Implementing scoping rules

Based on the slides of Maurizio Gabbrielli

Implementing scope rules

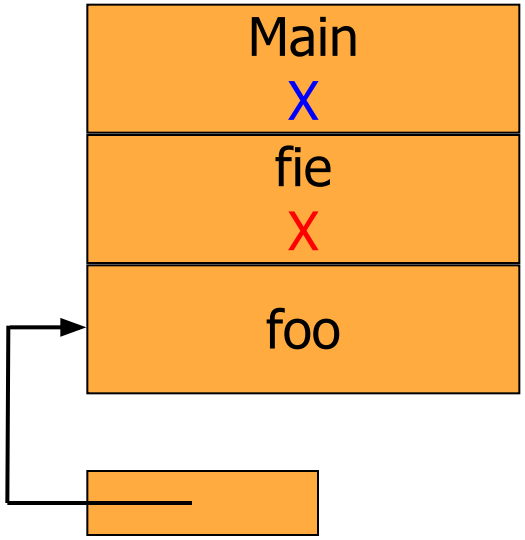
- Static Scope
  - Static chain
  - Display
- Dynamic Scope
  - A-List
  - Central Referencing Environment Table (CRT)

How do you determine the correct link?

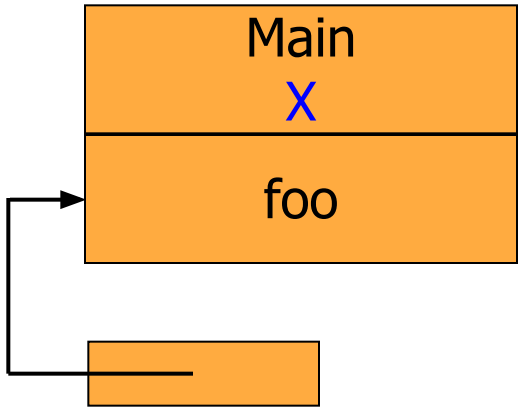
```
{int x= 10;
void foo () {
   x++;
   }
void fie () {
   Int x= 0;
   foo ();
   }
fie ();
foo ();
}
```

- The Code of `foo` must always access the same variable x
- This x is stored in a certain AR (in this case in the *Main*)
- At the top of the stack we have the AR of `foo` (because `foo` is running)

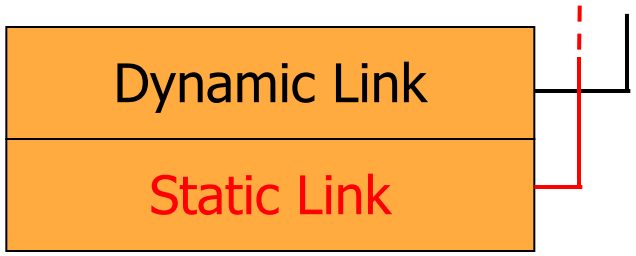
First case:
foo called in fie



Second case:
foo called from main



Activation Record for static scoping

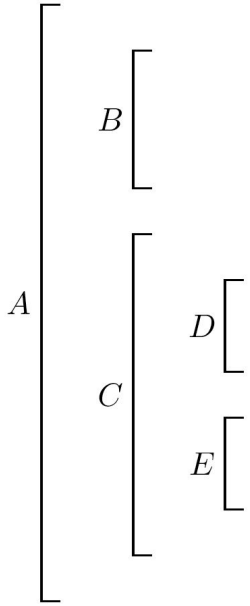


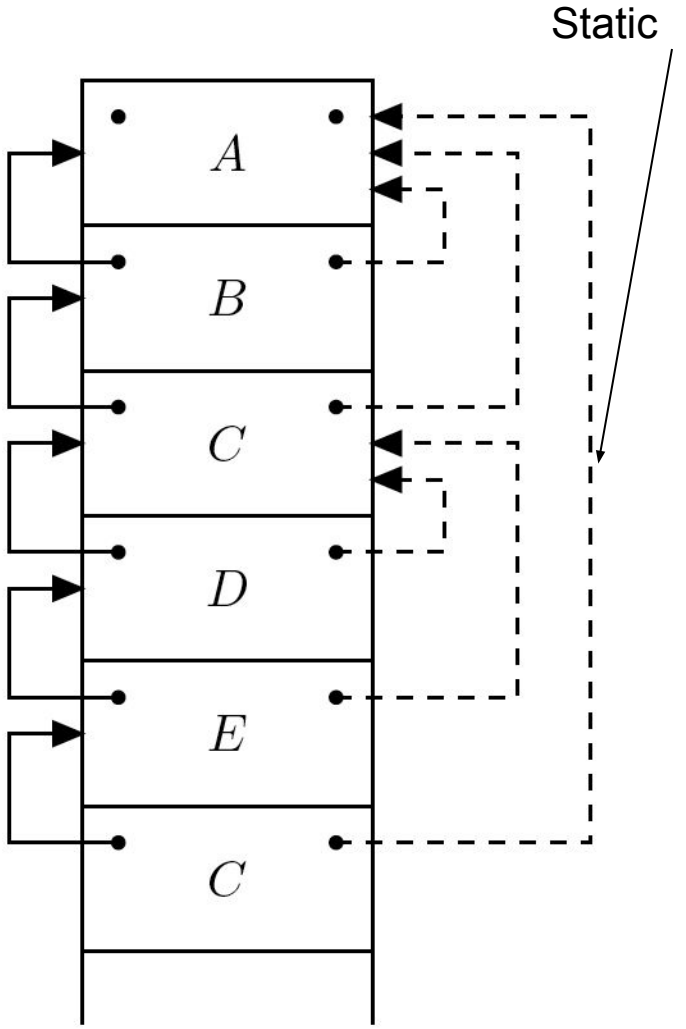
- Dynamic Link:
  - Pointer to previous AR on stack (caller AR)
- Static Link:
  - Pointer to the AR block that immediately contains the text of the running block

- Dynamic link depends on the sequence of execution of the program
- Static link depends on static nesting (in text) of procedure declarations

Static Chain: Example

- Sequence of calls to run time
  A, B, C, D, E, C





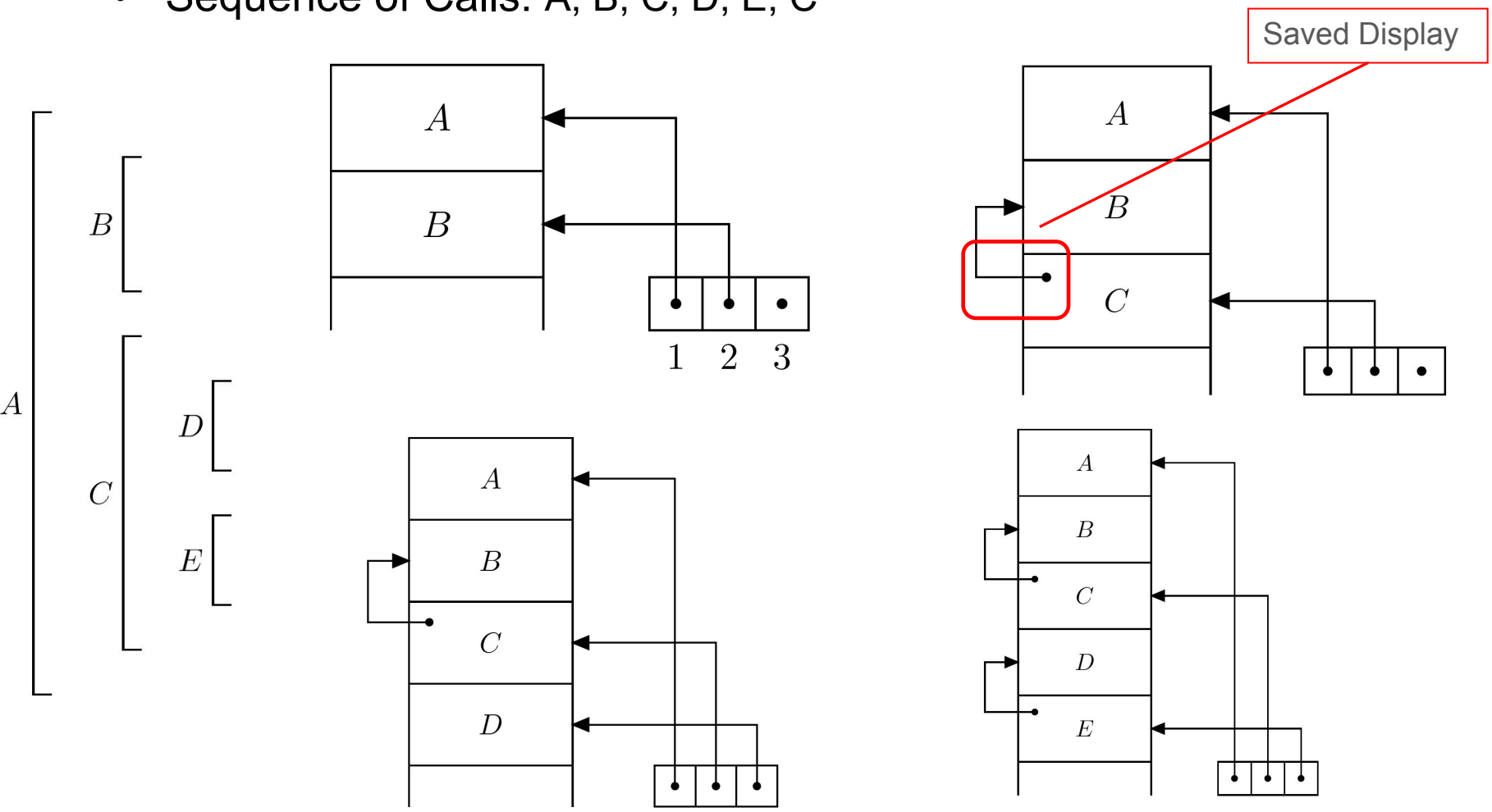
If a sub-program is nested at the level *K*, then the chain is long *K*

Allocation of tasks

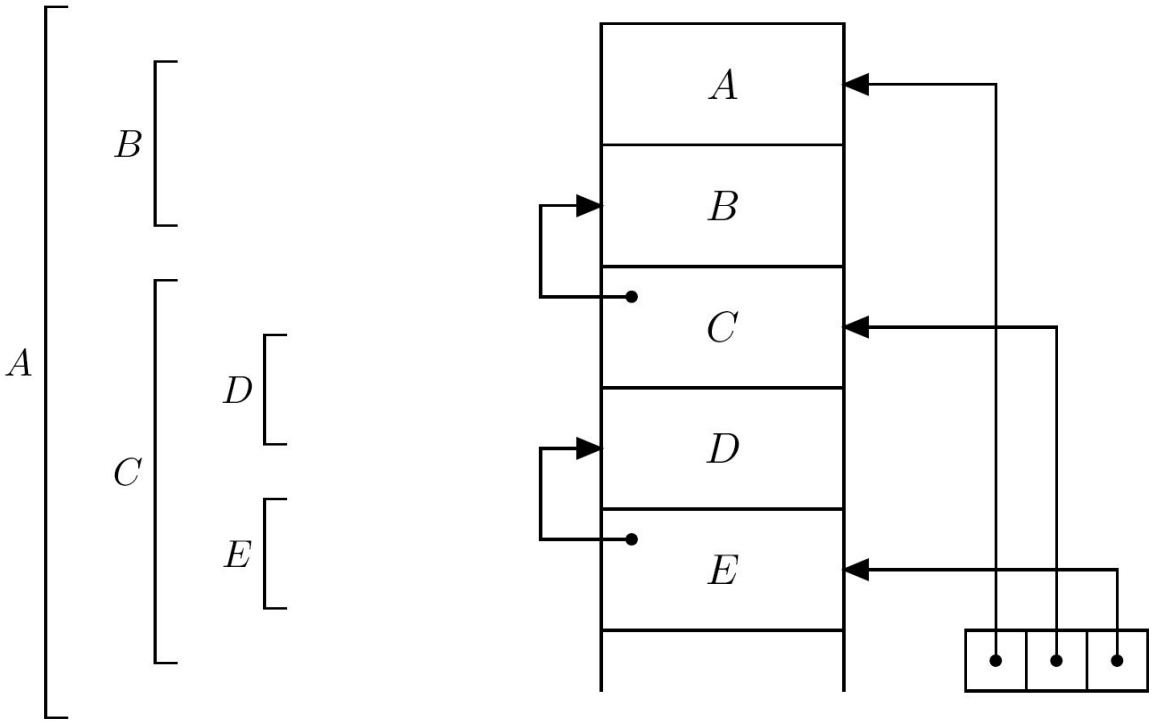
- Compiler:
  - Associates the information K at every call
  - Associates to every name an index K:
    - K = 0: local name
    - K ≠ 0: Non-local name defined h blocks above
- Costs
  - For every access to a non-local variable
    - K static chain steps more than access to a local

We try to reduce costs: the *Display*

- *Display*[i] = pointer to AR of procedure level i, last active
- Sequence of Calls: A, B, C, D, E, C



Display



With *Display*, an object is found with two accesses: one for the display and one for the object

Display or static chain?

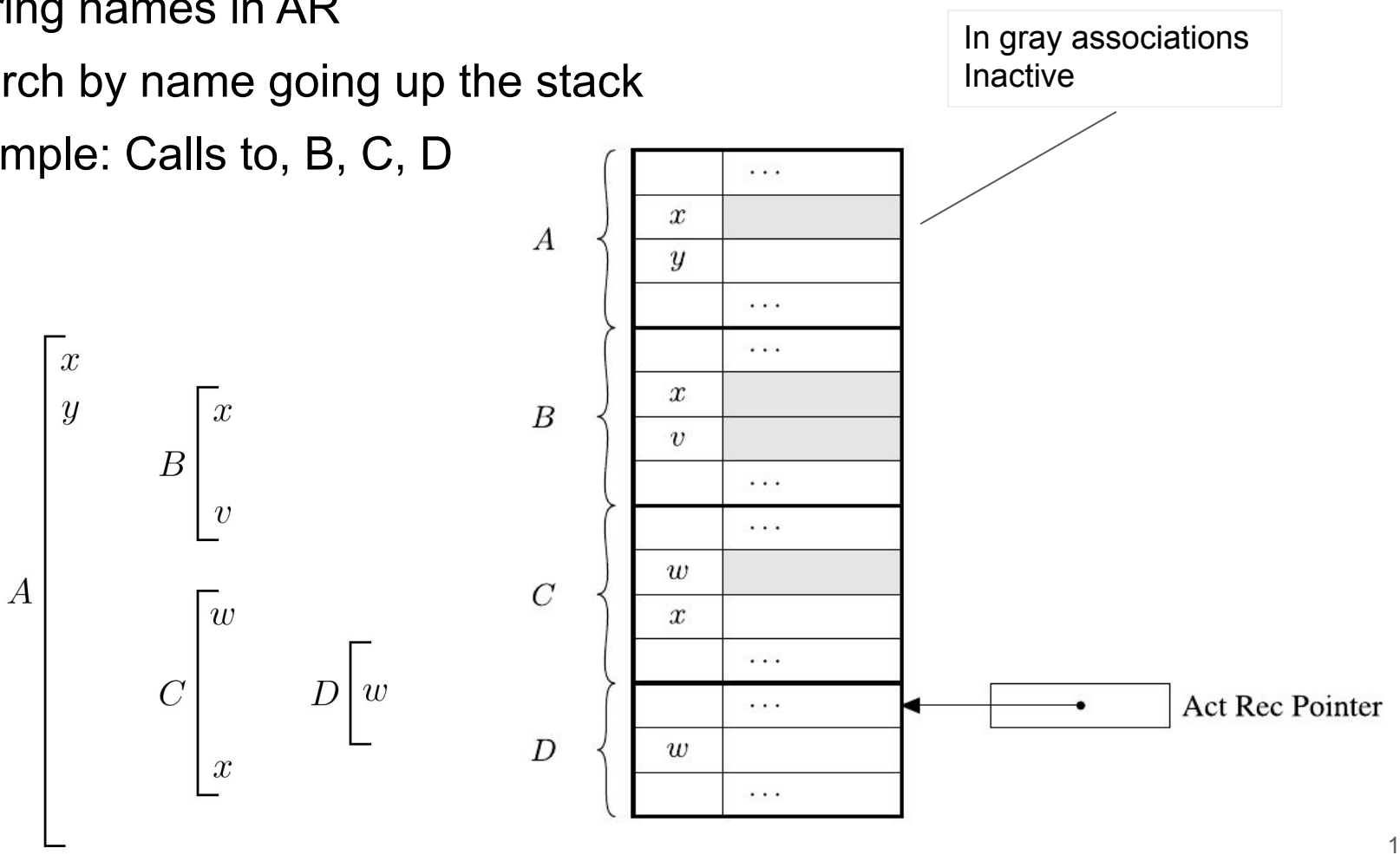
- Rare depth nesting > 3, so max length of static chain = 3
- Optimization techniques can improve access to frequently used chains (keeping pointers in registers)
- The display is more expensive to maintain than the static chain in the call sequence...
- Conclusion: display little used in modern implementations...

Dynamic Scope

- With dynamic scope the name-object association depends on
  - the flow of control at run-time
  - the order in which the sub-programmes are called
- The general rule is simple: the current binding for a name is the last one determined in the execution (not yet destroyed)

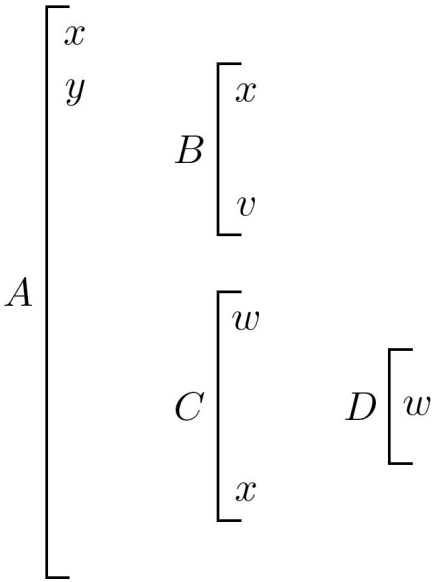
Obvious implementation

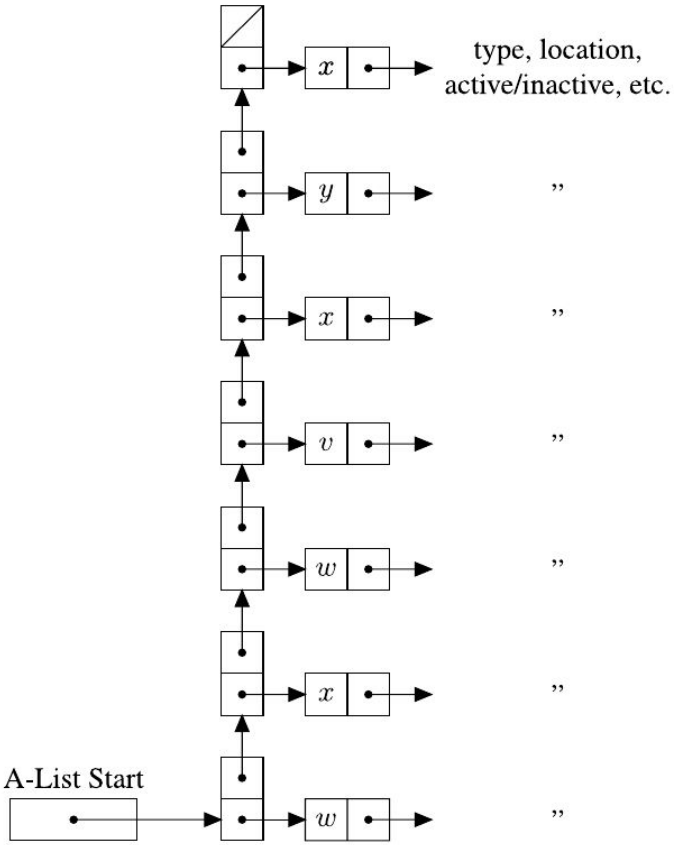
- Storing names in AR
- Search by name going up the stack
- Example: Calls to, B, C, D



Variant: A-List

- The associations are stored in a special structure, used like a stack
- Example: Calls to, B, C, D





A-List costs

- Very easy to implement
- Occupation Memory:
  - names explicitly present
- Cost of management
  - entering and exiting from the block
    - inserting/removing blocks on the stack
- Access time
  - Always linear in the depth of the A-list

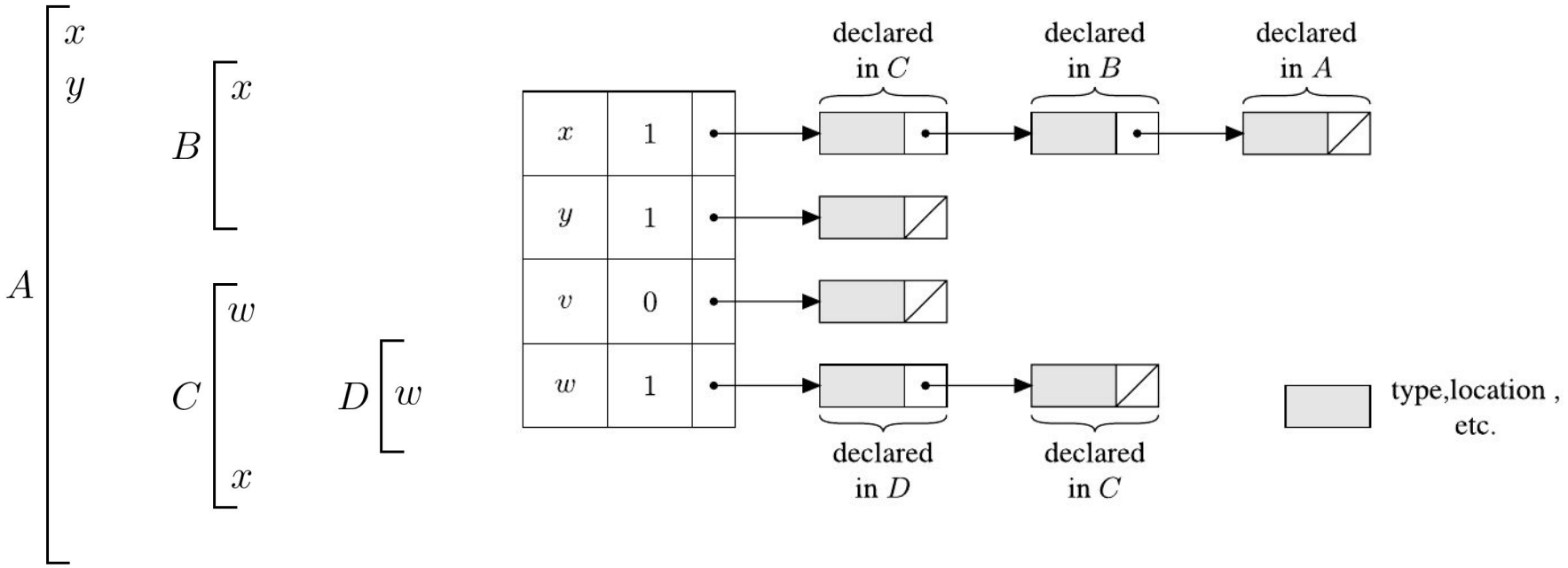
Can we do it better?

Central referencing table (CRT)

- Avoid long scans of A-list
- A table maintains all the distinguished names of the program
  - If the names are known statically, you can access the table element in constant time
  - Otherwise, hash access
- Each name is associated with the list of associations of that name
  - The most recent is the first
  - The others (deactivated) follow
- Constant access time

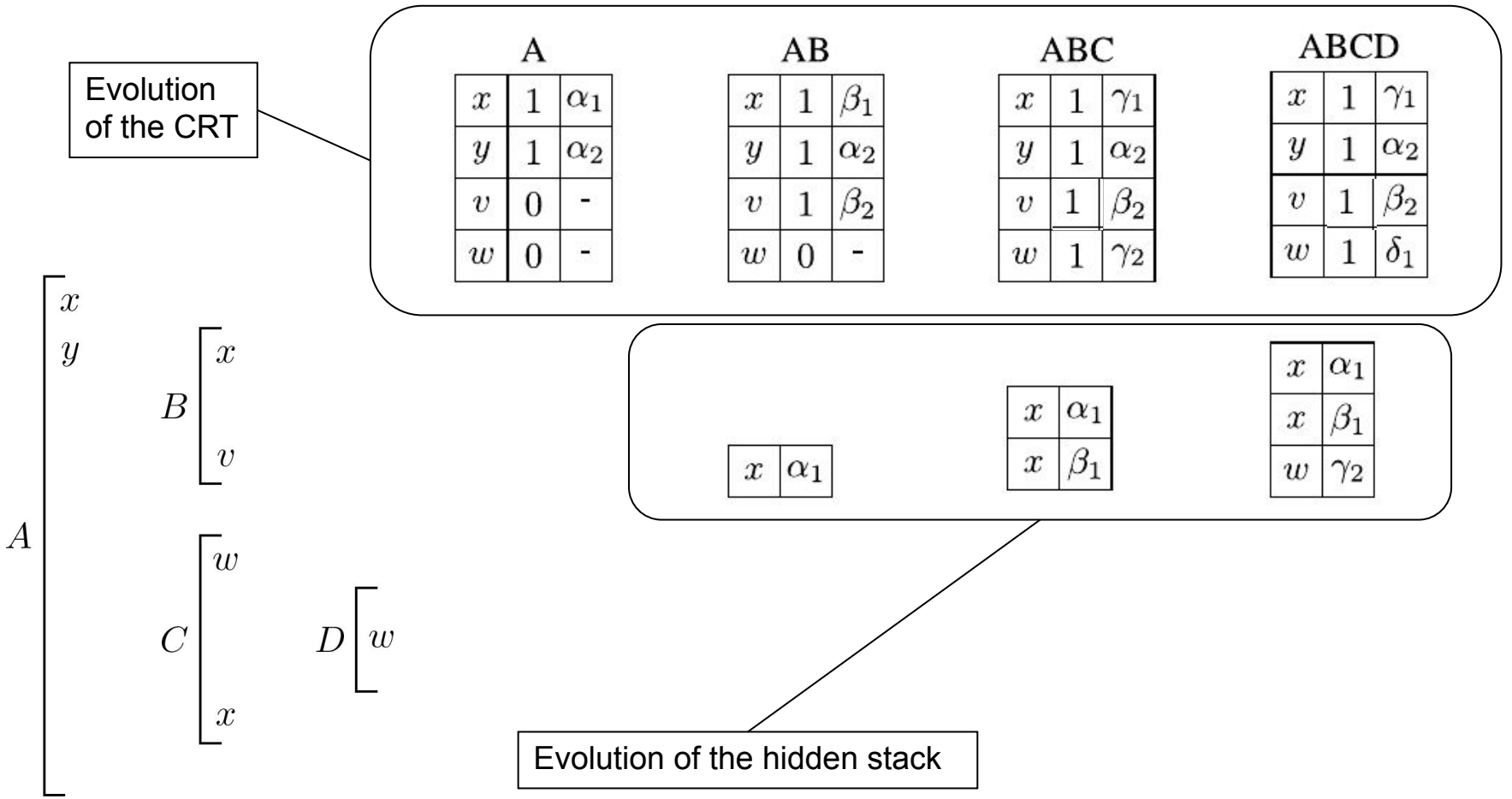
Example (CRT)

- Example: Calls to, A, B, C, D



CRT with hidden stack

- Example: Calls to A, B, C, D



CRT costs

- More complex management than A-List
- Less memory occupancy:
  - If names are statically known, names are not needed
  - In any case, each name stored only once
- Cost of management
  - entering and exiting from the block
    - managing all lists of names defined in the block
- Access time
  - Constant (two indirect accesses)

Suggested Exercises

- Chapter 5 exercises 1,2,3,4,6