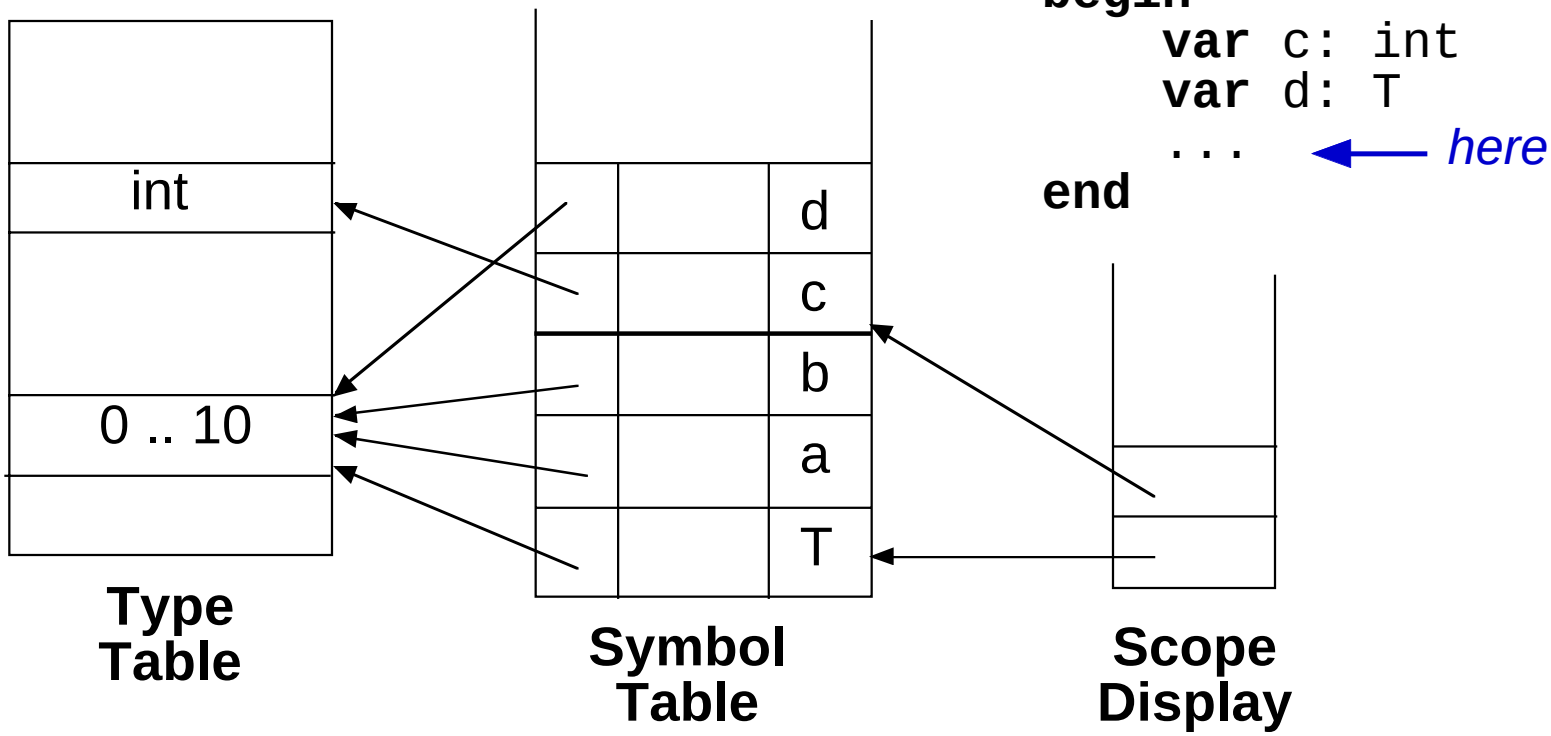
# Recall ...

- Symbol table for Algol family of languages uses a stack-structured Symbol Table, organized into frames by a Scope Display
- Separate Type Table allows for user-defined types

```
type T = 0..10
var a,b: T
begin
    var c: int
    var d: T
    ...          <-- here
end
```



# Exports, Public & Private

- Modular and OO languages (Euclid, Turing, Modula, C++, Java, ...) have nested scopes that can be modified by explicit user control
- All of these languages have export controls which permit some symbols to be visible from outside of the scope (e.g. public in Java) and some not (private)
- Example (Turing):
  - only exported (public) symbols can be used outside a module

```
module M
    export P
    procedure Q
      ...
    end Q
    procedure P
      ...
    end P
    procedure R
      ...
    end R
end M


    .
    .
    .

  M.P       ok
  M.Q       error!
```

# Imports

- Some of these languages (Euclid, Turing, Ada) also have import controls that determine which symbols from the outer scope may be used inside
- Scopes without import controls are called open scopes, and act like normal Pascal scopes (everything outside is visible inside)
- Scopes with import controls the are closed scopes - in a closed scope, only those symbols outside the scope that are explicitly imported into it may be accessed

# Imports

- Example (Turing):
  - only imported symbols can be used inside a module

```
var x: int
var y: int

module M
    export P
    import x

    var w := x      ok, x imported
    var v := y      error, y not imported

    procedure Q
     var z := w     ok, open scope so w visible
    end Q

    procedure P
     var u := x     ok, x imported then open scope
     var t := y     error, y not imported
    end P
end M
```

# Imports with Access Control

- Some languages also allow for explicit *access control*
- Example (Turing):
  - imports may be *readonly*

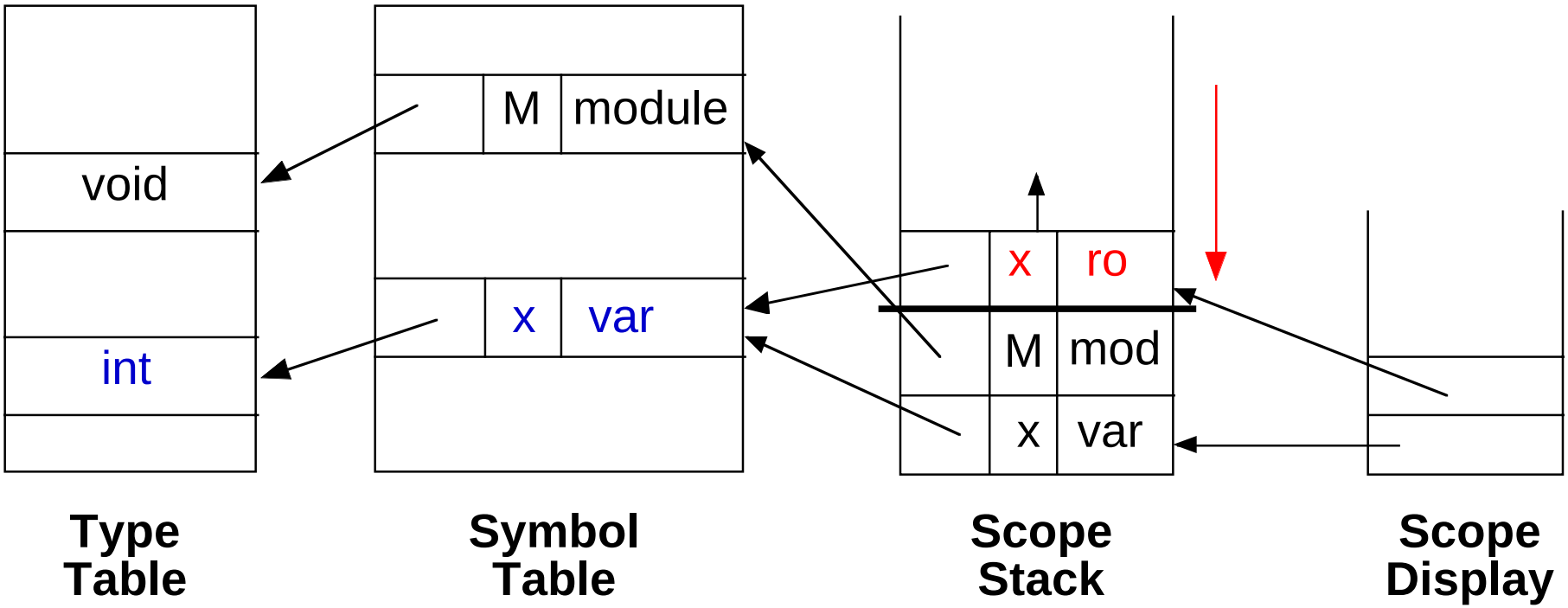
```
var x: int
var y: int

module M
    export P
    import readonly x

    procedure P
     var u := x      ok, x imported then open scope
     x := uc         error, x readonly here
    end P
end M

x := y               ok, x not readonly here
```

# Imports/Exports

- Import controls are handled by separating the Symbol Table into two parts, one to handle *visibility* of symbols in scopes, the other to hold their *definitions* (and other permanent attributes)
- The first part models the scope dependent information (visibility, scope dependent attributes such as *readonly*) for symbols - we call this the *Scope Stack*
- The second part holds the scope-independent information for the symbols - we continue to call this the *Symbol Table*
- Referenced symbols are looked for in the *Scope Stack* from the top down, stopping at the first closed scope boundary
- Their attributes are a combination of the scope dependent ones in the *Scope Stack* and the scope independent ones in the *Symbol Table*

# Imports with Access Control – Example

```
var x: int

module M
    import readonly x
    x := 5                    ← error!
end M
```



# Saved Scopes

- Information about inner scopes that can be later accessed using field selection or parameterization operations must be saved so that the items in the scopes are available later

- Examples:

  - record fields *R.x*

    – need *R*'s scope to find *x*
    (similarly for module exports, class members, etc.)

  - procedure arguments *P(y,z)*

    – need *P*'s formal parameter list to find the corresponding formal parameter types for arguments *y* and *z*

# Saved Scopes

- Recall that the parameters are part of the internal scope of the procedure, but the type for each parameter must be available outside (at call sites) so that it can be checked

- Similarly, record fields are in an internal scope (since a record field name may be the same as a variable in the outer scope), but the record field names must be available outside the record so that field references can be resolved - similarly for class members

```
var x:
  record
    a: int
    b: real
  end record
      .
      .
      .
```

```
      .
      .
      .
y := x.a
      .
      .
      .
```

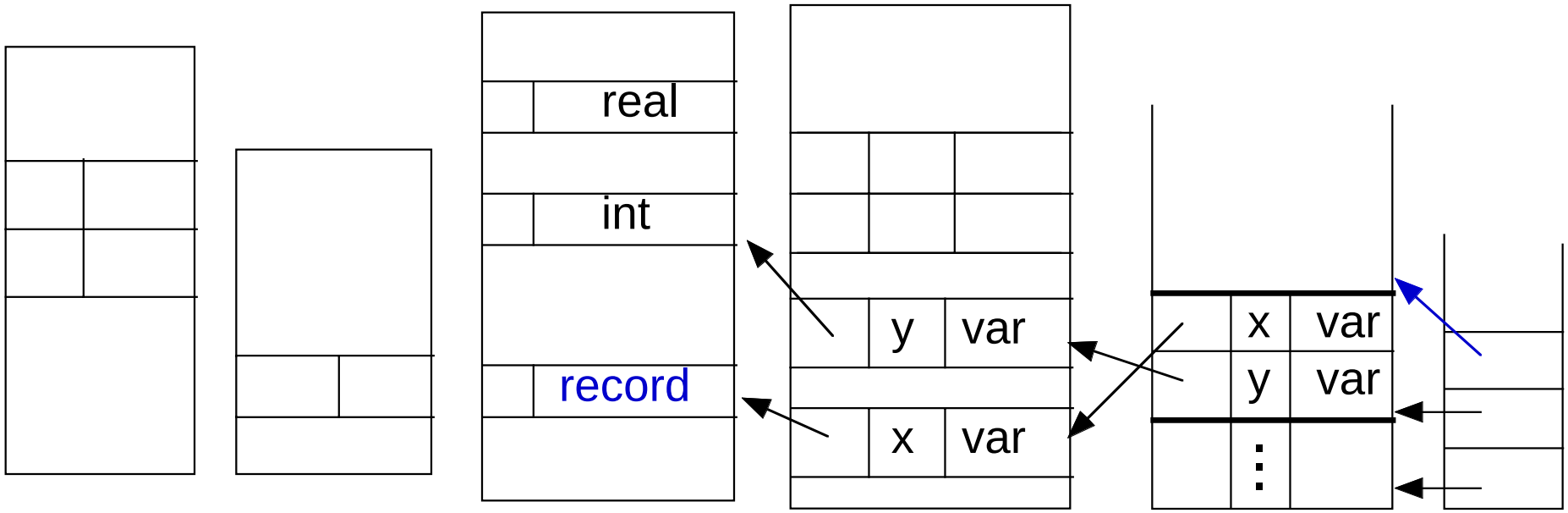
# Saved Scopes

- We can resolve this with a new table, designed to store scopes after we have processed them - the Scope Table
- When we hit a field selector (such as *x.y* where *x* is a record of type *R* with field *y*), we fetch the saved scope for record type *R* from the Scope Table and push it on the Scope Stack
- That is, we make its symbols visible again, as if they were in a new local scope
- We can then look up field *y* in that scope in the normal way (to see if it is a field of *x* or not) and find its attributes in the Symbol and Type Tables, just as if it were a local variable
- Once the field reference is resolved, we pop the scope from the Scope Stack

# Saved Scopes – Example

```
var y: int
var x:
  record
    a: int
    b: real
  end record
```

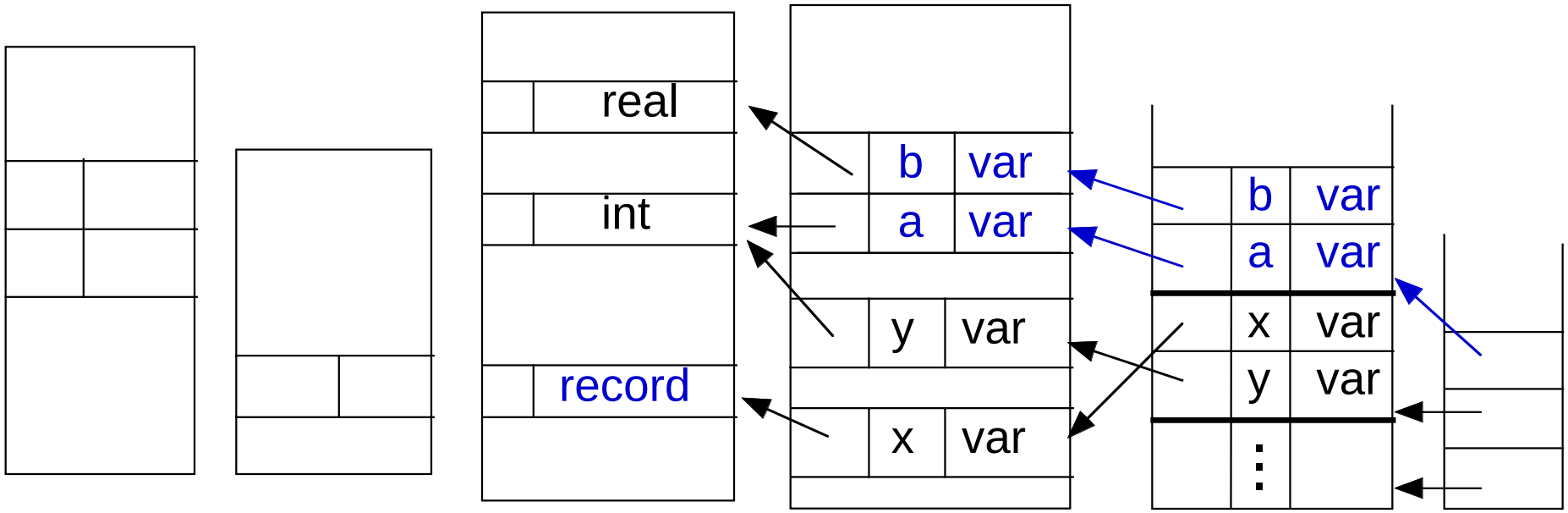
```
...
y := x.a
...
```



# Saved Scopes – Example

```
var y: int
var x:
  record
    a: int
    b: real
  end record
```

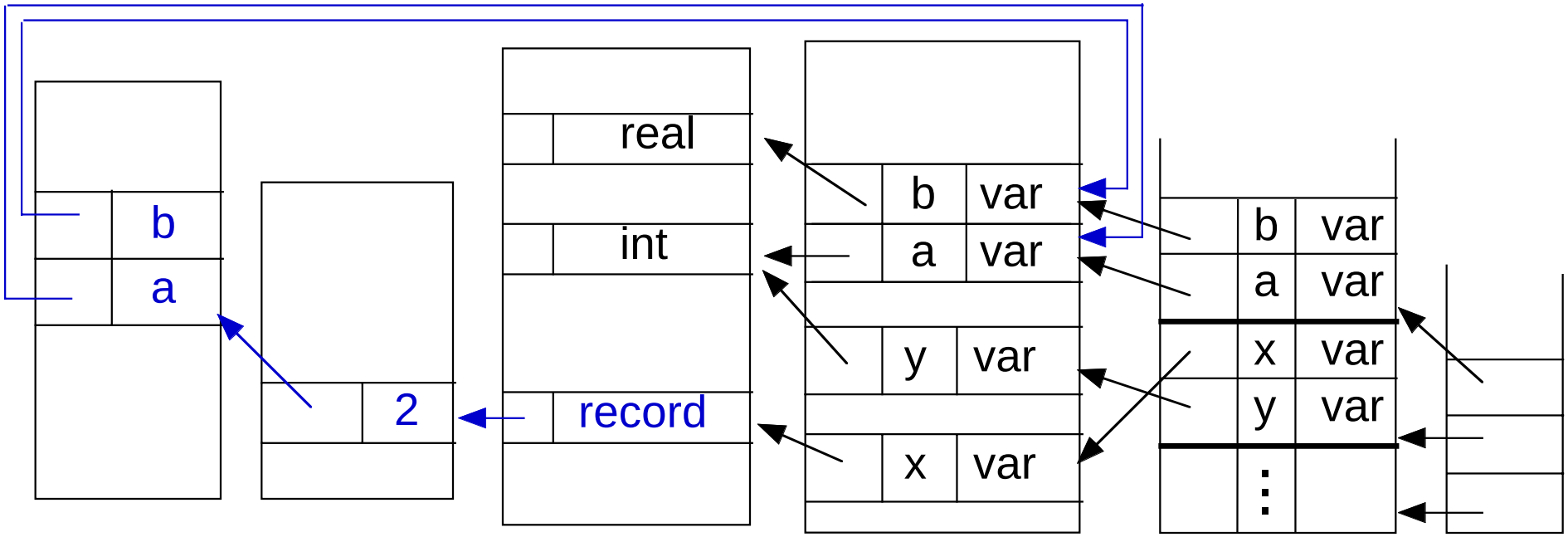
```
...
y := x.a
...
```



# Saved Scopes – Example

```
var y: int
var x:
  record
    a: int
    b: real
  end record
```

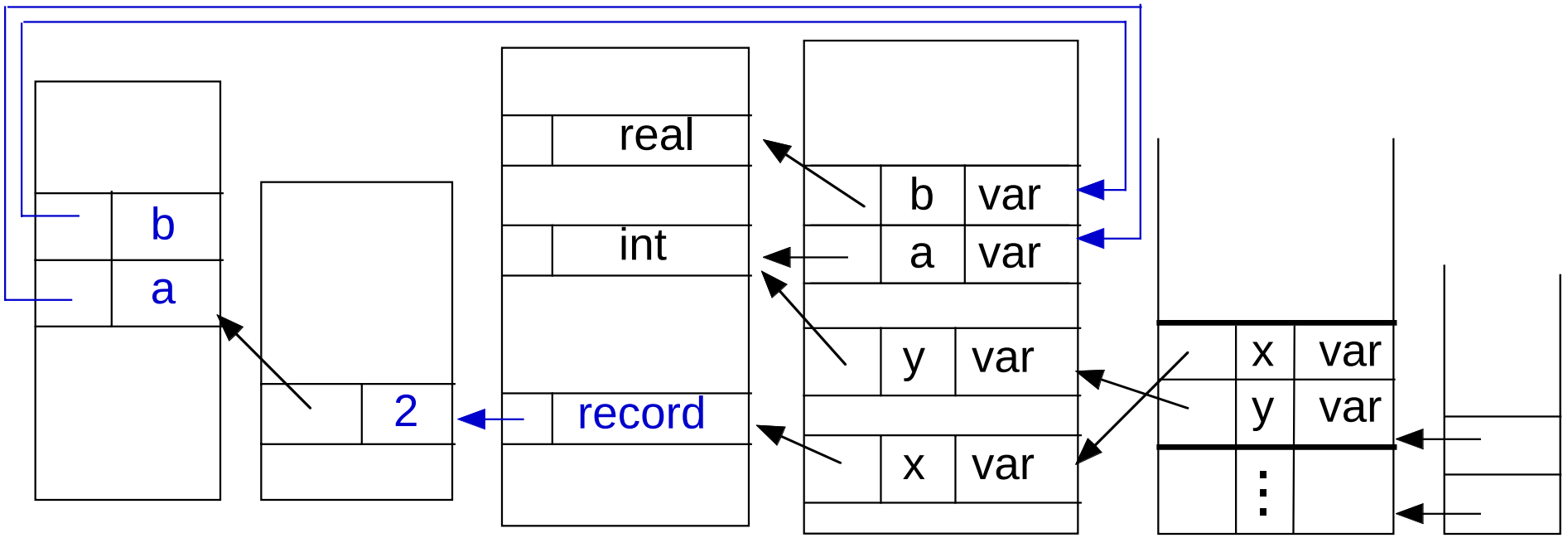
```
...
y := x.a
...
```



# Saved Scopes – Example

```
var y: int
var x:
  record
    a: int
    b: real
  end record
```

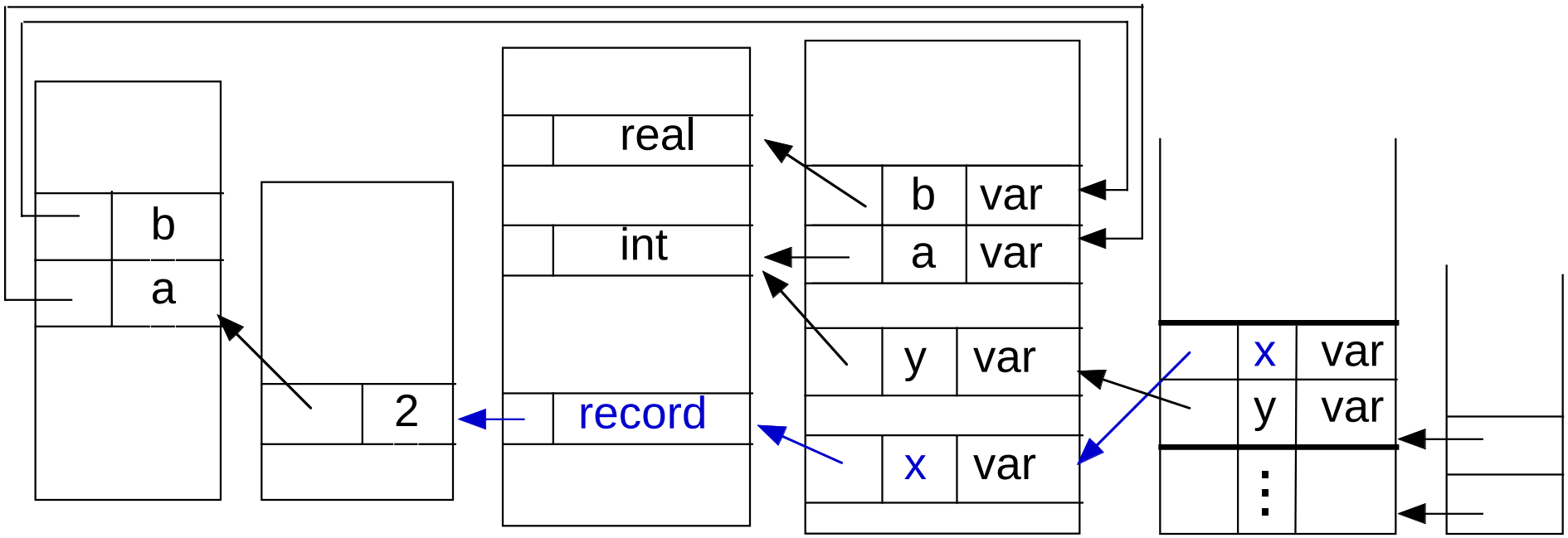
```
...
y := x.a
...
```



# Saved Scopes – Example

```
var y: int
var x:
  record
    a: int
    b: real
  end record
```

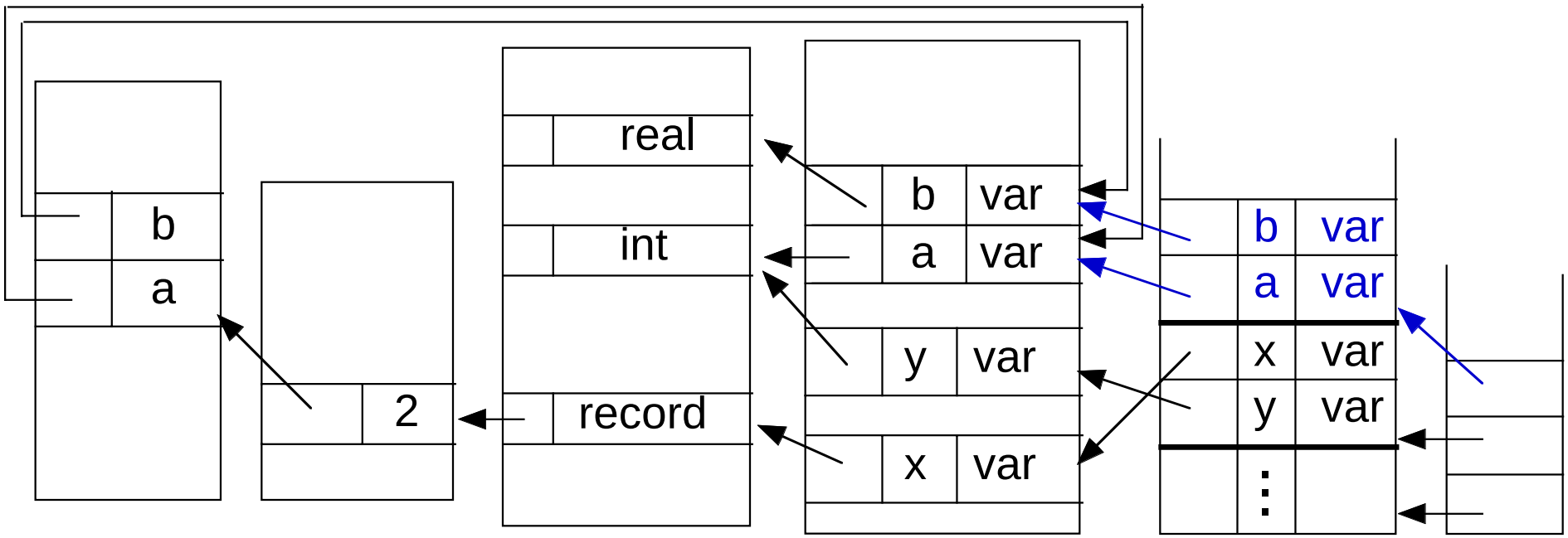
```
...
y := x.a
...
```



# Saved Scopes – Example

```
var y: int
var x:
  record
    a: int
    b: real
  end record
```

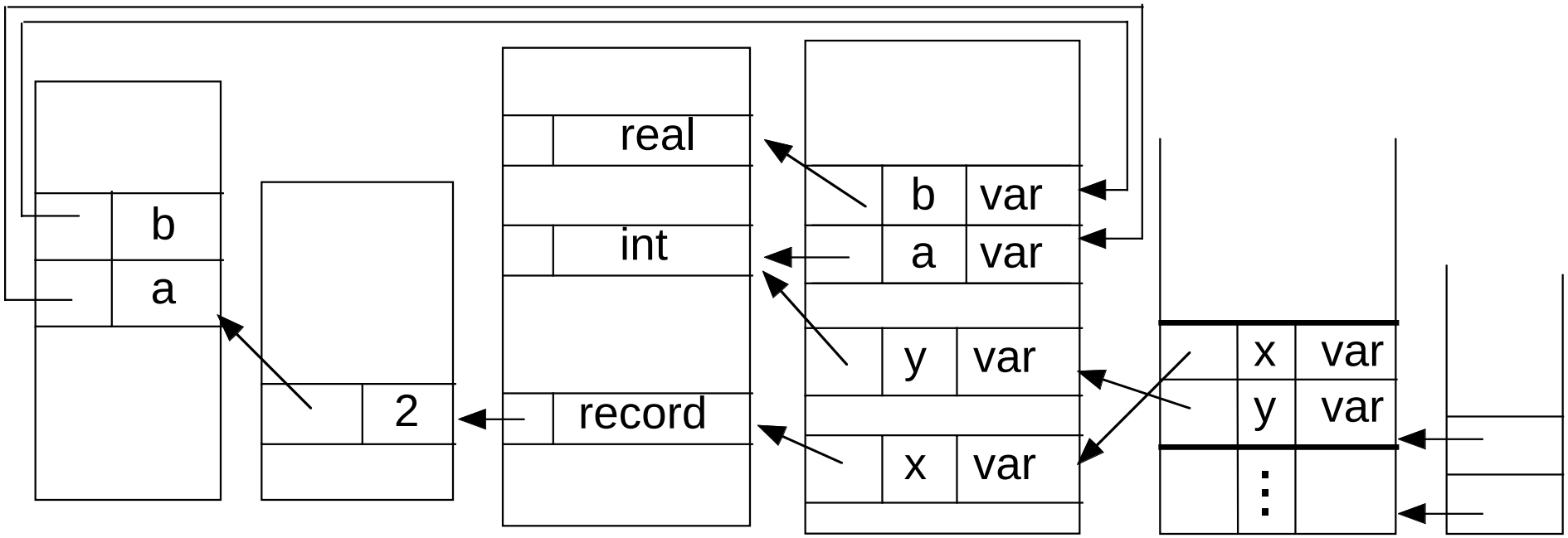
```
...
y := x.a
...
```



# Saved Scopes – Example

```
var y: int
var x:
  record
    a: int
    b: real
  end record
```

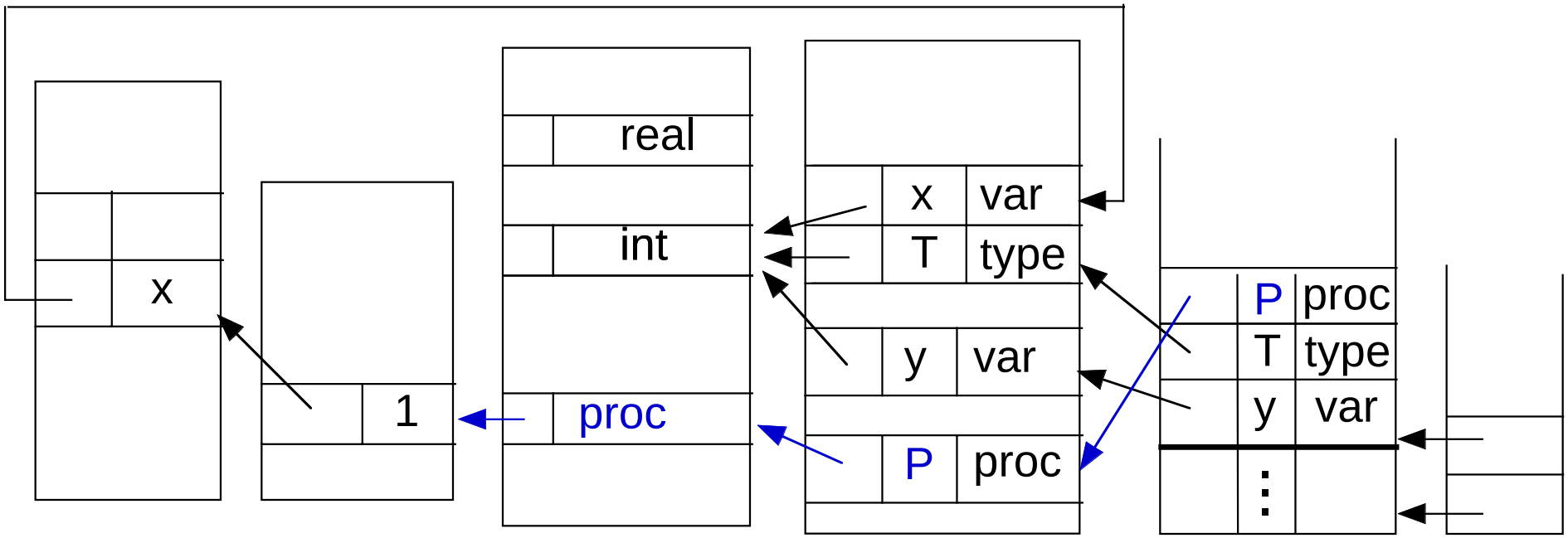
```
...
y := x.a
...
```



# Saved Scopes – Example 2

```
var y: int
type T: int

procedure P(var x: T)
   ...

end P
```

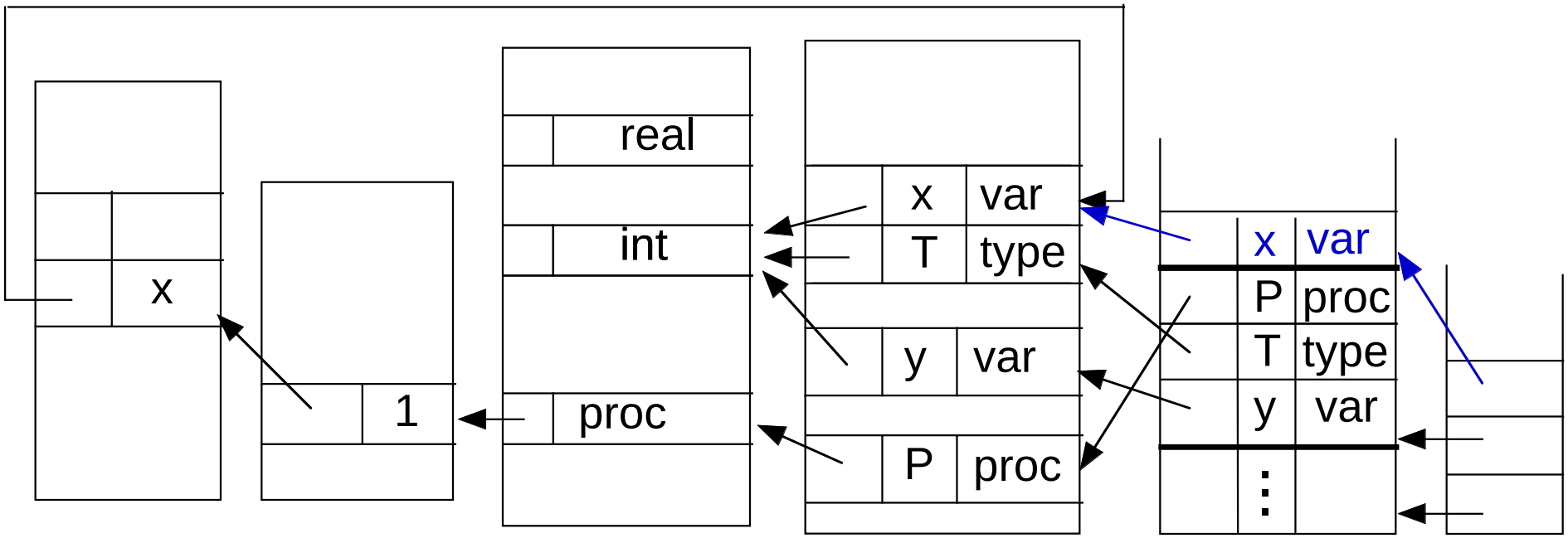
```
...

P(y)

...
```



# Saved Scopes – Example 2

```
var y: int
type T: int

procedure P(var x: T)
   ...

end P
```

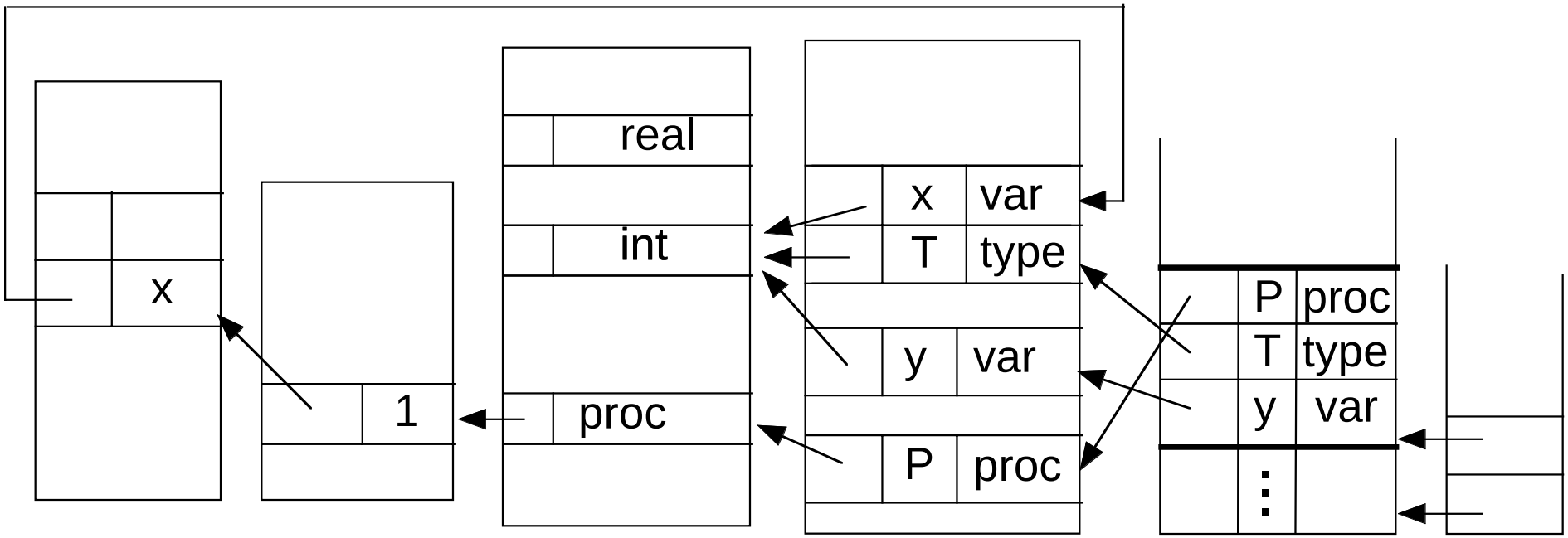
```
...

P(y)

...
```



# Saved Scopes – Example 2

```
var y: int
type T: int

procedure P(var x: T)
   ...

end P
```

```
...

P(y)

...
```



# Of Course ...

- In practice the process is optimized, and we don't run around copying scopes
- The saved scope is only conceptually pushed back onto the Scope Stack, and fields are looked up directly in the saved Scope Table using special operations
- In the PT Pascal compiler, formal parameters are simply stored directly in the Symbol Table beside the procedure, and are accessed when needed from there using special semantic operations
- They are made invisible in the outer scope by unlinking their identifiers from the identSymbolTblRef direct lookup array when processing the end of a procedure (leaving the procedure identifier itself linked since it is visible outside itself)
- We are doing similarly for public methods of classes in JT, making variables and non-public functions invisible in the outer scope by unlinking their identifiers from identSymbolTblRef at the end of the class, but leaving public functions linked since they are visible outside the class

# Summary

- Imports and access control are handled by separating the scope-dependent attributes from the scope-independent ones
- A separate *Scope Stack* is used to implement visibility and scope-dependent attributes, while pointing at entries of the *Symbol Table* which contains the original declared attributes
- Inner scopes of records, classes and parameter lists may have to be re-opened for field selection or argument passing
- A *Saved Scope Table* saves these scopes for use when needed
- In practice we don't really copy scopes, we design to avoid copying (like PT)
- **Next**:
  - Implementing the Run-time model