Semantics & Intermediate Representation

Arrays

(Homogeneous non-atomic elements)

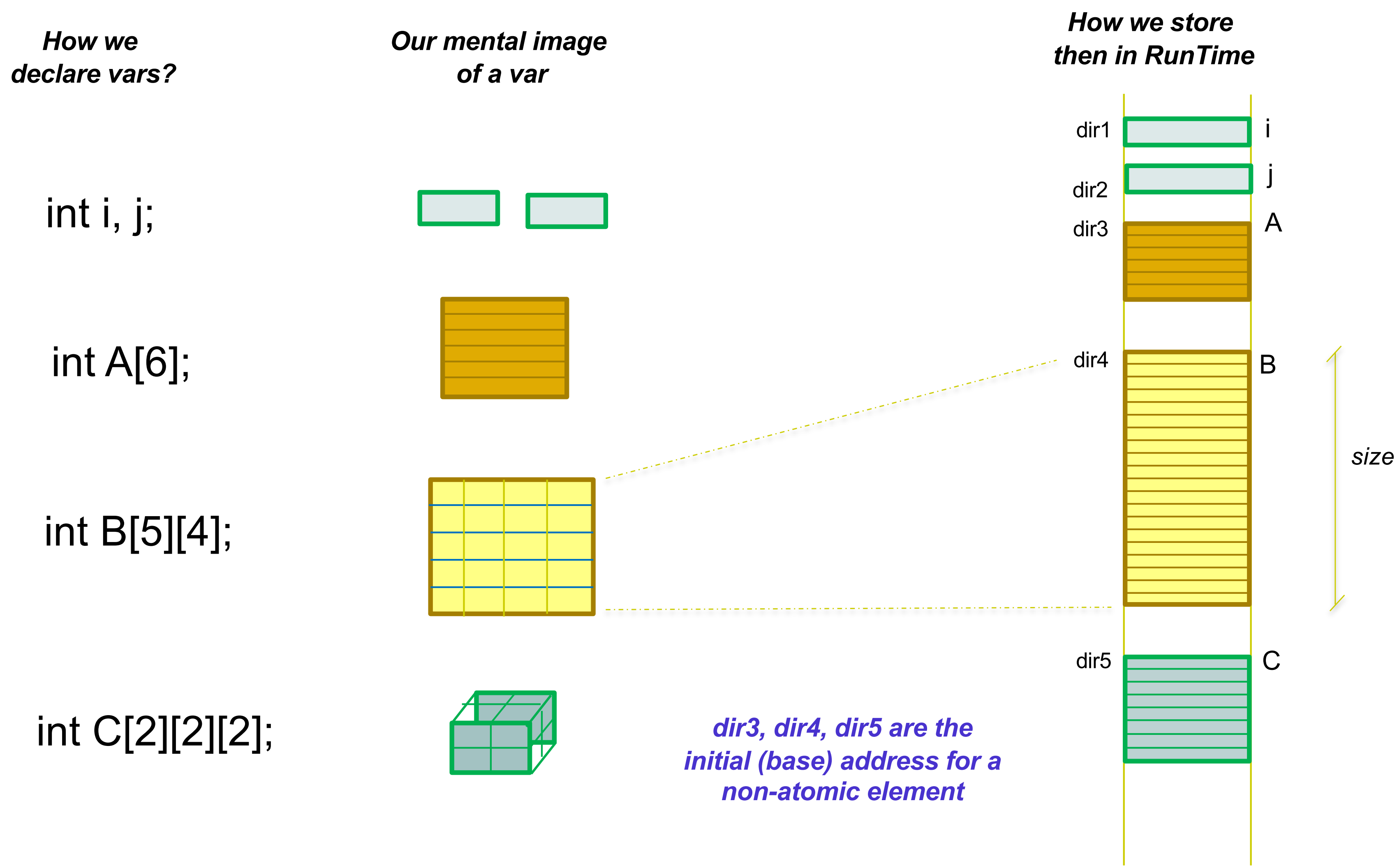
ARRAYS

- ***One of the most important features in any programming language within the Imperative Paradigm (and also in OO) is the possibility to identify under ONE single name a "memory chunk" that stores multiple homogeneous values.***

- ***This feature allow us to avoid declaring multiple variables to store "related-values".***

- ***Its a "chunk" of continuous memory that stores same-type (homogeneous) values.***

How do we store Arrays?



How do we store Arrays?

How we declare vars?

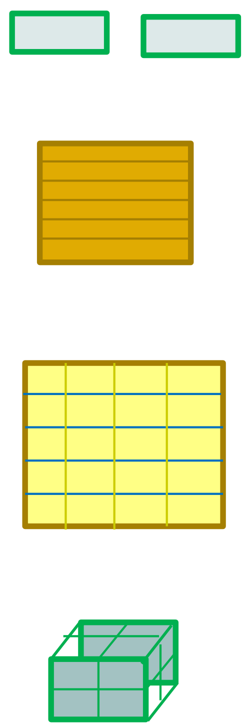
int i, j;

int A[6];

int B[5][4];

int C[2][2][2];

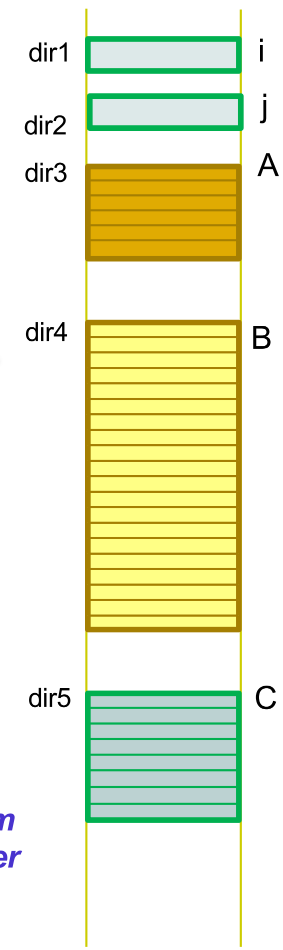
Our mental image of a var



It requires a function that "flattens" the non-atomic element

How to determine, in compile-time, which is the specific cell for the item B [i + j] [j * 2-i]? (knowing that neither i nor j have values yet).

How we store then in RunTime



¿ B [i + j] [j * 2-i] ?

How do we access an element of the Array?

- ***We have the initial address of the array and, based on, that we must "calculate" the required offset (f) to access a specific element within the array.***

- ***The main issue in Compile-time is that we don't have VALUES, so there's no way we can "calculate" anything; but we can add all the calculations to the intermediate code and let the Virtual Machine execute them to address a specific cell.***

$$\text{Address}(\ id[s_1,s_2,..s_n] = \text{BaseAddress}(id) + f(id[s_1,s_2,..s_n])$$

Formula Traslation for the MOST general Array declaration

Lets have this array declaration:

int **id[$linf_1..lsup_1$][$linf_2..lsup_2$] ... [$linf_n..lsup_n$]**

where **$linf_x$** stands for lower limit (not necessarily 0) and **$lsup_x$** for the upper limit

this variable is accessed as:

id [s_1 , s_2 ,.. s_n]

where **s_k** stands for the indexing expression

The formula to calculate the addess of a single cell is:

$$\mathbf{Address(\ id[s_1,s_2,..s_n] = BaseAddress(id) + \mathit{f}(id[s_1,s_2,..s_n])}$$

$$\mathbf{Address(\ id\ [s_1,s_2,..s_n]) = BaseA(id) + (s_1-linf_1)*d_2*d_3*....*d_n + (s_2 - linf_2)*d_3*d_4*.... * d_n + ... + (s_{n-1}- linf_{n-1})*d_n + (s_n- linf_n)}$$

where d_k stands for the "size" of dimension-i = **$lsup_k - linf_k + 1$**

Formula Traslation for the MOST general Array declaration

Let's apply a little algebra to make it simple:

$m_1 = d_2 * d_3 * .... * d_n$
$m_2 = d_3 * .... * d_n$
..
$m_{n-1} = d_n$
$m_n = 1$

$Address(\ id\ [s_1, s_2, .. s_n]) = BaseA(id) + s_1 * m_1 + s_2 * m_2 + ... + s_{n-1} * m_{n-1} + s_n + K$

where K stands for: **$K = -(linf_1 * m_1 + linf_2 * m_2 + ... + linf_n)$**

this K is a “constant” that transfer all dimensions to 0 (for non-zero lower-limits).

If the Array declaration is C-style, this constant is 0.

Formula Traslation for the MOST general Array declaration

Let's make it even simpler:

General formula for N-Dimensions

$Address(id [s_1,s_2,..s_n]) = BaseA(id) + s_1*m_1 + s_2*m_2 + ... + s_{n-1}*m_{n-1} + s_n + K$

Formula for 1 Dimension:

$Address(id [s_1]) = BaseA(id) + s_1 + K$

Formula for 2 Dimensions:

$Address(id [s_1,s_2]) = BaseA(id) + s_1*m_1 + s_2 + K$

Formula for 3 Dimensions:

$Address(id [s_1,s_2, s_3]) = BaseA(id) + s_1*m_1 + s_2 * m_2 + s_3 + K$

...

Formula Traslation for a C-style Array declaration

General formula for N-Dimensions

$\text{Address(id }[s_1,s_2,..s_n]) = \text{BaseA(id)} + s_1*m_1 + s_2*m_2 + ... + s_{n-1}*m_{n-1} + s_n$

Formula for 1 Dimension:

$\text{Address(id }[s_1]) = \text{BaseA(id)} + s_1$

Formula for 2 Dimensions:

$\text{Address(id }[s_1,s_2]) = \text{BaseA(id)} + s_1*d_2 + s_2$

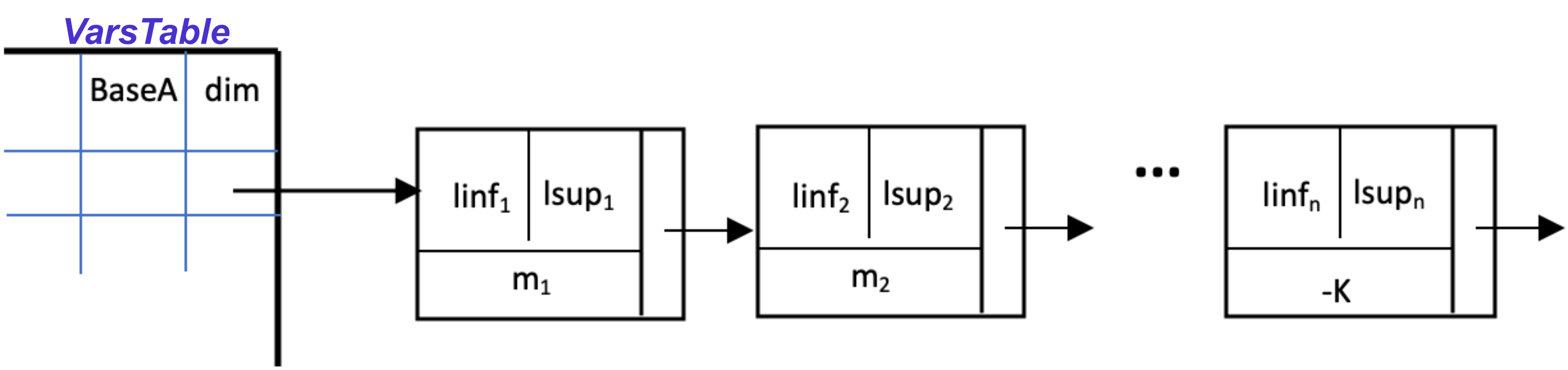
Formula for 3 Dimensions:

$\text{Address(id }[s_1,s_2, s_3]) = \text{BaseA(id)} + s_1*d_2 *d_3 + s_2 * d_3 + s_3$

...

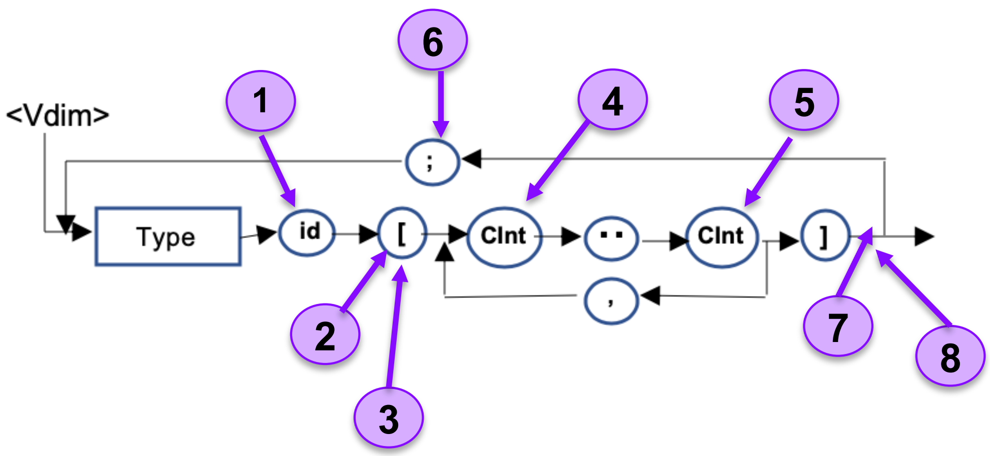
Important information to store for the MOST general Array declaration

- **In Compile-Time we can not perform any calculation because there are NO values.**
- **We'll have to convert the formula into Quadruples so the Virtual Machine can obtain results.**
- **That means we'll need to store all the elements (constant and calculated) in a structure, to make them available every time we try to address a specific cell from an Array.** **(trying to avoid extra-operations or even useless re-calculations)**
- **We'll add more attributes to our VarsTable.**



Intermediate Representation for an ARRAY

- ***Array Declaration***

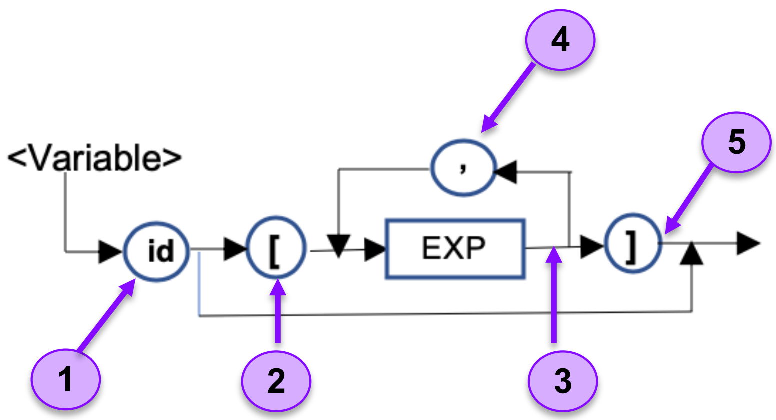


1. VarTable.add(id, type)
2. Set id as an array (isArray = true)
3. Get a new node to save info about dimensions and link it to the id.
   Set DIM=1, R=1
4. Store Li_{DIM}
5. Store Ls_{DIM}
   $R = (Ls_{DIM} - Li_{DIM} + 1) * R$
6. DIM = DIM + 1
   Get a new node and link it to the previous one.
7. Link the last node to null.
   Go to the first node in the list.
   DIM = 1 ; OffSet = 0 ; Size = R;
   REPEAT
      $m_{DIM} = R / (Ls_{DIM} - Li_{DIM} + 1)$
      Store m in the current node
      $R = m_{DIM}$
      $OffSet = OffSet + Li_{DIM} * m_{DIM}$
      DIM = DIM + 1 (Move to next node)
   UNTIL no mode nodes
   K = OffSet ** *K is the constant in the formula*
   Store **(- K**) in the last node.

8. Store **VirtualAddress** in the current **id** in VarsTable
   calculate next VirtualAddress = VirtualAddress + Size

Intermediate Representation for an ARRAY

- ***Array Access***



1. PilaO.push(id) and PTipos.push(type)
2. Id = PilaO.pop(); type = PTipos.pop()
   verify that id has dimensions
   DIM = 1
   PilaDim.push(id, DIM)
   Get the first node of dimensions (List).
   POper.push(FakeBottom)
3. Create Quadruple:
   Verify PilaO.Top Li_{DIM} Ls_{DIM}. *** Note: NO Pop at this time*
   If NextPointer(List)
   { aux = PilaO.Pop()
   Create quadruple: * aux. m_{DIM} T_j where T_j is next available temp.
   PilaO.Push (T_j) }
   If DIM > 1
   { aux2 = PilaO.Pop(); aux1 = PilaO.Pop();
   Create quadruple: + aux1 aux2 T_k where T_k is next available temp.
   PilaO.Push (T_k) }

4. DIM = DIM + 1
   Update DIM in PilaDIM
   Move to next node in List

5. Aux1= PilaO.Pop()
   Create quadruples:
   + aux1 K T_i where T_i is next available temp. K is the Offset
   + T_i VirtualAdress T_n where T_n is next available temp.
   PilaO.Push(**(** T_n **)**). *Contains an ADDRESS, that's why we use () to distinguish it.*
   POper.Pop(). *Eliminates FakeBottom*