Production rule	Semantic action	Output
$E \rightarrow id$	E.place := a	
$E \rightarrow id$	E.place := b	
$E \rightarrow E_1 + E_2$	E.place := t_1	$t_1 := a + b$
$E \rightarrow id$	E.place := c	
$E \rightarrow id$	E.place := d	
$E \rightarrow E_1 * E_2$	E.place := t_2	$t_2 := c * d$
$E \rightarrow E_1 - E_2$	E.place := t_3	$t_3 := t_1 - t_2$
$S \rightarrow id := E$		$g := t_3$

Review Questions

1. *Give the translation scheme for converting the assignments into three address code.*

AU : May-11, 13, Marks 8

2. *How would you convert the following into intermediate code ? Give a suitable example : Assignment statements.*

AU : May-14, Marks 8

AU : May-10, 15, Dec.-09, 10, 14, Marks 16

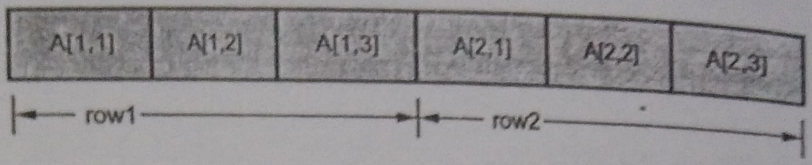
4.7 Arrays

As we know array is a collection of contiguous storage of elements. For accessing any element of an array what we need is its address. For statically declared arrays it is possible to compute the relative address of each element. Typically there are two representations of arrays -

1. Row major representation.
2. Column major representation.

These representations are as shown below -

Row major representation



Column major representation

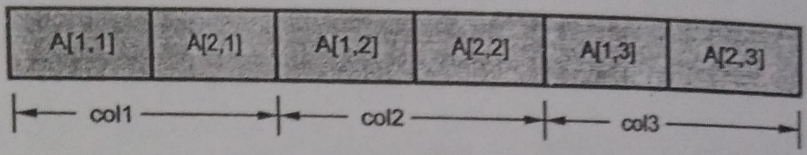


Fig. 4.7.1 Row major and column major representation

To compute the address of any element ;

Let base is the address at a[ ] and w is the width of the element (required memory units) then to compute i^{th} address of a[ ]

base+(i – low) × w

where low is lower bound on subscript. Here a[low]=base

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

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

Let c = base – low × w is computed at compile time. Then the relative address of a[i] can be computed as,

c + (i × w)

Similarly, for calculation of relative address of two dimensional array we need to consider i and j subscripts. Considering, row major representation we will compute the relative address for a[i,j] using following formula :

$$a[i,j] = \text{base} + ((i - \text{low}_1) \times n_2 + (j - \text{low}_2)) \times w$$

where, low_1 and low_2 are the two lower bounds on values of i and j and n_2 is number of values that j can take. In other words;

$$n_2 = \text{high}_2 - \text{low}_2 + 1$$

where, high_2 is the upper bound on j.

Assuming, that i and j are not known at compile time, we can re-write the formula as

$$a[i,j] = ((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

The term $(base - ((i - low_1 \times n_2) + low_2) \times w)$ can be computed at compile time.

For example : Consider an array A of size 10×20 assuming $low_1 = 1$ and $low_2 = 1$. The computation of A[i,j] is possible by assuming that $w = 4$ as :

$$A[i,j] = ((i \times n_2) + j)\ w + (base - ((low_1 \times n_2) + low_2) \times w)$$

Given, $n_2 = 20$

$w = 4$

$low_1 = 1$

$low_2 = 1$

$$A[i,j] = ((i \times 20) + j) \times 4 + (base - ((1 \times 20) + 1) \times 4$$

$$A[i,j] = 4 \times (20i + j) + (base - 84)$$

The value base – 84 can be computed at compile time.

The generalized formula for multi-dimensional array by considering row major representation is -

$$a[i_1, i_2, i_3 \ldots i_k] = ((\ldots((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_k + i_k) \times w +$$
$$base(\ldots((low_1 n_2 + low_2) n_3 + low_3) \ldots) n_k + low_k) \times w$$

where any value of j for n

$n_j = high_j - low_j + 1$

and computation of the term base $(\ldots((low_1 n_2 + low_2) n_3 + low_3) \ldots) n_k + low_k) \times w$ is done at compile time, after getting the base address.

Let us now discuss the translation scheme for generating three address code for array references. We have to consider the context free grammar for array references :

$$L \rightarrow id\ [List] \mid id$$

$$List \rightarrow List, E \mid E$$

List represents the list of index having limit of n_j and E could be any arithmetic expression. This grammar should support any dimensional array reference. Hence we will modify the grammar and rewrite it as :

$$L \rightarrow [List] \mid id$$

$$List \rightarrow List, E \mid id\ [E$$

That means array name is associated with leftmost index expression (i.e. List).The complete grammar for array reference can be given as :

$S \rightarrow L := E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow List\ ]$

$L \rightarrow id$

$List \rightarrow List_1, E$

$List \rightarrow id\ [E$

The translation scheme for generating three address code is given by using appropriate semantic actions.

Rule No.	Production rule	Semantic rule

1)	$S \rightarrow L := E$	{if L.offset := NULL /* L is id only */ then
		append(L.place ':=' E.place)
		else
		append(L.place '[' L.offset']':=E.place)
		}

2)	$E \rightarrow E_1 + E_2$	{E.place := newtemp(); append(E.place ':=' E_1.place'+' E_2.place) }
3)	$E \rightarrow (E_1)$	{E.place := E_1.place }

4)	$E \rightarrow L$	{if L.offset = NULL then
		E.place:=L.place
		else
		{
		E.place := newtemp()
		append(E.place ':=' L.place'[' L.offset']')
		}
		}

5)	$L \rightarrow List\ ]$	{L.place :=newtemp()
		L.offset:=newtemp()
		append(L.place ':=' c(List.array));
		append(L.offset ':=' List.place '*' Size(List.array))
		}

6)	$L \rightarrow id$	{L.place = id.place L.offset = NULL }

7)	$List \rightarrow List_1, E$	{t=newtemp();
		dim=List$_1$.ndim+1;
		append(t := List$_1$.place "*" Limit(List$_1$.array,dim));
		append(t := t "+" E.place)
		List.array := List$_1$.array;
		List.place := t;
		List.ndim := dim;
		}

8)	$List \rightarrow id\ [E$	{List.array := id.place List.place := E.place List.ndim := 1 }

- For the rule 1) if L is simple name then normal assignment is done otherwise the indexed assignment is done at specified location in L. The attribute *offset* will give the relative address for the corresponding entry in the symbol table. And the attribute *place* will refer to a pointer to symbol table entry for that name. Here append is a function used to generate the three address code and append it to output file.
- The rule 2) is for arithmetic operation, in which the newtemp() function is used to generate the temporary variable t_1, t_2... The value of E can be denoted by E.place and computed using E_1.place + E_2.place.
- For the rule $E \rightarrow L$ we want r-value of L. Hence in the semantic actions computation of r-value of L can be done. If L.offset is NULL then it indicates that L is simple id. Otherwise we will use indexing such as L.offset to obtain the E.place.
- To represent the first term (of an expression for a $[i_1, i_2, ..., i_k]$ i.e. $((... (i_1 n_2 + i_2) n_3 + i_3) ...) n_k + i_k) \times w$) L.offset is used and here L.offset is new temporary. Similarly the second term.
- Base $(... ((low_1 n_2 + low_2) n_3 + low_3) ...) n_k + low_k) \times w$ can be represented by L.place. The function c(List.array) is used to obtain the L.place. The function Size(List.array) is used to return the value of width w.
- The L.offset = NULL indicates simple name of id.

In rule (7) the List.ndim indicates number of index expressions (dimensions).Function Limit(List$_1$.array, dim) returns n_j. The n_j means number of

elements along j^{th} dimension of array. The E.place denotes the temporary value obtained by computing value from index expression.

- Consider formula $a[i_1, i_2, i_3 \ldots i_k] = (((\ldots((i_1 n_2 + i_2) n_3 + i_3)\ldots) n_k + i_k) \times w + base\ (\ldots((low_1 n_2 + low_2) n_3 + low_3)\ldots) n_k + low_k) \times w$

The List for expressions can be produced by

$$(\ldots((i_1 n_2 + i_2) n_3 + i_3 \ldots) n_m + i_m$$

Using recurrence relation

$$e_1 = i_1$$

$$e_m = e_m{-}1\ n_m + i_m$$

By rule (7) $List_1$.place corresponds to $e_m{-}1$ and List.place corresponds to e_m

$$\therefore \quad e_m = e_m{-}1\ n_m$$

List.place := $List_1$.place * Limit($List_1$.array, dim)

In rule (8) The value of E is finalized.Hence E.place holds the value of expression E and value of E for having dimension m = 1.

Example for Understanding

Example 4.7.1 *Generate the three address code for the expression x:=A[i,j] for an array 10×20. Assume $low_1 = 1$ and $low_2 = 1$.*

Solution : Given that

$low_1 = 1$ and $low_2 = 1$

$n_1 = 10, n_2 = 20$

$$A[i, j] = ((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

$$A[i, j] = ((i \times 20) + j) \times 4 + (base - ((1 \times 20) + 1) \times 4$$

$$A[i, j] = 4 \times (20i + j) + (base - 84)$$

The three address code for this expression can be,

t_1 := i * 20

t_1 := t_1 + j

t_2 := c /*computation of c = base – 84 */

t_3 := 4 * t_1

t_4 := t_2 [t_3]

x := t_4

The annotated parse tree can be drawn as follows

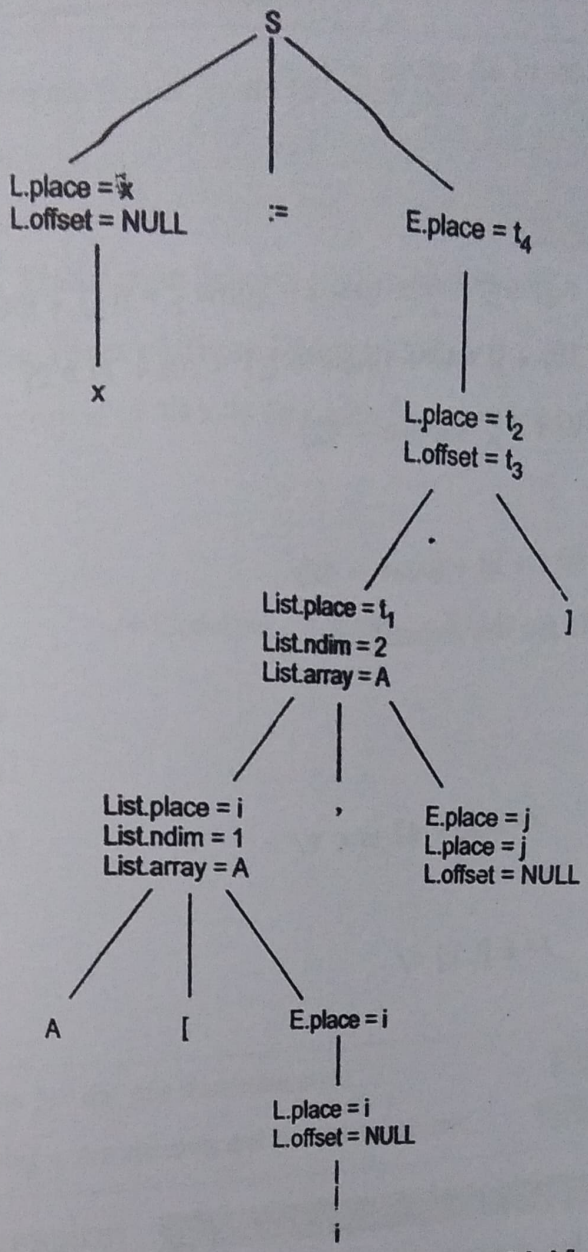


Fig. 4.7.2 Annotated parse tree for x := A[i, j]

Example 4.7.2 *Translate the following integer array operation into three address code. A [i, j] := B [i, j] + C [k] where A and B are of size 10 × 20 and C contains 50 elements.* **AU : May-10, Marks 16**

Solution : We will assume $low_1 = 1$ and $low_2 = 1$ and it is given that $n_1 = 10$ and $n_2 = 20$. Integer element

$$\therefore \quad w = 4 \text{ bytes}$$

$$A[i, j] = ((i \times n_2) + j) \times w + \left(base_A - ((low_1 \times n_2) + low_2) \times w\right)$$

$$= ((i \times 20) + j) \times 4 + \left(base_A - ((1 \times 20) + 1) \times 4\right)$$

$$= 4 \times (20i + j) + (base_A - 84)$$

Hence the three address code for A [i, j] will be

$t_0 := i * 20$

$t_1 := t_0 + j$

$t_2 := c_1$ /* computation of $c_1 = base_A - 84$ */

$t_3 := 4 * t_1$

$t_4 := t_2[t_3]$ /* A [i, j] */

Similarly the value for B [i, j] can be computed as

$t_5 := c_2$ /* computation of $c_2 = base_B - 84$ */

$t_6 := t_5[t_3]$ /* B [i, j] */

The value for C [k] will be obtained as follows

$$k \times w + (base_c - low_1 \times w)$$

$$= k \times 4 + (base_c - 1 \times 4)$$

$$C[k] = 4k + c_3$$

Hence three address code for c[k] will be

$t_7 := 4 * k$

$t_8 := c_3$ /* computation of $c_3 = base_c - 4$ */

$t_9 := t_8[t_7]$

Hence the 3 address code for given expression will be -

$t_0 := j * 20$

$t_1 := t_0 + j$

$t_3 := 4 * t_1$

$t_5 := c_2$

$t_6 := t_5[t_3]$ /* B [i, j] */

$t_7 := 4 * k$

$t_8 := c_3$

$t_9 := t_8[t_7]$ /* C [k] */

$t_{10} := t_6 + t_9$ /* B [i, j] + C[k] */

$t_2 := c_1$

$t_2 [t_3] := t_{10}$ /* A [i, j] := B [i, j] + C [k] */