Chapter-06

Intermediate-Code Generation

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

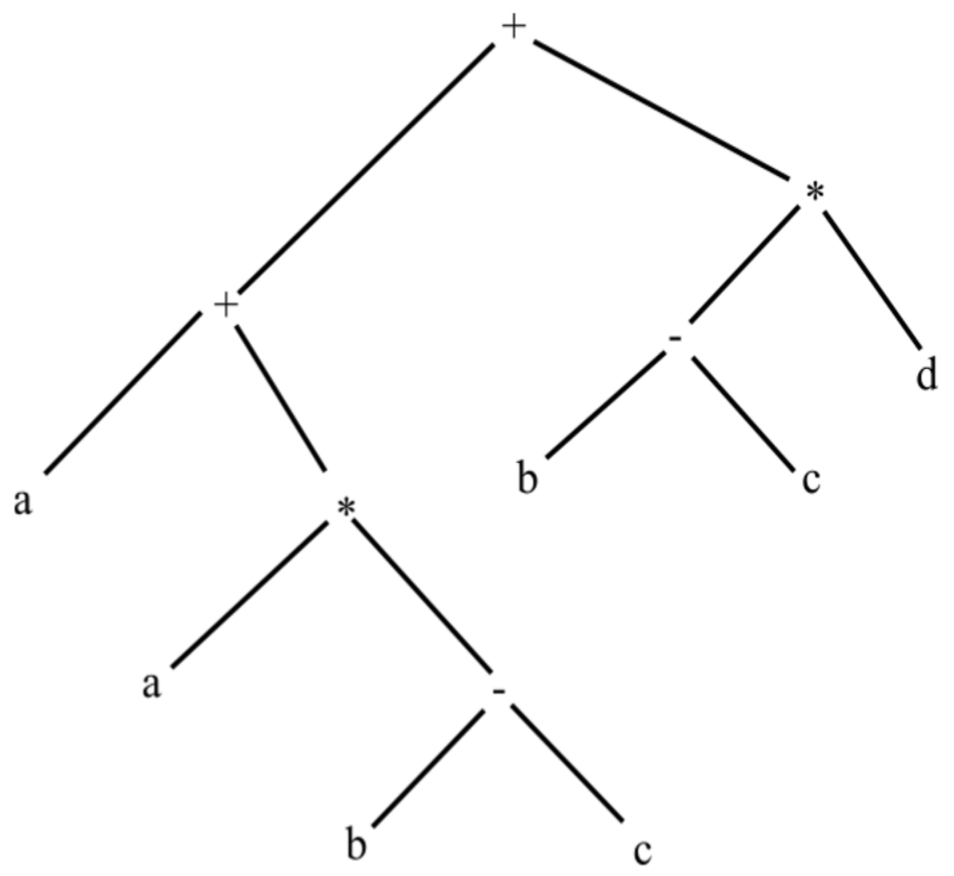
Directed Acyclic Graphs for Expressions

- 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

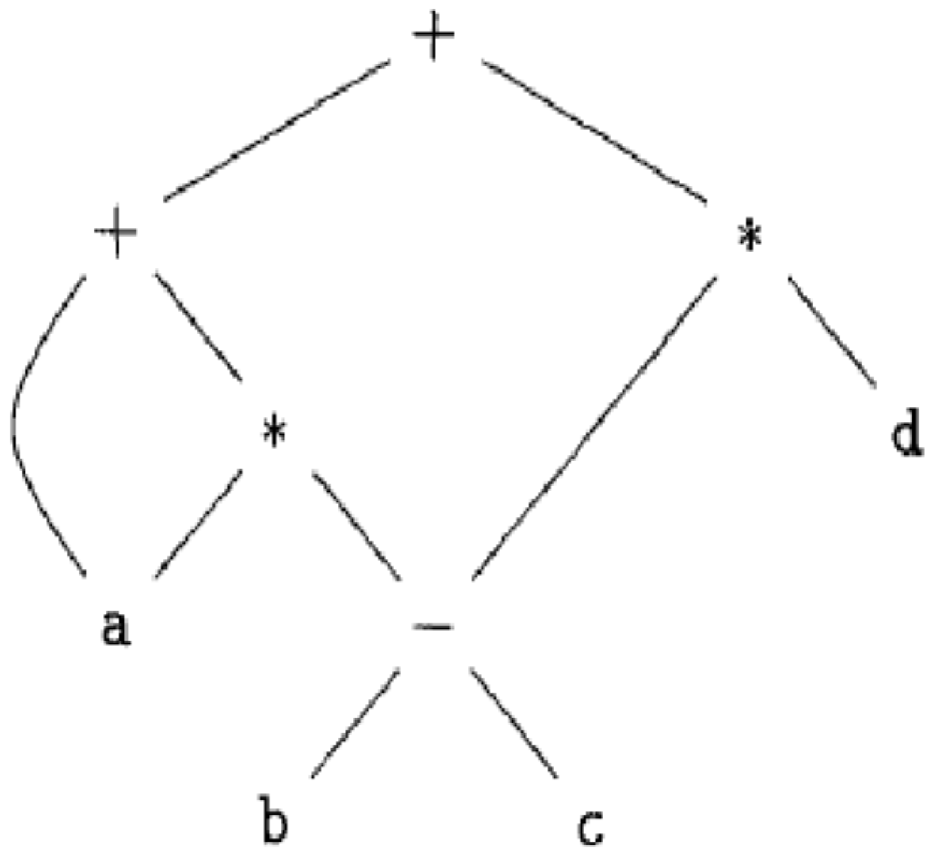
Example 6.1

Figure shows the **Syntax Tree** and **DAG** for the expression,

a + a∗(b−c) + (b−c)∗d



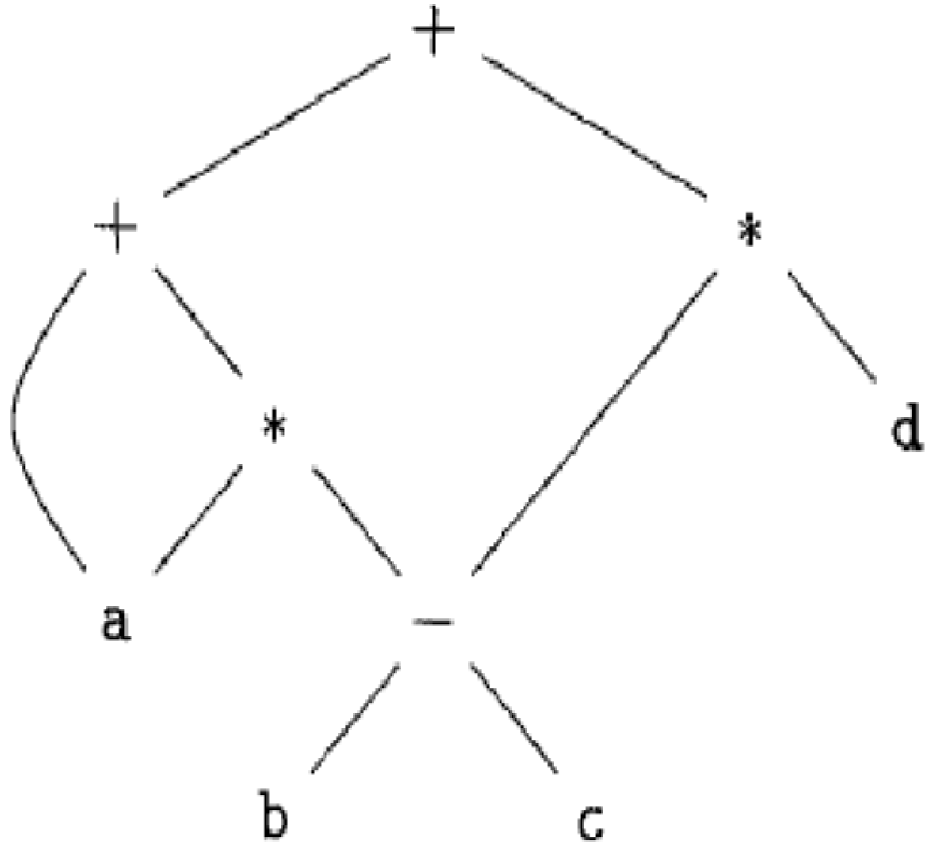
Syntax Tree



DAG

Example 6.1(Cont…)

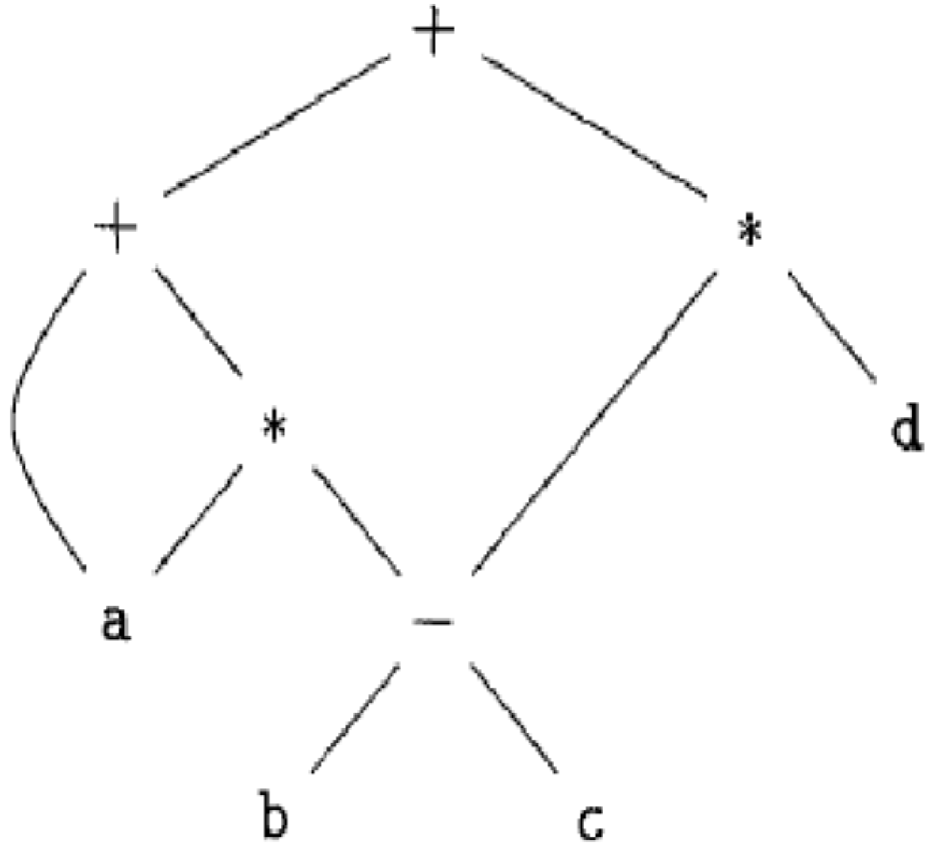
- The leaf for ‘a’ has two parents, because ‘a’ appears twice in the expression
- More interestingly, the two occurrences of the common subexpression ‘b−c’ are represented by one node, the node labeled ‘−’



DAG

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’



DAG

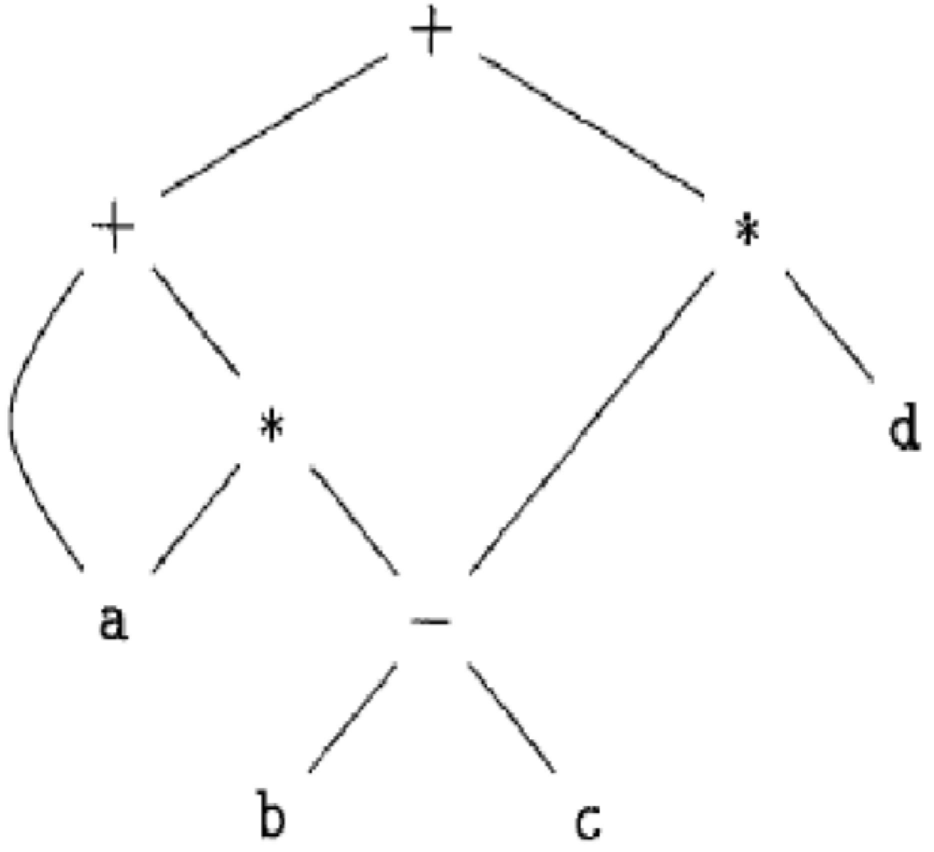
Example 6.1(Cont…)

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

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

Example 6.2

The sequence of steps shown in figure constructs the DAG



DAG

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

Steps for Constructing DAG

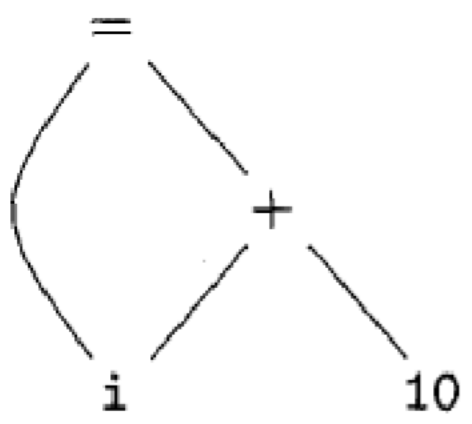
The Value-Number Method for Constructing DAG’s

- 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

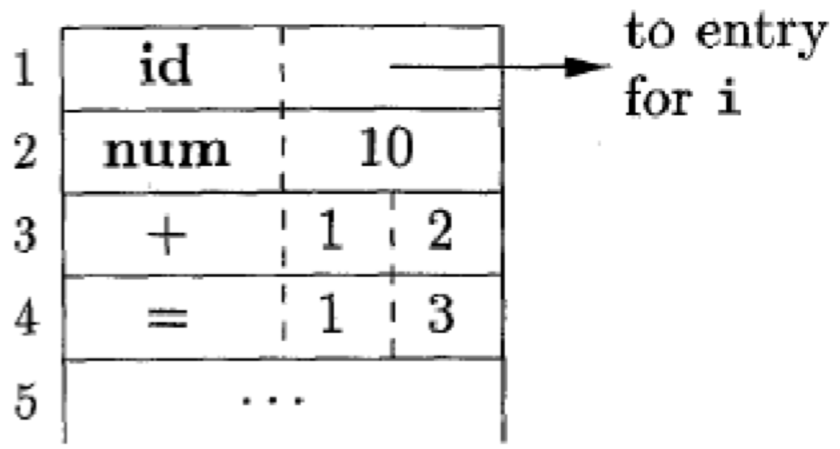
The Value-Number Method for Constructing DAG’s(Cont…)

- Leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and interior nodes have two additional fields indicating the left and right children
- 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…)



(a) DAG

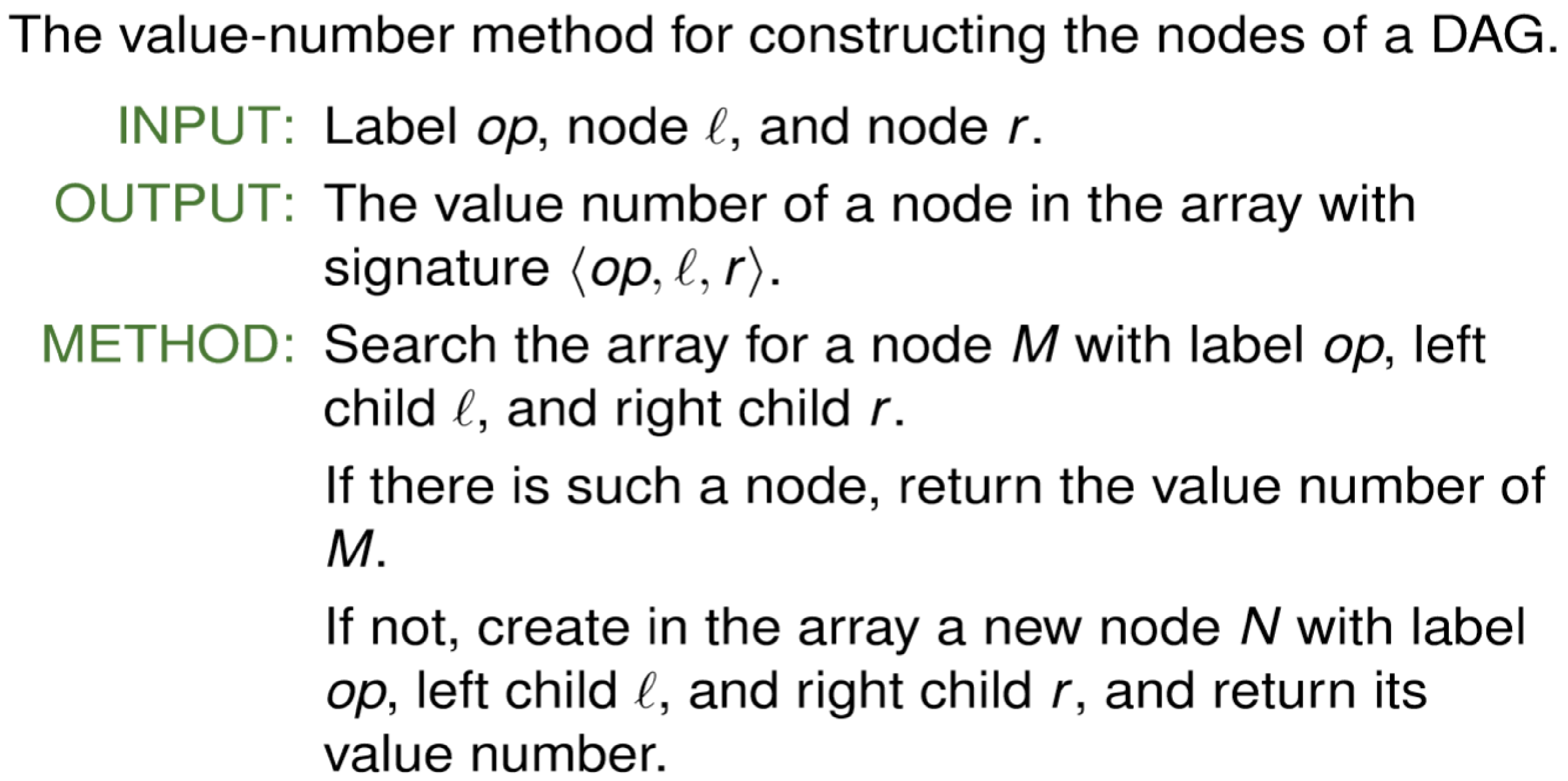


1	id	→ to entry for i	
2	num	10	
3	+	1	2
4	=	1	3
5	⋯		

(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 ℓ, and node *r*.

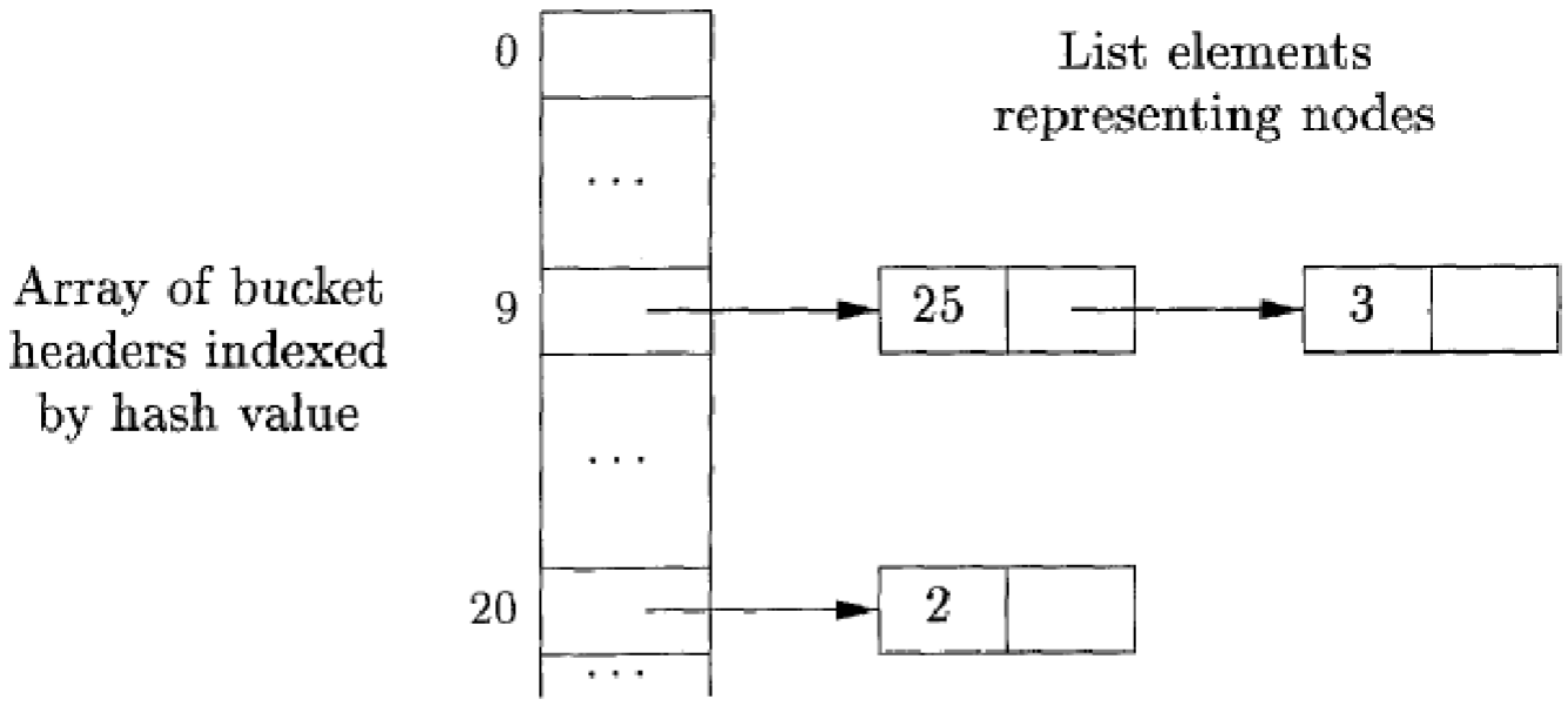
OUTPUT: The value number of a node in the array with signature $\langle op, \ell, r \rangle$.

METHOD: Search the array for a node *M* with label *op*, left child 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.

The Value-Number Method for Constructing DAG’s(Cont…)


0
…
Array of bucket headers indexed by hash value
9
…
20
…
List elements representing nodes
25
3
2

Data structure for searching buckets

Three-Address Code

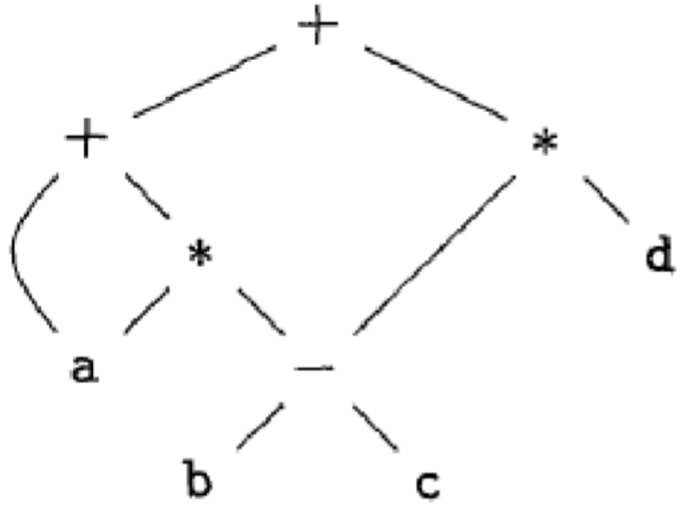
- In three-address code, there is at most one operator on the right side of an instruction
- Thus a source-language expression like ‘$x + y * z$’ might be translated into the sequence of three-address instructions
    - $t1 = y * z$
    - $t2 = x + t1$

  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

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

(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
  - Constant
  - Compiler-generated temporary

Addresses and Instructions(Cont…)

Here is a list of the common three-address instruction forms:

- 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
- Assignments of the form x = 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

Addresses and Instructions(Cont…)

Here is a list of the common three-address instruction forms:

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

Addresses and Instructions(Cont…)

Here is a list of the common three-address instruction forms:

- Conditional jumps of the form if x goto L and if False 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…)

Here is a list of the common three-address instruction forms:

- 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

Addresses and Instructions(Cont…)

Here is a list of the common three-address instruction forms:

- 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…)

- Their typical use is as the sequence of three-address instructions
  - param X_1
  - param X_2
  - ...
  - param X_n
  - 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

Addresses and Instructions(Cont…)

- 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 = 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
$t_1$ = call g, 1
param $t_1$
$t_2$ = call f, 2
n = $t_2$
```

Addresses and Instructions(Cont…)

Here is a list of the common three-address instruction forms:

- 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…)

Here is a list of the common three-address instruction forms:

- 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, perhaps a temporary, that denotes an expression with an l-value such as A[i][j] and x is a pointer name or temporary

Addresses and Instructions(Cont…)

Here is a list of the common three-address instruction forms:

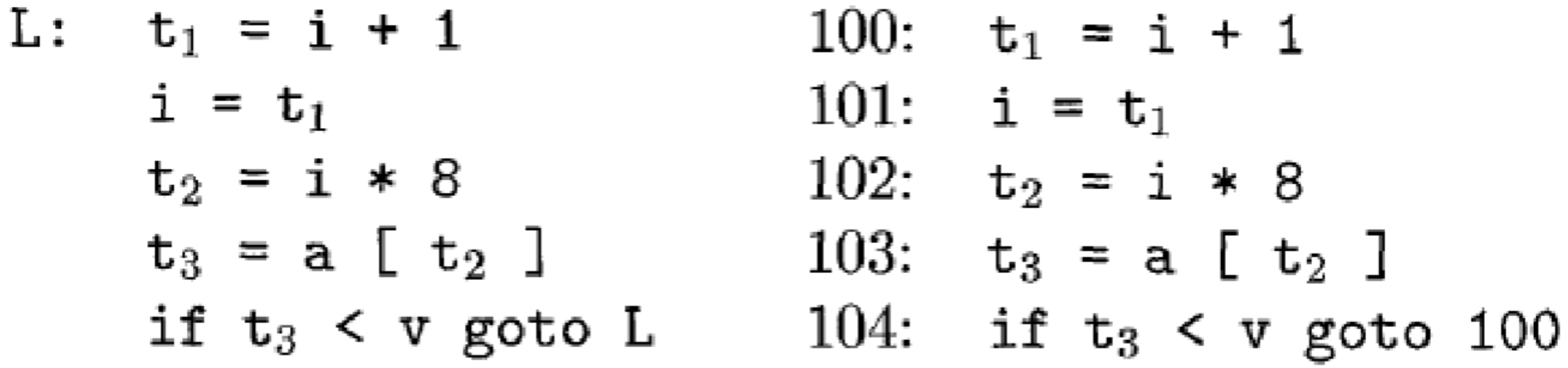
- Address and pointer assignments of the form $x = \&y$, $x = *y$ and $*x = y$
  - In the instruction $x = *y$, presumably y is a pointer or a temporary whose r-value is a location
  - The r-value of x is made equal to the contents of that location
  - Finally $*x = y$ sets the r-value of the object pointed to by x to the r-value of y

Example 6.5

- Consider the statement

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

- Two possible translations of this statement are shown in figure



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

(a) Symbolic labels. (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, arg1, arg2, 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 arg1, z in arg2, and x in result.

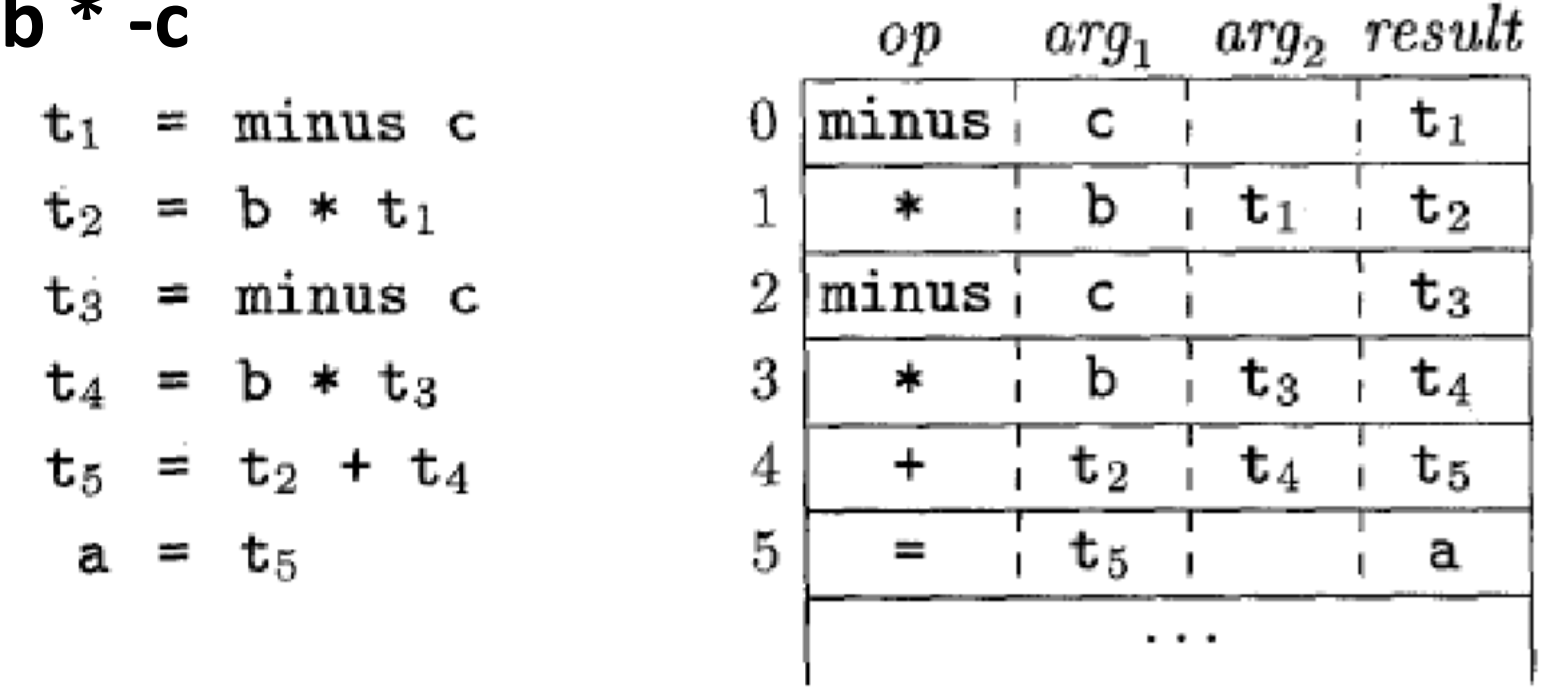
Quadruples(Cont…)

- The following are some exceptions to this rule:
- Instructions with unary operators like x = minus y or x = y do not use arg2
- For a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied
- Operators like param use neither arg2 nor result
- Conditional and unconditional jumps put the target label in result

Example 6.6

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

a = b * -c + b * -c



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

	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
		. . .		

(a) Three-address code (b) Quadruples

Three-address code and its quadruple representation

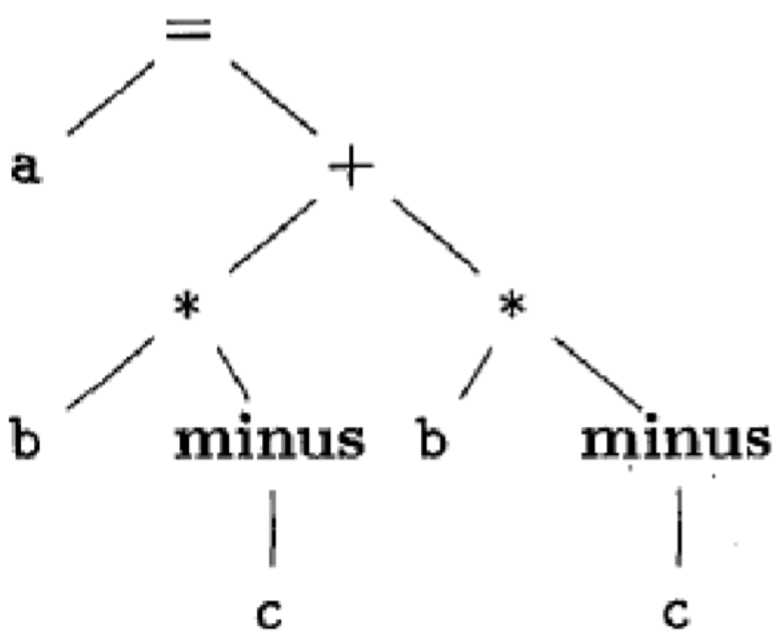
Triples

- A triple has only three fields, which we call op, arg1, arg2
- 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 a triple representation would refer to position (0)
- Parenthesized numbers represent pointers into the triple structure itself

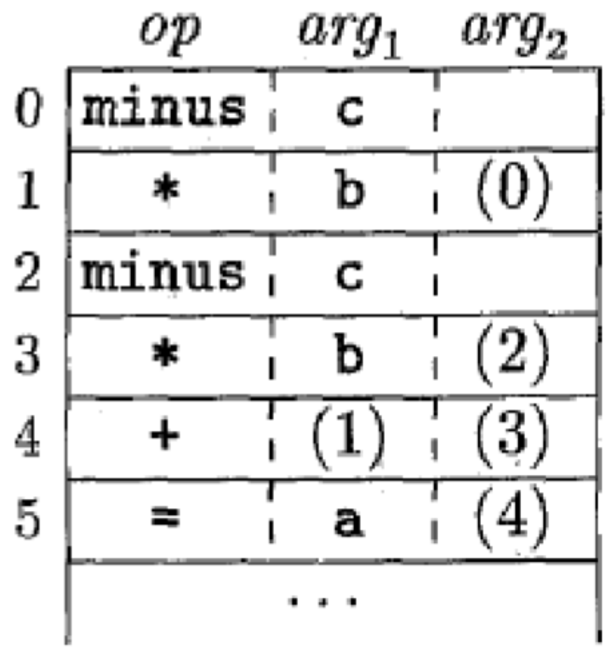
Example 6.7

Figure shows the **Syntax Tree** and **Triples** for the expression,

a = b * -c + b * -c



(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

Triples(Cont…)

- In the triple representation in (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
- 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.

Benefit of Quadruples over Triples

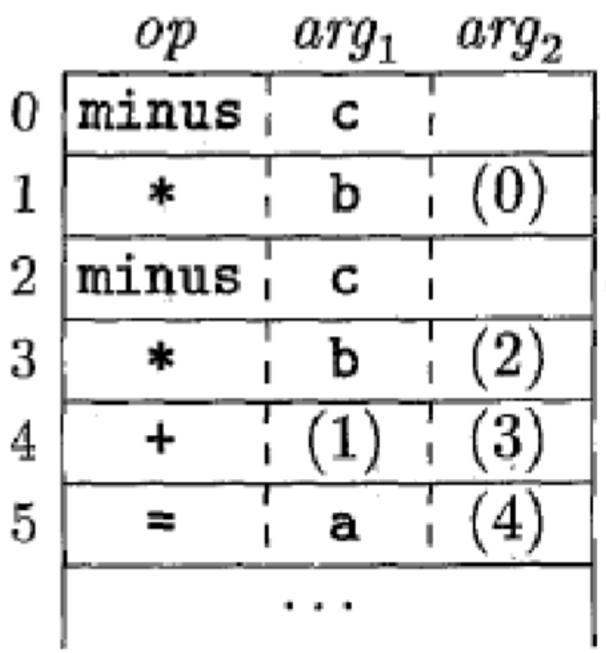
- A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around
- With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change
- With triples, the result of an operation is referred to 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

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

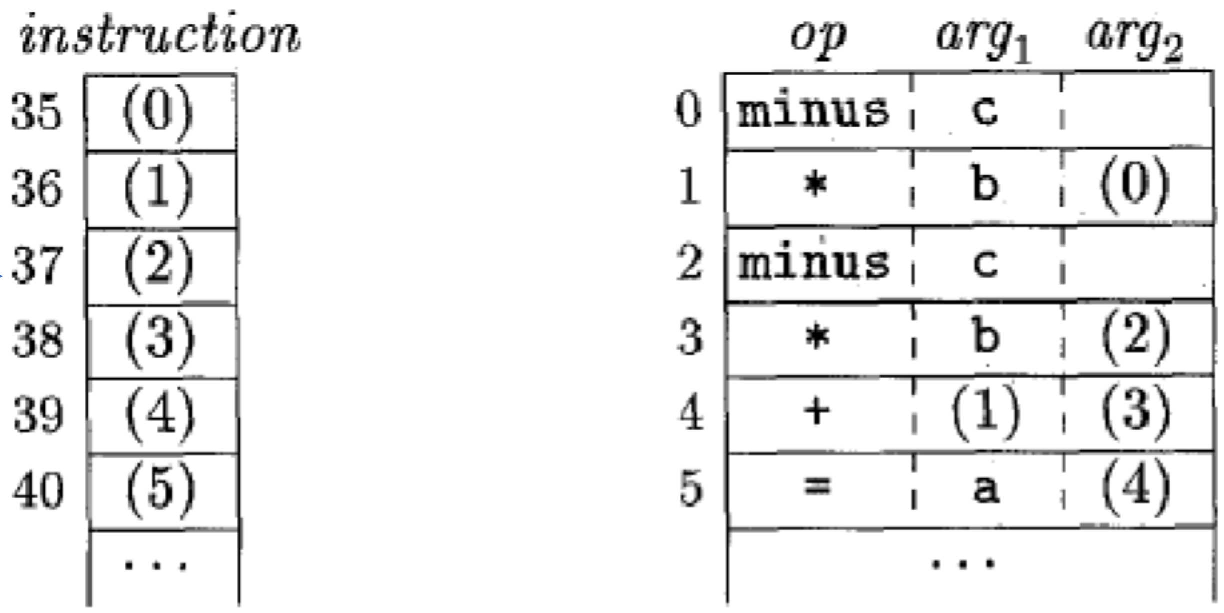
Indirect Triples(Cont…)

- Then, the triples in (b) might be represented as in this figure



	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



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

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		…	

Indirect triples representation of three-address code

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…)

- Figure 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 ϕ-function to combine the two definitions of x:

if (flag) x_1 = -1; else x_2 = 1;

$x_3 = \phi(x_1, x_2)$

- 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 **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

Practice Problem-1

Expression: **((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))**

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

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…)

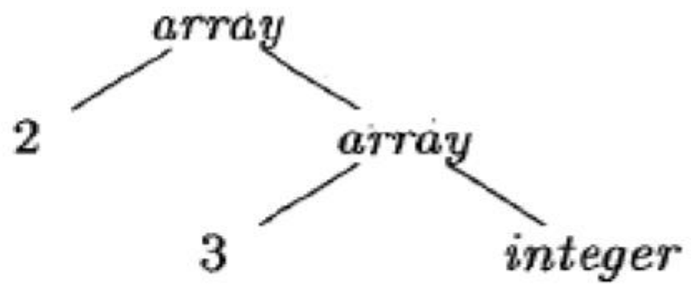
- The applications of types can be grouped under checking and translation
  - **Translation Applications:** From the type of a name, a compiler can determine the storage that will be needed for that name at run time
  - Type information is also needed to calculate the address denoted by an array reference,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

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



Type expression for **int**[2][3]

- 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;
int i;
float m;
void main()
```

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];

  array float[2][3];

  array float[2][3][4];

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:

```
float x;
record (float x; float y;) p;
record (int tag; float x; float y;) q;
```

Type Expressions(Cont…)

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

function int f(float x);
float→int

function float g(int a, float b, int c);
int×float×int→float

Type Expressions(Cont…)

- We shall use the following definition of type expressions:
  - If s and t are type expressions, then their Cartesian product s×t is a type expression
  - They can be used to represent a list or tuple of types (e.g., for function parameters)
  - We assume that **×** associates to the left and that it has higher precedence than **→**.
  - **Type expressions may contain variables whose values are type expressions**
  - A convenient way to represent a type expression is to use a graph

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

Name Equivalence(Cont…)

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

Name Equivalence(Cont…)

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

- This means that statements such as:
    - **X = Y** and **M = N** would be valid but
    - Statements such as **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{                    struct T{
        int a;                        int a;
        float b;                      float b;
        }X,Y;                         }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 and will give error for above assignment)

Declarations

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

$$\begin{array}{lcl} D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\ T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\ B & \rightarrow & \textbf{int} \mid \textbf{float} \\ C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C \end{array}$$

Declarations(Cont…)

$$\begin{array}{lcl} D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\ T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\ B & \rightarrow & \textbf{int} \mid \textbf{float} \\ C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C \end{array}$$

- 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…)

$$\begin{aligned} D &\rightarrow T\ \textbf{id}\ ;\ D \mid \epsilon \\ T &\rightarrow B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\ B &\rightarrow \textbf{int} \mid \textbf{float} \\ C &\rightarrow \epsilon \mid [\ \textbf{num}\ ]\ C \end{aligned}$$

- 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

$$\begin{array}{lcl} D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\ T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\ B & \rightarrow & \textbf{int} \mid \textbf{float} \\ C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C \end{array}$$

- 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; }**

Storage Layout for Local Names

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

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

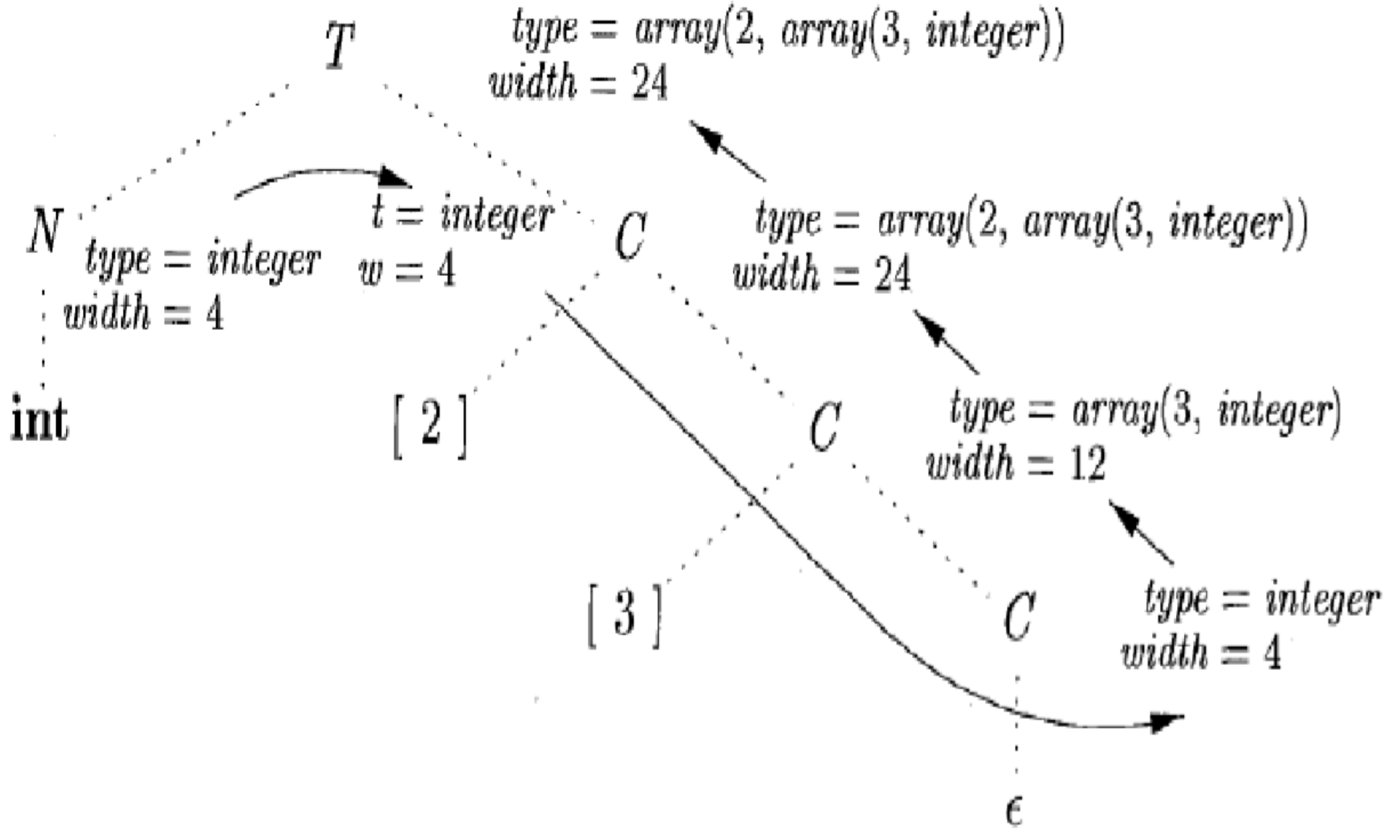
Computing types and their widths

Example 6.9

The parse tree for the type **int [2][3]** along with SDT is shown in the following figure

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

Computing types and their widths



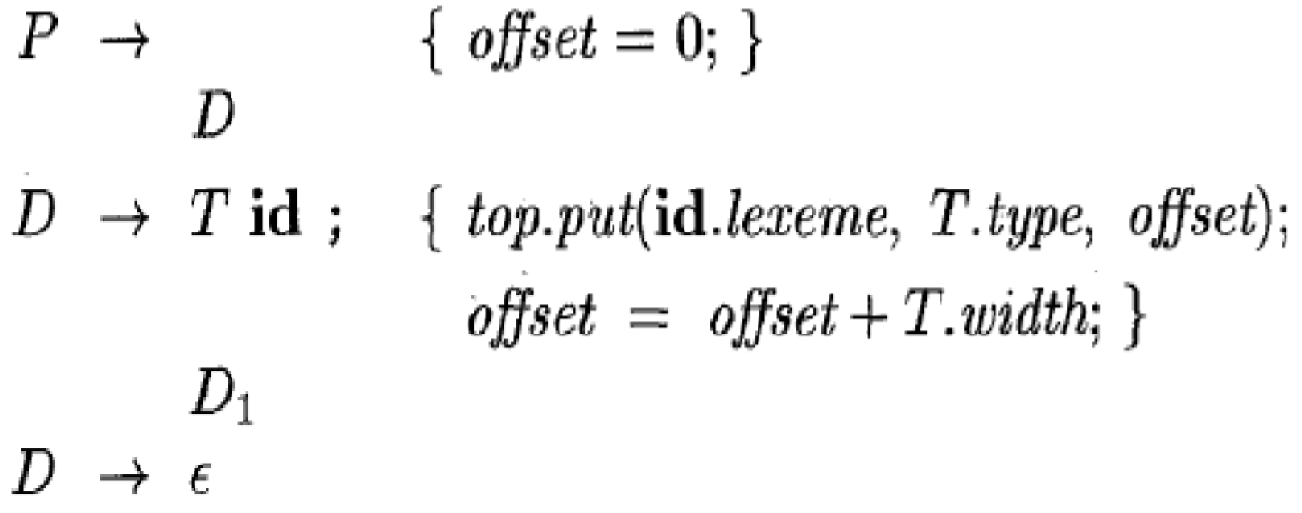
Syntax-directed translation of array types

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

Sequences of Declarations(Cont…)

- 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



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

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:

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

- Nonterminals generating **ϵ**, 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:

$\mathbf{P \rightarrow M\ D}$

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

Thank
you