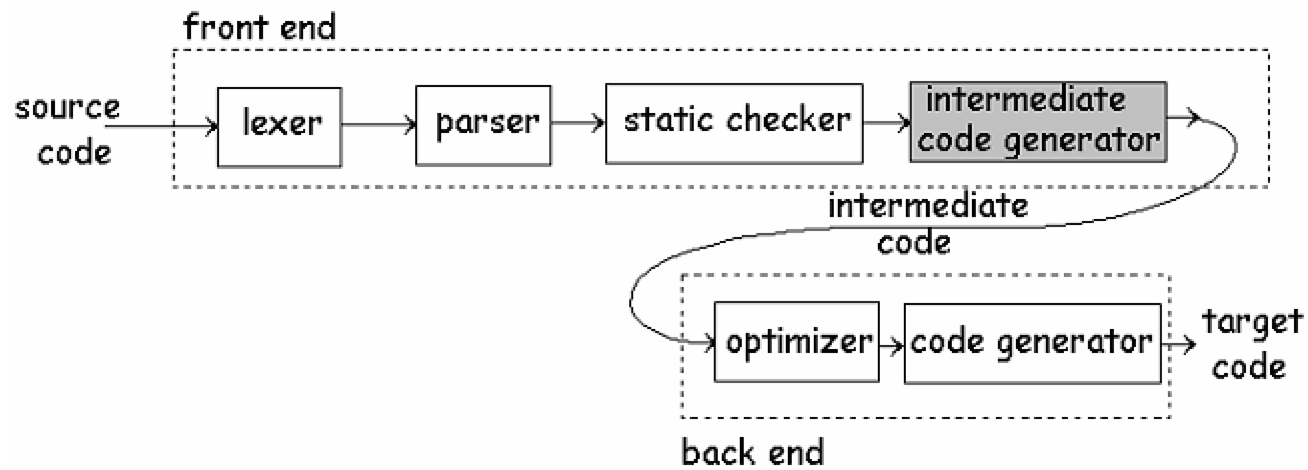
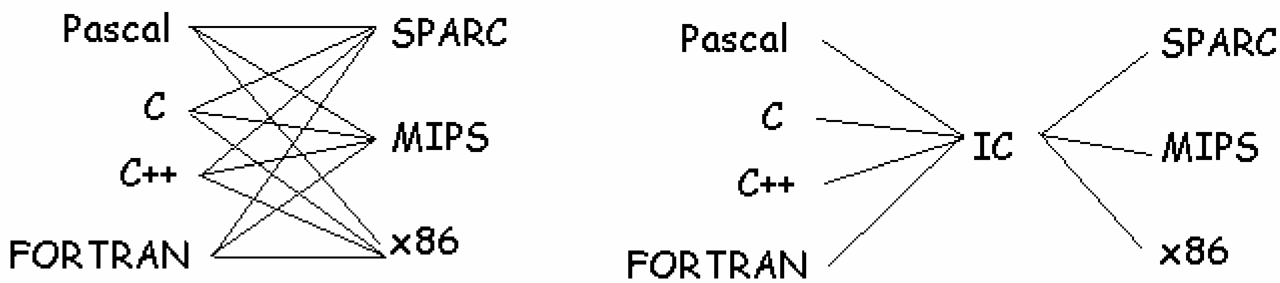
Intermediate Code Generation

- Most modern compilers are split into two:
  - the front end translates a source program to an intermediate representation
  - the back end then generates machine code for target architecture



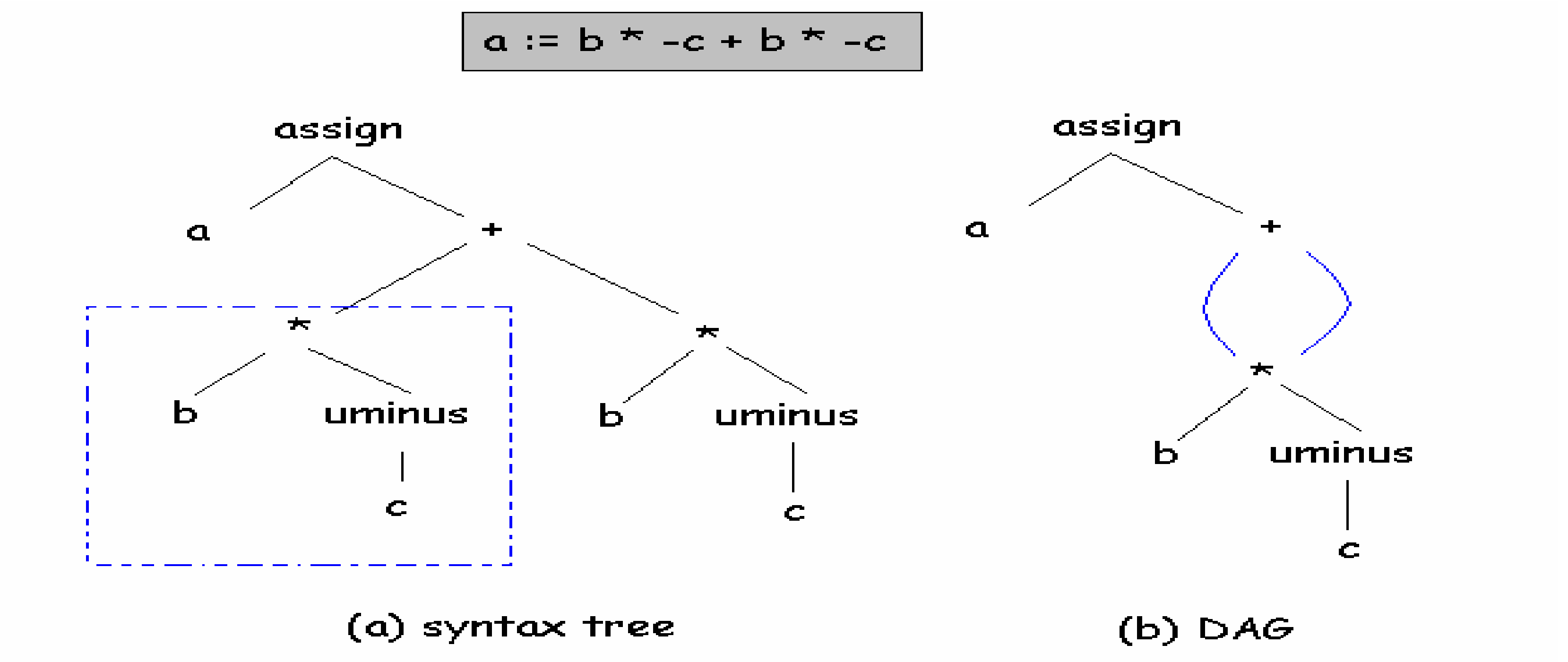
- The advantage is given by retargeting facility: it is easier to write different back ends for different target machines
- The disadvantage is that the compiling become a little slower (because of this intermediate step)



- Method used for translation into intermediate code: syntax directed translation. We will translate the constructs of the language:
  - Declarations;
  - Assignments
  - Flow-of-control statements
- Intermediate representation in the literature include:
  - syntax tree
  - postfix
  - three address code
  - others (AST, DAG, Control Flow Graphs (CFG), Program dependence Graph (PDG), Static Single Assignment Form, stack code)

Graphical Representation

- Graphical representation: syntax tree and more compact, DAG



- The postfix notation is a linearized representation of the syntax tree (a list of nodes of the tree where a node is put immediately after their children):

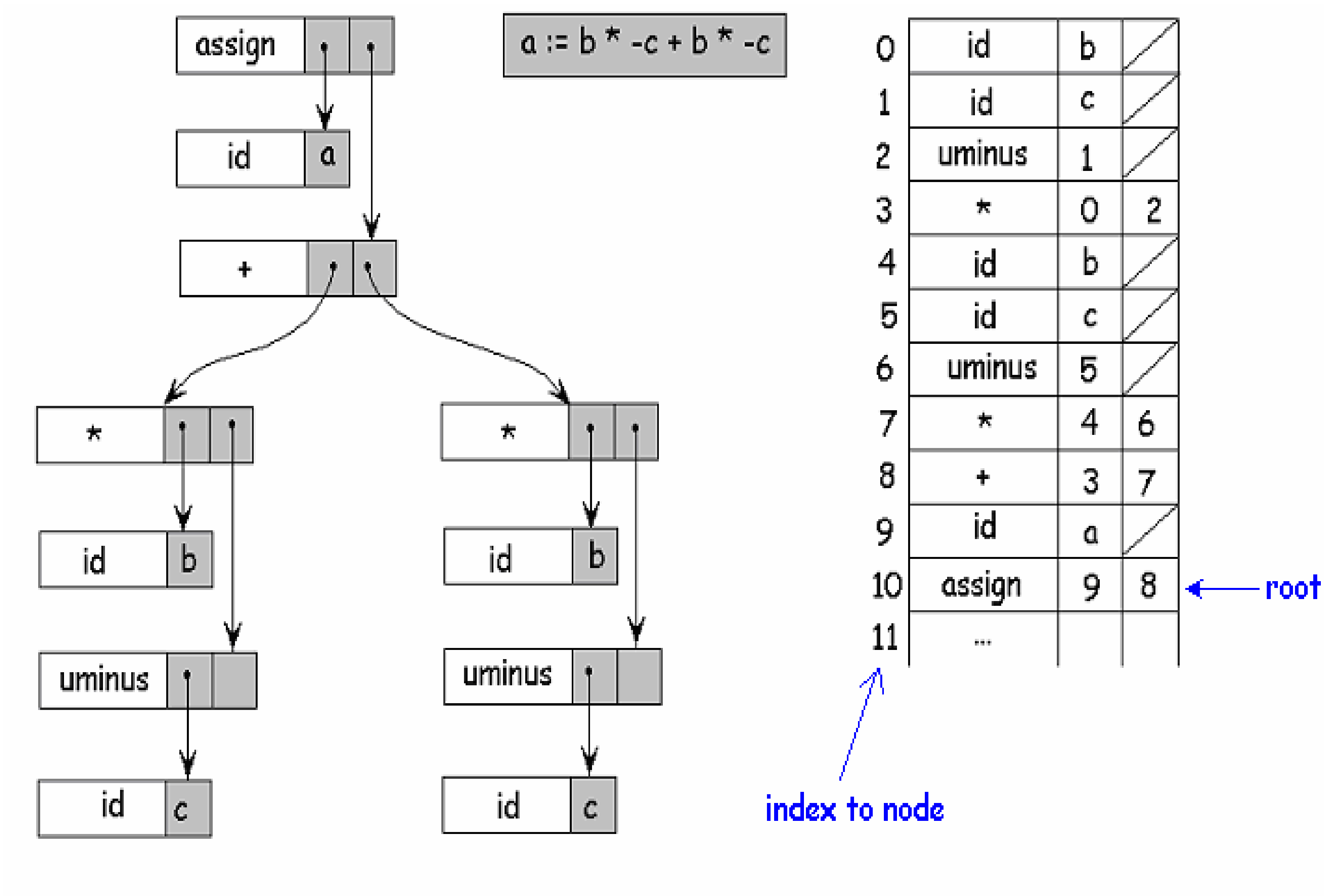
a b c uminus * b c uminus * + assign

- Syntax trees can be produced by syntax directed definition. E.g:

Production	Semantic Rule
$S \rightarrow \textbf{id} := E$	$S.nptr := mkenode(\text{'assign'}, \text{mkleaf}(\textbf{id}, \textbf{id}.place), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr := mknode(\text{'+'}, E_1.nptr, E2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr := mknode(\text{'*'}, E_1.nptr, E2.nptr)$
$E \rightarrow - E_1$	$E.nptr := mkunode(\text{'uminus'}, E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr := E_1.nptr$
$E \rightarrow \textbf{id}$	$E.nptr := mkleaf(\textbf{id}, \textbf{id}.place)$

- The attribute place from **id** points to the symbol-table for identifier
- The node has two basic attributes:
  - **place** : indicates the location of expression or location of symbol
  - code is generated usually by a **bottom-up traversal** of the tree.
- **Note**: when a node is visited, the codes are generated for the children first and only after that for the node itself

- Representations of the syntax tree :



(a) records

(b) array

Three addresses code and three address statements

- Three address code is used to describe a sequence of statements (*op* is for numeric operator: +, -, *, or, and, etc.):

  x : = y *op* z ; x, y, z are names, constants or compiler generated temporaries

- Other types of three-address statement

unary x : = *op* z

copy x := y

indexed x := y[i] or x[i] := y

address x := &y

pointer x := *y or *x := y

jumps unconditional goto L

conditional if x *relop* y goto L

procedure statements param x

- call p, n

- return y

$$\left.\begin{array}{l} \text{param } x_1 \\ \text{param } x_2 \\ \dots \\ \text{param } x_n \\ \text{call } p, n \end{array}\right\} p(x_1, x_2, \dots, x_n)$$

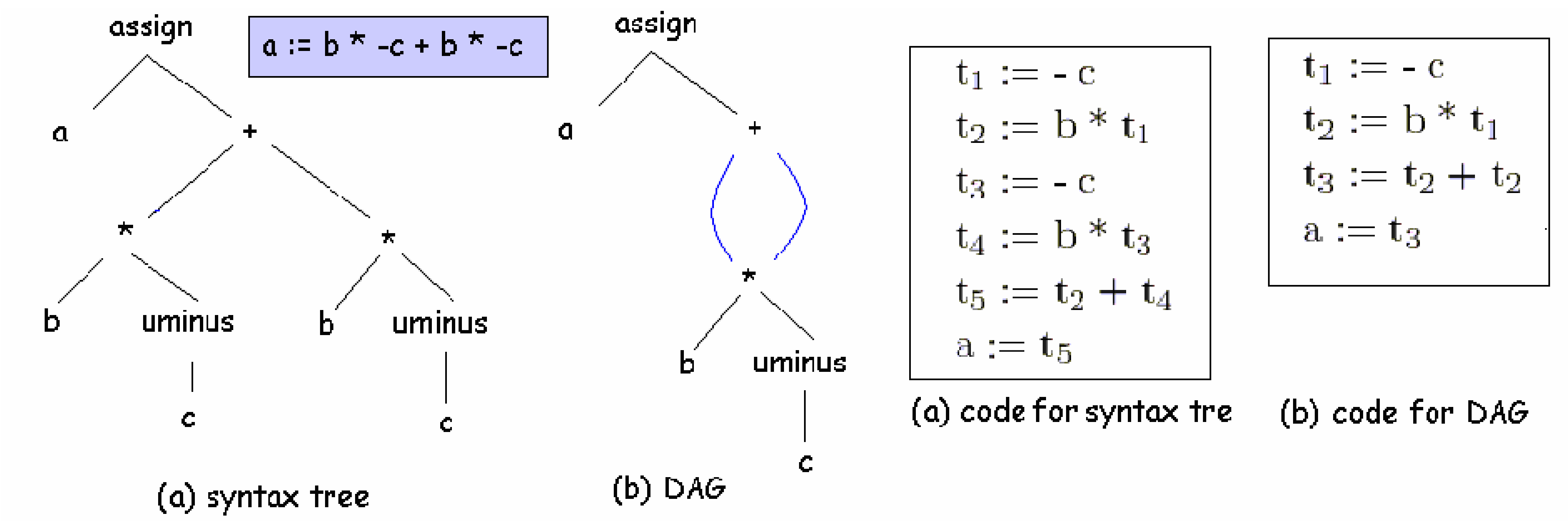
- A symbolic label represents the index of three-address statement in the array holding the intermediate code

- Compiler generated temporaries are used to evaluate expressions : x + y * z

$$t_1 := y * z$$

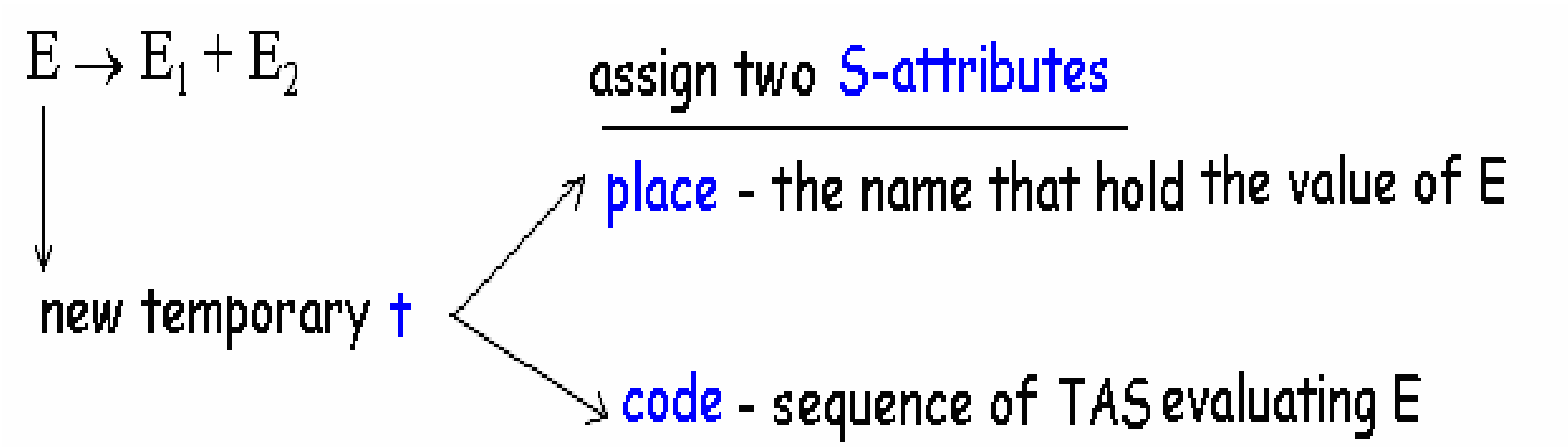
$$t_2 := x + t_1$$

- **Three-address code** represents a linear view of syntax trees or DAG when the explicit names correspond to a node in the graph



Syntax-directed translation into three-address code

- When the Three-Address Code (TAC) is generated, the interior node of the syntax tree get an assignation of a new name



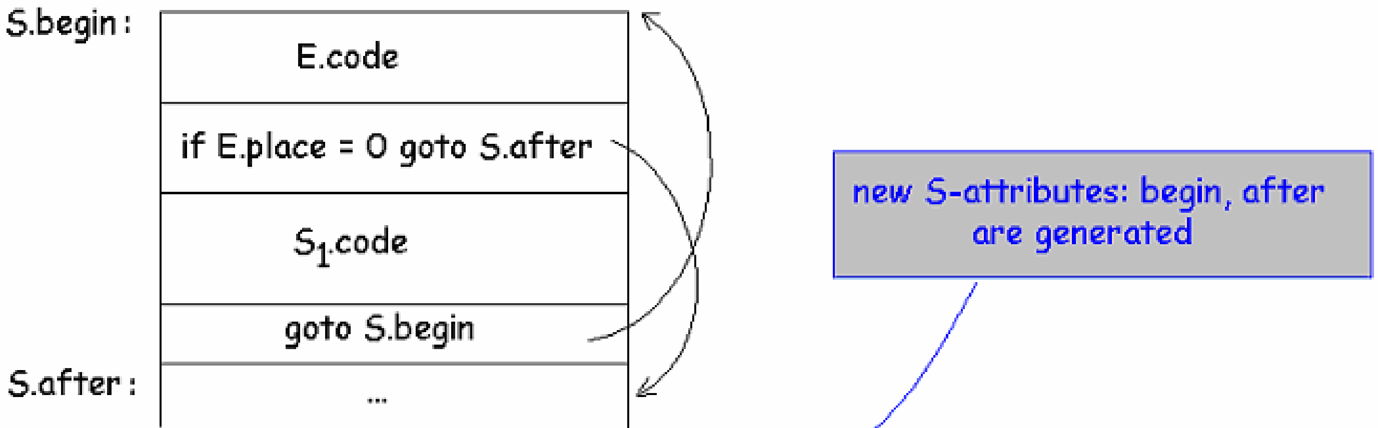
Syntax Directed Translation

Production	Semantic Rules
$S \rightarrow$ **id** := E	S.code := E.code || *gen*(**id**.place ':=' E.place)
$E \rightarrow E_1 + E_2$	E.place := *newtemp*; E.code := E_1.code || E_2.code || *gen*(E.place ':=' E_1.place '+' E_1.place)
$E \rightarrow E_1 * E_2$	E.place := *newtemp*; E.code := E_1.code || E_2.code || *gen*(E.place ':=' E_1.place '*' E_1.place)
$E \rightarrow - E_1$	E.place := *newtemp*; E.code := E_1.code || *gen*(E.place ':=' 'uminus' E_1.place)
$E \rightarrow (E_1)$	E.place := E_1.place; E.code := E_1.code;
$E \rightarrow$ **id**	E.place := id.place; E.code := ' '

gen(x ':=' y '+' z) represents x := y + z

Syntax directed definition for TAS

- The function ***newtemp***() return a sequence of distinct names: t_1, t_2, …in successive calls
- The procedure ***gen***() generate intermediate code
- Flow-of-control statements are depicted as below:



PRODUCTION	SEMANTIC RULES
S → **while** E **do** E_1	S.begin := newlabel; S.after := newlabel; S.code := gen(S.begin ': ') || E.code || gen('if' E.place '= ' '0' 'goto' S.after) S_1.code || gen('goto' S.begin) || gen(S.after ': ')

Implementation of three-address statements

- A TAS is an abstract form of intermediate code.
- In compiler, these statements can be implemented as records having fields for operator and the operands.
- Record representations of this form comprises quadruples, triples and indirect triples

	op	*arg 1*	*arg 2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quadruples

arg1, arg2 and results are normally pointers to entries in the symbol table

- Triples are used in order to avoid the entering temporary names into symbol tables.
- We can refer to a temporary value by its position in the statement

- E.g.

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

(a) code for syntax tree

	op	*arg* 1	*arg* 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

pointers to TAS struct

(b) Triples

- Ternary operation x[i] := y requires two entries in the triple structure and x := y[i] requires two operations.

	op	*arg* 1	*arg* 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) `x[i] := y`

	op	*arg* 1	*arg* 2
(0)	=[ ]	y	i
(1)	assign	x	(0)

(b) `x := y[i]`

More triple representations

- Other representations. Indirect triples use a list of pointers to triples instead of triple themselves

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

	op	*arg* 1	*arg* 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements.

- The usage of a different representation is a matter of how much indirection we want to use in our compiler

- It could be a problem when we use optimization in output code.

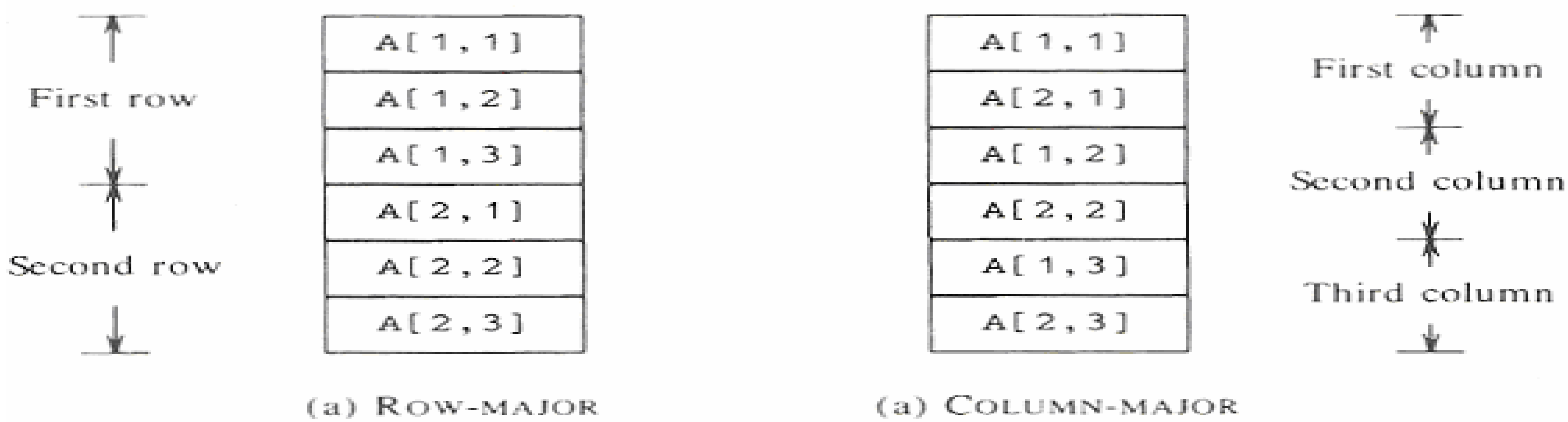
- Moving a temporary that represents a statement requires to change all the references in the triples

Addressing array elements

- Arrays are stored normally in block of contiguous locations
- A 1-D array A[*low…high*], with elements of size *w*, the i^{th} element A[i] begin at location:

  $base + (i\text{-}low) \times w = i \times w + (base - low \times w) = i \times w + c$

- A 2-D array is normally stored row-major (row-by-row) or column major (column-by-column).



Layouts for a two-dimensional array.