

Declarations, Procedures, & Code Generation

Dr. William Harrison

harrisonwl@missouri.edu

CS 4430 Compilers I

Today

- These slides concern source languages called “Micro” and “Micro+”
 - simple & straightforward
 - Imp's ugly stepsister
- Also, a target language called “Tuple”
 - ThreeAddr's ugly stepsister.
- Today, important extensions to Micro and how to compile them; they are:
 - declarations
 - procedures

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 list>}
<statement>	→ ID := <expression>
<statement>	→ read (<id list>)
<statement>	→ write (<expr>)
<id list>	→ ID { , ID }
<expr list>	→ <expression> { , <expression> }
<expression>	→ <primary> { <op> <primary> }
<primary>	→ (<expression>)
<primary>	→ ID
<primary>	→ INTLITERAL
<add op>	→ PLUSOP
<add op>	→ MINUSOP
<system goal>	→ <program> \$CANEOF

**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

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

Call this language **Micro+**

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

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

A typical Micro+ program has the form:

Type id; ... ; Type id; Stmt ; ... ; Stmt

└──────────────────┘ └──────────────────┘

declarations statements

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>...
```

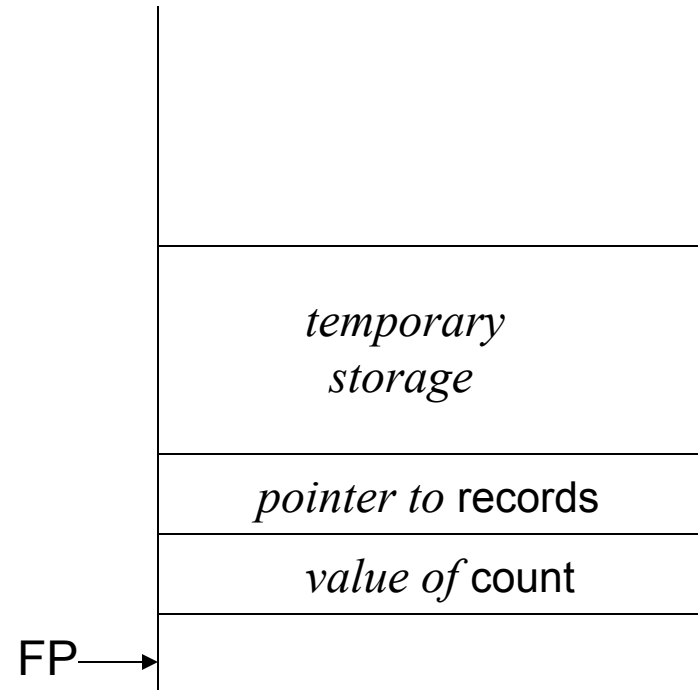
- Given a Micro+ program, we have to decide on how that variables are represented in memory
 - That is, where & how each variable is stored and accessed

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



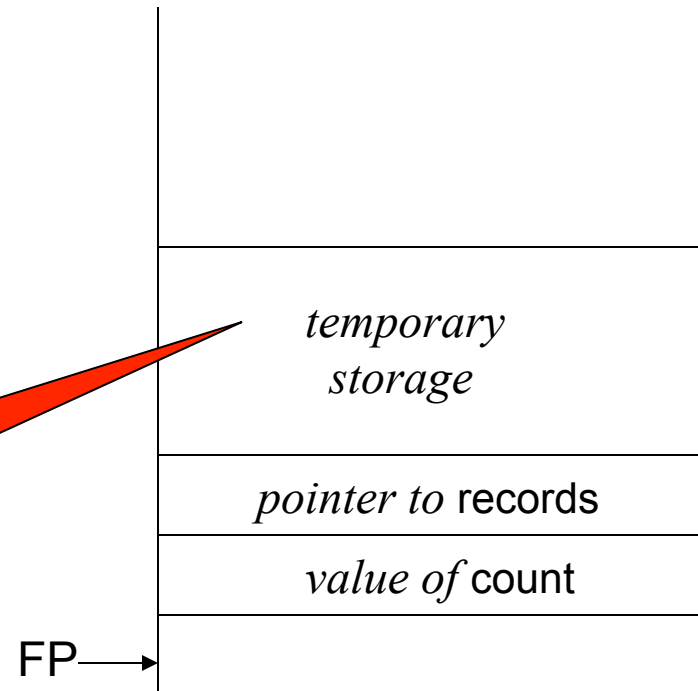
Example

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

generated virtual
registers might be
stored here

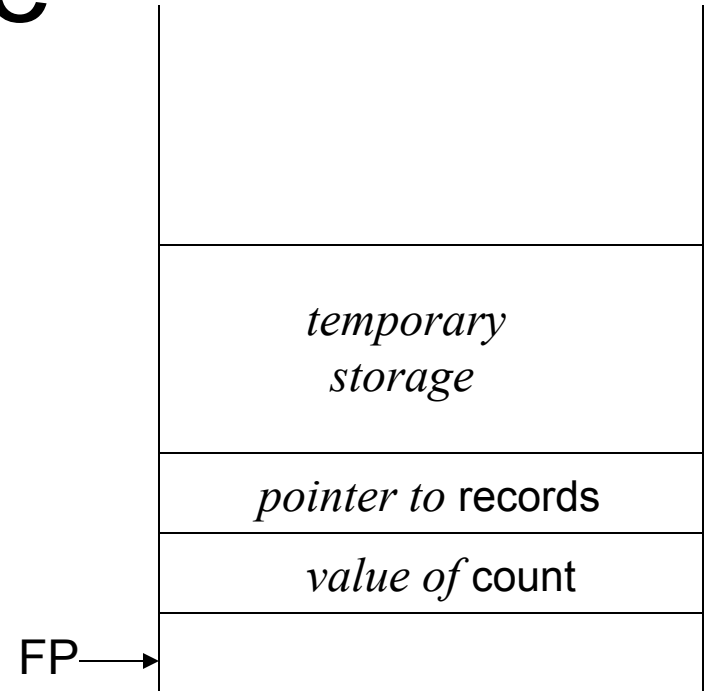
ass

- 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?
 - and, 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.*

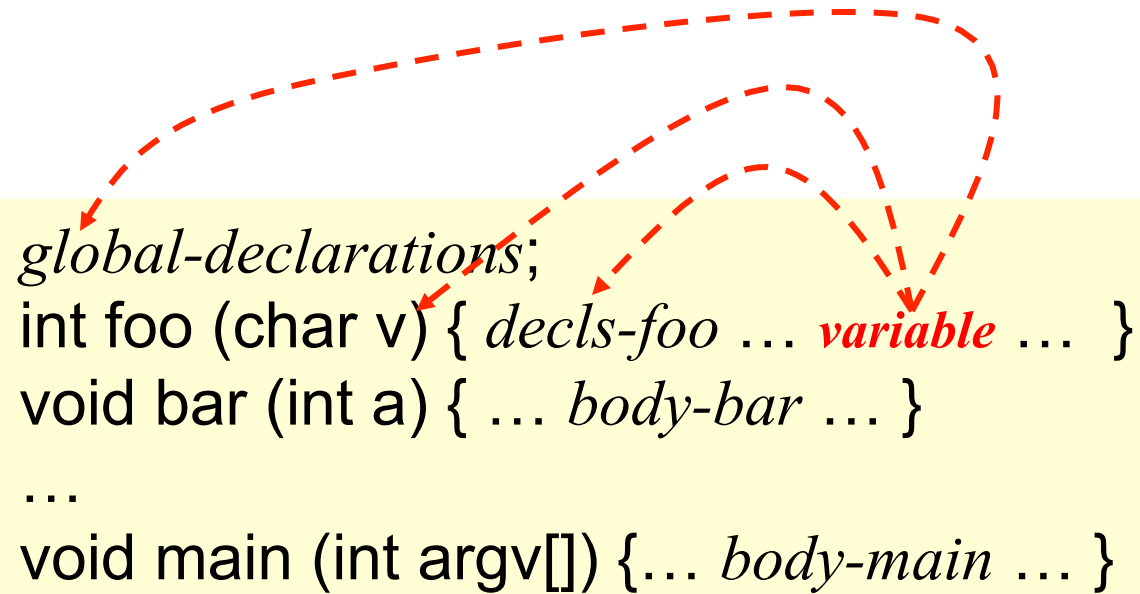
Variable scoping in C

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



The diagram illustrates variable scoping in C. It shows a sequence of code blocks: *global-declarations*;, *decls-foo*, *body-bar*, and *body-main*. Red dashed arrows indicate the scope of variables: one arrow points from the *global-declarations* block to the *body-main* block, another from the *decls-foo* block to the *body-foo* block, and a third from the *body-bar* block to the *body-bar* block. The word *variable* is written in red in the *decls-foo* block.

```
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?

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 ... }
```

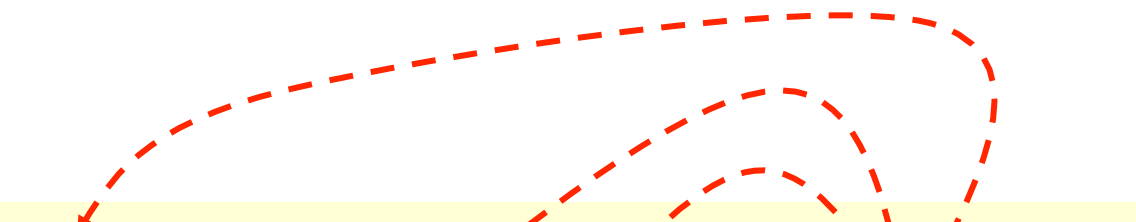
proc-call must refer to:

- 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



The diagram illustrates variable scoping in C using red dashed arrows. One arrow points from the text *global-declarations* to the *global-declarations* line in the code. Another arrow points from the parameter *v* in `int foo (char v)` to the *variable* label in the *decls-foo* block. A third arrow points from the *variable* label to the *body-foo* block. A fourth arrow points from the *variable* label to the *body-main* block. The code is as follows:

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

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

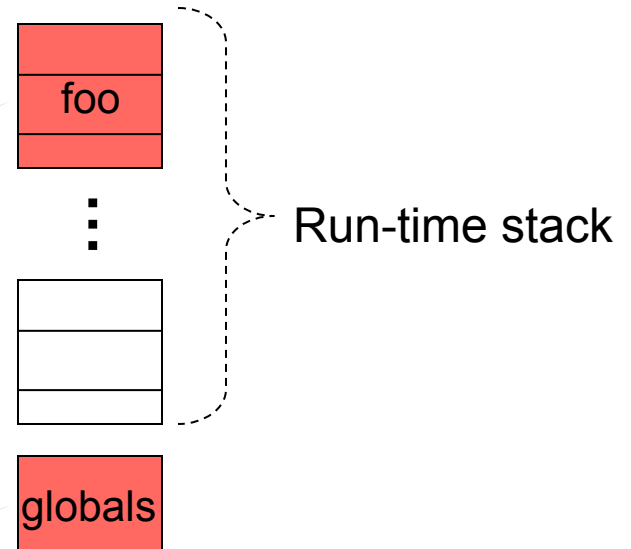
- **Locally**: it's either a procedure parameter (like “v”) or is declared locally (as in *decls-foo*)
- **Globally**: e.g., it's declared in *global-declarations*

...as a consequence

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

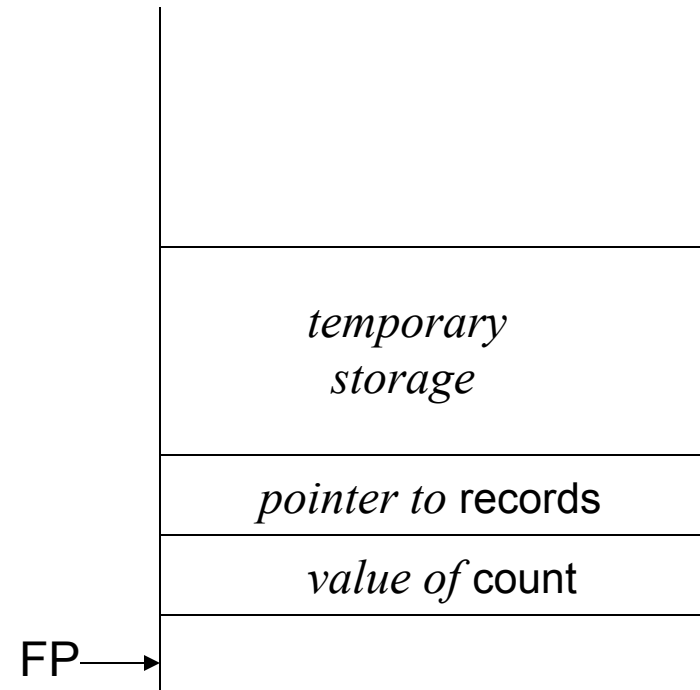
variable is
stored **here**

...or there



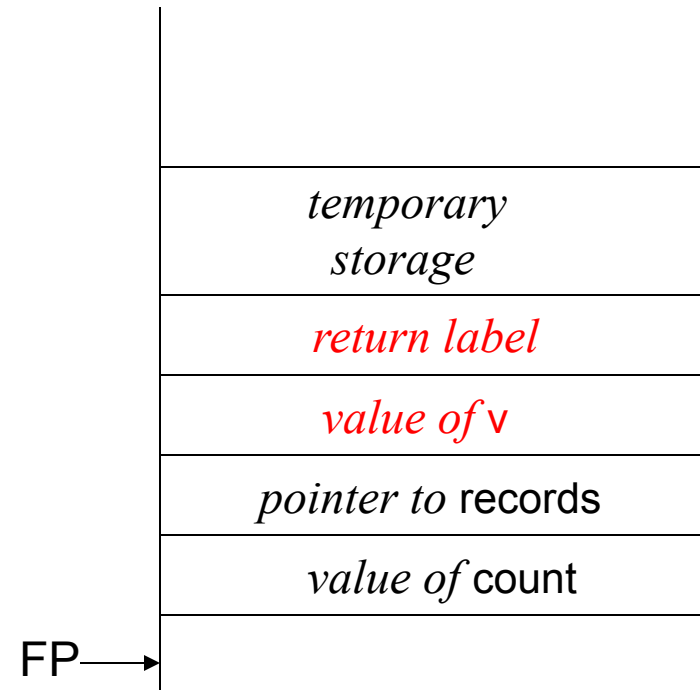
Recall the Micro+ mem. model

```
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>...  
}
```



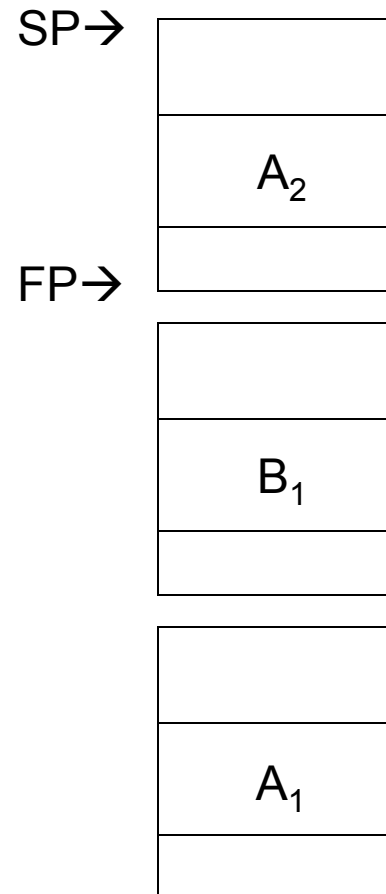
adds book-keeping details for actual arguments and returns

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”

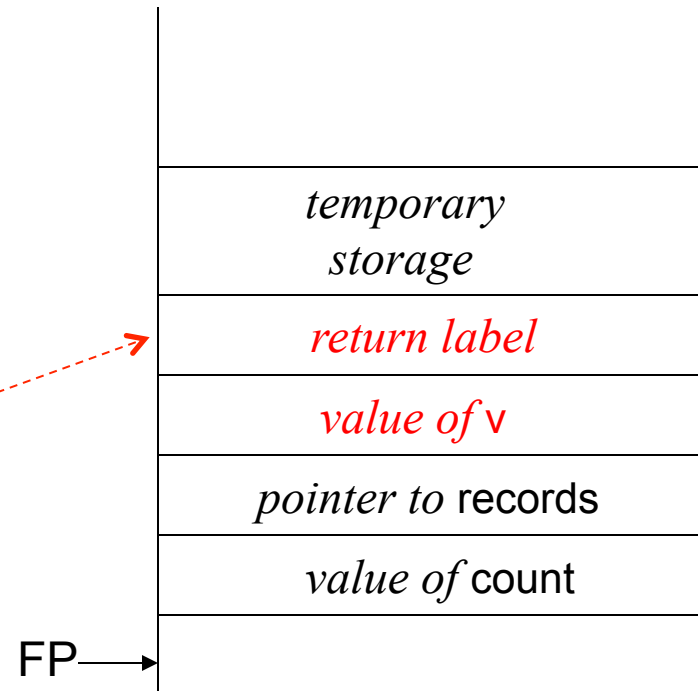
- with frame pointer FP
- stack pointer SP
- calling code must prepare this new AR



Activation record

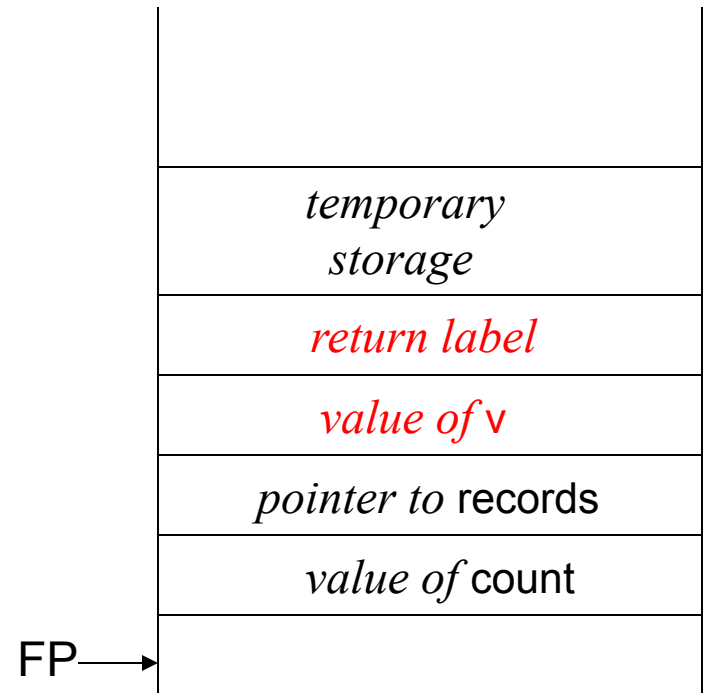
```
void foo (char v) {  
    int count ;  
    int[] records[100] ;  
    count := 1 ;  
    record[count] := 5 ;  
    ...<rest of foo>...  
}
```

This is
label
from
the
caller's
code



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

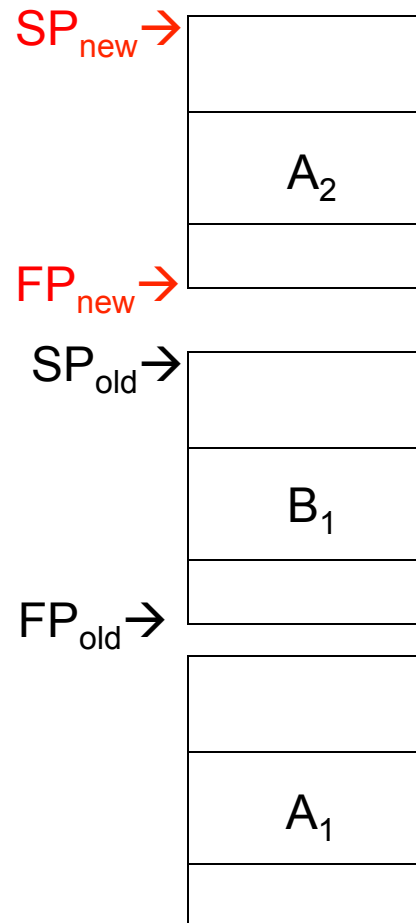
- Locally declared variables
 - are at a fixed offset from FP
 - note that this offset is determined statically
 - this offset is stored in the **symbol table** for each variable
- Procedure parameters
 - also at fixed FP-offset
 - store in symbol table
 - note that this offset may be determined statically
 - 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

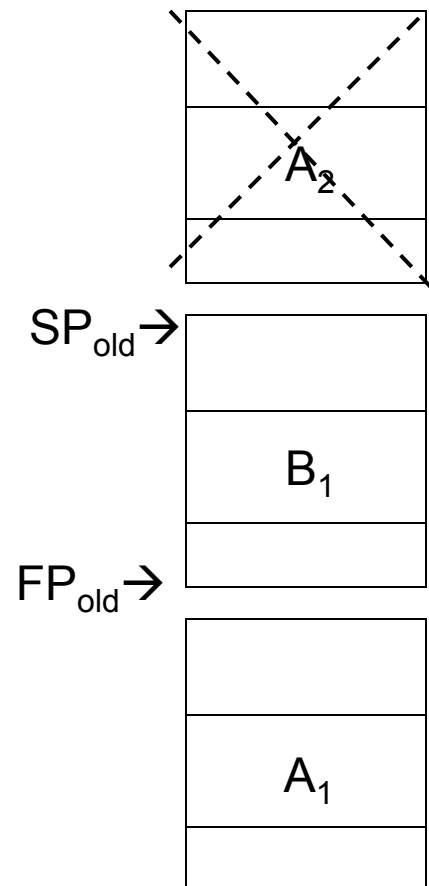


Returning from call

```
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

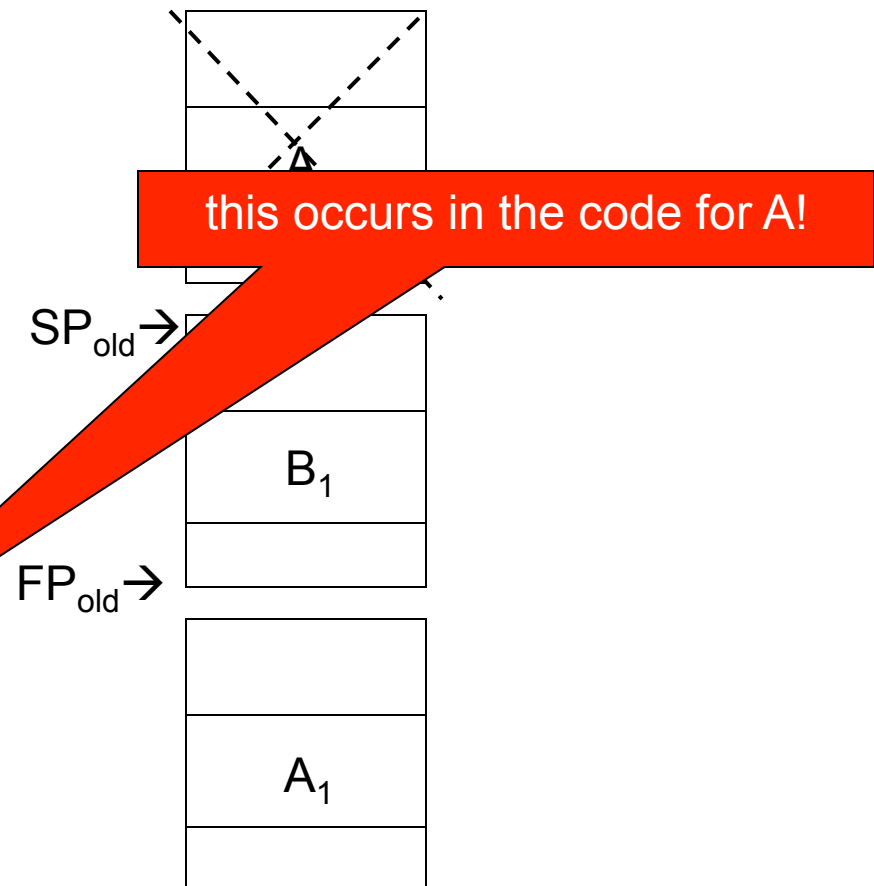


Returning from call

```
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

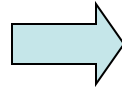


Compiling procedure declarations

- 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 }
```



```
Lfoo : <code for body-foo >  
  
Lbar : <code for body-bar >  
  
Lmain : <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 may refer to function arguments or globals

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)’”

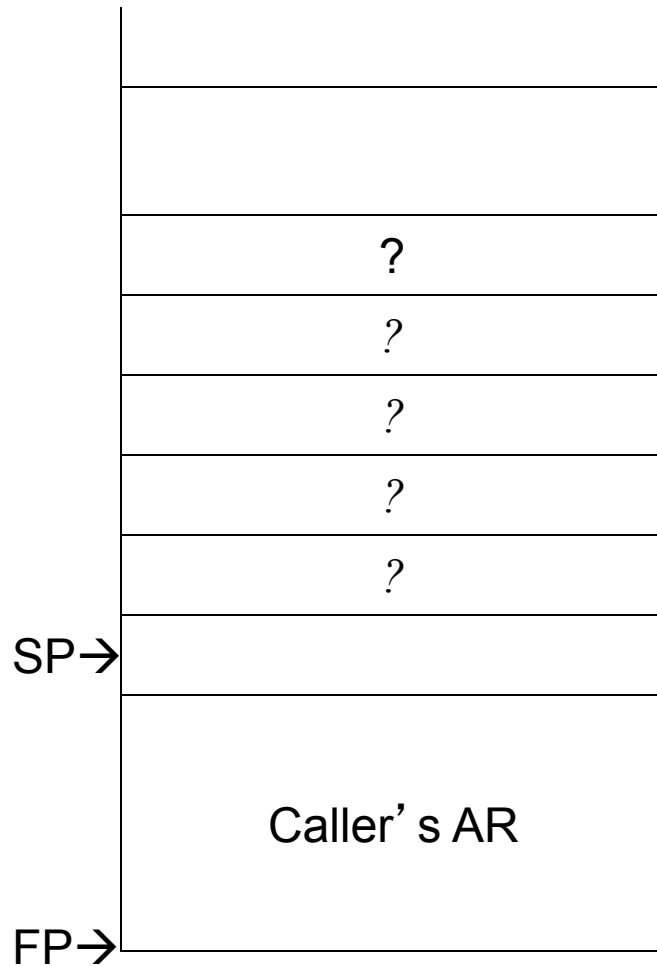
```
void foo (char v) {  
    int count ;  
    int[] records[100] ;  
    count := 1 ;  
    record[count] := 5 ;  
    ...<rest of foo>...  
}
```

Code to call foo(‘a)

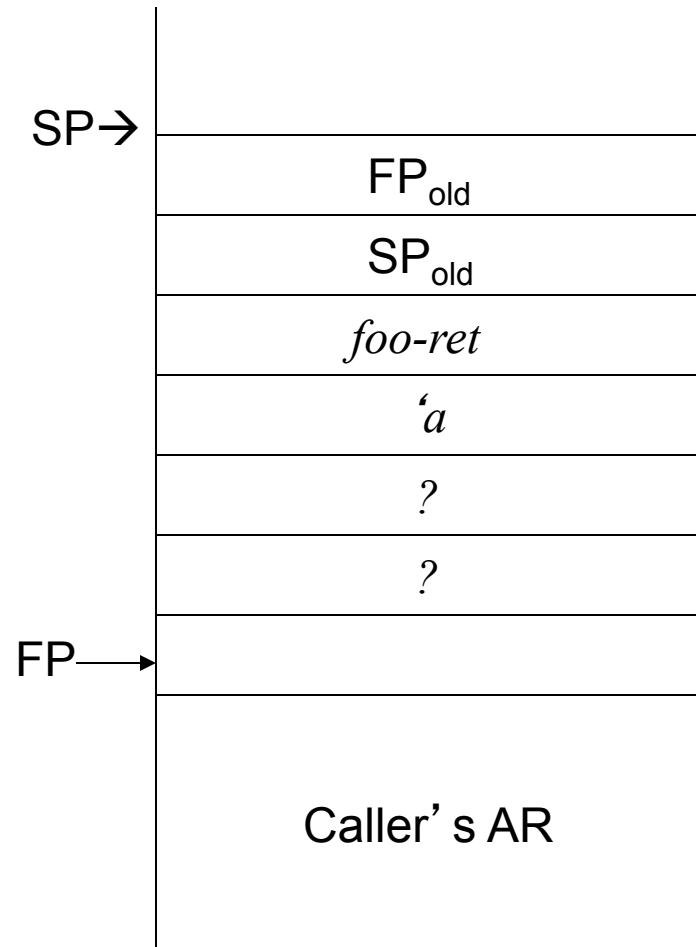
```
[SP+3] := ‘a  
[SP+4] := foo-ret  
[SP+5] := SP  
[SP+6] := FP  
FP := SP  
SP := SP+7  
call foo-label foo-ret  
Label foo-ret
```

Activation record for “foo(‘a)’”

Before AR construction



After



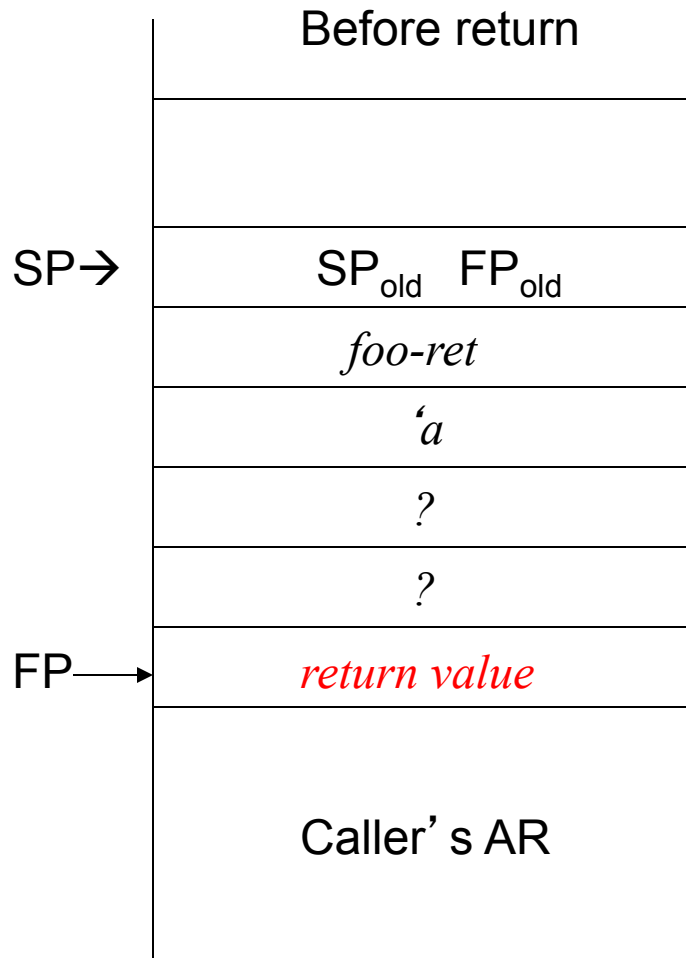
what if “foo(‘a)’” returns a value?

```
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]

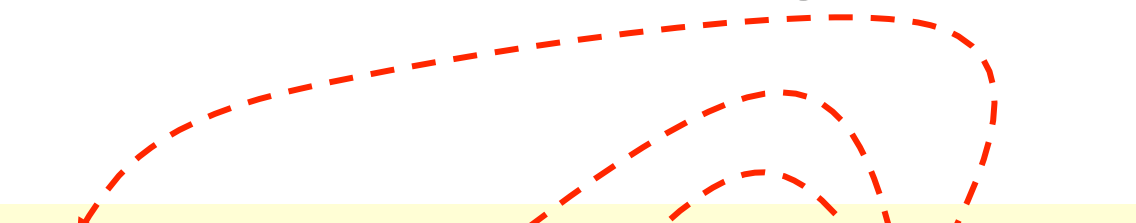


Or, return in a register

Summary

- Showed how C-like procedures could be added to Micro+ compiler
 - Activation records \approx 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



The diagram illustrates variable scoping in C using red dashed arrows. One arrow points from the word *global-declarations* to the *global-declarations* section of the code. Another arrow points from the word *variable* to the parameter *v* in the function *foo*. A third arrow points from the word *variable* to the *decls-foo* section of the function *foo*. A fourth arrow points from the word *variable* to the *body-main* section of the *main* function.

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

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

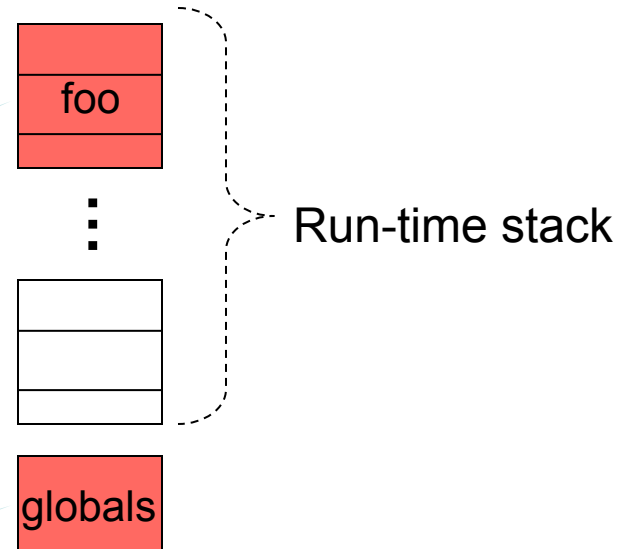
- **Locally**: it's either a procedure parameter (like “v”) or is declared locally (as in *decls-foo*)
- **Globally**: e.g., it's declared in *global-declarations*

...as a consequence

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

variable is
stored **here**

...or there



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

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

The diagram illustrates the scope of declarations in nested procedure declarations. Red dashed arrows point from the opening curly brace of each procedure to the declarations within its body. Specifically, an arrow points from the opening brace of `global-decls` to the `foo-decls` block. Another arrow points from the opening brace of `proc foo` to the `bar-decls` block. A third arrow points from the opening brace of `proc bar` to the `variable` declaration. This shows that the `variable` is only visible within the scope of `proc bar`.

Q: where is *variable* defined?

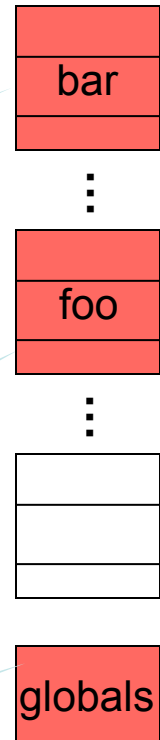
...as a consequence

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

variable is
stored **here**

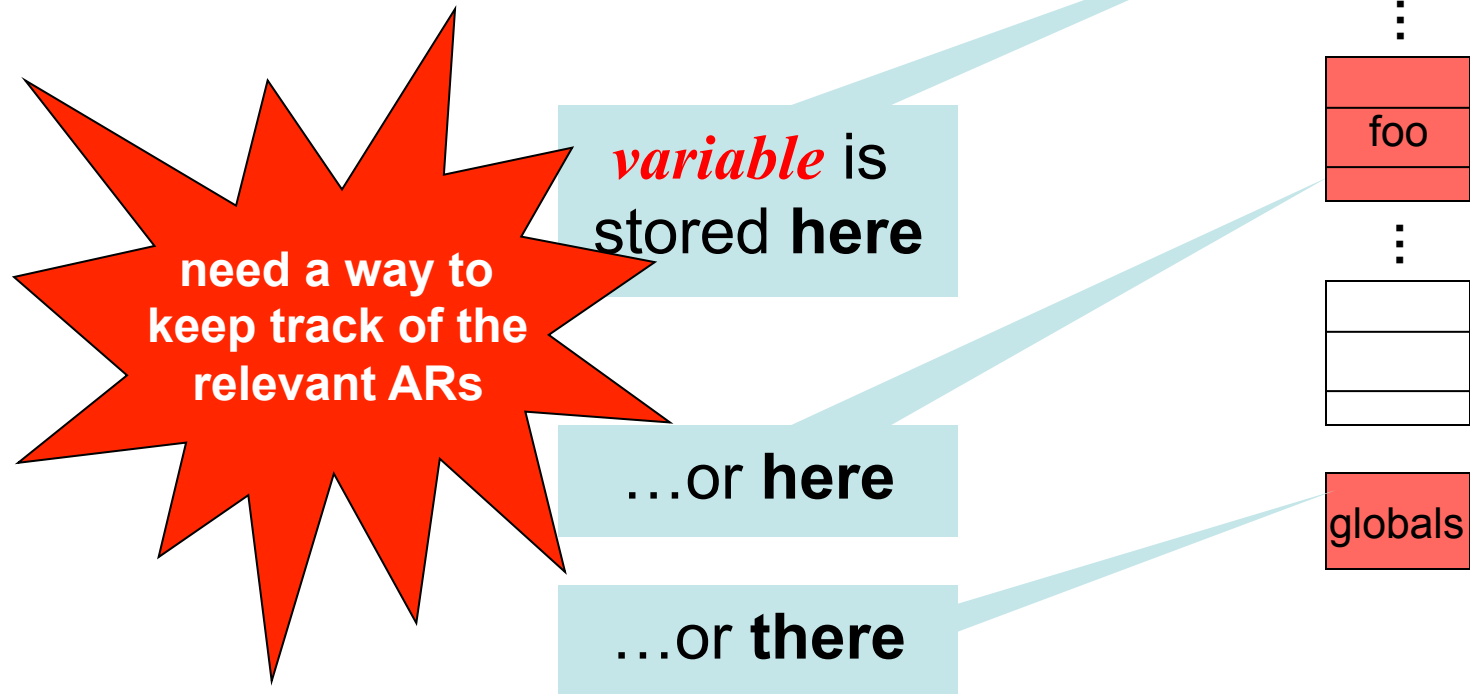
...or **here**

...or **there**



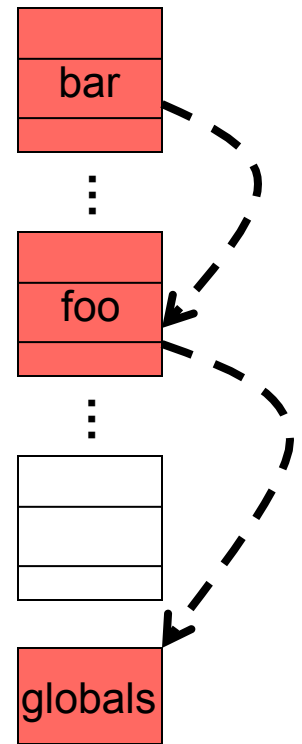
...as a consequence

```
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

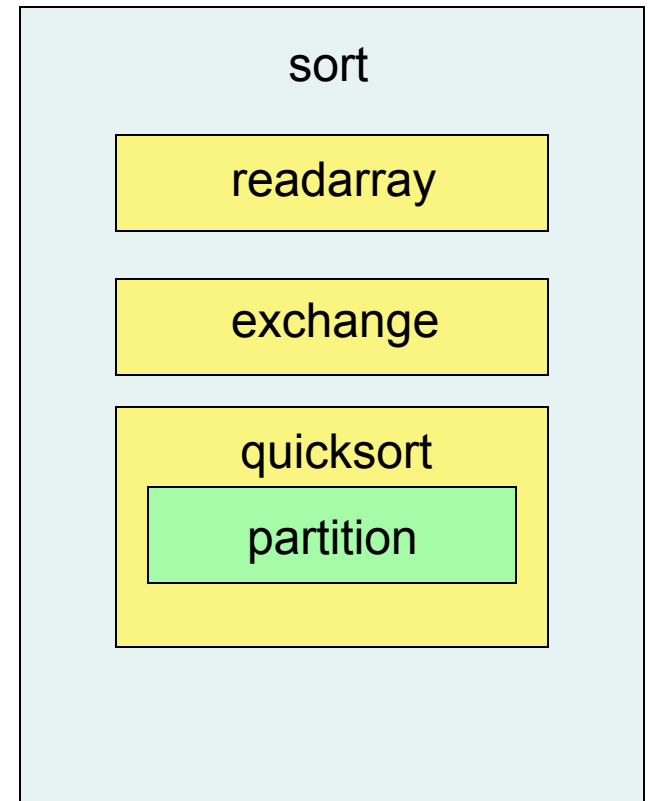
  procedure quicksort(m,n : integer);
    var k,v : integer;
    function partition( y,x : integer): integer;
      var i,j : integer;
      begin ... a ...
          ... v ...
          ... exchange(i,j); ...
      end;

    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


```

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

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:

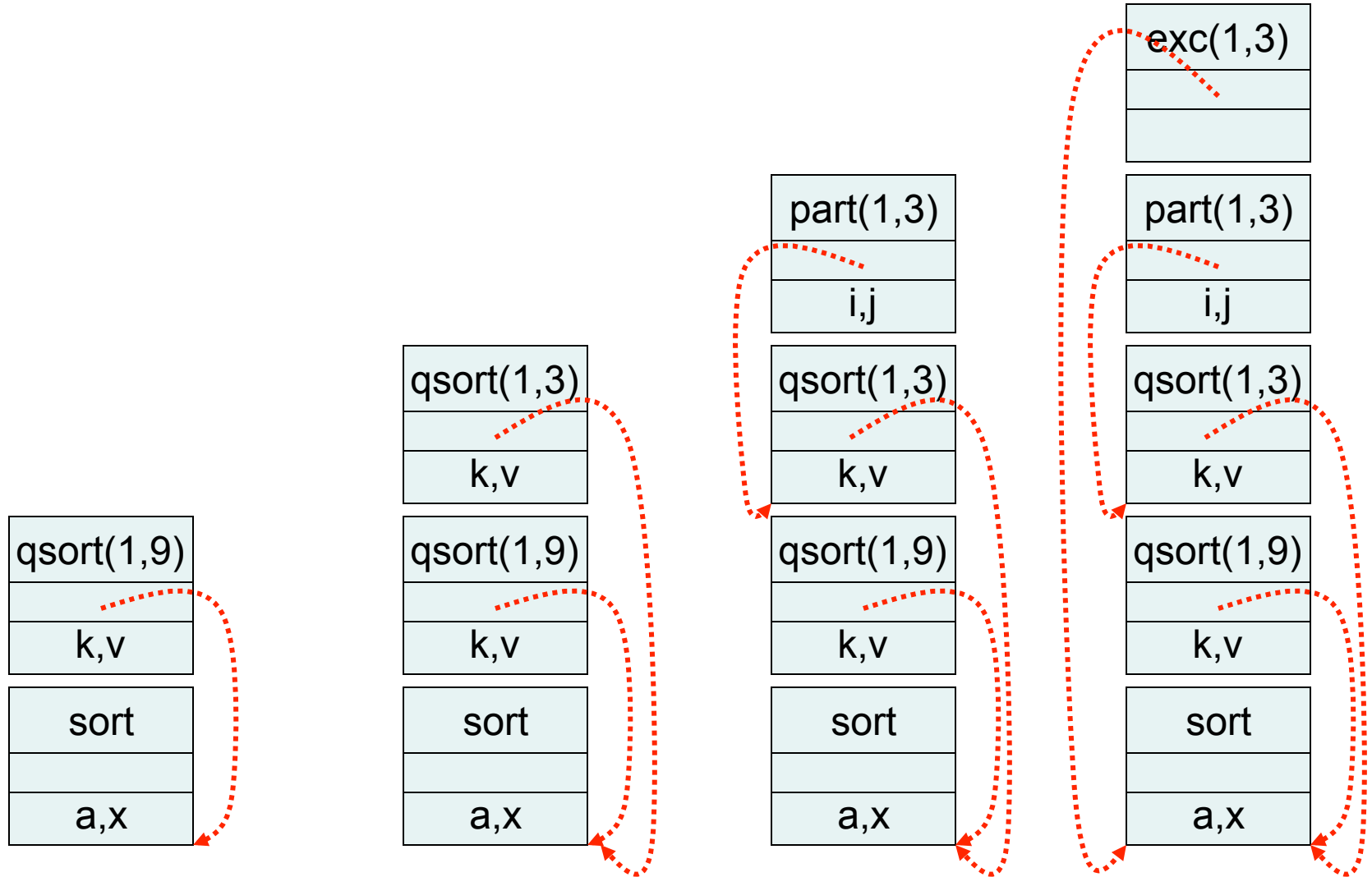
sort calls **quicksort(1,9)**

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

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

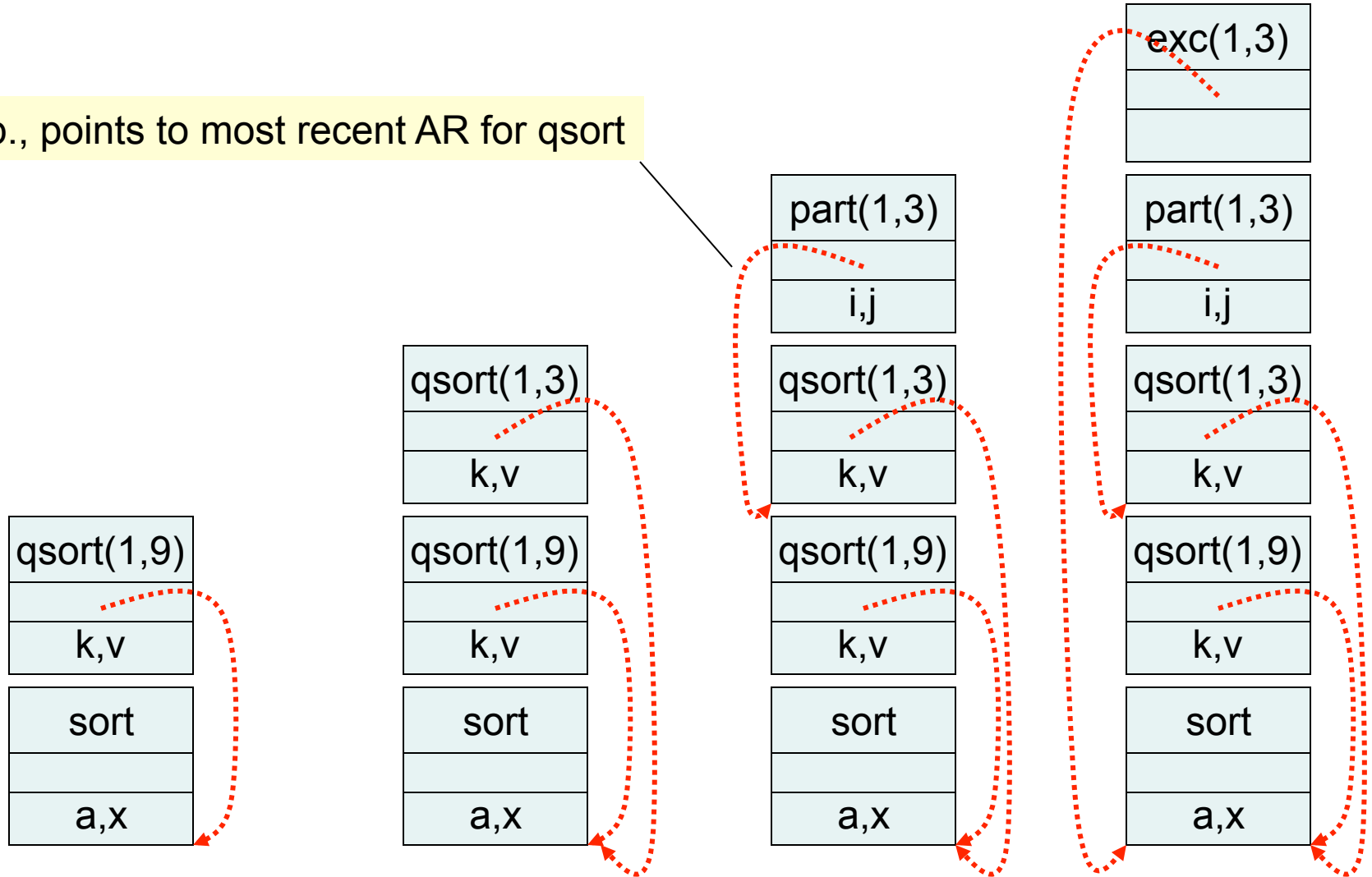
partition(1,3) calls **exchange(1,3)**

Quicksort snapshots



Quicksort snapshots

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

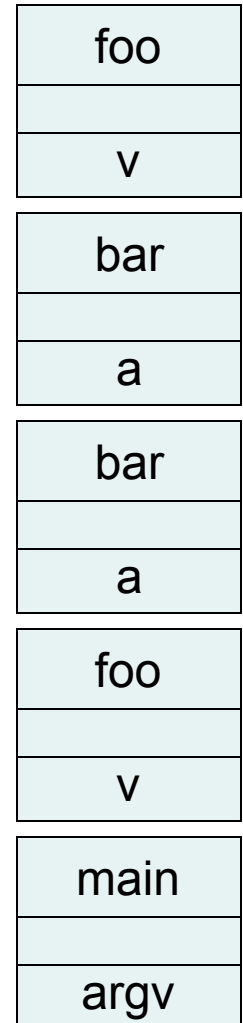


Review: C-style procedures

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 }
```



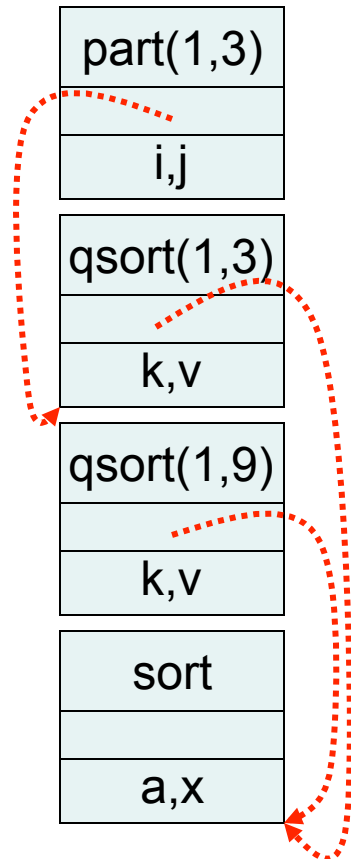
Non-local references

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

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

Ex: $\text{nestdep}(\mathbf{part}) - \text{nestdep}(\mathbf{a}) = 3 - 1 = 2$

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

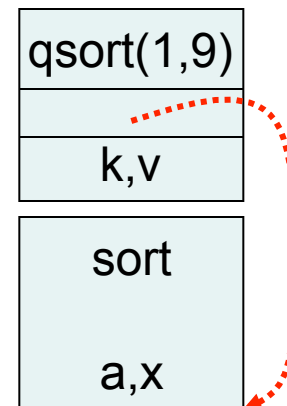


Setting up access links (i.e., part of a procedure call)

Let **p** (nesting depth n_p) call procedure **q** (nesting depth n_q)

- Case $n_p < n_q$
 - must be that **q** is declared within **p**
 - \therefore accesslink(**q**) should point to caller **p**

Ex:



NOTE: $ND(\text{sort}) < ND(\text{qsort})$

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

Let **p** (nesting depth n_p) call procedure **q** (nesting depth n_q)

- Case $n_p \geq n_q$
 - ...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;  
    ...
```

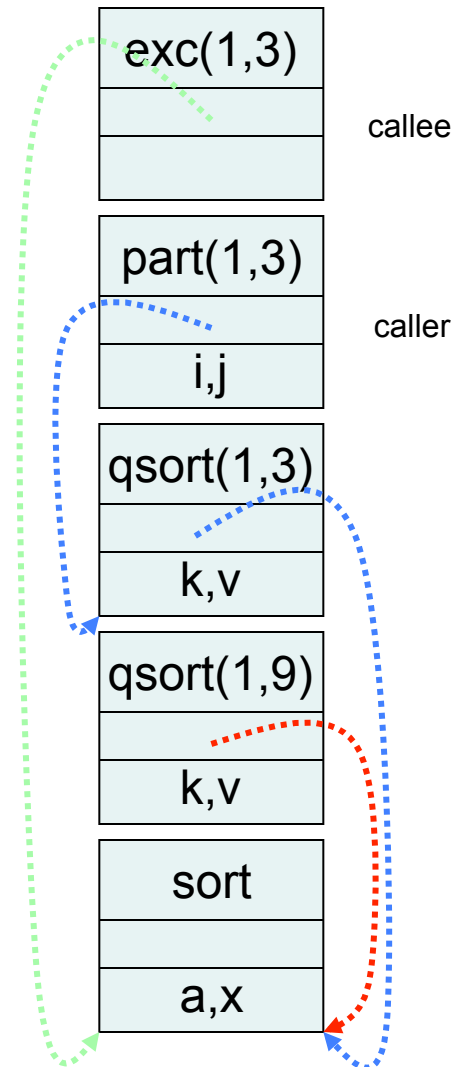
Note: $\text{nestdep}(\text{partition}) \geq \text{nestdep}(\text{exchange})$

Calculating the access link

...this AR will be found by following
 $(\text{nestdep}(\mathbf{partition}) - \text{nestdep}(\mathbf{exchange}) + 1)$ links

Ex: follow $3 - 2 + 1 = 2$ links

N.b., $(\text{nd}(\mathbf{p}) - \text{nd}(\mathbf{e}) + 1)$ can be calculated
at compile time



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

Let **p** (nesting depth n_p) call procedure **q** (nesting depth n_q)

- Case $n_p \geq n_q$
 - follow $(n_p - n_q + 1)$ access links
 - set `accesslink(q)` to the result
- Case $n_p < n_q$
 - must be that **q** is declared within **p**
 - \therefore `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