Semantic Analysis 3

Declarations, Procedures, & Code Generation

Dr. William Harrison
harrisonwl@missouri.edu
CS 4430 Compilers I

Today

- Last time, we introduced translation schema for “Micro”
  - simple & straightforward
- Today, we’ll consider important extensions to Micro and how to compile them; they are:
  - declarations
  - procedure

Extended CFG for Micro

<program>	→ begin <statement list> end
<statement list>	→ <statement> {<statement list>}
<statement>	→ ID := <expression> ;
<statement>	→ read (<id list>) ;
<statement>	→ write (<expr list>) ;
<id list>	→ ID { , ID }
<expr list>	→ <expression> { , <expression> }
<expression>	→ <primary> { <add op> <primary> }
<primary>	→ (<expression>)
<primary>	→ ID
<primary>	→ INTLITERAL
<add op>	→ PLUSOP
<add op>	→ MINUSOP
<system goal>	→ <program> SCANEOF

** Note: some things left out here like if-then-else, booleans, etc.*

Extended CFG for Micro

<program> → begin <statement list> end
<statement list> → <statement> {<statement lis }
<statement> → ID := <expression> ;
<statement> → read (<id lis) ;
<statement> → write (<expr
<id list> → ID { , ID }
<expr list> → <expres
<expression> → <primary>
<primary> → (<ex
<primary> → ID
<primary> → INTLITE
<add op> → PLUS
<add op> → MINUSOP
<system goal> → <program> SCANEOF

Q: How are variables declared in Micro?
Q: What about procedures

** Note: some things left out here like if-then-else, booleans, etc.*

Variable declaration in Micro

- …all variables are **undeclared**
  - we assume that they are used consistently
    - i.e., "`x := 1 ; x := 1.0`" doesn't make sense
    - this consistency check may be performed statically
- most languages insist on **explicit** variable declarations
  - one reason for this is that it simplifies the compiler writer's task
    - it makes static checks like type inference easier
      - this, in turn, leads to more informative error messages (e.g., "error on line 12: x declared float…")
    - corresponds directly to simple memory model

“Memory model”?

IR

```
(READI, A)
(READI, B)
(GT,A,B,t1)
(JUMP0,t1,L1)
(ADDI,A,5,C)
(JUMP,L2)
(LABEL,L1)
(ADDI,B,5,C)
(LABEL,L2)
(SUBI,C,1,t2)
(MULTI,2,t2,t3)
(WRITEI,t3)
```

The **memory model** describes where these virtual registers are represented in memory:

- are the stored in registers?
- memory locations?
- how is memory organized during program execution?

A Micro-like Language with Declarations

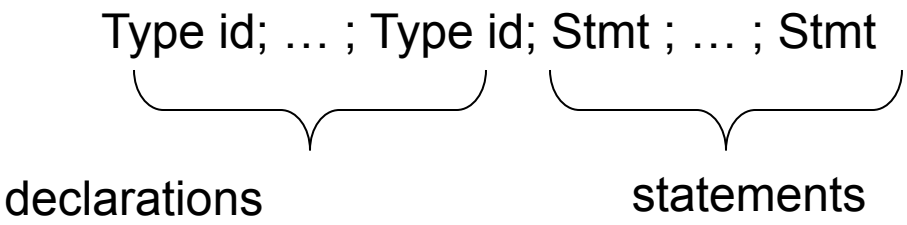
Program	→ *TypeDef* Stmt**	*CExp*	→ *Exp* ***cop*** *Exp*
TypeDef	→ *Type* ***id*** **;**	*Exp*	→ *Exp* ***op*** *Exp*
Type	→ **int**		→ ***id***
	→ **int [** ***INT_LITERAL*** **]**		→ ***INT_LITERAL***
Stmt	→ ***id*** **:=** *Exp* ;		→ ***id*** **[** *Exp* **]**
	→ ***id*** **[** *Exp* **] :=** *Exp* **;**		→ **(** *Exp* **)**
	→ **print (** *Exp* **) ;**		
	→ **while (** *CExp* **) {** *Stmt** **}**		
	→ **if (** *CExp* **) {** *Stmt** **} else {** *Stmt** **}**		
	→ **if (** *CExp* **) {** *Stmt** **}**		

Call this language **Micro+**

Micro+ programs just like in Micro, but with leading declarations

Program	→ *TypeDef* Stmt**	*CExp*	→ *Exp* **cop** *Exp*
TypeDef	→ *Type* ***id*** **;**	*Exp*	→ *Exp* **op** *Exp*
Type	→ **int**		→ ***id***
	→ **int [** ***INT_LITERAL*** **]**		→ ***INT_LITERAL***
Stmt	→ ***id*** **:=** *Exp* **;**		→ ***id*** **[** *Exp* **]**
	→ ***id*** **[** *Exp* **] :=** *Exp* **;**		→ **(** *Exp* **)**
	→ **print (** *Exp* **) ;**		
	→ **while (** *CExp* **) {** *Stmt** **}**		
	→ **if (** *CExp* **) {** *Stmt** **} else {** *Stmt** **}**		
	→ **if (** *CExp* **) {** *Stmt** **}**		

A typical Micro+ program has the form:



Translating Declarations

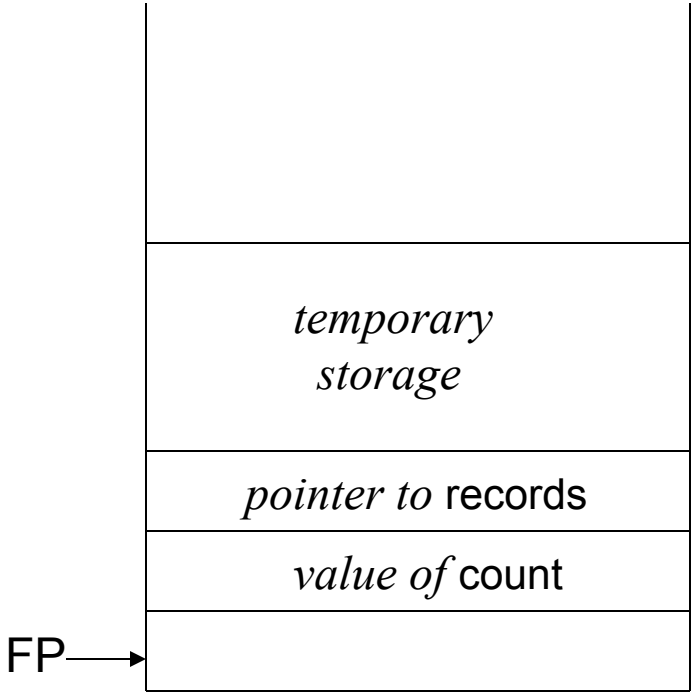
- Note that Micro+ has two types occurring in declarations: int and int[]

```
int count ;
int[] records[100] ;
count := 1 ;
record[count] := 5 ;
    …<rest of the program>…
```

- Given a Micro+ program, we have to decide on how that variables are represented in memory
  - That is, where & how each variable is stored and accessed

Example

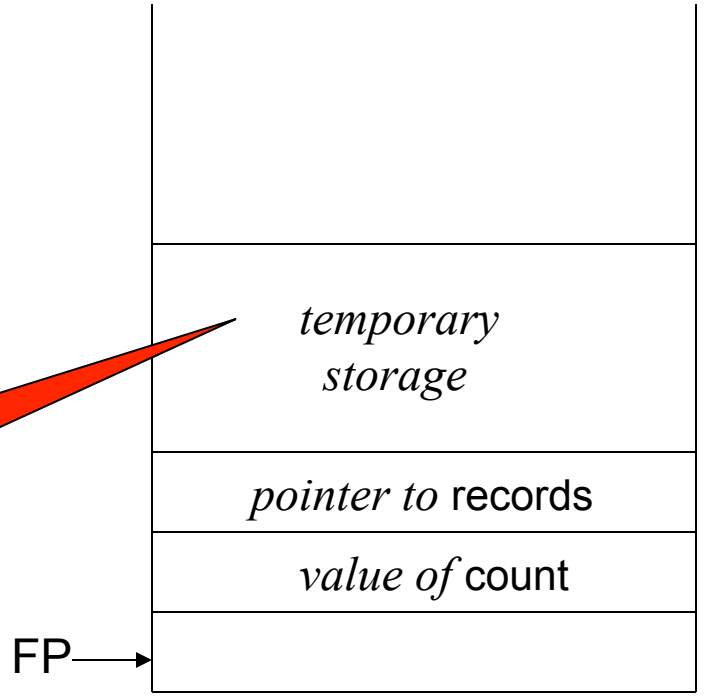
```
int count ;
int[] records[100] ;
count := 1 ;
record[count] := 5 ;
    …<rest of the program>…
```



assuming
- 1 word integers here
- a “frame pointer” register FP

Example

```
int count ;
int[] records[100] ;
count := 1 ;
record[count] := 5 ;
    …<rest of the program>…
```

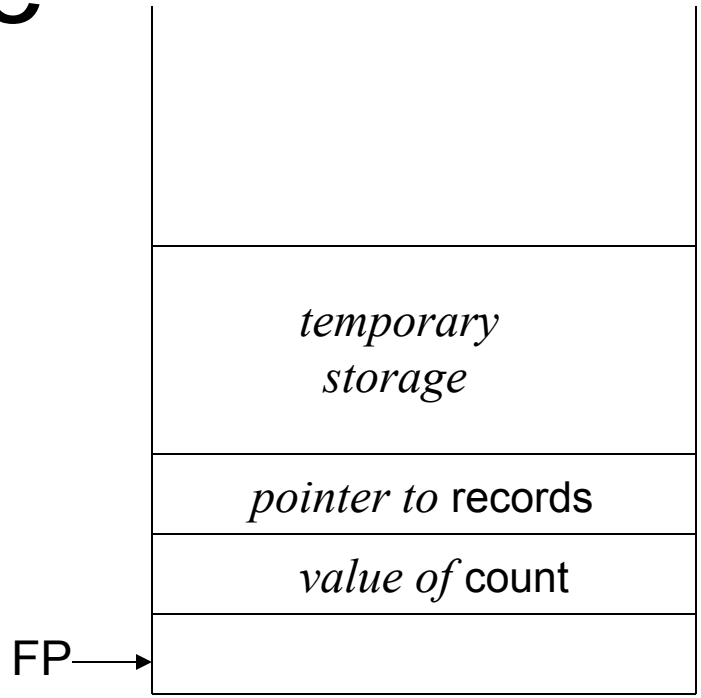


generated virtual registers might be stored here

ass
- 1 word integers here
- a “frame pointer” register FP

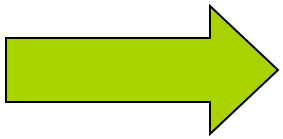
Example

```
int count ;
int[] records[100] ;
count := 1 ;
record[count] := 5 ;
    …<rest of the program>…
```



allows virtual registers to become less virtual

```
(ADDI,count,0,count)
```



```
(ADDI,[FP+1],0,[FP+1])
```

What about Procedures?

- Language design has vast implications for the implementation of procedures
- To what extent is a procedure a “value” in the language?
  - Can a procedure be passed as to another procedure?
  - Can a procedure be returned as a value by another procedure?
- How are procedures declared?
  - is it declared globally?
  - are procedure declarations nested?
- What is the big issue for compiler writers?
  - resolving variable references; that is,
    - where is the code for procedure p?
    - or, where is variable “x” stored?

C-style procedures

A high-level view of a C program (i.e., ignoring separate files) is:

global-declarations;
int foo (char v) { … *body-foo* … }
void bar (int a) { … *body-bar* … }
…
void main (int argv[]) {… *body-main* … }

** ignoring C's procedure pointers by which one may hack downward/upward fun-args.*

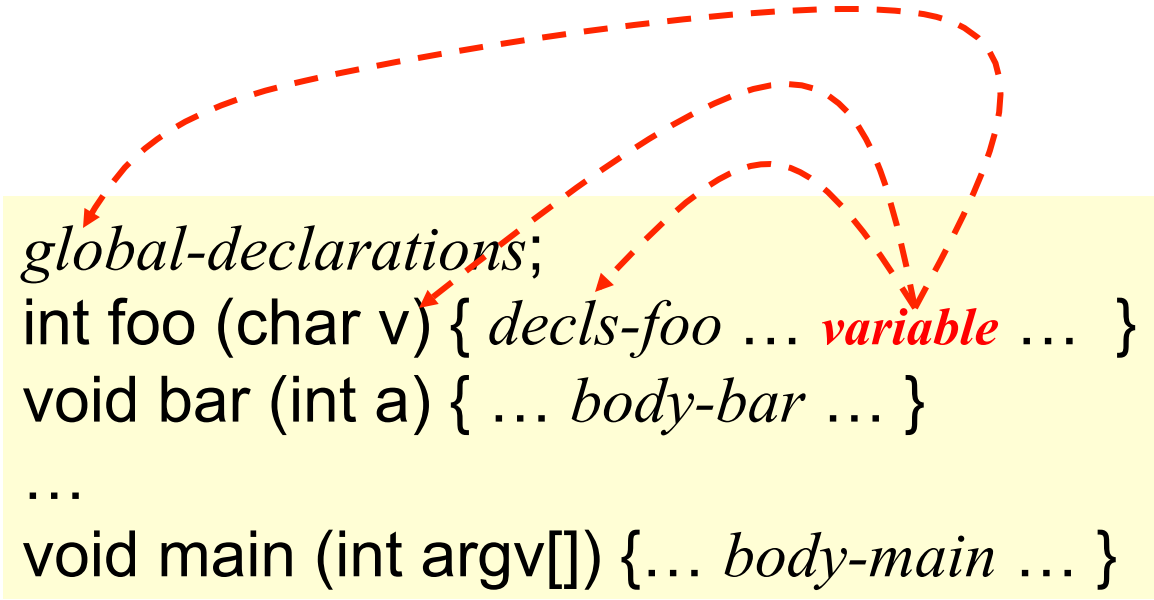
Variable scoping in C

Scoping in C is particularly simple:

global-declarations;
int foo (char v) { *decls-foo* … ***variable*** … }
void bar (int a) { … *body-bar* … }
…
void main (int argv[]) {… *body-main* … }

Question: if a variable is used in *body-foo*, where is it declared?

Variable scoping in C



Question: if a variable is used in *body-foo*, where is it declared?

procedure scoping

Procedure names are variables as well

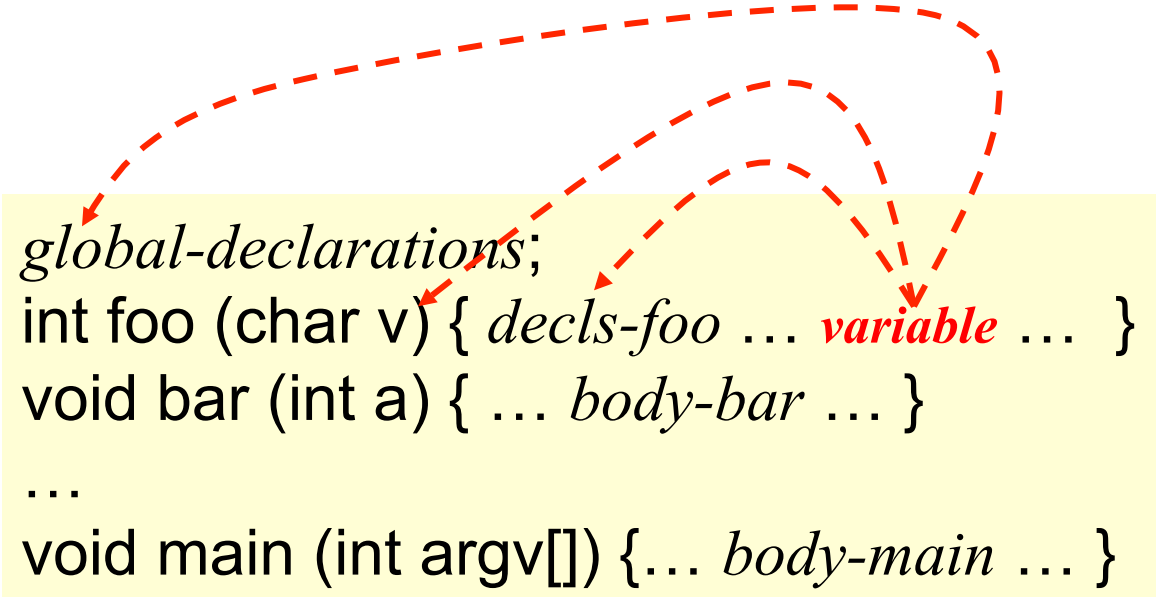
global-declarations;
int foo (char v) { … *proc-call* … }
void bar (int a) { … *body-bar* … }
…
void main (int argv[]) {… *body-main* … }

proc-call must refer to:
- user defined procedures (foo, …), or
- (global) library routines

Implications of C language design for compiler writer

- Compiling procedures confronts two issues
  - compiling procedure **declarations**
    - similar to what we've done for Micro+
    - this is deliberate – Micro+ programs look like the bodies of C procedures
    - code must be stored at a new label.
  - compiling procedure **calls**
    - create "activation record" for call
    - keep track of return label
    - jump to the code
- The simple scoping of C-procedures simplifies their compilation
  - main issue: keeping track of variables during execution

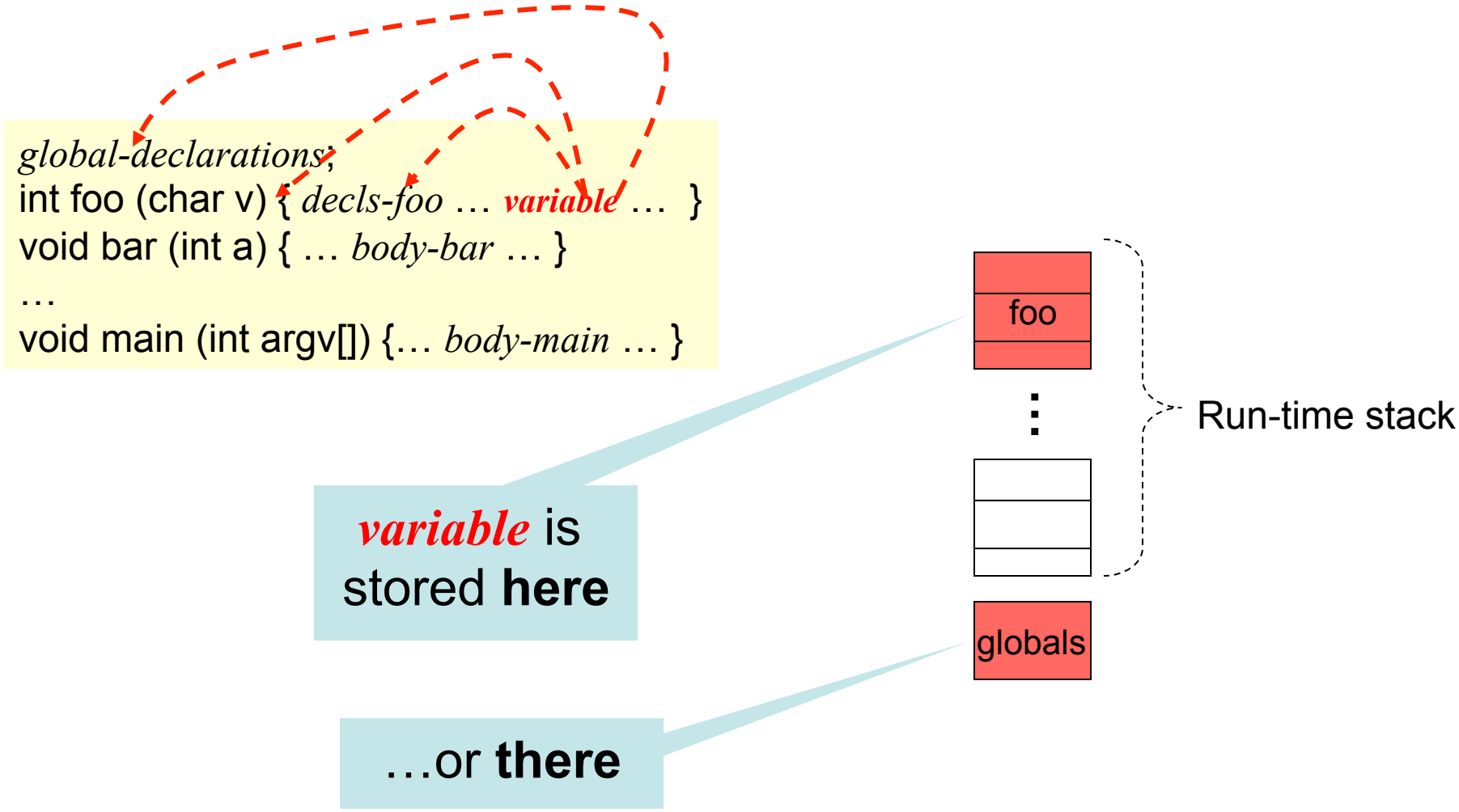
Variable scoping in C



For a C variable reference, it is declared in one of two places:

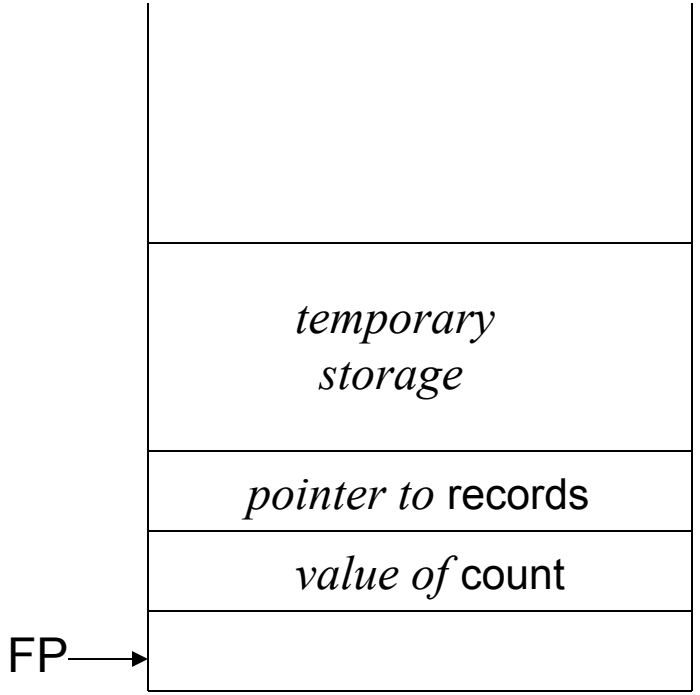
- **Locally**: it's either a procedure parameter (like "v") or is declared locally (as in *decls-foo*)
- **Globally**: e.g., it's declared in *global-declarations*

…as a consequence



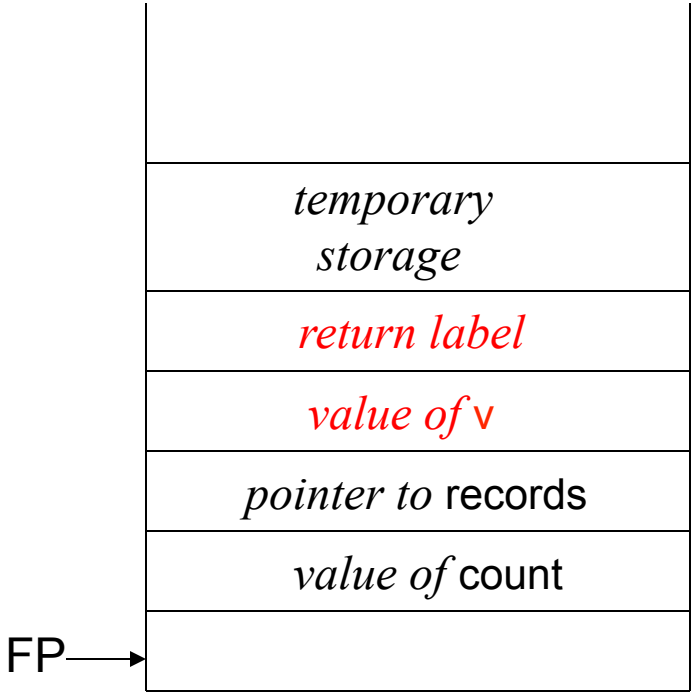
Recall the Micro+ mem. model

```
int count ;
int[] records[100] ;
count := 1 ;
record[count] := 5 ;
    …<rest of the program>…
```



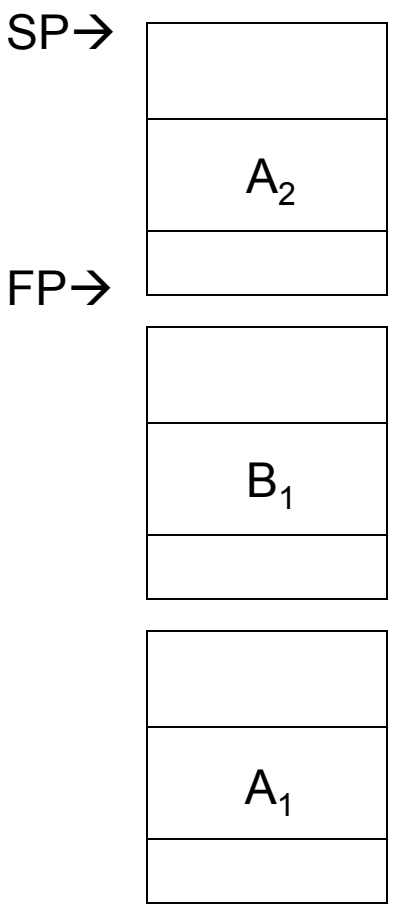
Activation record

```
void foo (char v) {
    int count ;
    int[] records[100] ;
    count := 1 ;
    record[count] := 5 ;
     …<rest of foo>…
}
```



Execution generates a new activation record for **each** call

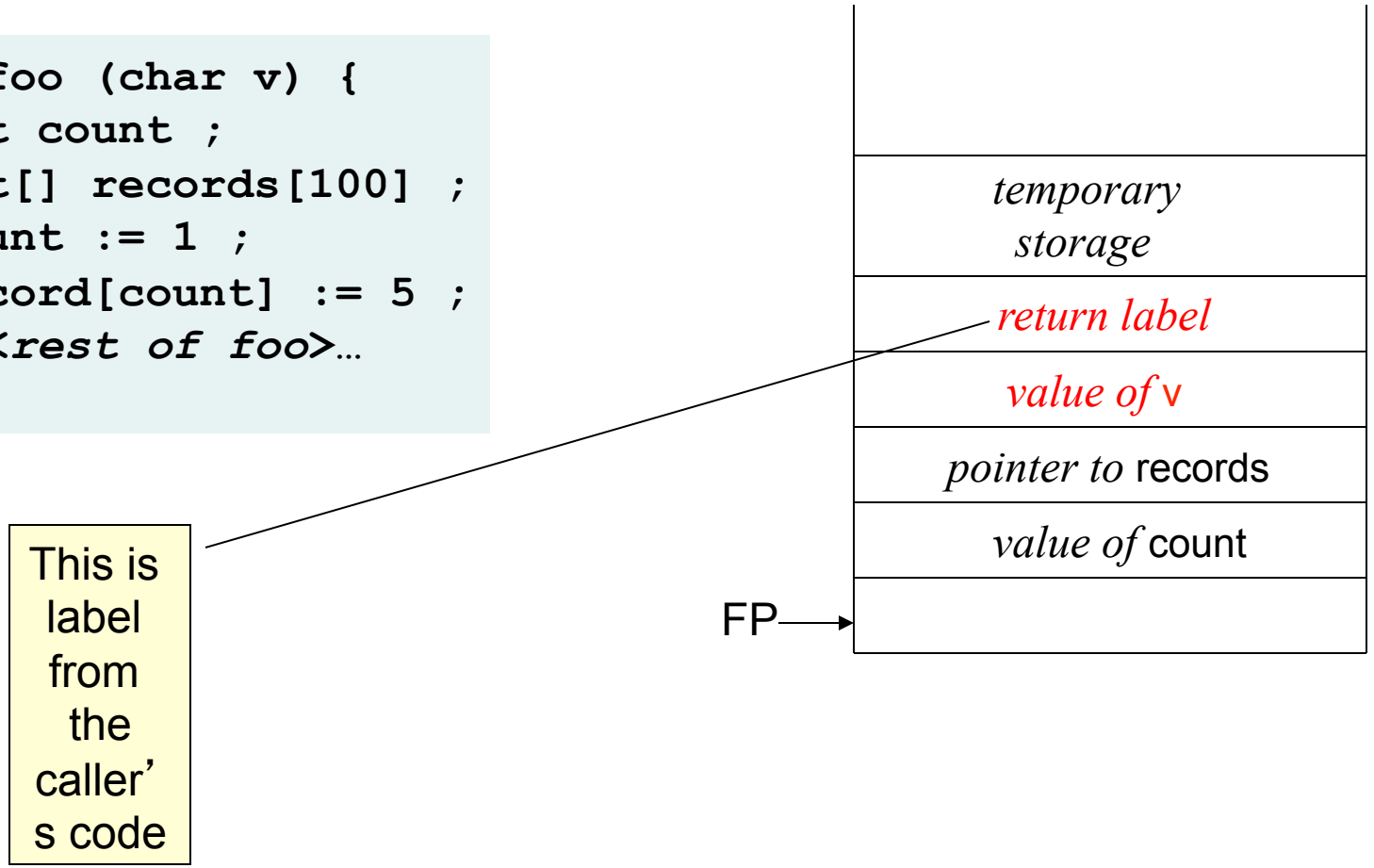
```
proc A (aargs) {…B(bv)…}
proc B (bargs) {…A(av)…}
begin
    A (…); // a procedure call
end
```



Keep track of the "current activation"
- with frame pointer FP
- stack pointer SP
- calling code must prepare this new AR

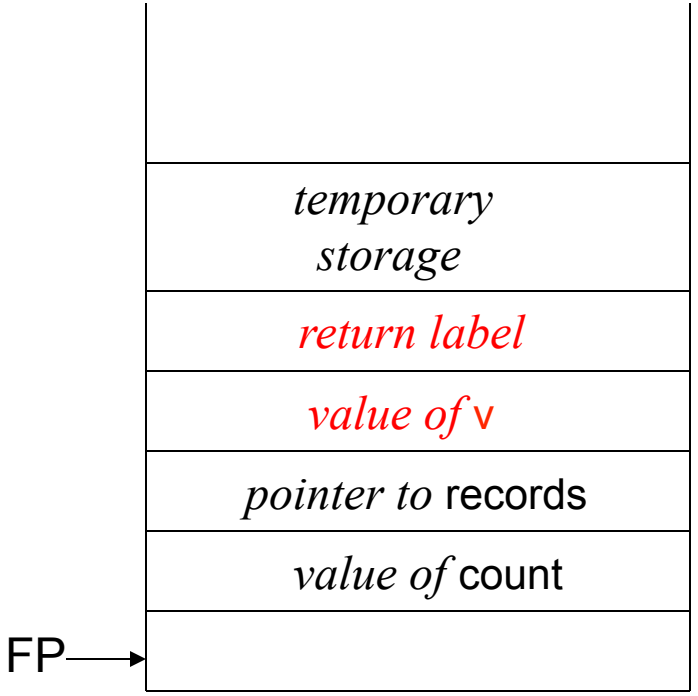
Activation record

```
void foo (char v) {
    int count ;
    int[] records[100] ;
    count := 1 ;
    record[count] := 5 ;
     …<rest of foo>…
}
```



Activation record

```
void foo (char v) {
    int count ;
    int[] records[100] ;
    count := 1 ;
    record[count] := 5 ;
     …<rest of foo>…
}
```



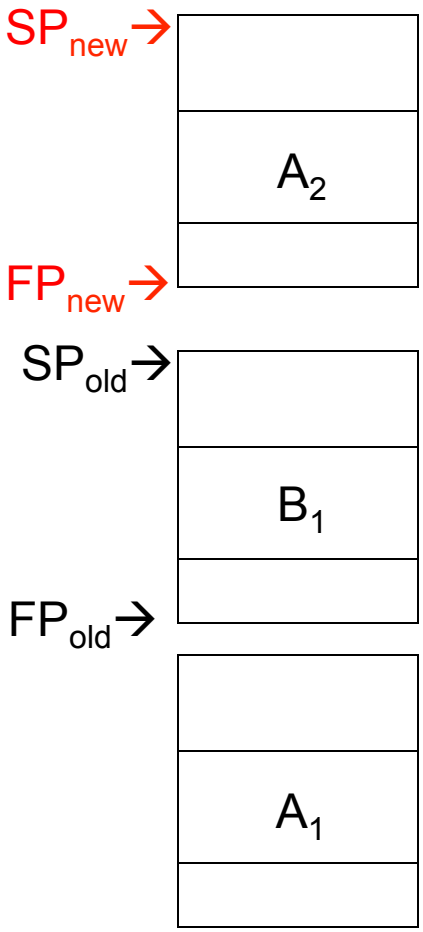
- Each time a procedure is called, such an activation is created on the **run-time stack**
- N.b., this AR format will be extended (slightly)
- Many possible choices for AR format

The symbol table & variable and procedure references: the case for C

- Locally declared variables
  - are at a fixed offset from FP
    - note that this offset may be determined statically
  - this offset is stored in the symbol table for each variable
- Procedure parameters
  - also at fixed FP-offset
    - store in symbol table
    - note that this offset may be determined statically
  - some languages have restrictions on whether they are "write-able"
- Global identifiers
  - fixed location also stored in the symbol table

making a call: when B calls A

```
proc A (aargs) {…B(bv)…}
proc B (bargs) {…A(av)…}
begin
    A (…);
end
```

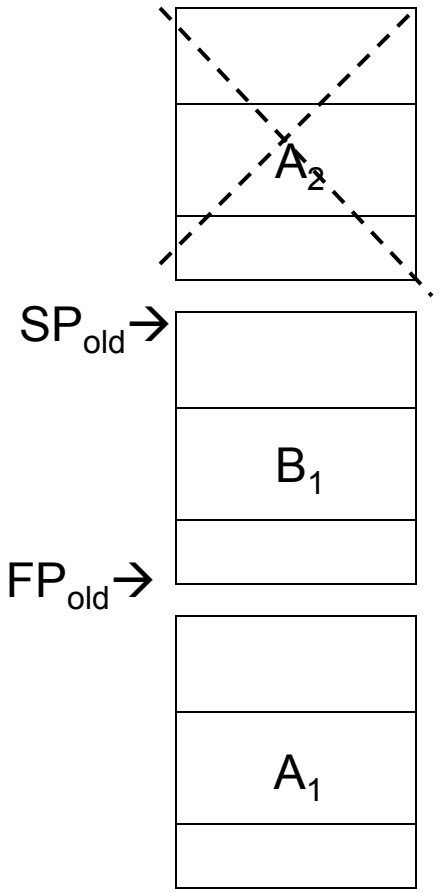


Code for call must

- allocate AR space
- Store current FP, SP there
- store arguments there
- set SP,FP to new values

Returning from call

```
proc A (aargs) {…B(bv)…}
proc B (bargs) {…A(av)…}
begin
    A (…);
end
```



Code for procedure must

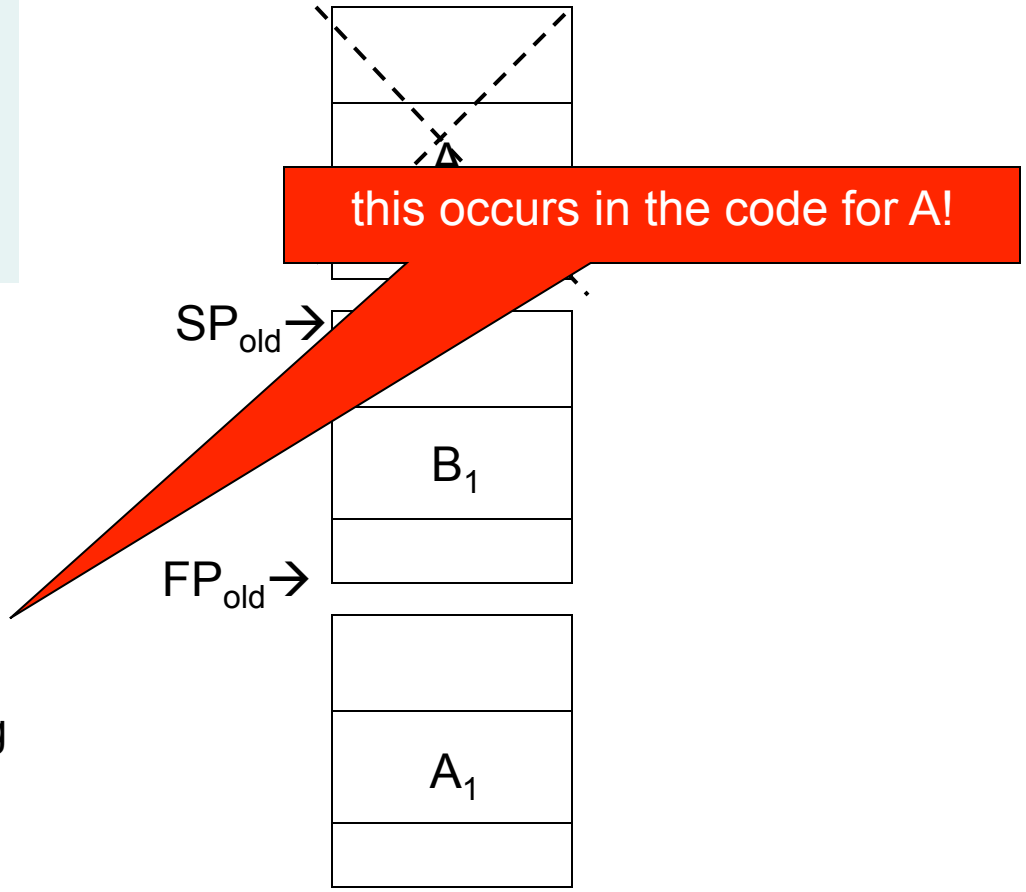
- Restore previous FP, SP
  - these in AR for A
- deallocation of AR generally not necessary
  - accomplished by resetting FP,SP registers

Returning from call

```
proc A (aargs) {…B(bv)…}
proc B (bargs) {…A(av)…}
begin
    A (…);
end
```

Code for procedure must

- Restore previous FP, SP
  - these in AR for A
- deallocation of AR generally not necessary
  - accomplished by resetting FP,SP registers

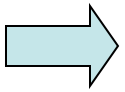


Compiling procedure declarations

- Produces code for the procedure body
  - this “lives” at its own new address
  - quite similar to compiling straight-line Micro+ programs
- Main difference: de-referencing variables is (slightly) more complicated.
  - recall that a program variable in C is one of the following:
    - globally declared,
    - procedure argument, or
    - locally declared
      - in this case, handled just like in Micro+

Compiling procedure declarations

global-declarations;
int foo (char v) { *body-foo* }
void bar (int a) { *body-bar* }
…
void main (int argv[]) { *body-main* }



L_{foo} : <code for *body-foo* >

L_{bar} : <code for *body-bar* >

L_{main} : <code for *body-main* >

Here, the code for the procedure bodies is compiled just as Micro+, except:
- There may be a "return" value. Usually passed in a designated register.
- …and variable references

More complicated scope rules complicate implementation

Example: Nested procedures in Pascal

```
procedure A (x:real) {
    procedure B() { …x… }
    B();
}

begin
    A(9.9);…
end
```

Procedure B is local to A

Calling A results in a call to B

Upshot: code for B must "climb down" the runtime stack to find x

Compiling procedure calls

Caller's code must

- prepare an activation record for the "callee"
  - this includes filling in the AR with
    - argument values
    - return label
    - values of FP,SP to be restored
- Set values of FP,SP
- use a "call"
- Upon returning, save any returned value

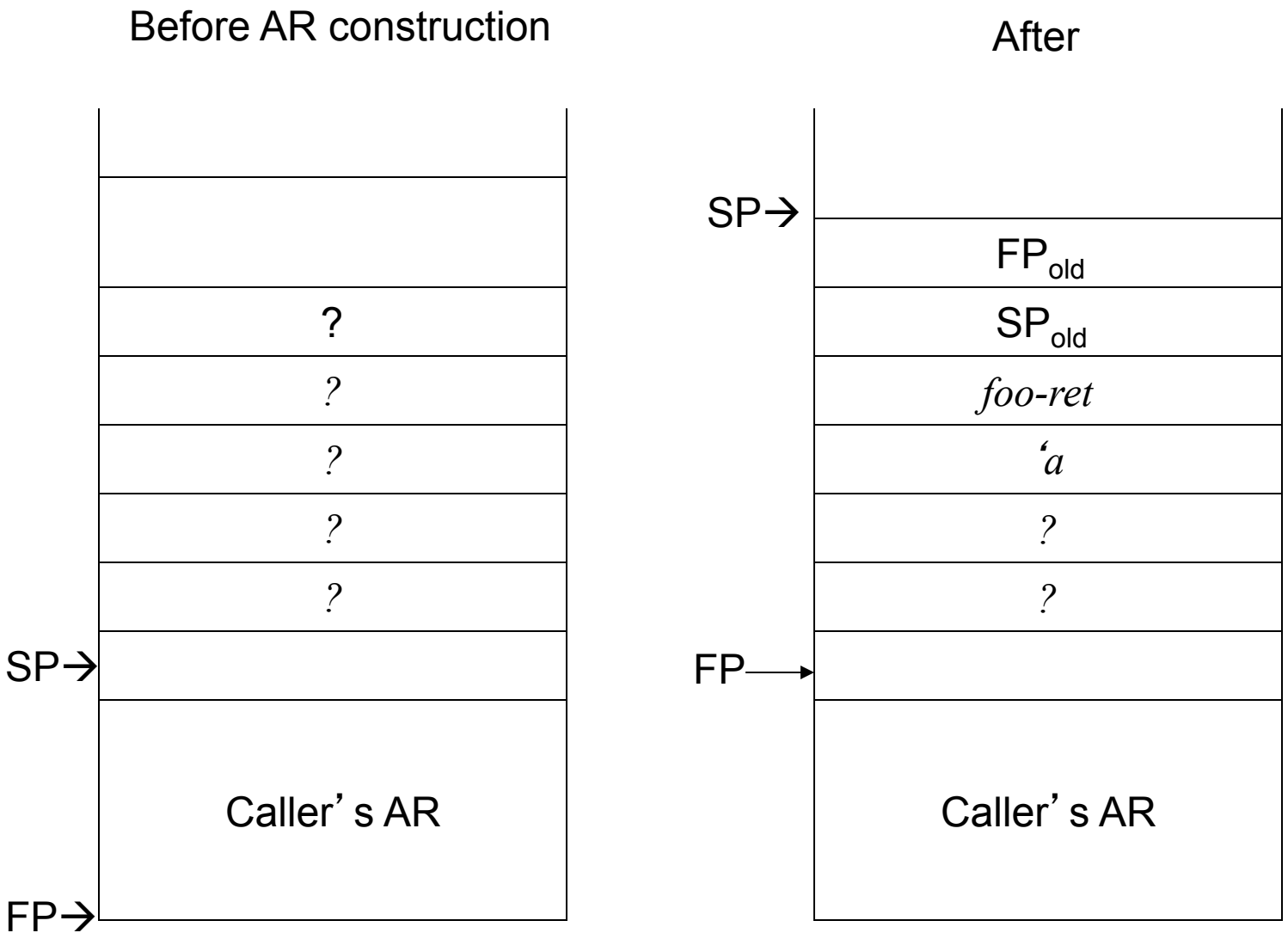
Example: compiling “foo(‘a)”

```
void foo (char v) {
   int count ;
   int[] records[100] ;
   count := 1 ;
   record[count] := 5 ;
    …<rest of foo>…
}
```

Code to call foo(‘a)

[SP+3] := ‘a
[SP+4] := foo-ret
[SP+5] := SP
[SP+6] := FP
FP := SP
SP := SP+7
call foo-label foo-ret
Label foo-ret

Activation record for “foo(‘a)”



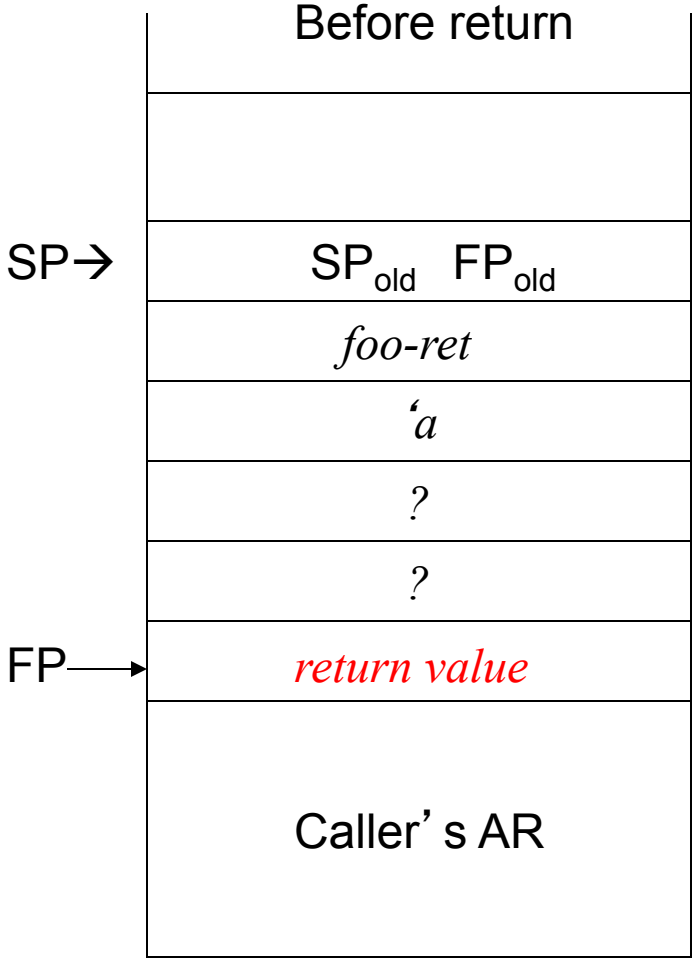
what if “foo(‘a)” returns a value?

```
int foo (char v) {
   int count ;
   int[] records[100] ;
   count := 1 ;
   record[count] := 5 ;
    …<rest of foo>…
}
```

[SP+3] := ‘a
[SP+4] := foo-ret
[SP+5] := SP
[SP+6] := FP
FP := SP
SP := SP+7
call foo-label foo-ret
Label foo-ret
RetVal := [SP+1]

Upshot: caller & callee must agree on how to communicate returned values

Let the callee write the return value to [FP]

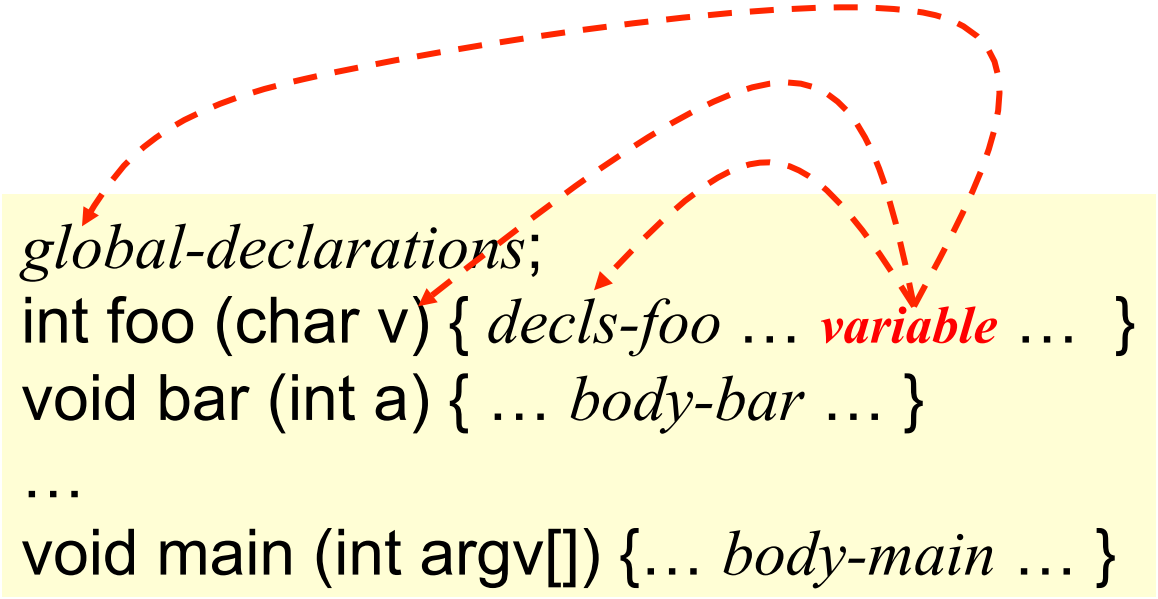


Or, return in a register

Summary

- Showed how C-like procedures could be added to Micro+ compiler
  - Activation records ≈ memory model for Micro+
  - Procedure compiled by
    - compiling procedure declaration
    - compiling procedure calls
  - We considered stack allocation of activation records
    - i.e., there is a stack built from contiguous memory, and each AR is "right next to" other ARs
    - How would compilation method differ if we used dynamic memory allocation with garbage collection?
- The language design (particularly the scoping) had a big effect on how the language is implemented
  - in this case, simple scoping made the implementation easier
    - how would this differ for more expressive languages? Pascal, …

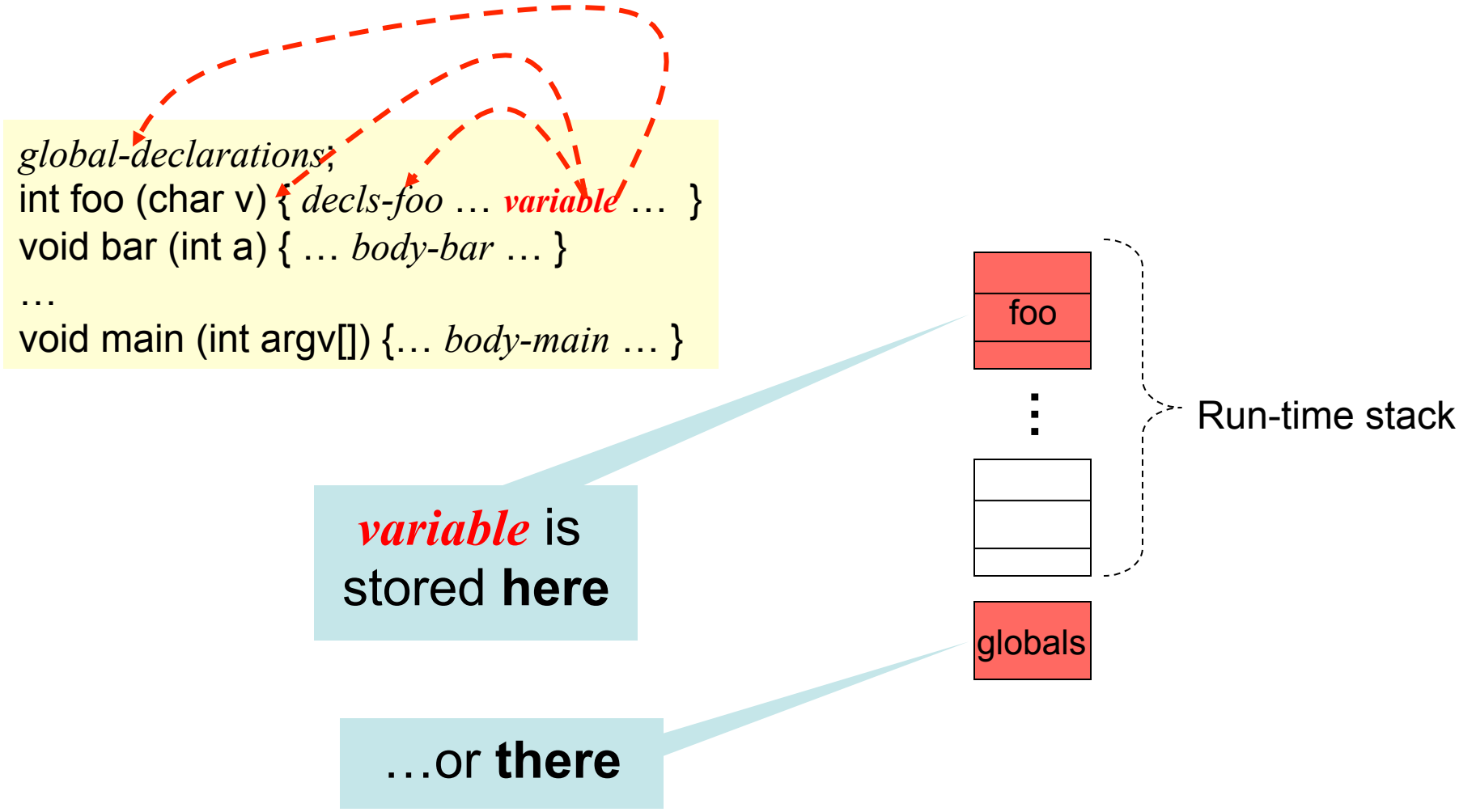
Variable scoping in C



For a C variable reference, it is declared in one of two places:

- **Locally**: it's either a procedure parameter (like "v") or is declared locally (as in *decls-foo*)
- **Globally**: e.g., it's declared in *global-declarations*

…as a consequence



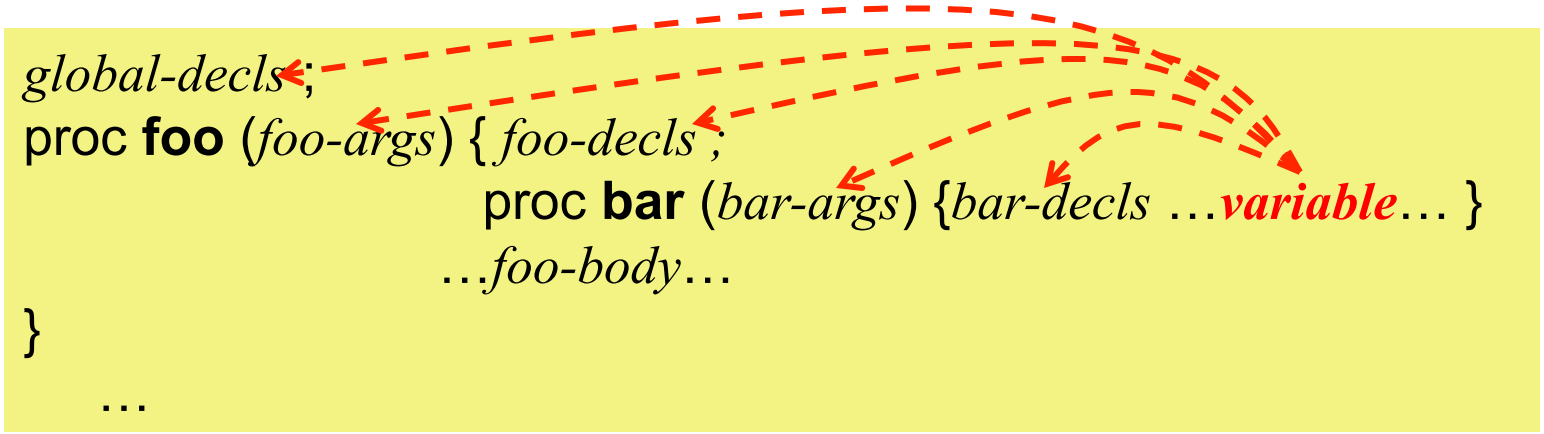
NESTED PROCEDURES

Nested Procedure Declarations

global-decls ;
proc **foo** (*foo-args*) { *foo-decls ;*
proc **bar** (*bar-args*) {*bar-decls* …***variable***… }
…*foo-body*…
}
…<Note: can't call **bar** from here>…

Q: where is ***variable*** defined?

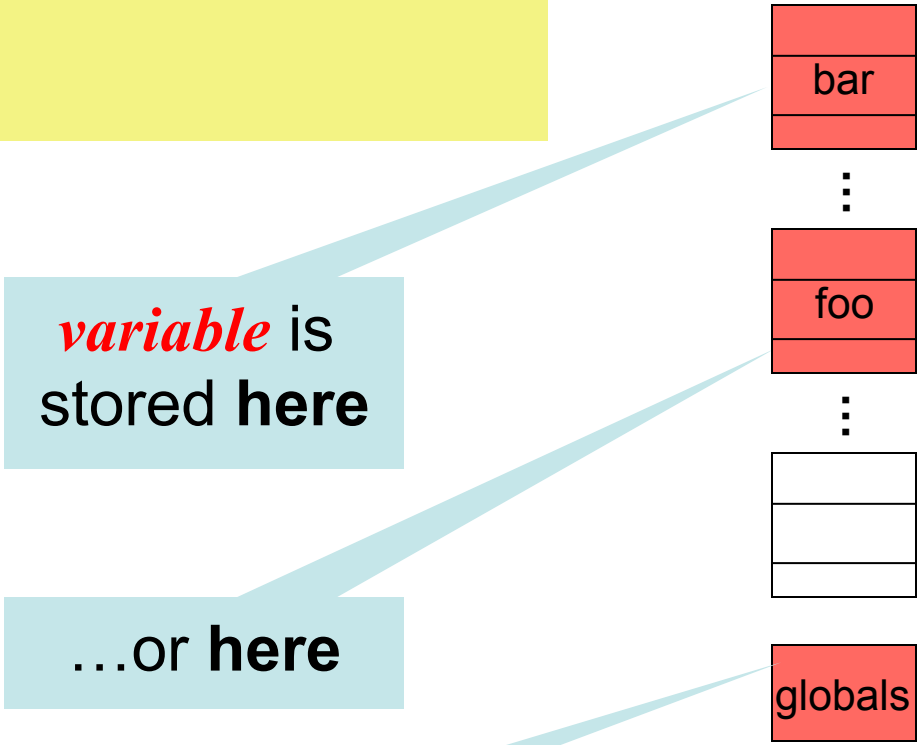
Nested Procedure Declarations



Q: where is ***variable*** defined?

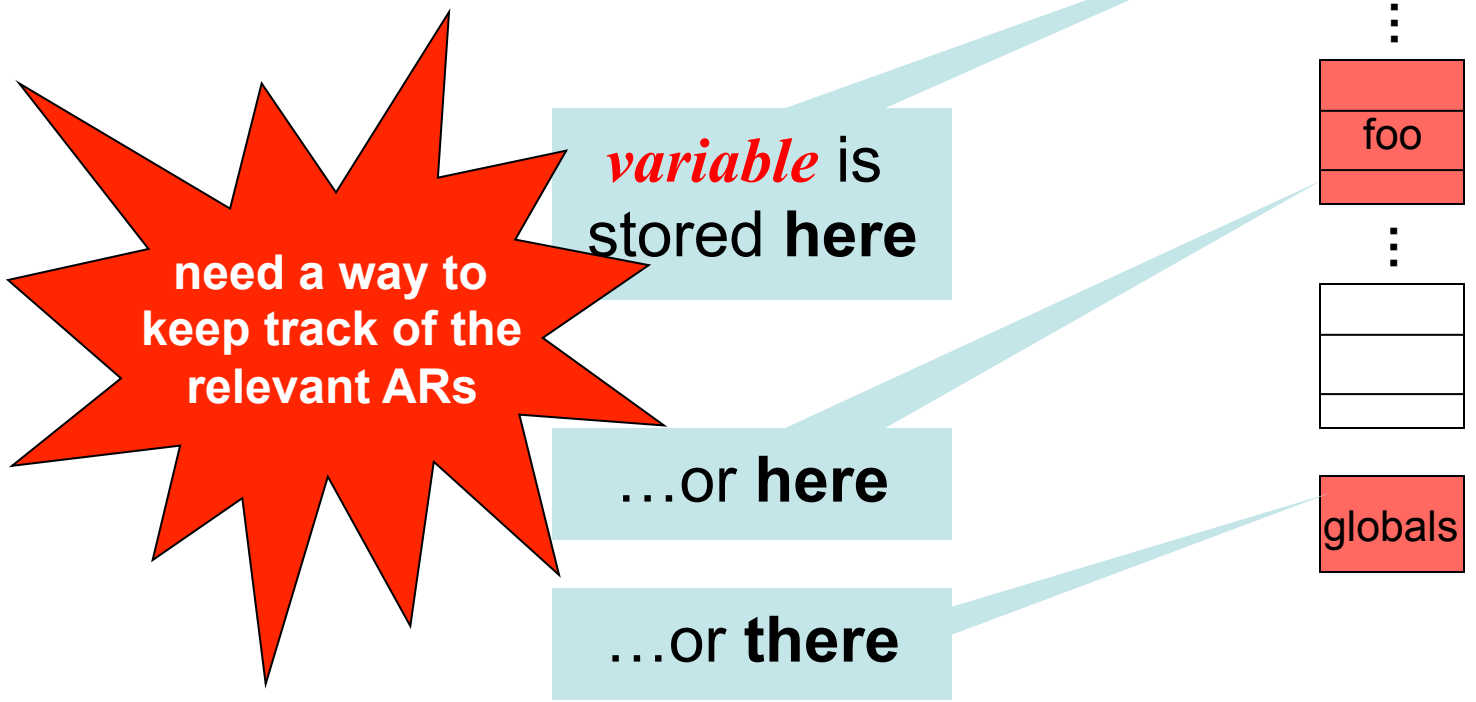
…as a consequence

```
global-decls ;
proc foo (foo-args) { foo-decls ;
                      proc bar (bar-args) {bar-decls …variable… }
                    …foo-body…
}
    …
```



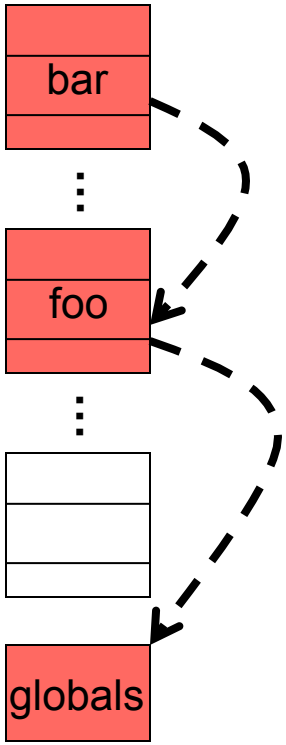
…as a consequence

```
global-decls ;
proc foo (foo-args) { foo-decls ;
                      proc bar (bar-args) {bar-decls …variable… }
                    …foo-body…
}
    …
```



use “access links”

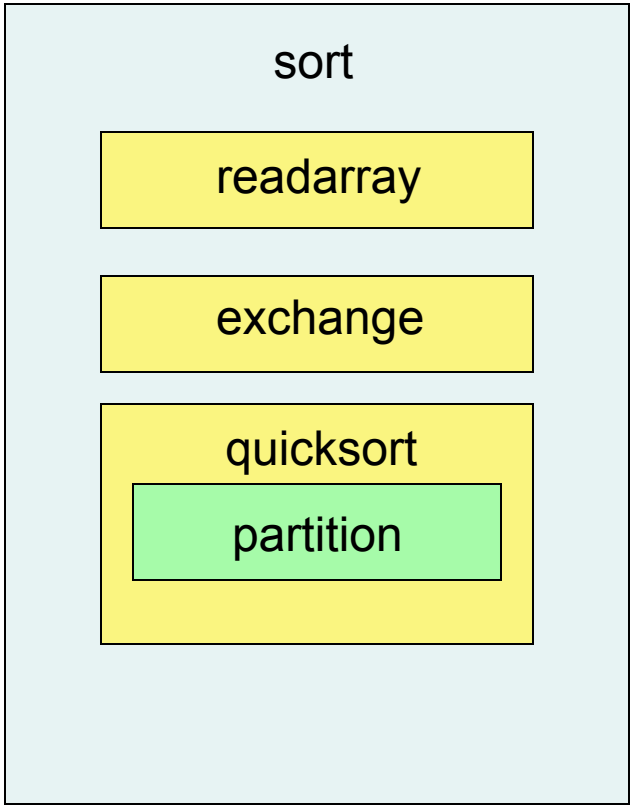
```
global-decls ;
proc foo (foo-args) { foo-decls ;
                      proc bar (bar-args) {bar-decls …variable… }
                     …foo-body…
}
    …
```



** a.k.a., “static links”*

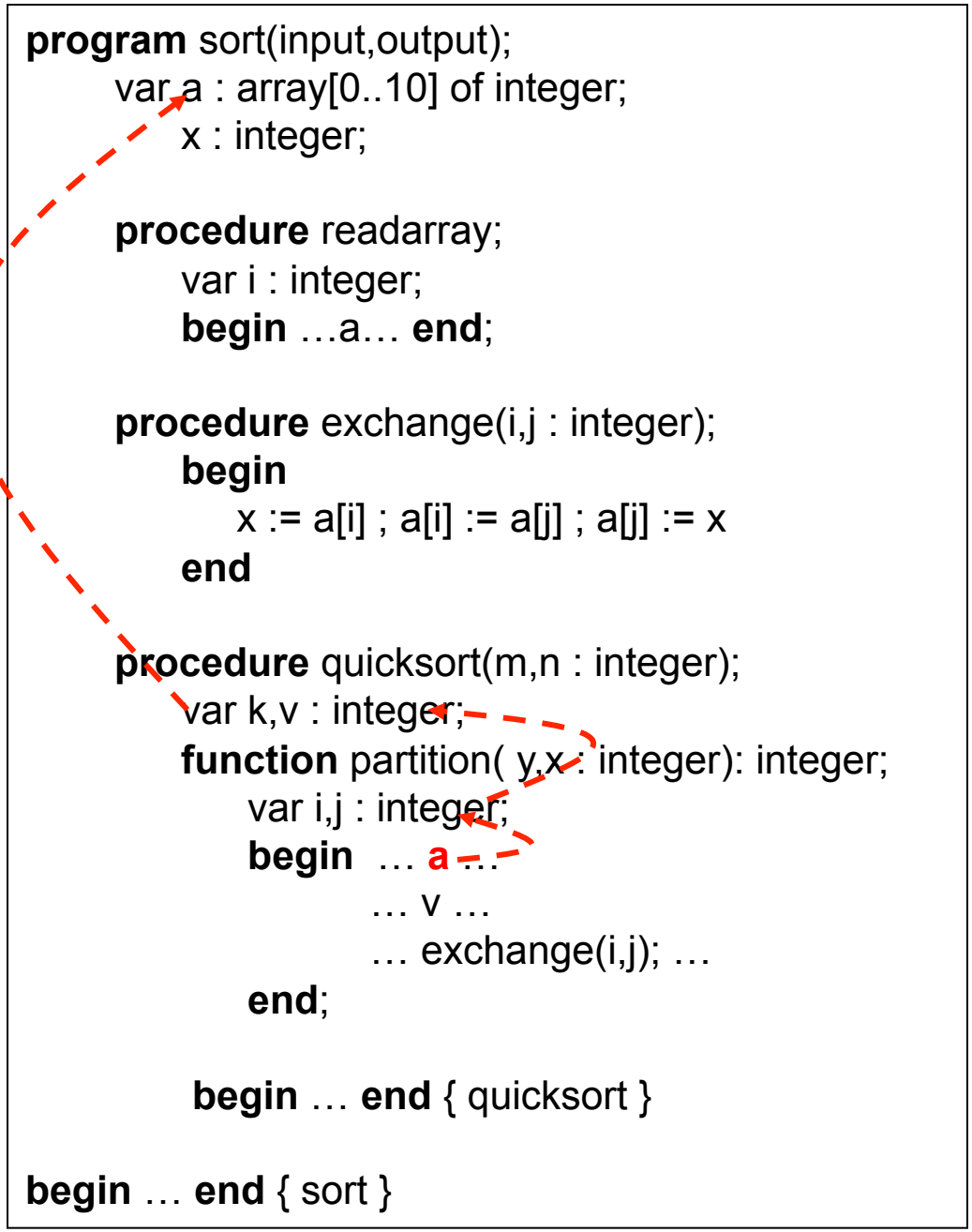
```
program sort(input,output);
    var a : array[0..10] of integer;
        x : integer;

    procedure readarray;
        var i : integer;
        begin …a… end;

    procedure exchange(i,j : integer);
        begin
            x := a[i] ; a[i] := a[j] ; a[j] := x
        end

    procedure quicksort(m,n : integer);
        var k,v : integer;
        function partition( y,x : integer): integer;
            var i,j : integer;
            begin  … a …
                   … v …
                   … exchange(i,j); …
            end;

         begin … end { quicksort }

begin … end { sort }
```

This program has the following "shape" which has an impact on its implementation



```
program sort(input,output);
    var a : array[0..10] of integer;
        x : integer;

    procedure readarray;
        var i : integer;
        begin …a… end;

    procedure exchange(i,j : integer);
        begin
            x := a[i] ; a[i] := a[j] ; a[j] := x
        end

    procedure quicksort(m,n : integer);
        var k,v : integer;
        function partition( y,x : integer): integer;
            var i,j : integer;
            begin  … a …
                   … v …
                   … exchange(i,j); …
            end;

         begin … end { quicksort }

begin … end { sort }
```

Lexical scope:
A variable reference refers to its most closely nested declaration



Lexical scope:
A variable reference refers to its most closely nested declaration

Ex: **a** refers to the top-most declaration

```
program sort(input,output);
    var a : array[0..10] of integer;
        x : integer;

    procedure readarray;
        var i : integer;
        begin …a… end;

    procedure exchange(i,j : integer);
        begin
            x := a[i] ; a[i] := a[j] ; a[j] := x
        end

    procedure quicksort(m,n : integer);
        var k,v : integer;
        function partition( y,x : integer): integer;
            var i,j : integer;
            begin  … a …
                   … v …
                   … exchange(i,j); …
            end;

         begin … end { quicksort }

begin … end { sort }
```

Nesting Depth of Procedures: Starting at 1, how many lexically enclosing proc's

proc/fun	n.d.
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

```
program sort(input,output);
    var a : array[0..10] of integer;
        x : integer;

    procedure readarray;
        var i : integer;
        begin …a… end;

    procedure exchange(i,j : integer);
        begin
            x := a[i] ; a[i] := a[j] ; a[j] := x
        end

    procedure quicksort(m,n : integer);
        var k,v : integer;
        function partition( y,x : integer): integer;
            var i,j : integer;
            begin … a …
                    … v …
                    … exchange(i,j); …
            end;

         begin … end { quicksort }

begin … end { sort }
```

Nesting Depth of a Variable: identical to the nesting depth of its defining procedure

var.	n.d.
a	1
v	2

```
program sort(input,output);
    var a : array[0..10] of integer;
        x : integer;

    procedure readarray;
        var i : integer;
        begin …a… end;                          /* 1. */

    procedure exchange(i,j : integer);
        begin
            x := a[i] ; a[i] := a[j] ; a[j] := x      /* 2. */
        end

    procedure quicksort(m,n : integer);
        var k,v : integer;
        function partition( y,x : integer): integer;
            var i,j : integer;
            begin … a …                          /* 3. */
                  … v …                          /* 4. */
                  … exchange(i,j); …
            end;

         begin … end { quicksort }

begin … end { sort }
```

Distance from definition:
ND of procedure where var. ref. occurs - ND of var.:
ND(proc) - ND(var)

var. ref.		distance
1.	a	1
2.	a	1
3.	a	2
4.	v	1

Access links

- **Access links** are pointers used to implement lexical scope for nested procedures
  - added as a field in the activation record
  - if procedure **p** is declared immediately within procedure **q** in the source text
    - the access link for a call to **p** always points to the most recent (i.e. top-most) AR for **q**

Quicksort example

A partial run of sort consists of the following calls:

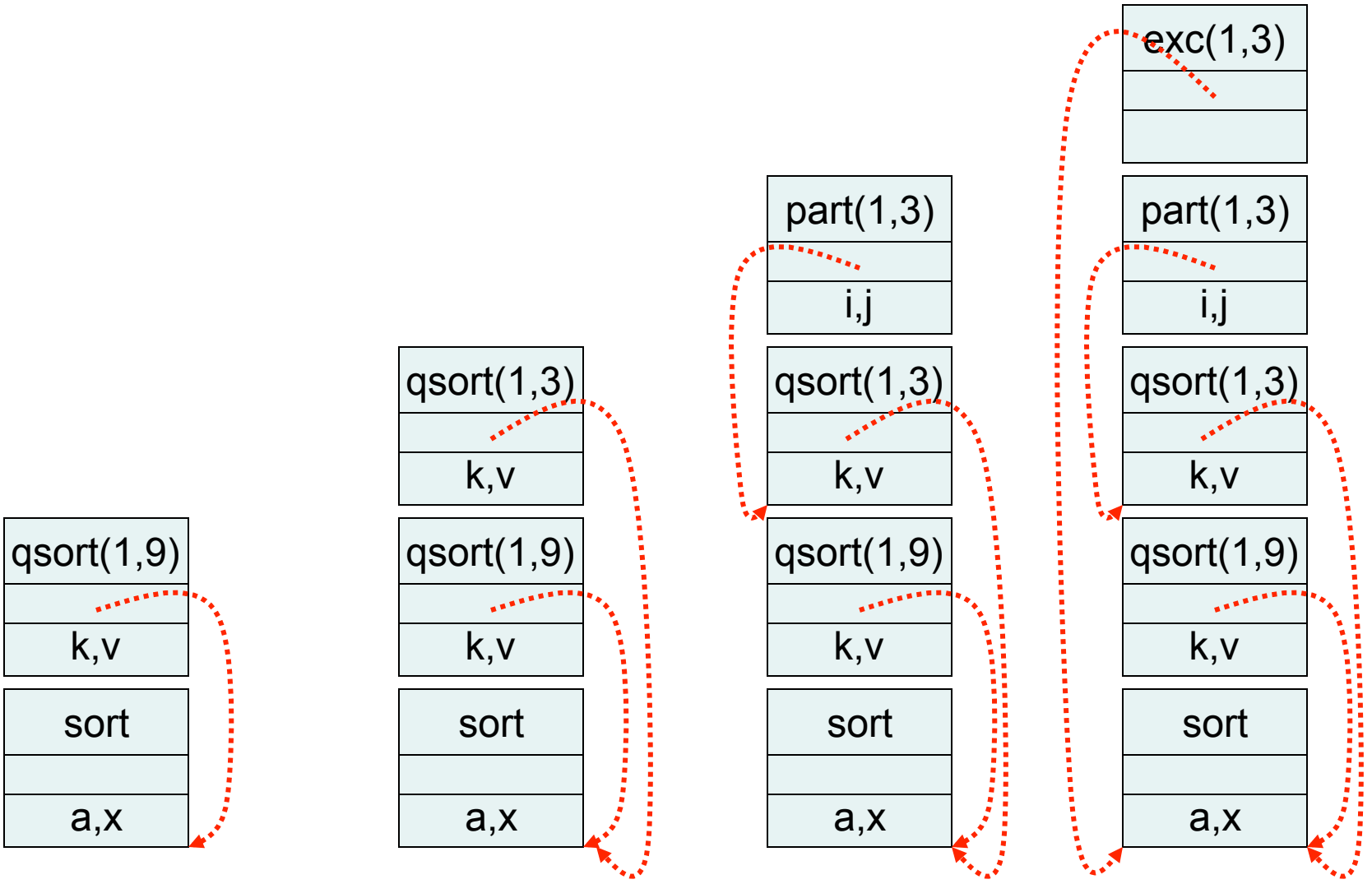
sort calls **quicksort(1,9)**

quicksort(1,9) calls **quicksort(1,3)**

quicksort(1,3) calls **partition(1,3)**

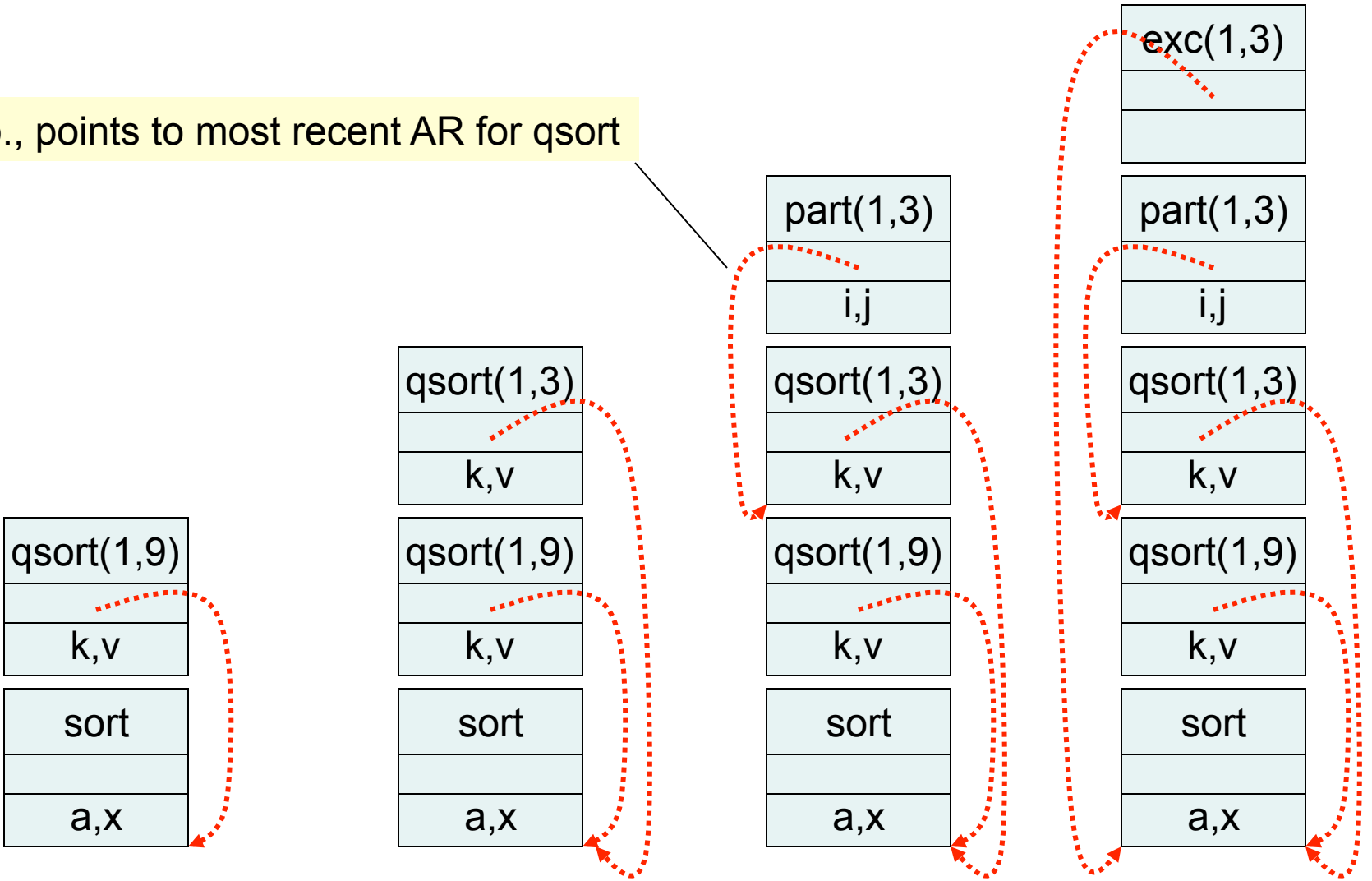
partition(1,3) calls **exchange(1,3)**

Quicksort snapshots



Quicksort snapshots

N.b., points to most recent AR for qsort

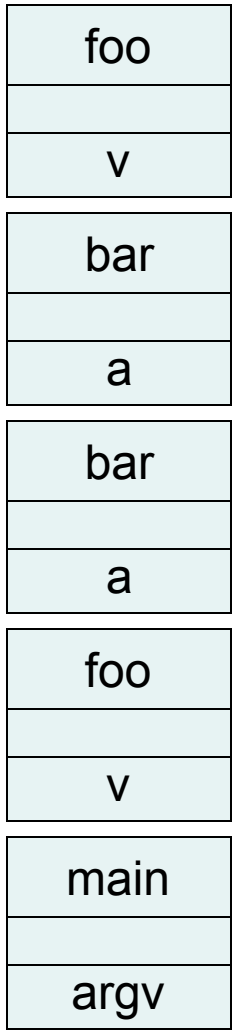


Review: C-style procedures

For non-nested procedures, access links are unnecessary

That is, there's nothing for the access links to point to.

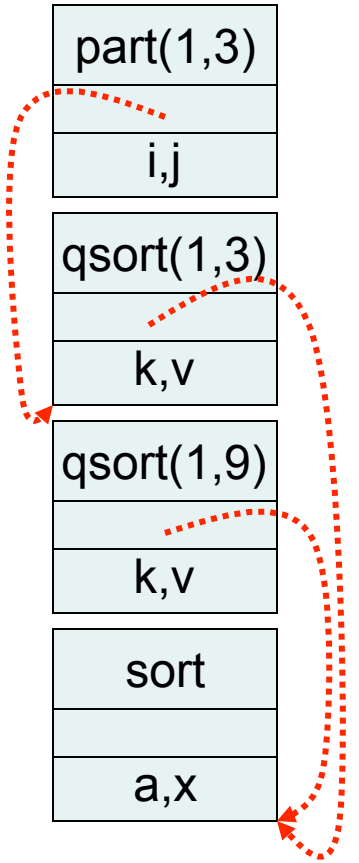
global-declarations;
int foo (char v) { *body-foo* }
void bar (int a) { *body-bar* }
…
void main (int argv[]) { *body-main* }



Non-local references

If procedure **p** refers to non-local **a**, then

- it must be that nestdep(**a**) ≤ nestdep(**p**)
- code for p must
  1. follow nestdep(**p**) - nestdep(**a**) links
  2. look up **a** in that AR from some fixed offset
- Note nestdep(**p**) - nestdep(**a**) can be determined at compile time!
- This changes how a procedure *declaration* is compiled



Ex: nestdep(**part**) - nestdep(**a**) = 3 - 1 = 2

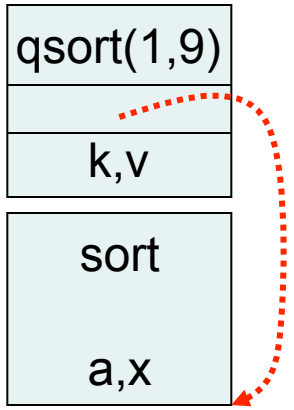
∴ *to look up* **a** *in* **partition**, *follow* 2 *access links*

Setting up access links (i.e., part of a procedure call)

Let **p** (nesting depth np) call procedure **q** (nesting depth nq)

- Case np < nq
  - must be that **q** is declared within **p**
  - ∴ accesslink(**q**) should point to caller **p**

Ex:



NOTE: ND(sort) < ND(qsort)

Setting up access links (i.e., part of a procedure call)

Let **p** (nesting depth np) call procedure **q** (nesting depth nq)

- Case np ≥ nq
  - …look at the following example…

Calculating the access link

partition calls **exchange** : **exchange**'s access link should point to the most recent AR of the "parent" **sort**

```
program sort(input,output);
        …
        procedure readarray;
            … begin … end;
        procedure exchange(i,j : integer);
                begin … end
        procedure quicksort(m,n : integer);
            …
            function partition( y,x : integer): integer;
                …
                begin … exchange(i,j); … end;
            …
```

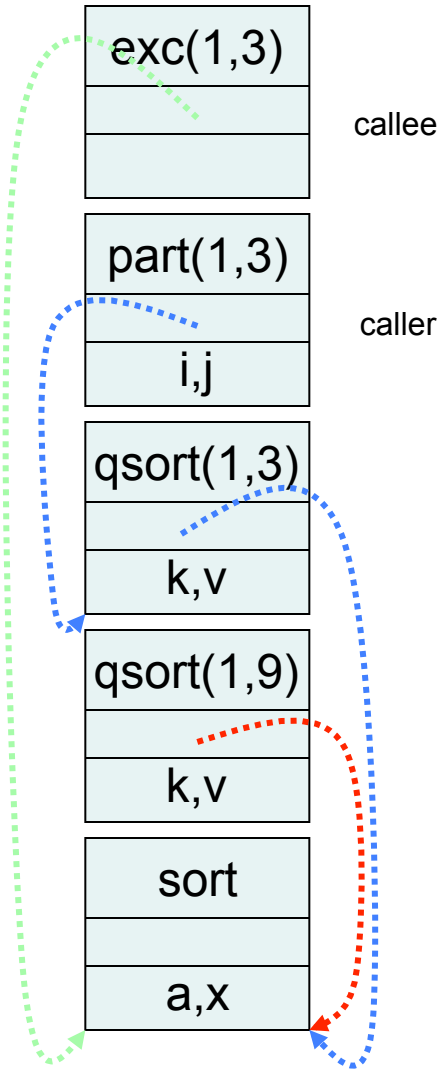
Note: nestdep(partition) ≥ nestdep(exchange)

Calculating the access link

…this AR will be found by following
(nestdep(**partition**) – nestdep(**exchange**) + 1) links

Ex: follow 3 – 2 + 1 = 2 links

N.b., (nd(**p**) – nd(**e**) + 1) can be calculated at compile time



Setting up access links (i.e., part of a procedure call)

Let **p** (nesting depth np) call procedure **q** (nesting depth nq)

- Case np ≥ nq
  - follow (np – nq + 1) access links
  - set accesslink(**q**) to the result
- Case np < nq
  - must be that **q** is declared within **p**
  - ∴ accesslink(**q**) should point to caller **p**

Summary

- Without nesting, compiling procedure calls and definitions is straightforward
- With nesting, resolving non-local variable references becomes more complicated
  - Lexical scope determines how access links are set up and how non-local storage is accessed
- For more, see pages 415-419 of “Compilers: Principles, Tools, and Techniques”
  - AKA “The Dragon Book” (1st Edition)
  - in 2nd edition, 430-435
    - Presentation in 1st edition is (alas) better