$	$\{ T.type = C.type; T.width = C.width; \}$
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$
$C \rightarrow [\text{num}] C_1$	$\{ C.type = \text{array}(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width; \}$

Computing types and their width



Syntax-directed translation of array types



# Comparison of Example 6.9 and Example 5.13

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
$C$	$\{ T.type = C.type; T.width = C.width; \}$
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$
$C \rightarrow [\text{num}] C_1$	$\{ C.type = \text{array}(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width; \}$

**Computing types and their width**

PRODUCTION	SEMANTIC RULES
$T \rightarrow B C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \text{int}$	$B.t = \text{integer}$
$B \rightarrow \text{float}$	$B.t = \text{float}$
$C \rightarrow [\text{num}] C_1$	$C.t = \text{array}(\text{num.val}, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

$T$  generates either a basic type or an array type

# Sequences of Declarations

- Languages such as C and Java allow all the declarations in a single procedure to be processed as a group
- We can use a variable say *offset* to keep track of the next available relative address
- The translation scheme of figure deals with a sequence of declarations of the form *T id* where *T* generates a type as in previous figure

$$\begin{aligned} P &\rightarrow D \quad \{ \textit{offset} = 0; \} \\ D &\rightarrow T \textit{id} ; \quad \{ \textit{top.put}(\textit{id.lexeme}, T.\textit{type}, \textit{offset}); \\ &\quad \textit{offset} = \textit{offset} + T.\textit{width}; \} \\ D &\rightarrow D_1 \\ D &\rightarrow \epsilon \end{aligned}$$

Computing the relative addresses of declared names

# Sequences of Declarations (Cont...)

- ❑ The initialization of offset is more evident if the first production appears on one line as:

$P \rightarrow \{\text{offset} = 0;\} D$

- ❑ Nonterminals generating  $\epsilon$ , called marker nonterminals can be used to rewrite productions so that all actions appear at the ends of right sides

- ❑ Using a marker nonterminal M the previous equation can be restated as:

$P \rightarrow M D$

$M \rightarrow \epsilon \{\text{offset} = 0;\}$

# Translation of Expressions

# Three-address code for expressions - SDD

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E ;$	$S.code = E.code \parallel$ $gen(top.get(\mathbf{id.lexeme}) '=' E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new Temp}()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr '=' E_1.addr '+' E_2.addr)$
$  \quad - E_1$	$E.addr = \mathbf{new Temp}()$ $E.code = E_1.code \parallel$ $gen(E.addr '=' 'minus' E_1.addr)$
$  \quad ( E_1 )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$  \quad \mathbf{id}$	$E.addr = top.get(\mathbf{id.lexeme})$ $E.code = ''$

Task-1: Draw Annotated parse tree for  $a = b + -c$  and find S.code

# Incremental Translation - SDT

$$S \rightarrow \text{id} = E ; \quad \{ \text{gen}( \text{top.get}(\text{id.lexeme}) \neq E.addr ); \}$$
$$E \rightarrow E_1 + E_2 \quad \{ E.addr = \text{new Temp}(); \\ \text{gen}(E.addr \neq E_1.addr \neq + E_2.addr); \}$$
$$\mid - E_1 \quad \{ E.addr = \text{new Temp}(); \\ \text{gen}(E.addr \neq \text{'minus'} E_1.addr); \}$$
$$\mid ( E_1 ) \quad \{ E.addr = E_1.addr; \}$$
$$\mid \text{id} \quad \{ E.addr = \text{top.get}(\text{id.lexeme}); \}$$

Task-2: Draw  
Annotated parse  
tree for  $a = b + -c$   
and output the  
three address  
code.

# Translation of Array References - SDT

$S \rightarrow \text{id} = E ; \quad \{ \text{gen}(\text{top.get}(\text{id.lexeme}) \neq E.\text{addr}); \}$

$| \quad L = E ; \quad \{ \text{gen}(L.\text{array.base} '[' L.\text{addr} ']' \neq E.\text{addr}); \}$

$E \rightarrow E_1 + E_2 \quad \{ E.\text{addr} = \text{new Temp}();$   
 $\quad \text{gen}(E.\text{addr} \neq E_1.\text{addr} + E_2.\text{addr}); \}$

$| \quad \text{id} \quad \{ E.\text{addr} = \text{top.get}(\text{id.lexeme}); \}$

$| \quad L \quad \{ E.\text{addr} = \text{new Temp}();$   
 $\quad \text{gen}(E.\text{addr} \neq \text{array name} '[' L.\text{addr} ']); \}$

$L \rightarrow \text{id} [ E ] \quad \{ L.\text{array} = \text{top.get}(\text{id.lexeme});$   
 $\quad L.\text{type} = L.\text{array.type.elem};$   
 $\quad L.\text{addr} = \text{new Temp}();$   
 $\quad \text{gen}(L.\text{addr} \neq E.\text{addr} * L.\text{type.width}); \}$

$| \quad L_1 [ E ] \quad \{ L.\text{array} = L_1.\text{array};$   
 $\quad L.\text{type} = L_1.\text{type.elem};$   
 $\quad t = \text{new Temp}();$   
 $\quad L.\text{addr} = \text{new Temp}();$   
 $\quad \text{gen}(t \neq E.\text{addr} * L.\text{type.width});$   
 $\quad \text{gen}(L.\text{addr} \neq L_1.\text{addr} + t); \}$

Task-3: Draw Annotated parse tree for  $C + A[i][j]$  and Three-Address code considering  $A$  is an array declaration of  $2 \times 3$ .

# Practice

- Use the Translation of Array References for the following assignment to generate Three-address code.
  - $X = a[i] + b[j]$
  - $X = a[i][j] + b[i][j]$



End