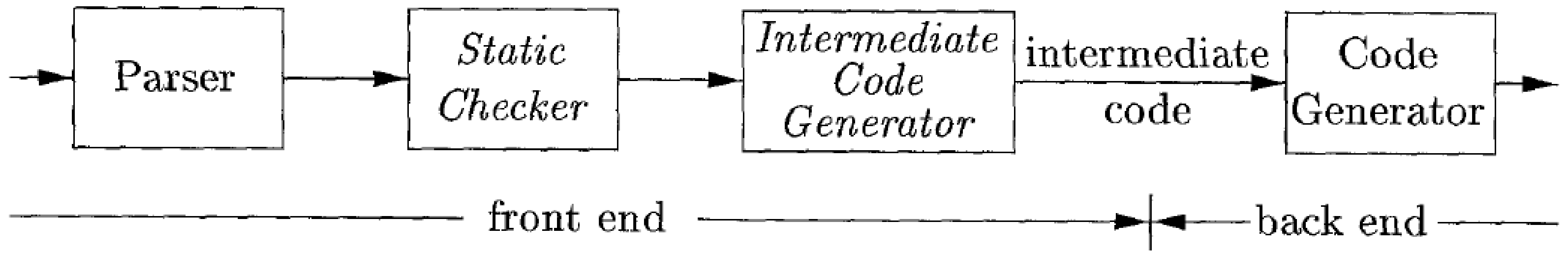
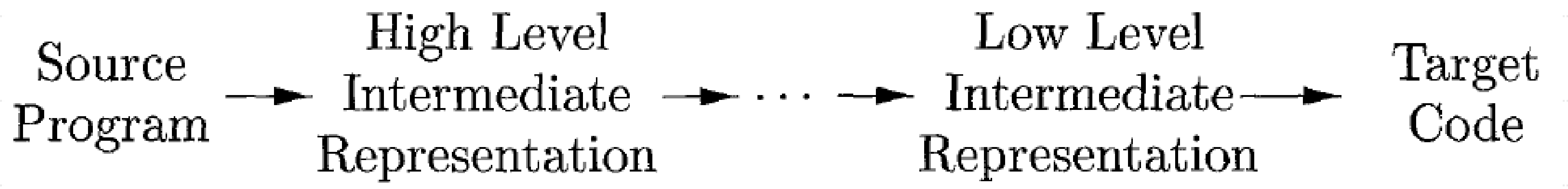
Chapter 10: Intermediate-Code Generation

陳奇業 成功大學資訊工程系

- This chapter deals with intermediate representations, static type checking, and intermediate code generation.

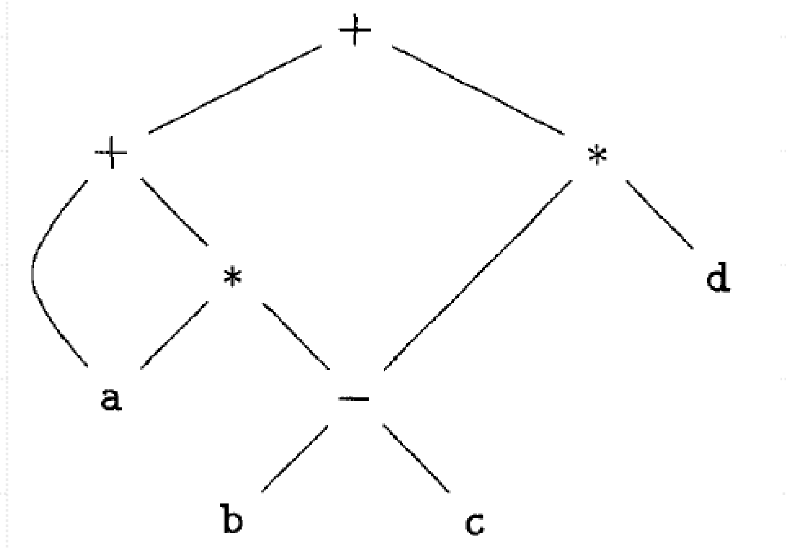


- In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations. High-level representations are close to the source language and low-level representations are close to the target machine.


Source Program
High Level Intermediate Representation
· · ·
Low Level Intermediate Representation
Target Code

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.



Directed Acyclic Graphs for Expressions

- Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators.
- The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.

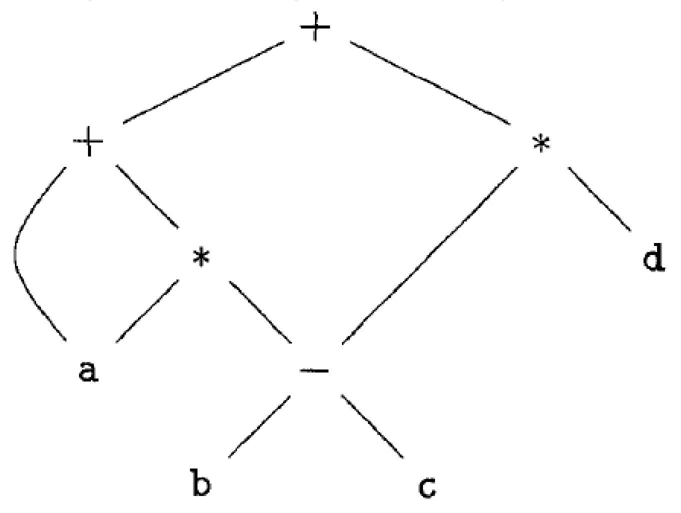


Figure 6.3: Dag for the expression $a + a * (b - c) + (b - c) * d$

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

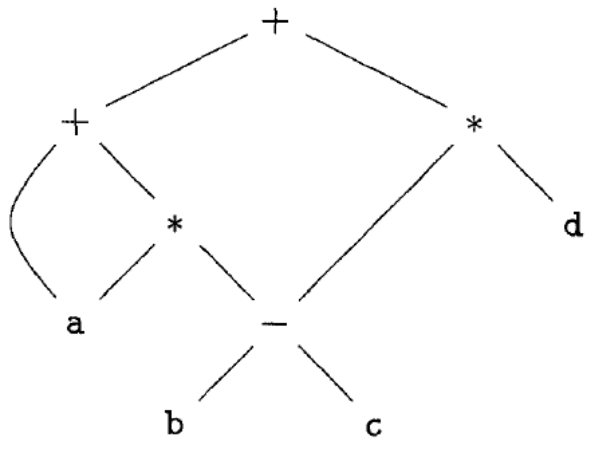
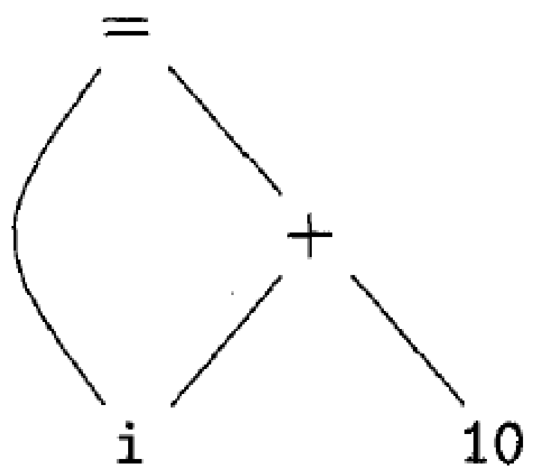


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

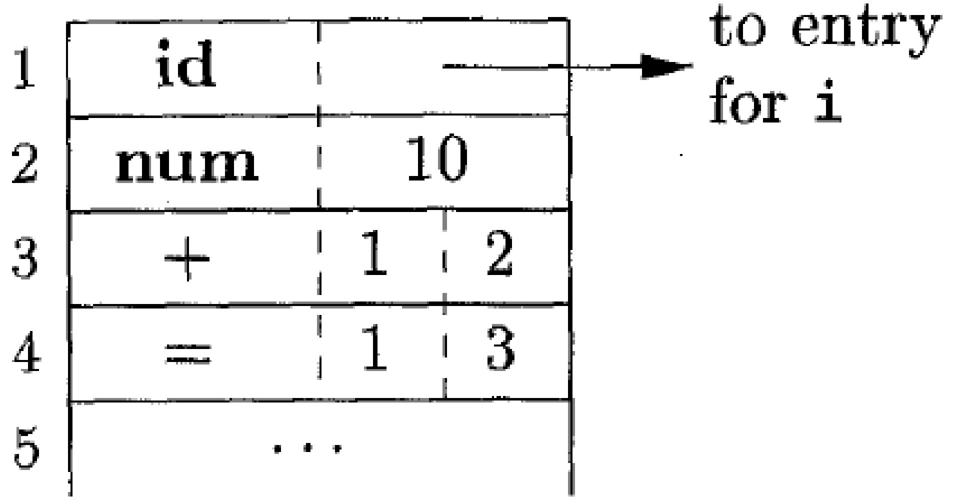
1) $p_1 = \textit{Leaf}\,(\mathbf{id}, \textit{entry-a})$
2) $p_2 = \textit{Leaf}\,(\mathbf{id}, \textit{entry-a}) = p_1$
3) $p_3 = \textit{Leaf}\,(\mathbf{id}, \textit{entry-b})$
4) $p_4 = \textit{Leaf}\,(\mathbf{id}, \textit{entry-c})$
5) $p_5 = \textit{Node}('-', p_3, p_4)$
6) $p_6 = \textit{Node}('*',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

The Value-Number Method for Constructing DAG's

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



(a) DAG

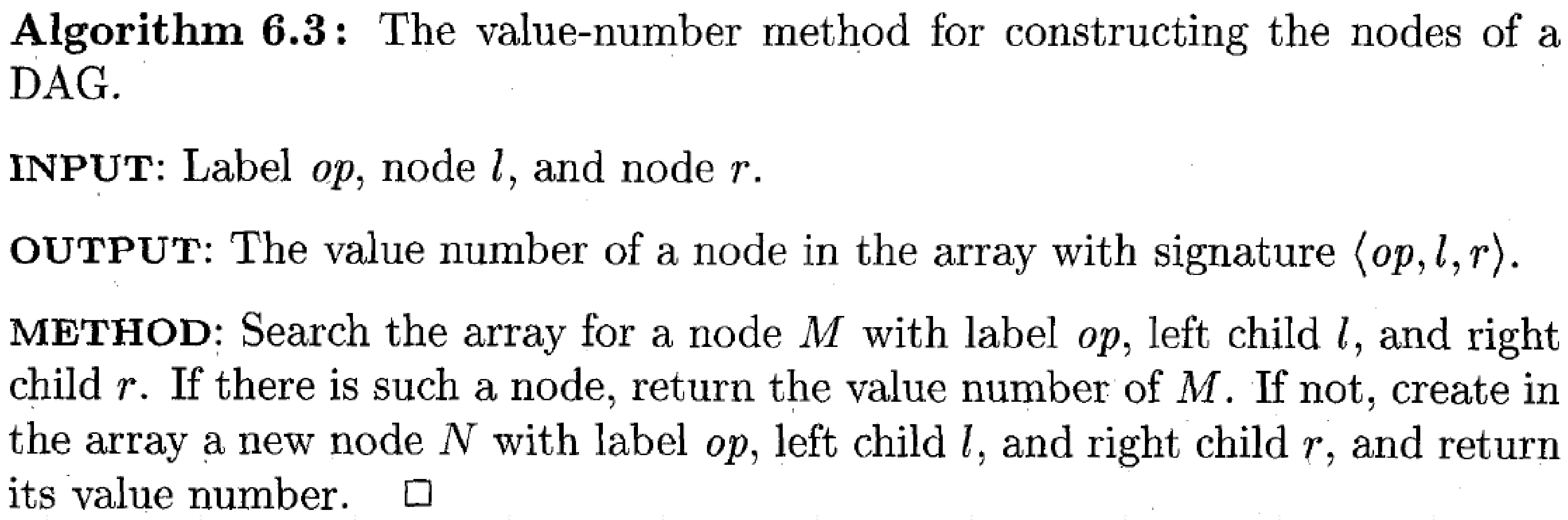


1	id		
2	num	10	
3	+	1	2
4	=	1	3
5	. . .		

to entry for i

(b) Array.

The Value-Number Method for Constructing DAG's



Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

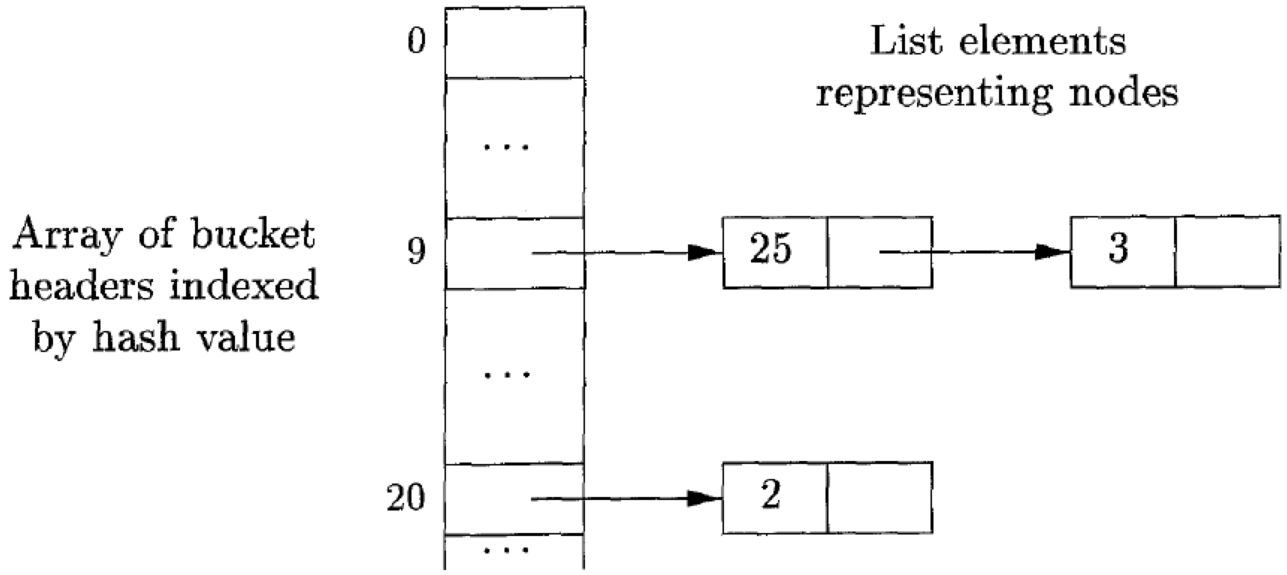
INPUT: Label *op*, node l, and node r.

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. □

The Value-Number Method for Constructing DAG's

- A more efficient approach is to use a hash table, in which the nodes are put into "buckets," each of which typically will have only a few nodes.

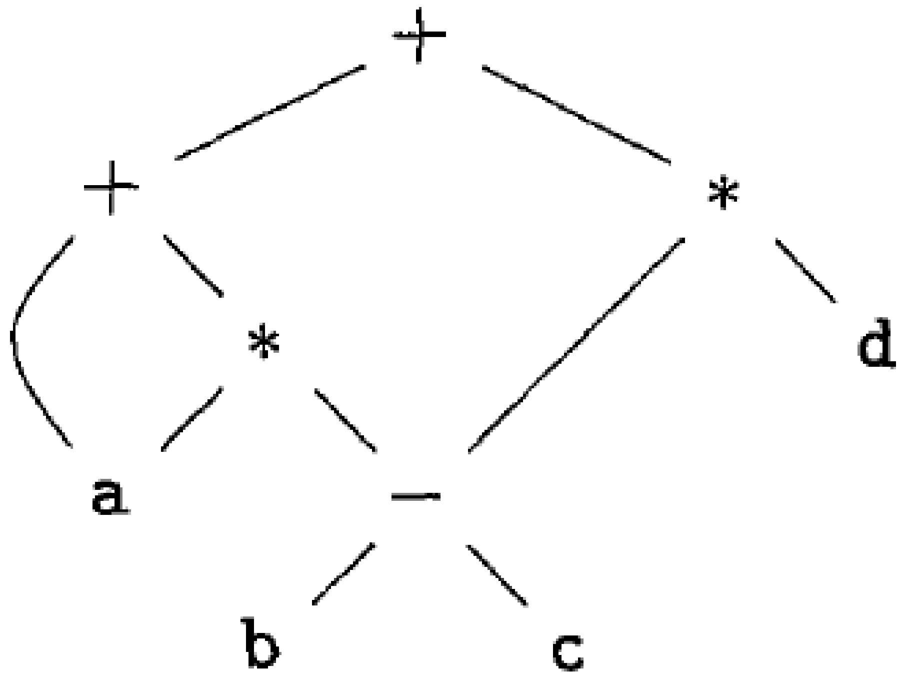

0
9
20
...
Array of bucket headers indexed by hash value
List elements representing nodes
25
3
2

Three-Address Code

- In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus, a source-language expression like $x + y * z$ might be translated into the sequence of three-address instructions

$$
\begin{aligned}
t_1 &= y * z \\
t_2 &= x + t_1
\end{aligned}
$$

  where t_1 and t_2 are compiler-generated temporary names.



(a) DAG

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

(b) Three-address code

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

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

Addresses and Instructions

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

Addresses and Instructions

- 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.
  - Copy instructions of the form $x = y$, where x is assigned the value of y.
  - An unconditional jump $goto\ L$.
  - Conditional jumps of the form if x goto L and ifFalse x goto L.
  - 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.

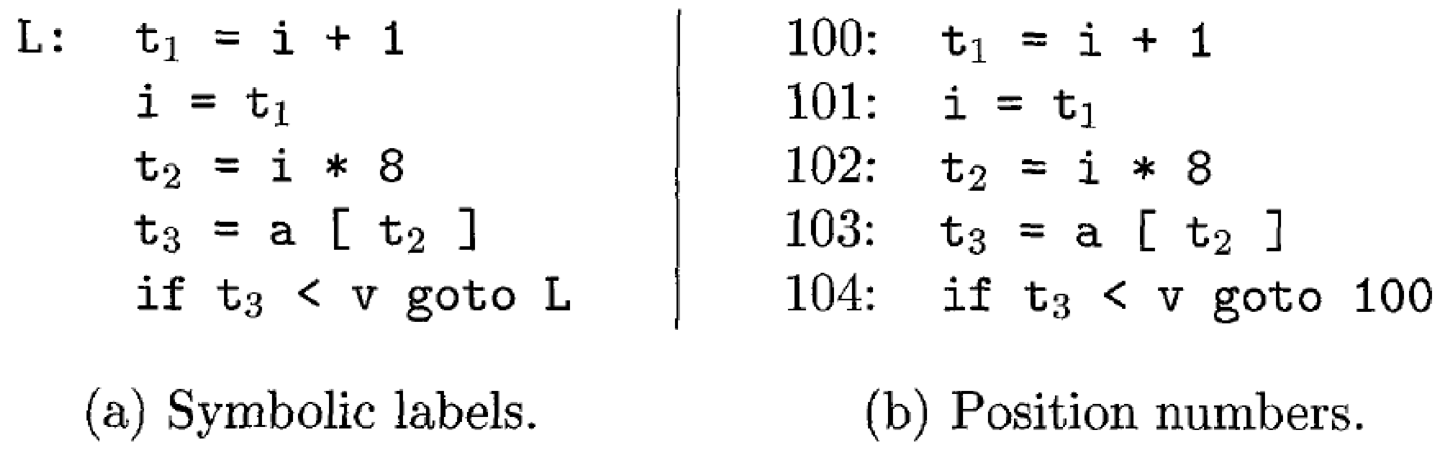
Addresses and Instructions

- Here is a list of the common three-address instruction forms:
  - Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$.
  - 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.
    - 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

- Consider the statement

$$\text{do } i = i + 1;\ \text{while } (a[i] < v)\ ;$$



```
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.

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

Intermediate Code Generation

- Three Address Code <-> (Two Address code => Triples)
- **Quadruples** (a collective data structure, each unit is with 4 fields)

Operator	Arg1	Arg2	Result
=+			
=-			
=*			
=/			
=%			
[]=			
=[]			
….			

Note: The entries of operator column are integers that represent individual operators. The entries of Arg1 (operand1) Arg2 (operand2) and Result are index (pointer) to the symbol table.

Kinds of three-address codes:

1. A = B op$^{(1)}$ C (op is a binary arithmetic or logical operation)
2. A = op$^{(2)}$ B (op is a unary operation, e.g. minus, negation, shift operators, conversion operator, identity operator)
3. goto L (unconditional jump, execute the Lth three-address code)
4. if A relop B goto L (relop denotes relational operators, e.g., <, ==, >, >=, !=, etc.)
5. param A and call P,n (used to implement a procedure call)
6. A = B [i]
7. A[i] = B
8. A = &B
9. A = *B
10. *A = B

	Operator	Arg1	Arg2	Result
A = B op$^{(1)}$ C	op$^{(1)}$	B	C	A
A = op$^{(2)}$ B	op$^{(2)}$	B		A
goto L	goto			L
if A relop B goto L	relopgoto	A	B	L
param A and call P,n	param	A		
	call	P	n	
A = B [i]	=[]	B	i	A
A[i] = B	[]=	B	i	A
A = &B	=&	B		A
A = *B	=*	B		A
*A = B	*=	B		A

Example: D = A*B+C

The generated three address code is:

⇨ T1 = A * B

⇨ T2 = T1 + C

⇨ D = T2

D = A + B * C

T1 = B * C

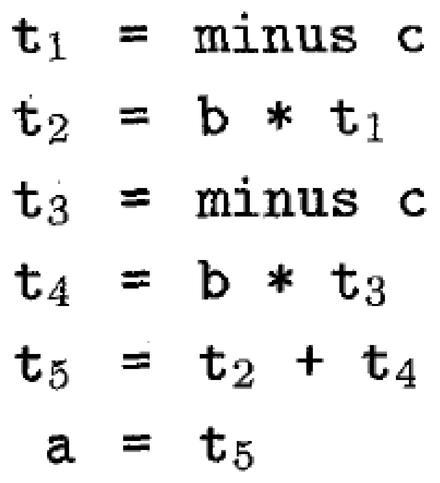
T2 = A + T1

D = T2

Operator	Arg1	Arg2	Result
=*	A	B	T1
=+	T1	C	T2
=	T2		D

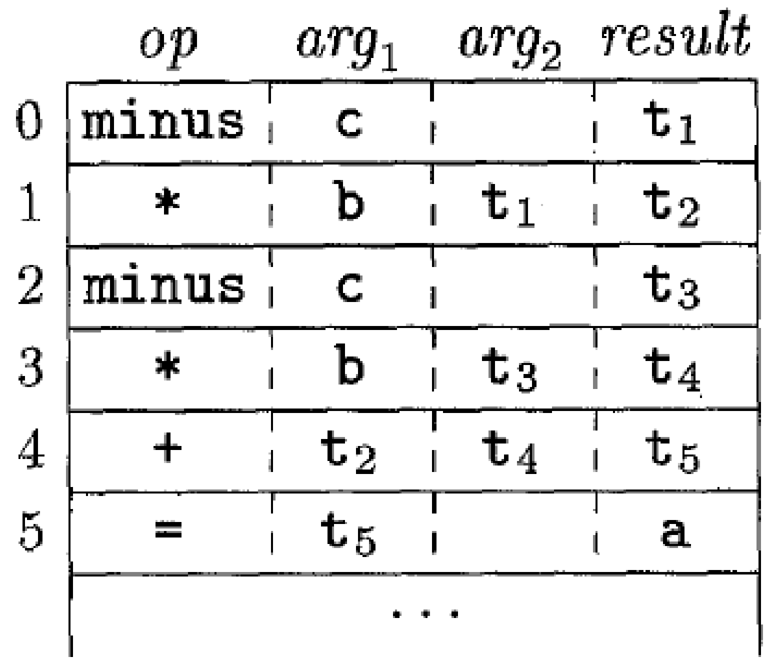
* T1 and T2 are compiler-generated temporary variables and they are also saved in the symbol table.

- Three-address code for the assignment $a = b * -c + b * - c$;



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

(a) Three-address code



	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

(b) Quadruples

Actually, in implementation the quadruples look as:

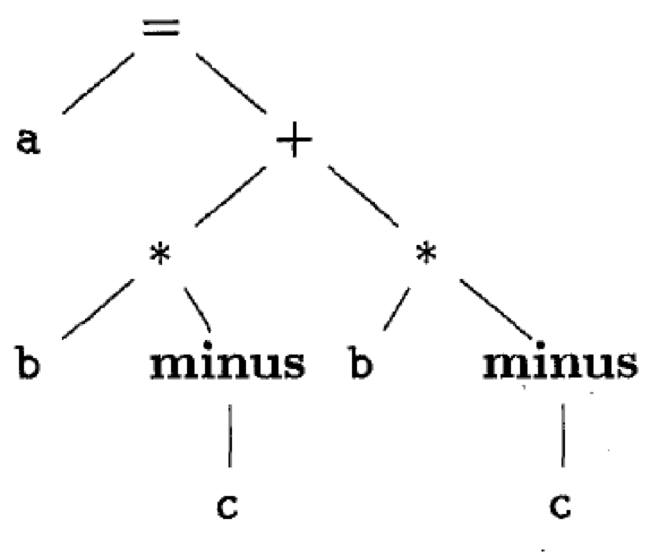
Operator	Arg1	Arg2	Result
8	6	7	9
15	9	8	11
3	11		10

in symbol table:

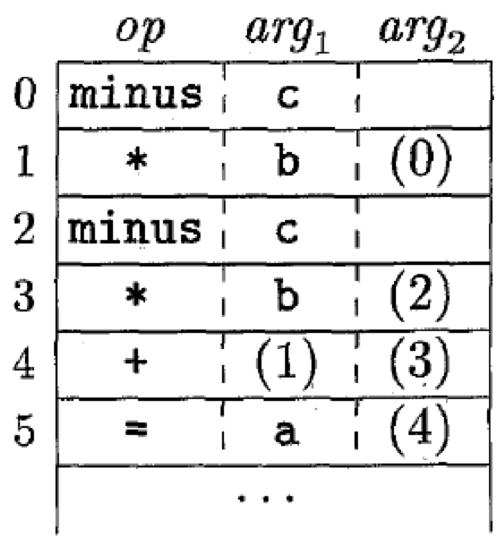
index	**identifier**	**attributes**
0	twa	
1	K	
..	..	
..	..	
6	**A**	
7	**B**	
8	**C**	
9	T1 /* compiler generated temporary variable */	
10	**D**	
11	T2 /* compiler generated temporary variable */	

Triples

- A triple has only three fields, which we call op, arg , and $\mathrm{arg2}$.



(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

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

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.
  - 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 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.
- Ex. The array type int [2][3] can be read as "array of 2 arrays of 3 integers each" and written as a type expression array(2, array(3, integer)).

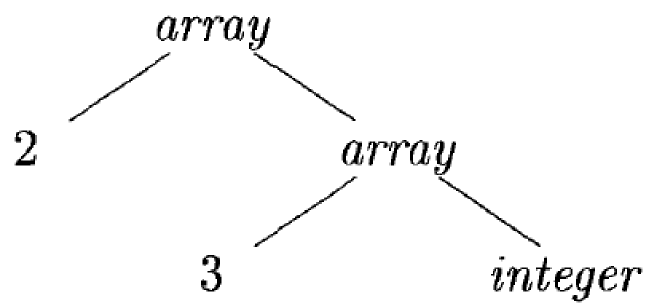


Figure 6.14: Type expression for `int[2][3]`

Declarations

- We shall study types and declarations using a simplified grammar that declares just one name at a time; declarations with lists of names can be handled as the grammar:

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

Storage Layout for Local Names

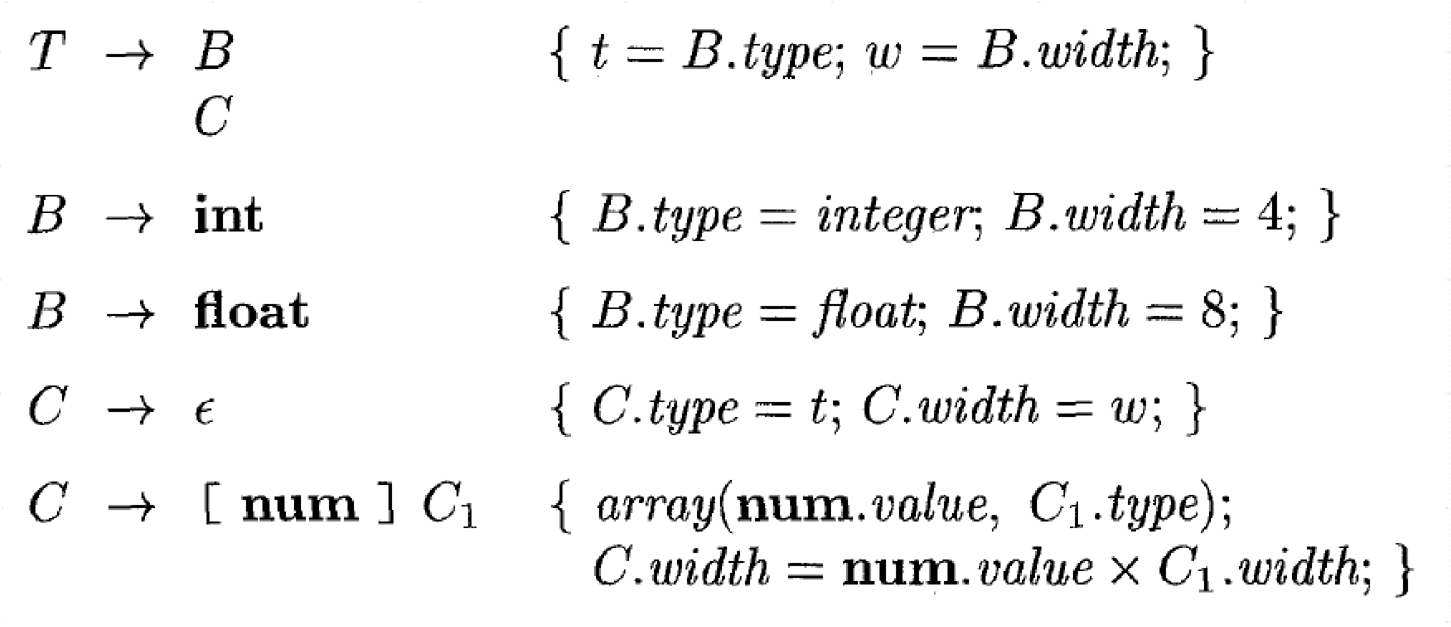
- At compile time, we can use these amounts to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name.
- Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by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

Storage Layout for Local Names

- Suppose that storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory.
- The width of a type is the number of storage units needed for objects of that type. A basic type, such as a character, integer, or float, requires an integral number of bytes.

Storage Layout for Local Names

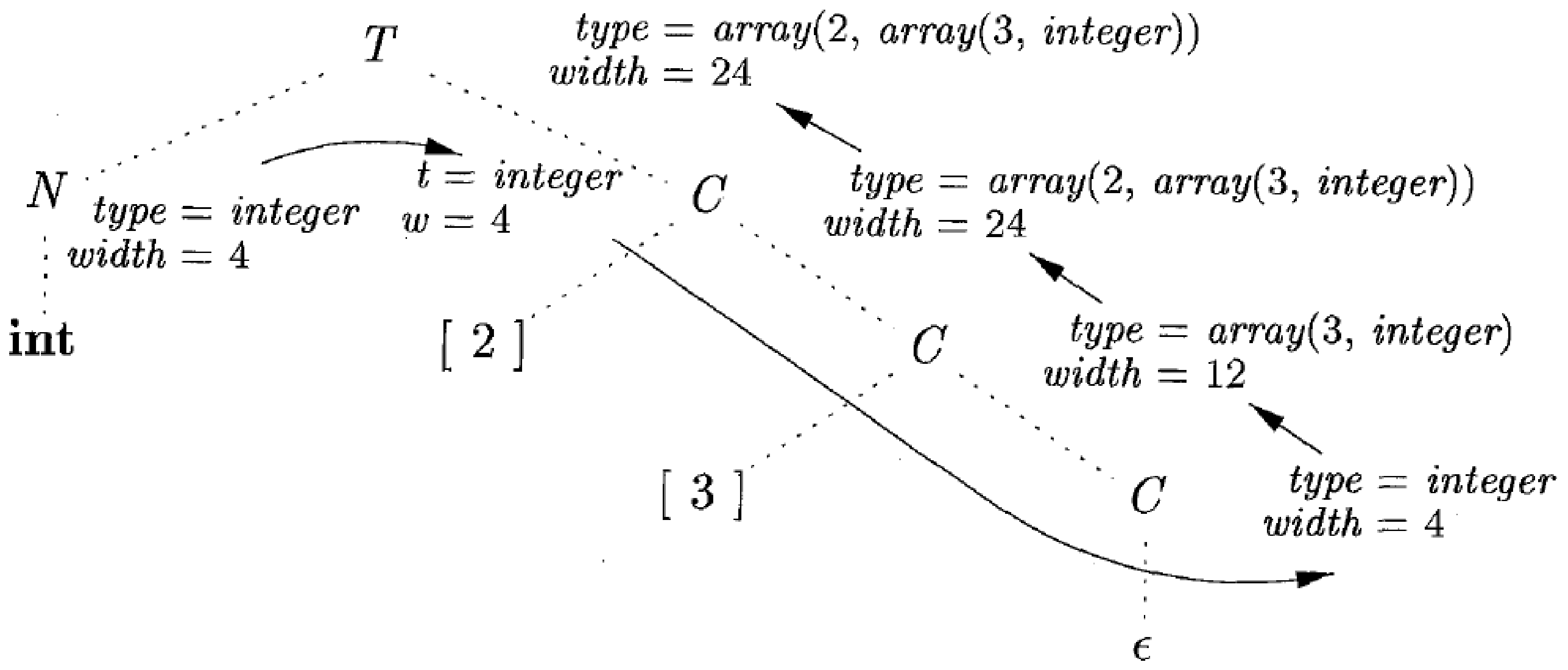
- The translation scheme uses synthesized attributes type and width for each nonterminal and two variables t and w to pass type and width information from a B node in a parse tree to the node for the production $C \rightarrow \epsilon$. In a syntax-directed definition, t and w would be inherited attributes for C.



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

Example

- The parse tree for the type int [2][3]



Sequences of Declarations

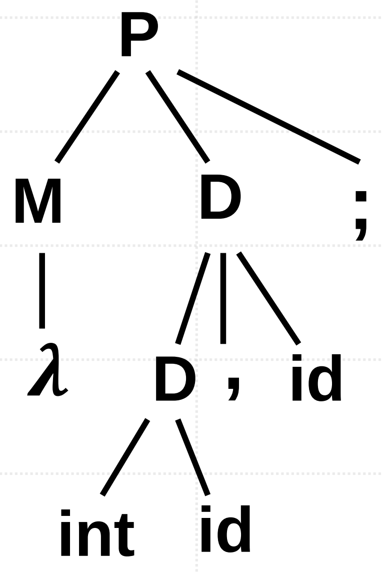
$P \rightarrow MD;$

$M \rightarrow \lambda$ /* empty string */

$D \rightarrow D, id$

$\quad | \; int\ id$

$\quad | \; float\ id$



int x , y ;

Sequences of Declarations

$P \rightarrow MD$; {/* do nothing */}
$M \rightarrow \lambda$ { if offset was not initialized then $offset = 0$;}

$D \longrightarrow int\ id$ { **enter ($\boldsymbol{id.name, int, offset}$);**
/* a function entering type “int” and
particular offset to the entry $id.name$
of the symbol table */
$D.type = int$;
$offset = offset + 4$; /*bytes, width of int*/
$D.offset = offset$; }

```
$D \rightarrow float\ id$ { enter ($id.name, float, offset$);
            $D.type = float$;
            $offset = offset + 8$;
            /*bytes, width of float*/
            $D.offset = offset$ ; }
$D \rightarrow D_1, id$ { enter ($id.name, D_1.type, D_1.offset$);
            $D.type = D_1.type$;
            If $D_1.type == int$
                $D.offset = D_1.offset + 4$;
            else if $D_1.type == float$
                $D.offset = D_1.offset + 8$;
            $offset = D.offset$;}
```

Note: We can construct a data structure to store the information (attributes) of D. (i.e., $D.type$ and $D.offset$)

Fields in Records and Classes

- The translation of declarations carries over to fields in records and classes. Record types can be added to the grammar by adding the following production

$$T \rightarrow \textbf{record}\ '\{'\ D\ '\}'$$

$$\begin{array}{llll} T & \rightarrow & \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env(); \\ & & & \quad Stack.push(offset);\ offset = 0;\ \} \\ & & D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\ & & & \quad top = Env.pop();\ offset = Stack.pop();\ \} \end{array}$$

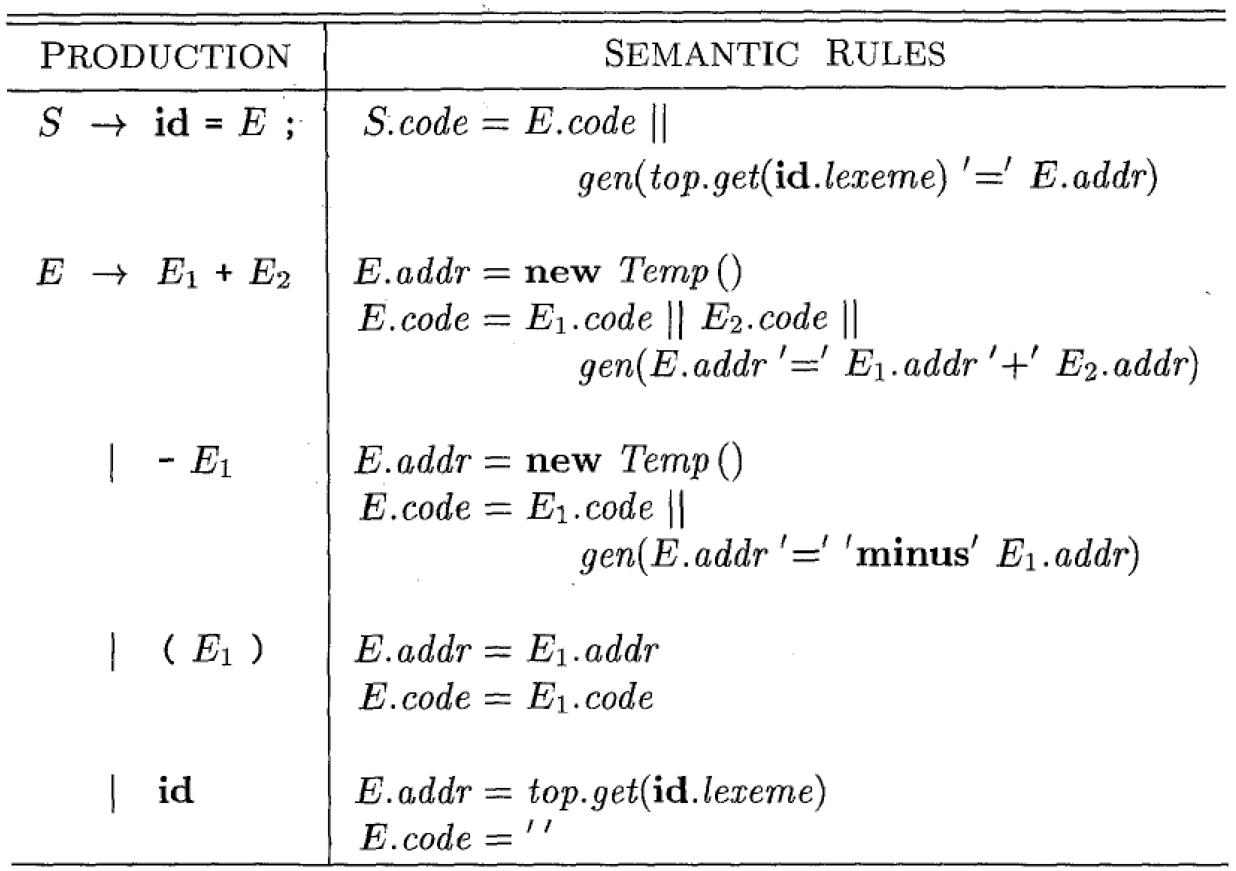
Translation of Expressions

- We begin in this section with the translation of expressions into three-address code. An expression with more than one operator, like $a + b * c$, will translate into instructions with at most one operator per instruction. An array reference $A[i][j]$ will expand into a sequence of three-address instructions that calculate an address for the reference.

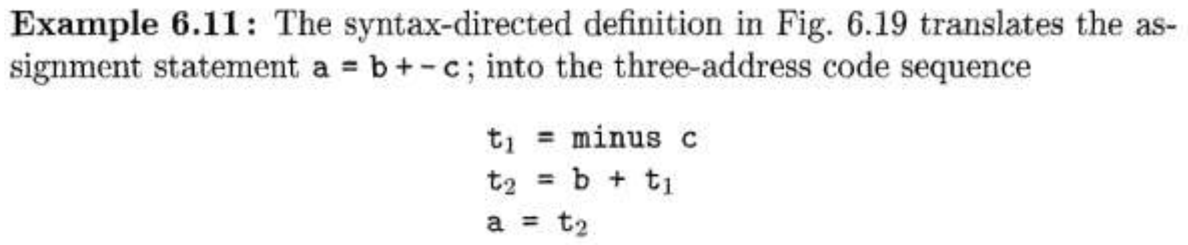
Operations Within Expressions

- The syntax-directed definition builds up the three-address code for an assignment statement S using attribute code for S and attributes $addr$ and $code$ for an expression E.
- Attributes $S.code$ and $E.code$ denote the three-address code for S and E, respectively.
- Attribute $E.addr$ denotes the address that will hold the value of E. An address can be a name, a constant, or a compiler-generated temporary.

Operations Within Expressions



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



Example 6.11 : The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c;` into the three-address code sequence

$$
\begin{aligned}
&t_1 = \texttt{minus}\ c\\
&t_2 = b + t_1\\
&a = t_2
\end{aligned}
$$

Incremental Translation

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E \; ; & \{ \; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \} \\
 & & & \\
E & \rightarrow & E_1 + E_2 & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & & & \quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\
 & & & \\
 & | & - \, E_1 & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & & & \quad gen(E.addr \; '=' \; '\textbf{minus}' \; E_1.addr); \; \} \\
 & & & \\
 & | & (\; E_1 \;) & \{ \; E.addr = E_1.addr; \; \} \\
 & & & \\
 & | & \textbf{id} & \{ \; E.addr = top.get(\textbf{id}.lexeme); \; \}
\end{array}
$$

Like to build a syntax tree

Addressing Array Elements

- Array elements can be accessed quickly if they are stored in a block of consecutive locations.
- If the width of each array element is w, then the ith element of array A begins in location

$$base + i \times w$$

where $base$ is the relative address of the storage allocated for the array

Addressing Array Elements

- The relative address of $A[i_1][i_2]$ can then be calculated by the formula

$$base + i_1 \times w_1 + i_2 \times w_2$$

- In k dimensions, the formula is

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k$$

Addressing Array Elements

- In two dimensions (i.e., $k = 2$ and $w = w_2$), the location for $A[i_1][i_2]$ is given by

$$base + (i_1 \times n_2 + i_2) \times w$$

- In k dimensions, the formula is

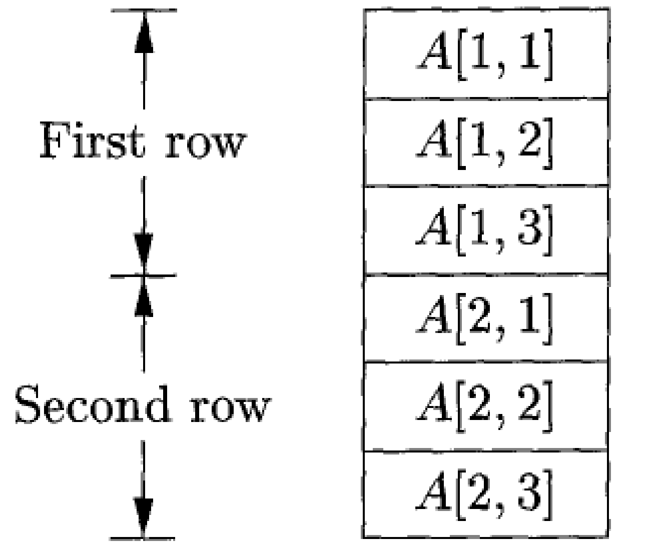
$$base + ((\cdots(i_1 \times n_2 + i_2) \times n_3 + i_3) \cdots) \times n_k + i_k) \times w$$

Addressing Array Elements

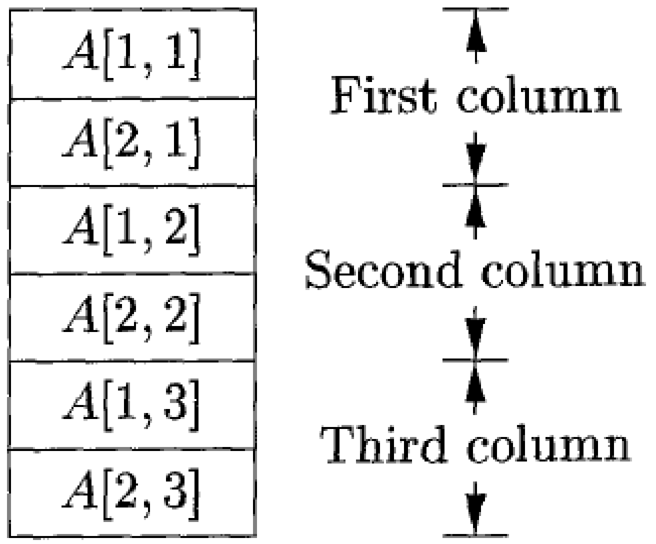
- More generally, array elements need not be numbered starting at 0. In a one-dimensional array, the array elements are numbered $low, low + 1, \ldots, high$ and $base$ is the relative address of $A[low]$. The address of $A[i]$ is

$$base + (i - low) \times w$$

Addressing Array Elements



(a) Row Major



(b) Column Major

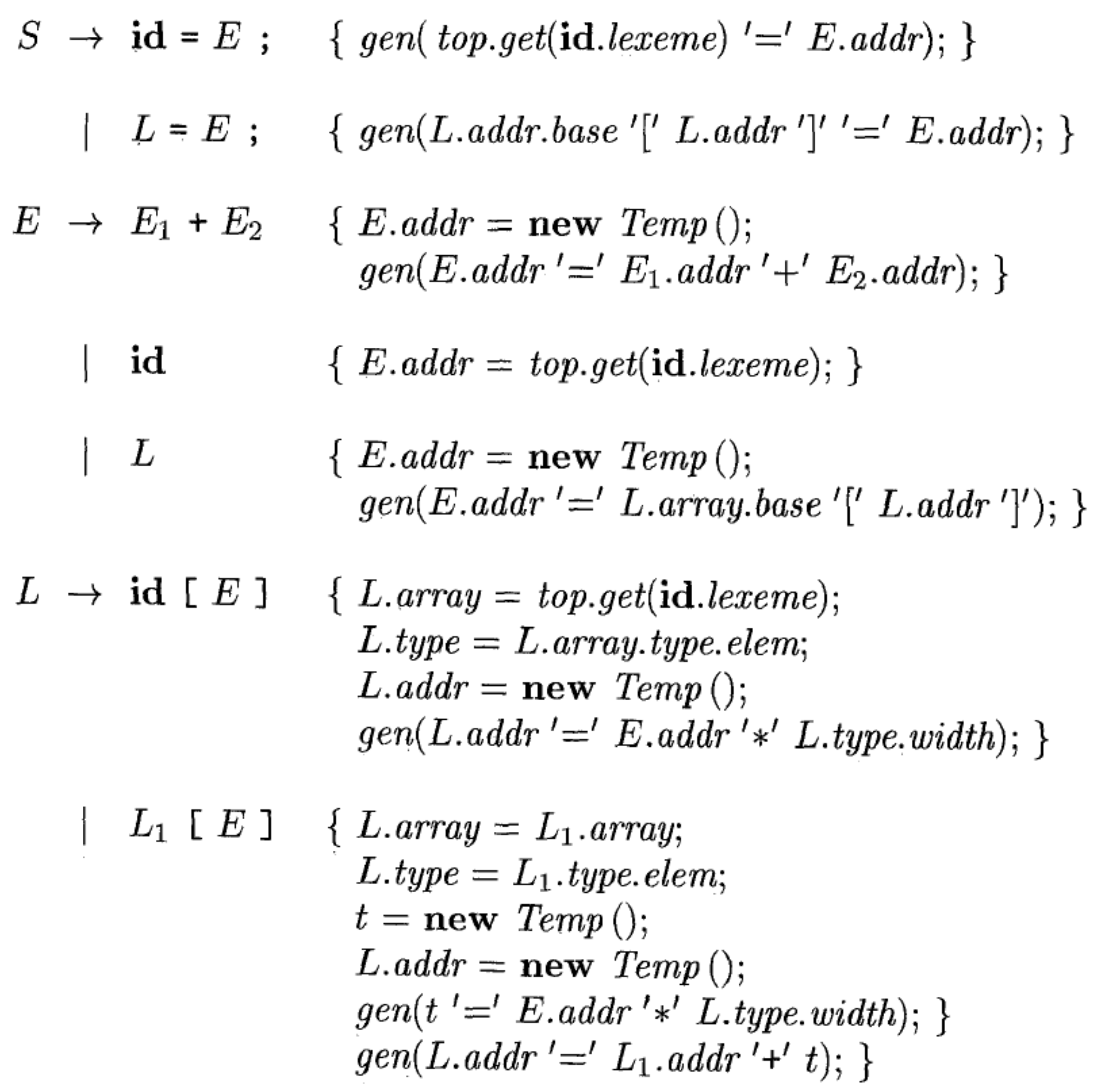
Translation of Array References

- Let nonterminal L generate an array name followed by a sequence of index expressions:

$$L \rightarrow L\ [\ E\ ]\ \mid\ \mathbf{id}\ [\ E\ ]$$

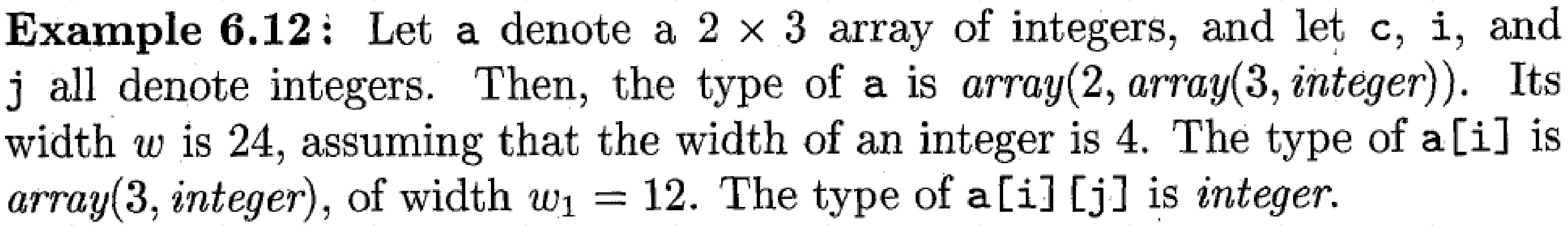
Translation of Array References

- $L.addr$ denotes a temporary that is used while computing the offset for the array reference by summing the terms $i_j \times w_j$.
- $L.array$ is a pointer to the symbol-table entry for the array name. The base address of the array, say, $L.array.base$ is used to determine the actual 1-value of an array reference after all the index expressions are analyzed.
- $L.type$ is the type of the subarray generated by L. For any type t, we assume that its width is given by $t.width$. For any array type t, suppose that $t.elem$ gives the element type.

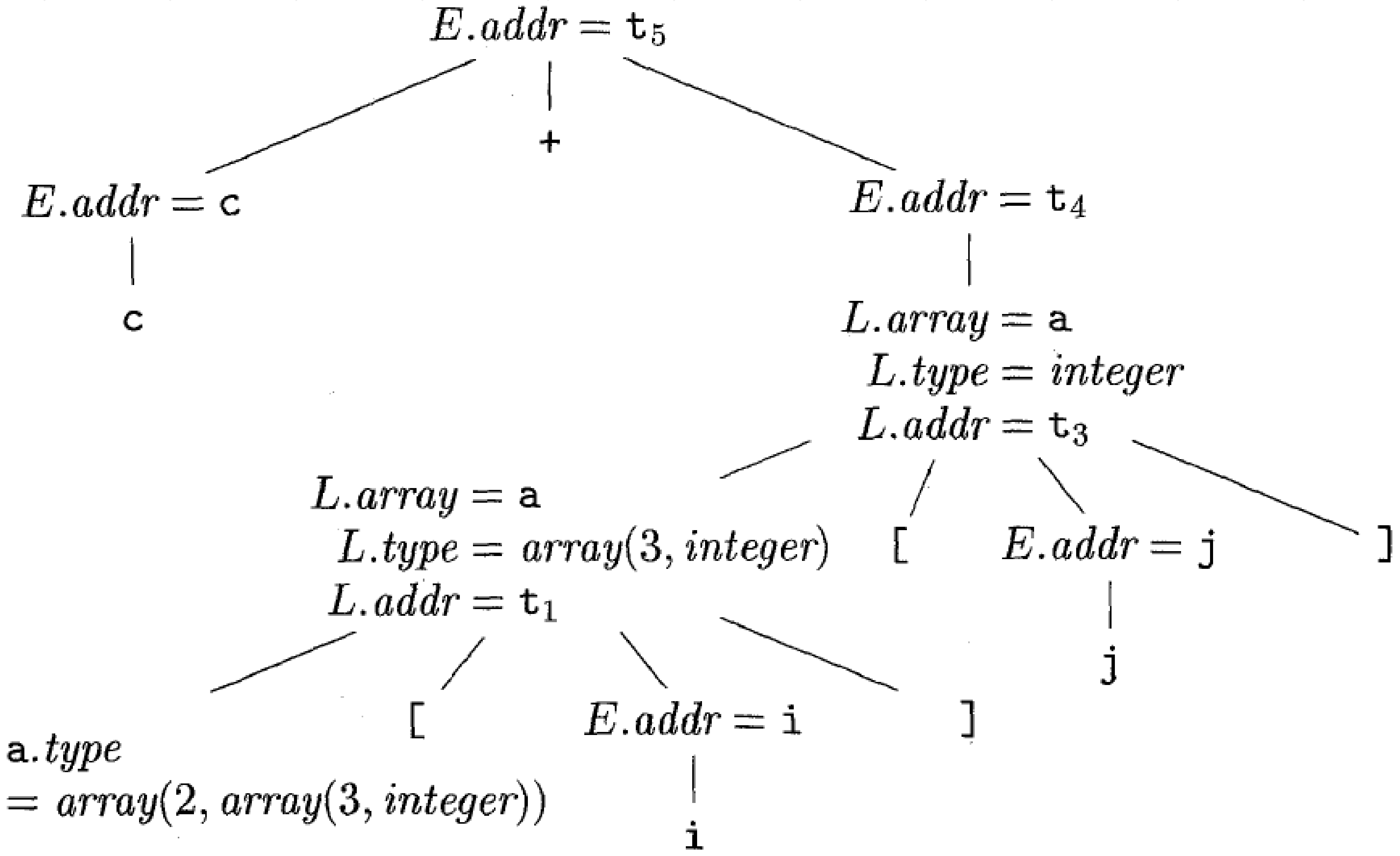


```
S → id = E ;      { gen( top.get(id.lexeme) '=' E.addr); }

  | L = E ;       { gen(L.addr.base '[' L.addr ']' '=' E.addr); }

E → E₁ + E₂       { E.addr = new Temp();
                    gen(E.addr '=' E₁.addr '+' E₂.addr); }

  | id            { E.addr = top.get(id.lexeme); }

  | L             { E.addr = new Temp();
                    gen(E.addr '=' L.array.base '[' L.addr ']'); }

L → id [ E ]      { L.array = top.get(id.lexeme);
                    L.type = L.array.type.elem;
                    L.addr = new Temp();
                    gen(L.addr '=' E.addr '*' L.type.width); }

  | L₁ [ E ]      { L.array = L₁.array;
                    L.type = L₁.type.elem;
                    t = new Temp();
                    L.addr = new Temp();
                    gen(t '=' E.addr '*' L.type.width); }
                    gen(L.addr '=' L₁.addr '+' t); }
```

Example

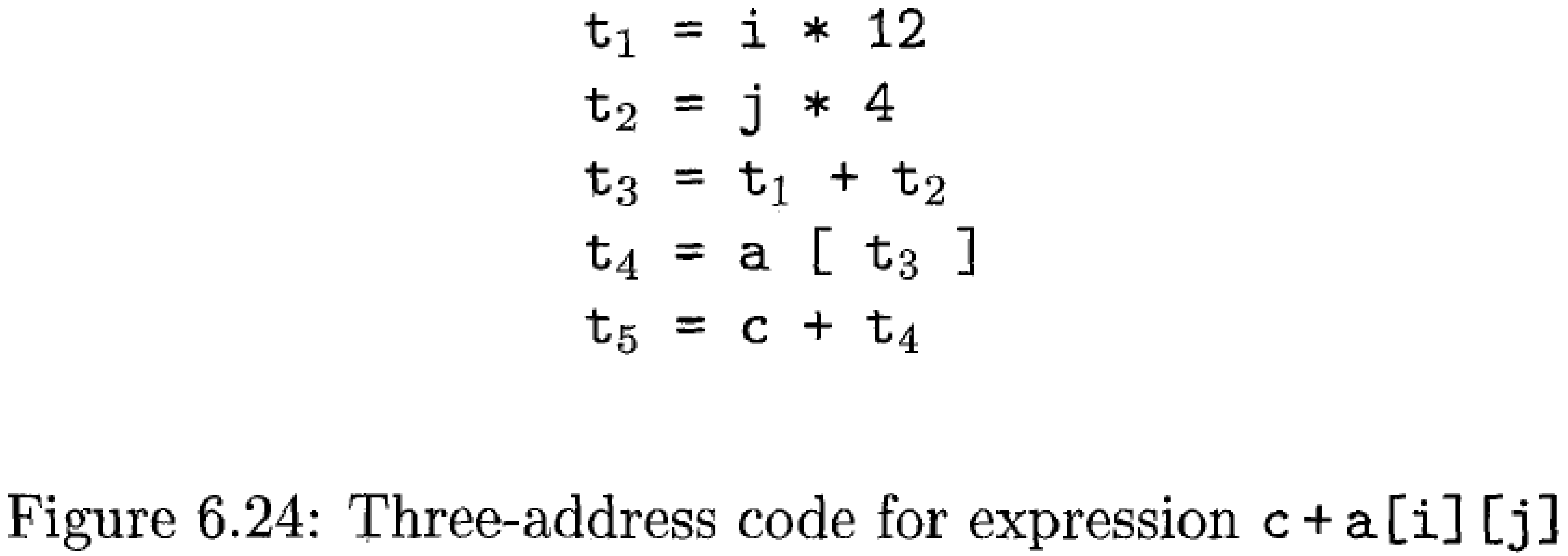


Example 6.12: Let `a` denote a 2×3 array of integers, and let `c`, `i`, and `j` all denote integers. Then, the type of `a` is $array(2, array(3, integer))$. Its width w is 24, assuming that the width of an integer is 4. The type of `a[i]` is $array(3, integer)$, of width $w_1 = 12$. The type of `a[i][j]` is $integer$.



$E.addr = \texttt{t}_5$
$E.addr = \texttt{c}$
+
$E.addr = \texttt{t}_4$
c
$L.array = \texttt{a}$
$L.type = integer$
$L.addr = \texttt{t}_3$
$L.array = \texttt{a}$
$L.type = array(3, integer)$
$L.addr = \texttt{t}_1$
[
$E.addr = \texttt{j}$
]
j
$\texttt{a}.type$
$= array(2, array(3, integer))$
[
$E.addr = \texttt{i}$
]
i

Example



$t_1 = i * 12$

$t_2 = j * 4$

$t_3 = t_1 + t_2$

$t_4 = a [t_3]$

$t_5 = c + t_4$

Figure 6.24: Three-address code for expression c + a[i][j]

Arithmetic Statements

A -> id = E

E -> $E^{(1)} + E^{(2)}$

E -> $E^{(1)} - E^{(2)}$

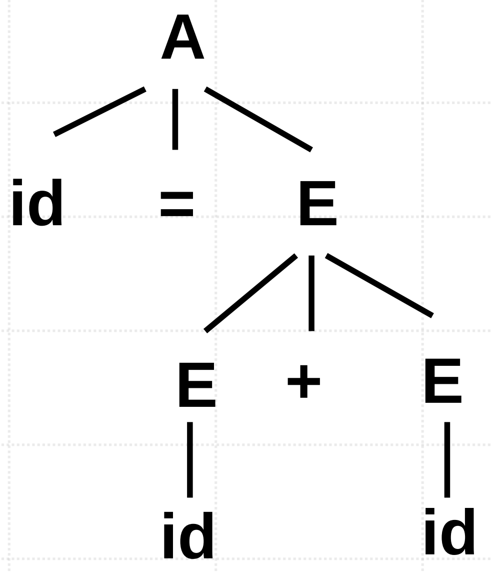
E -> $E^{(1)} * E^{(2)}$

E -> $E^{(1)} / E^{(2)}$

E -> - $E^{(1)}$

E -> ($E^{(1)}$)

E -> id



x = a + b $\longrightarrow$ **T1 = a + b**
x = T1

A -> id = E { GEN (id.addr = E. addr); }

/* GEN (argument) - a function used to save its argument into the quadruple. The implementation of E is a data structure with one field E.addr which holds the name that will hold the index value of the symbol table. */

E -> $E^{(1)}$ + $E^{(2)}$ { T = NEWTEMP();

/* NEWTEMP() - a function used to generate a temporary variable T and save T into symbol table and return the index value of the symbol table. */

E. addr = T;

/* T's index value in symbol table is assigned to E.addr */

$\underline{\text{GEN(E.addr = E}^{(1)}\text{.addr + E}^{(2)}\text{.addr);}}$ }

$$\downarrow$$

T = a + b

$E \rightarrow E^{(1)} * E^{(2)}$ { T = NEWTEMP();
E.addr = T;
GEN(E.addr = $E^{(1)}$.addr * $E^{(2)}$.addr); }

$E \rightarrow -E^{(1)}$ { T = NEWTEMP();
E.addr = T;
GEN(E.addr = -$E^{(1)}$.addr); }

$E \rightarrow (E^{(1)})$ { E.addr = $E^{(1)}$.addr; }

$E \rightarrow$ id { E.addr = id.addr; }
/*將id之符號表index值傳給E之field 'place' ; In implementation id.addr refers to the index value of id in the symbol table. */

Enhanced version for E -> E$^{(1)}$ op E$^{(2)}$

**注意in this version E所對應資料結構之設計 (應以array of struct of E之資料結構來儲存各個E之attributes, 並將對應之array index值儲存於E對應之value stack中)

```
{   T = NEWTEMP();
      if E(1).type == int and E(2).type == int then
       {  GEN (T = E(1).addr intop E(2).addr);
            E.type = int;
       }
      else if E(1).type == float and E(2).type == float then
      {  GEN (T = E(1).addr floatop E(2).addr);
```

```
        E.type = float;
      }
  else if $E^{(1)}$.type == int and $E^{(2)}$. type == float then
      { U = NEWTEMP();
        GEN (U = **inttofloat** $E^{(1)}$.addr);
        GEN (T = U **floatop** $E^{(2)}$.addr);
        E.type = float;
      }
  else /* $E^{(1)}$.type == float and $E^{(2)}$.type == int then
      { U = NEWTEMP();
        GEN (U = **inttofloat** $E^{(2)}$.addr);
        GEN (T = $E^{(1)}$.addr **floatop** U);
        E.type = float;
      }
}
```

Control Flow

- In programming languages, boolean expressions are often used to
  - Alter the flow of control. Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expressions is implicit in a position reached in a program. For example, in $if\ (E)\ S$, the expression E must be true if statement S is reached.
  - Compute logical values. A boolean expression can represent true or false as values. Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

Boolean Expressions

M -> ε

E -> E or M E

| E and M E

| not E

| (E)

| id

| id relop id

An example

```
if  p < q || r < s && t < u
        x = y + z;
k = m – n;
```

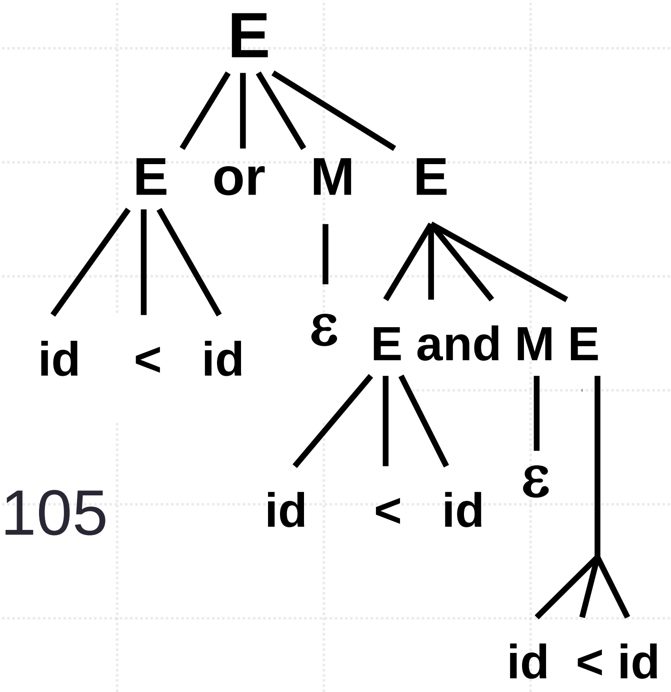
For the above boolean expression the corresponding contents in the quadruples are:

quadruples

```
if  p < q || r < s && t < u
        x = y + z;
       k = m – n;
```

counter = 100

Location	Three-Address Code
…	…………..
100	**if p < q goto** 106
101	**goto** 102
102	**if r < s goto** 104
103	**goto** 108
104	**if t < u goto** 106
105	**goto** 108 /*s.next = 105
106	t1 = y + z
107	x = t1
108	t2 = m - n
109	k = t2
…	………



NEXTQUAD – an integer variable used for saving the index (location) value of the next **available** entry of the quadruples.

E.true – an attribute of E that holds a set of indexes (locations) of the quadruples, each indexed quadruple saves the three-address code with ‘true’ boolean expression.

E.false – an attribute of E that holds a set of indexes of the quadruples, each indexed quadruple saves the three-address code with ‘false’ boolean expression.

GEN(x) – a function that translates x (a kind of three-address-code) into quadruple representation.

So, we need to construct a data structure for E which includes two fields, each field can save an unlimited number of integer.
Meanwhile, we need to construct an array of this E’s structure to store several Es’ attributes to be used in the same period of time .

1. M -> ε { M.quad = NEXTQUAD; }

   /* M.quad is a data structure associated with M */

2. E -> $E^{(1)}$ or M $E^{(2)}$

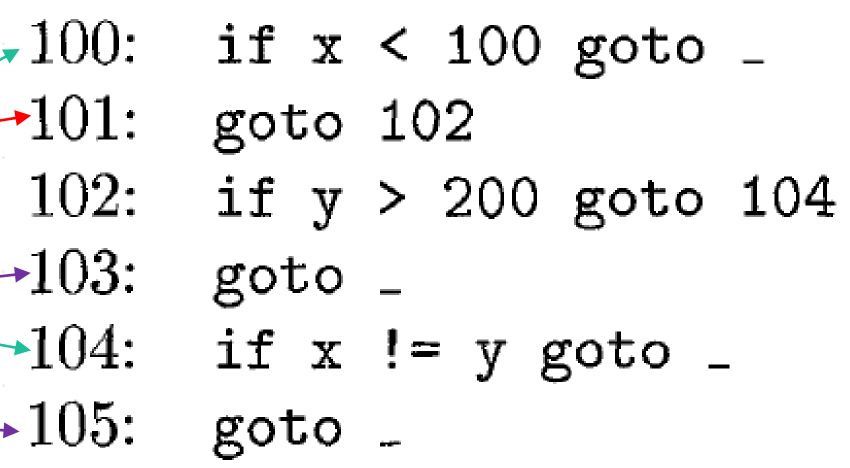
   {

   BACKPATCH ($E^{(1)}$.false, M.quad);

   E.true = MERGE ($E^{(1)}$.true, $E^{(2)}$.true);

   E.false = $E^{(2)}$.false;

   }



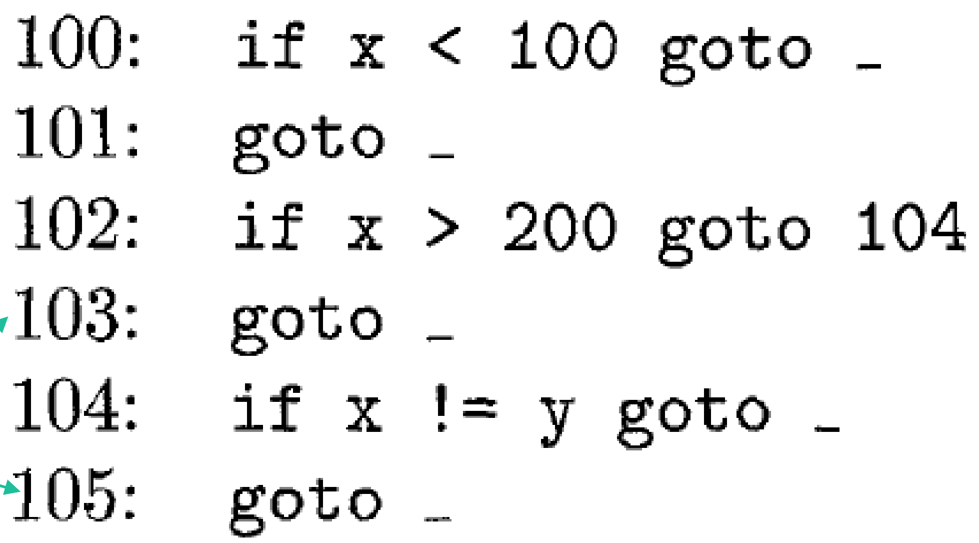
```
100:   if x < 100 goto _
101:   goto 102
102:   if y > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

/* **BACKPATCH (p, i)** – a function that makes each of the quadruple index values on the list pointed to by p take quadruple i as a target (i.e., **goto i**).*/

/* **MERGE (a, b)** – a function that takes the lists pointed to by a and b, concatenates them into one list, and returns a pointer to the concatenated list. */

3. E -> $E^{(1)}$ and M $E^{(2)}$

{
  BACKPATCH ($E^{(1)}$.true, M.quad);
  E.true = $E^{(2)}$.true;
  E.false = MERGE ($E^{(1)}$.false, $E^{(2)}$.false);
}



```
100:   if x < 100 goto _
101:   goto _
102:   if x > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

4. E -> not $E^{(1)}$

{ E.true = $E^{(1)}$.false; E.false = $E^{(1)}$.true;}

5. E -> ($E^{(1)}$)

{ E.true = $E^{(1)}$.true; E.false = $E^{(1)}$.false;}

6. E -> id

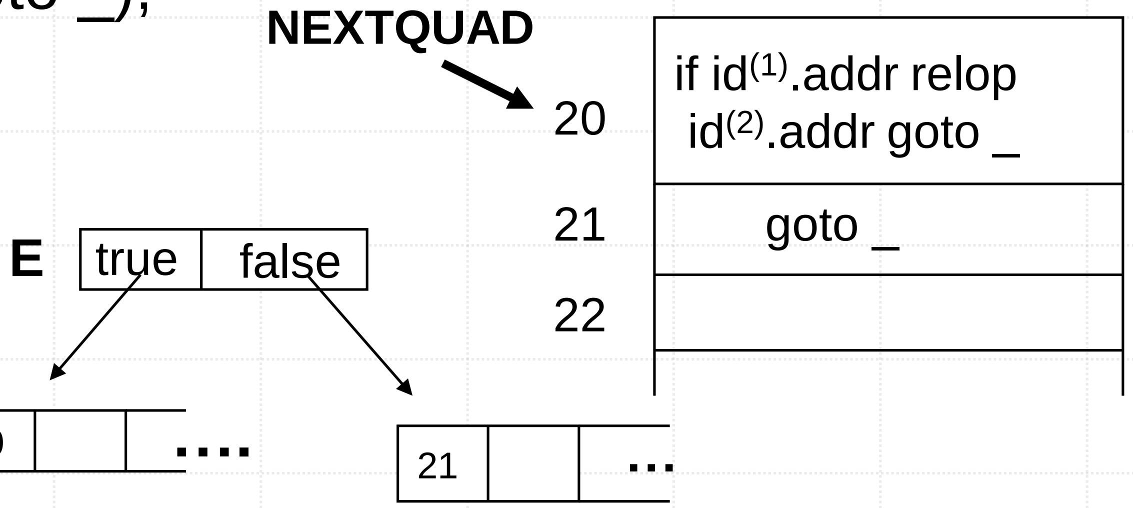
```
{
   E.true = MAKELIST (NEXTQUAD);
   E.false = MAKELIST(NEXTQUAD + 1);
   GEN (if id.addr goto _ );
   GEN (goto _);
}
```

/* MAKELIST (i) – a function that creates a list containing i, an index into the array of quadruples, and returns a pointer to the list it has made. */

/* GEN(x) – a function that translates x (a kind of three-address-code) into quadruple representation. */

7. E -> id$^{(1)}$ relop id$^{(2)}$

```
{
    E.true = MAKELIST (NEXTQUAD);
    E.false = MAKELIST(NEXTQUAD + 1);
    GEN (if id(1).addr relop id(2).addr goto _ );
    GEN (goto _);
}
```



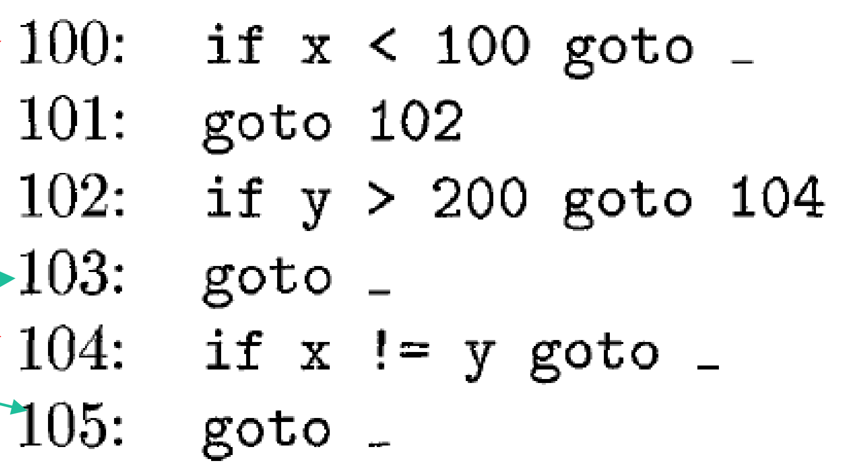
Flow-of-Control statements

A. Conditional Statements

```
S -> if E then S else S
    |  if E then S
    |  A
    |  begin L end
L -> S
    | L ; S

 /* A – denotes a general assignment statement
     L – denotes statement list
     S – denotes statement                    */
```

1. S -> if E then $M^{(1)}$ $S^{(1)}$ N else $M^{(2)}$ $S^{(2)}$

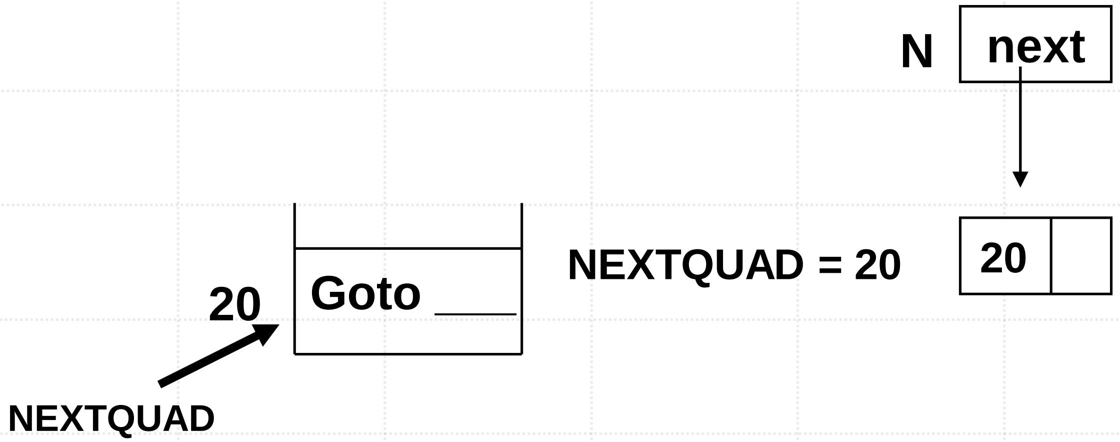
```
{
   BACKPATCH (E.true, M(1).quad);
   BACKPATCH (E.false, M(2).quad);
   S.next = MERGE (S(1).next, N.next, S(2).next);
}
```



```
100:   if x < 100 goto _
101:   goto 102
102:   if y > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

/* **S.next** is a pointer to a list of all conditional and unconditional jump (goto) to the quadruple following the statement S in execution order. */

```
2.  S -> if E then M S(1)
    {
       BACKPATCH (E.true, M.quad);
       S.next = MERGE (E.false, S(1).next)
     }

3.  M -> ε   { M.quad = NEXTQUAD; }

4.  N -> ε
    {
        N.next = MAKELIST (NEXTQUAD);
        GEN (goto _);
    }
```



```
5. S -> A
        { S.next = null; }
         /* initialize S.next to an empty list */

6.      L -> S  { L.next = S.next; }

7.      L -> L(1) ; M S
        {
           BACKPATCH (L(1).next, M.quad); // To resolve all
      quadruples with conditional & unconditional unresolved
      ‘goto _’
           L.next = S.next;
         }

8.      S -> begin L end  { S.next = L.next; }
```

Iterative Statement

S -> while E do S

9. S -> while $M^{(1)}$ E do $M^{(2)}$ $S^{(1)}$

```
{
   BACKPATCH (E.true, $M^{(2)}$.quad);
   BACKPATCH ($S^{(1)}$.next, $M^{(1)}$.quad);
   S.next = E.false;
   GEN (goto $M^{(1)}$.quad);
}
```

An example:

while (A<B) do if (C<D) then X = Y + Z;

E E

Index	Three-Address Code	
…	…..	
100	if (A<B) goto 102	2
101	goto __	//will be resolved (filling 107) later
102	if (C<D) goto 104	1 **If (C<D) then X=Y+Z;**
103	goto 100	3
104	T = Y + Z	
105	X = T	
106	goto 100	
107	…	

6. Procedure calls

1. call -> id (args)
2. args -> args , E
3. args -> E

```
1.  call -> id (args)
     { for each item p on QUEUE do
          GEN (param p);
       GEN (call id.addr, length of QUEUE); }
```

/* QUEUE is a data structure for saving the indexes of the symbol table containing the names of the arguments. The length of QUEUE is the number of elements in QUEUE */

```
2. args -> args , E
      { append E.addr to the end of QUEUE; }

3. args -> E
      { initialize QUEUE to contain only
         E.addr; }
```

/* Originally, QUEUE is empty and, after the reduction of E to args, QUEUE contains a single pointer to the symbol table location for the name that denotes the value of E. */

Structure Declarations

type -> **struct {** fieldlist**}**   /*Note: symbols with bold face are terminals */

 | **ptr**
 | **char**
 | **int**
 | **float**
 | **double**

```
struct  {  int  x;        //offset 0
           float y;      //offset 2
           char k[10];//offset 6
         }  m;
```

m.width = 16 bytes

fieldlist -> fieldlist field;
 | field;

field -> type **id** $\longrightarrow$ **int x**

 | field [**integer** /*a token denoting any string of digits*/]

$\searrow$ **int x [10]**   **or** $\underline{\textbf{int x [10]}}$ **[20] [30]**
field

```
field -> type id
        { field.width = type.width;
          field.name = id.name;
          W_enter(id.name, type.width);}
```

/* **W_enter(name,width)** enters ‘width’ as the width of each element of ‘name’. If ‘name’ is not an array, then its width is the number of locations taken by data of name’s type. */

```
      | field(1) [integer]
       { field.width = field(1).width * integer.val;
         field.name = field(1).name;
         D_enter(field(1).name, integer.val);}
```

/* **D_enter(name,size)** increases the number of dimensions for 'name' by one and enters the last dimension as 'size' in the symbol table entry for 'name'. */

```
fieldlist -> field; {O_enter (field.name, 0); fieldlist.width =
                   field.width;}
```

/* O_enter(name,offset) makes 'offset' the number for which field name 'name' stands. This information, also, is recorded in the symbol table entry for 'name'. */

```
            |fieldlist(1) field; { fieldlist.width = fieldlist(1).width
                                 + field.width;
                             O_enter(field.name, fieldlist(1).width);}
```

```
type -> struct '{' fieldlist '} '   { type.width = fieldlist.width; }

type -> char { type.width = 1; }  /* Assume characters take one
                                      byte.*/

type -> ptr {type.width = 4; }     /*Assume pointers take four
                                      bytes.*/

type -> int { type.width = 2; }    /* Assume integers take two
                                      bytes.*/

.......
```

Switch Statement

Syntax:

```
switch E
    {
        case V1: S1;
        case V2: S2;
        ………….
        ………….
        case Vn-1: Sn-1;
        default: Sn;
    }
```

When translated into three-address code:

```
100     Code to evaluate E into T   ←  Temporary variable
101     If T ≠ V1 goto 104
102     Code for S1
103     Goto 113
104      If T ≠ V2 goto 107
105     Code for S2
106     Goto 113
107      ...
108      ...
109      If T ≠ Vn-1 goto 112
110      Code for Sn-1
111     Goto 113
112      code for Sn
113      ….
```

Symbol Table

- The principal symbol table operations are insert, lookup, and delete; other operations may also be necessary.
  1. The insert operation is used to store the information provided by name declarations when processing these declarations.
  2. The lookup operation is needed to retrieve the information associated to a name when that name is used in the associated code.
  3. The delete operation is needed to remove the information provided by a declaration when that declaration no longer applies.

The Structure of the Symbol Table

- Typical implementations of dictionary structures include linear lists, various search tree structures (binary search trees, AVL trees, B trees), and hash tables.

- Linear lists are a good basic data structure that can provide easy and direct implementations of the three basic operations, with a constant-time operation (by always inserting at the front or rear) and lookup and delete operations that are linear time in the size of the list.

Hash Tables

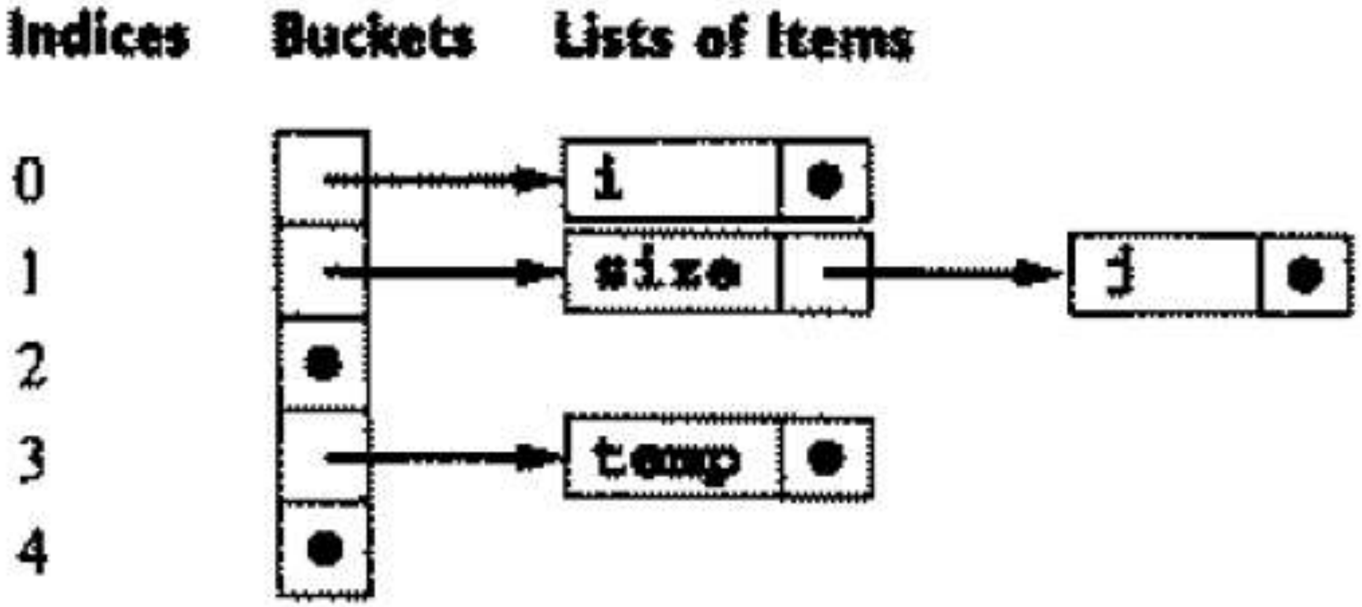


Figure 6.12

Separately chained hash table showing collision resolution

Declarations

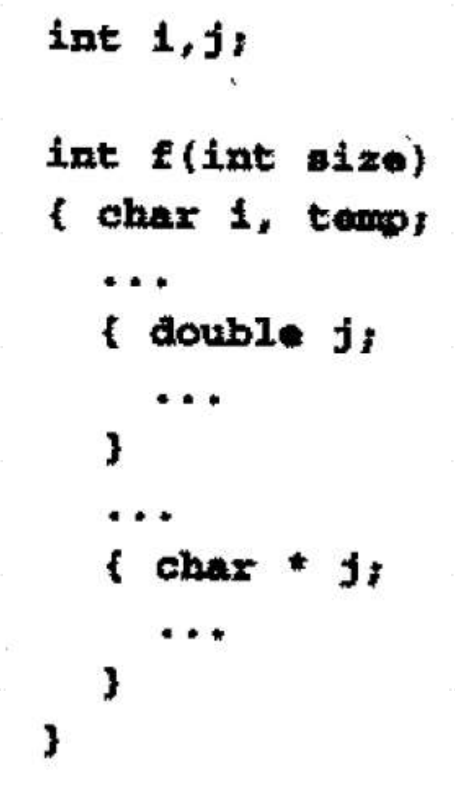
- There are four basic kinds of declarations that occur frequently in programming languages:
  - constant declarations
  - type declarations
  - variable declarations
  - procedure/function declarations

Scope Rules and Block Structure

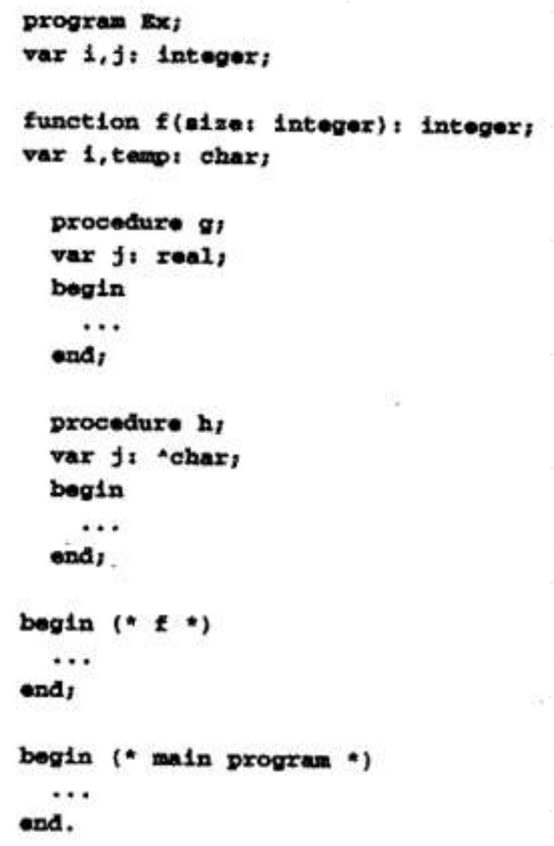
- Declaration before use is a common rule, used in C and Pascal, that requires that a name be declared in the text of a program prior to any references to the name.
- Declaration before use permits the symbol table to be built as parsing proceeds and for lookups to be performed as soon as a name reference is encountered in the code.
- If the lookup fails, a violation of declaration before use has occurred, and the compiler will issue an appropriate error message.

Scope Rules and Block Structure

- Block structure is a common property of modern languages.



```
int i,j;

int f(int size)
{ char i, temp;
    ...
    { double j;
        ...
    }
    ...
    { char * j;
        ...
    }
}
```

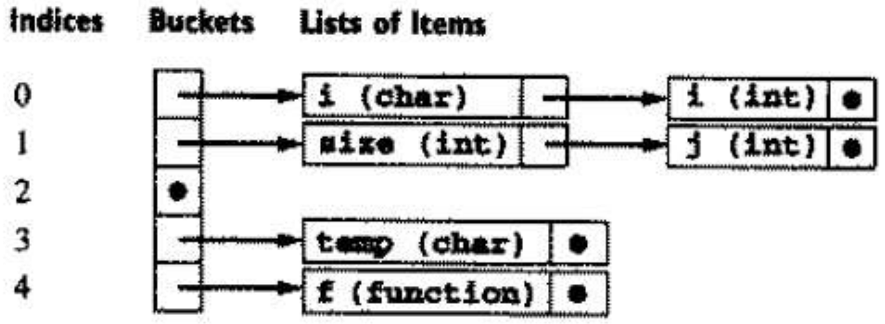


```
program Ex;
var i,j: integer;

function f(size: integer): integer;
var i,temp: char;

  procedure g;
  var j: real;
  begin
    ...
  end;

  procedure h;
  var j: ^char;
  begin
    ...
  end;

begin (* f *)
  ...
end;

begin (* main program *)
  ...
end.
```

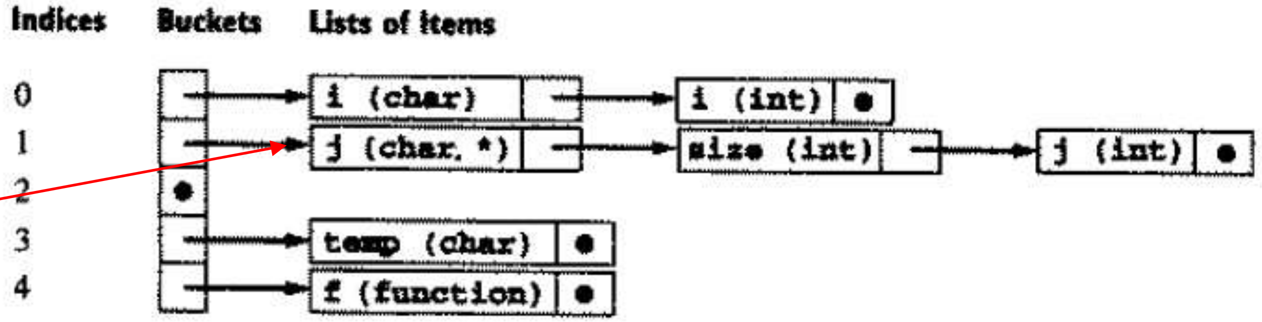
Scope Rules and Block Structure

- To implement nested scopes and the most closely nested rule, the symbol table insert operation must not overwrite previous declarations, but must temporarily hide them, so that the lookup operation only finds the most recently inserted declaration for a name.
- Similarly, the delete operation must not delete all declarations corresponding to a name, but only the most recent one, uncovering any previous declarations.

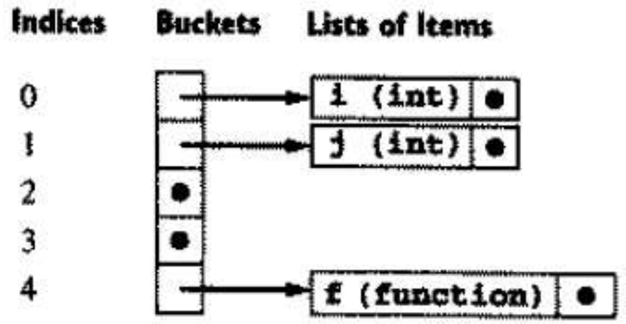
```
program Ex;
var i,j: integer;

function f(size: integer): integer;
var i,temp: char;

  procedure g;
  var j: real;
  begin
    ...
  end;

  procedure h;
  var j: ^char;
  begin
    ...
  end;

begin (* f *)
  ...
end;

begin (* main program *)
  ...
end.
```



(a) After processing the declarations of the body of f



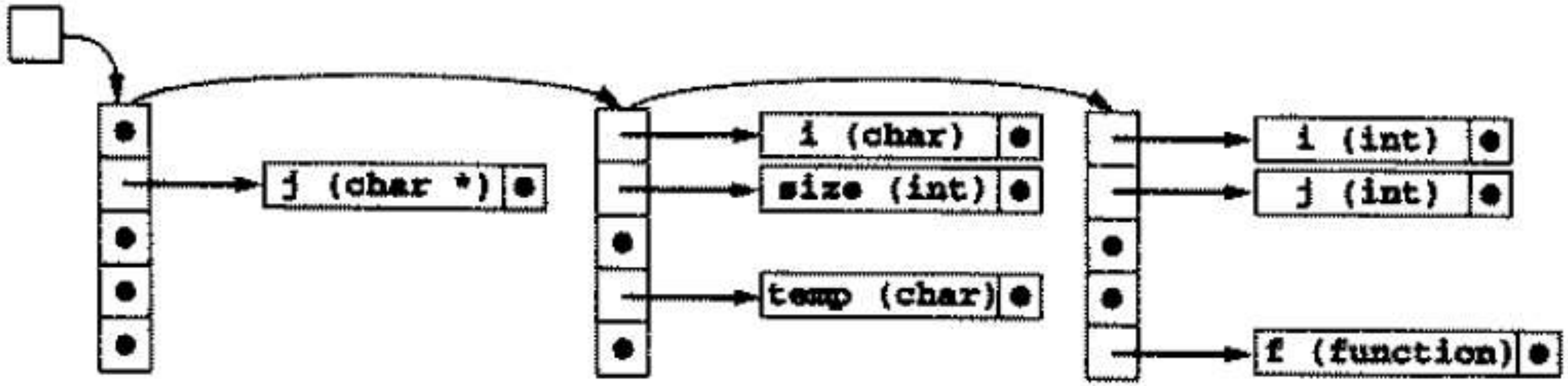
(b) After processing the declaration of the second nested compound statement within the body of f



(c) After exiting the body of f (and deleting its declarations)

Scope Rules and Block Structure

- A number of alternatives to this implementation fo nested scopes are possible. One solution is to build a new symbol table for each scope and to link the tables from inner to outer scopes together, so that the lookup operation will automatically continue the search with an enclosing table if it fails to find a name in the current table.


j (char *)
i (char)
size (int)
temp (char)
i (int)
j (int)
f (function)