Where to Store Variables?

Static Allocation

Variables created at compile-time
Size and address known at compile-time

Stack Allocation

Variables placed in activation records on a stack
Variables are created / destroyed in LIFO order

Heap Allocation

Size and address determined at run-time
Creation / destruction of data occurs in any order

CS-322 Code Generation-Part 2

Static Allocation

Early Languages (FORTRAN)

Each variable is placed in memory ("static allocation")
Fortran had routines, but...
- No stack
- Recursion was not possible

Values of a routine's variables are retained across invocations
    Initialization vs. re-initialization
Each variable's size must be known at compile-time
    Dynamic arrays?

© Harry H. Porter, 2006

Stack Allocation

Each variable is "*local*" to some routine

Invoke a routine?

Allocate storage for its variables
(and initialize it?)

Return?

Pop frame
(Variables are destroyed)

Consider one routine (e.g., "quicksort")

Many activations, many frames
$\Rightarrow$ Many copies of each local variable

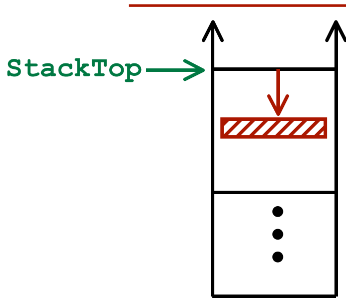
Local variables:
Each invocation has its own set of variables

The "currently active" invocation
Its variables will be in the frame on top of stack.

Every reference to a local variable...
will access data in the top frame

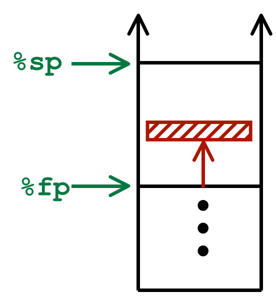
References to a local variable in the currently active routine...

General Idea



`*(StackTop + offset`$_X$`)`

In the SPARC




```
ld    [%fp-48],%l5
```

Laying Out the Frame

Each local (and temp) variable has a size

"C" int: 4 bytes,
double: 8 bytes, ...

Each local and temp variable needs an "**offset**"

```
for each procedure (or block) do
  offset = 0;
  for each local and temp variable do
    assign this variable to current offset
    offset = offset + this variable's length
  endFor
endFor
```


Laying Out the Frame

Each local (and temp) variable has a size

"C" int: 4 bytes,
double: 8 bytes, ...
"PCAT" all variables: 4 bytes

Each local and temp variable needs an "**offset**"

```
for each procedure (or block) do
  offset = 0;
  for each local and temp variable do
    assign this variable to current offset
    offset = offset + this variable's length
  endFor
endFor
```

May start at some other value (`-4`)

We'll use `-4`

Laying Out the Frame

Each local (and temp) variable has a size

"C"	int: 4 bytes,
	double: 8 bytes, ...
"PCAT"	all variables: 4 bytes

Each local and temp variable needs an "**offset**"

```
for each procedure (or block) do
  offset = 0;
  for each local and temp variable do
    assign this variable to current offset
    offset = offset + this variable's length
  endFor
endFor
```

May start at some other value (`-4`)

We'll use `-4`

We'll treat "main" body as just another routine.
It will have a frame

Global variables
Treat identically to local variables for procedures!

Example

Variable	Size	Offset
w	4	0
x	8	4
y	4	12
z	1	16
a	8	17
b	1	25
c	8	26
	34	34

Ignoring alignment issues

Example

Variable	Size	Offset	Offset with Alignment
w	4	0	0
x	8	4	8
y	4	12	16
z	1	16	20
a	8	17	24
b	1	25	32
c	8	26	40
	34	34	48

4 bytes of padding

3 bytes of padding

7 bytes of padding

Example

Variable	Size	Offset	Offset with Alignment
w	4	0	0
x	8	4	8
y	4	12	16
z	1	16	20
a	8	17	24
b	1	25	32
c	8	26	40
	34	34	48

4 bytes of padding

3 bytes of padding

7 bytes of padding

Re-order!

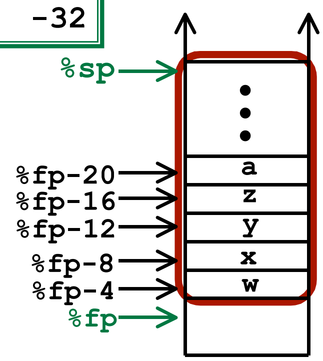
Place variables with most restrictive alignment first.

Variable	Size	Offset
x	8	0
a	8	8
c	8	16
w	4	24
y	4	28
z	1	32
b	1	33
	34	34

PCAT Example

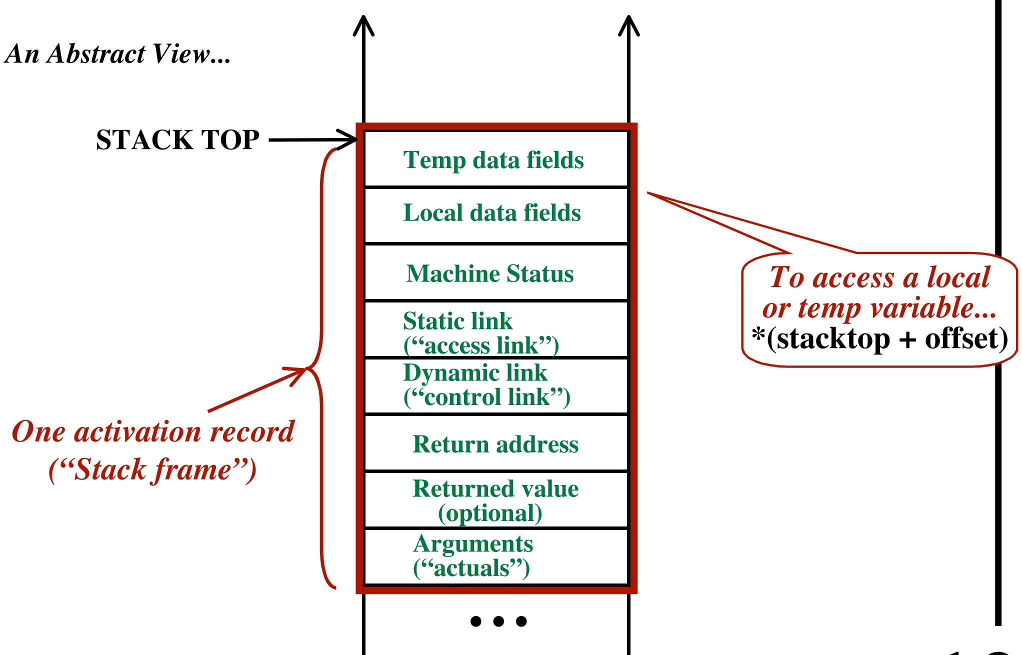
Initial offset: -4
Increment: -4

Variable	Size	Offset
w	4	-4
x	4	-8
y	4	-12
z	4	-16
a	4	-20
b	4	-24
c	4	-28
	28	-32



The Stack of Activation Records

An Abstract View...

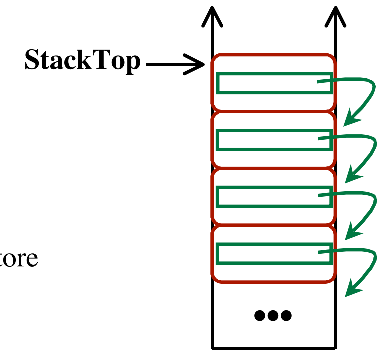


The Dynamic Link

When we return, we need to be able to go back to previous frame.

During a call:

- Save old StackTop
- Increment StackTop
  (Allocate new frame)
- Store old StackTop
  in Dynamic Link field



During a return:

- Use the Dynamic Link to restore
  the old StackTop

To access local variables:

`[StackTop + offset`$_x$`]`

To access variables in our caller's frame:

`[[StackTop + offset`$_{DynamicLink}$`] + offset`$_x$`]`

What do we need from the caller's frame?

- Arguments?
- Place to store a returned result?

The Static Link
used to access non-local variables (to be discussed later)

SPARC

Much of the Activation Record is cached in registers!!!

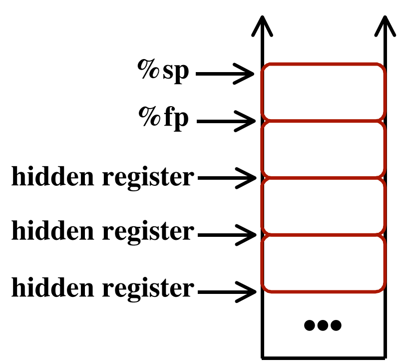
Dynamic Link
Stored in registers (%sp, %fp)

Return Address
Stored in registers (%i7, %o7)

Arguments
Some in registers %i0 ... %i5
Rest in caller's frame

Returned Value
32-bits: in register (%i0, %o0)

Machine Status
64 bytes
Architecture dependent & often not needed



The "Calling Sequence"

- Compute argument values
- Allocate new frame
- Initialize it
    - Move arguments into the new frame (optional)
    - Save machine state (optional)
    - Save return address
- Transfer control to new routine

The "Return Sequence"

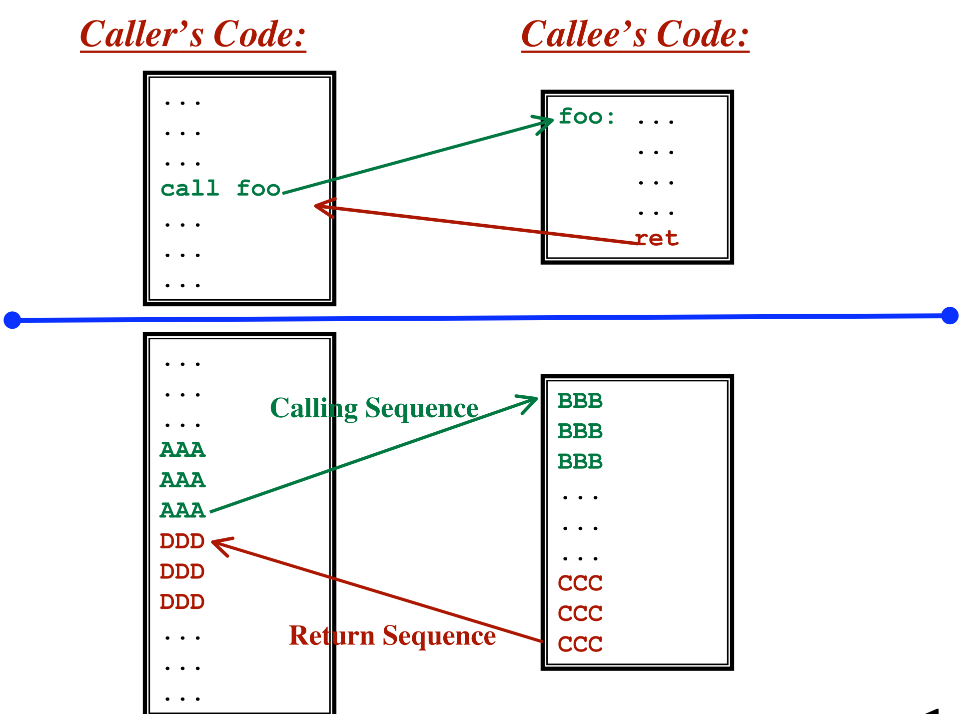
- Compute and move return value (optional)
- Pop stack / delete the top frame
- Resume execution in the caller's code

Flexibility as to who...

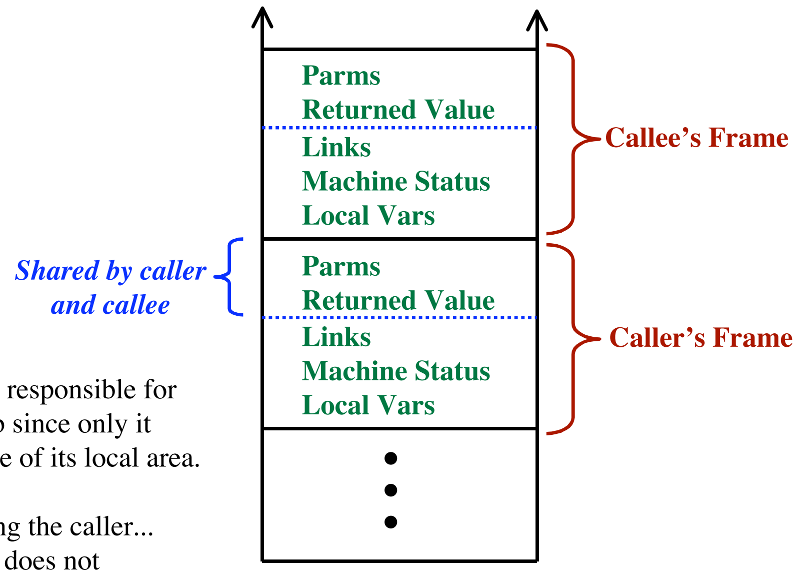
- ***caller***
- ***callee***

...does what...

- ***calling sequence***
- ***return sequence***



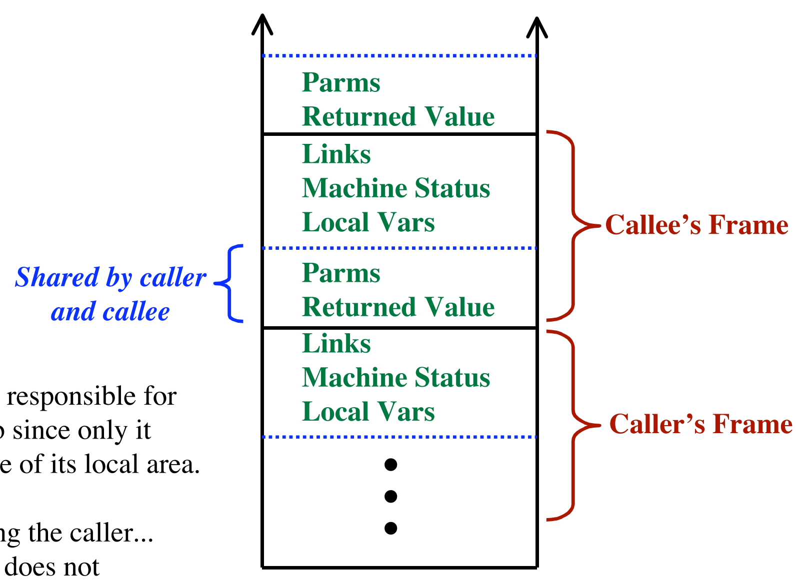
Where do you draw the line between the caller's frame and the callee's frame?



Callee must be responsible for
part of the job since only it
knows the size of its local area.

When compiling the caller...
The compiler does not
have the callee's code.

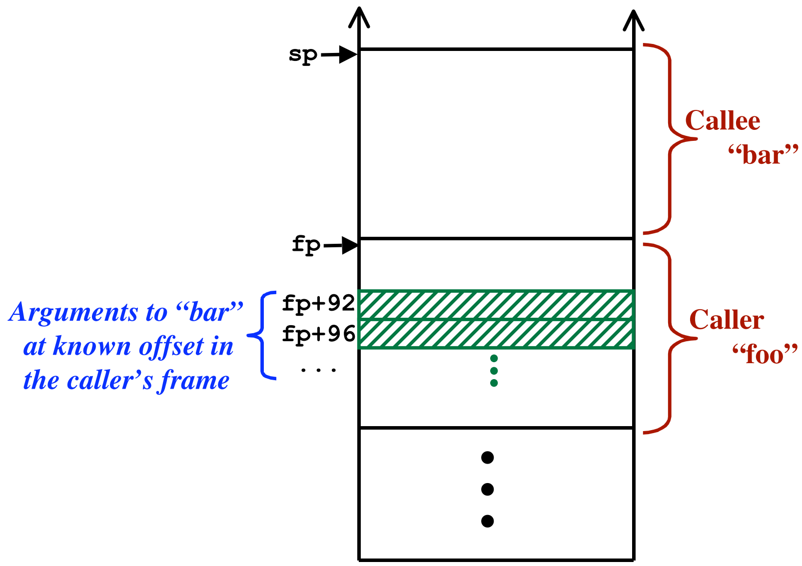
Where do you draw the line between the caller's frame and the callee's frame?



Callee must be responsible for
part of the job since only it
knows the size of its local area.

When compiling the caller...
The compiler does not
have the callee's code.

Parameter Passing in SPARC



Using a Stack for Expression Evaluation

Source Code:

```
x := y + (2 * z);
```

Target Code:

```
push y
push 2
push z
mult
add
pop x
```

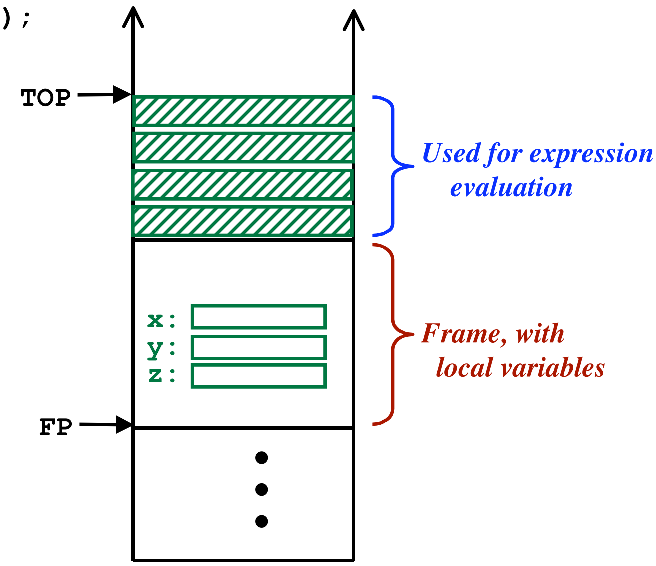
Using a Stack for Expression Evaluation

Source Code:

```
x := y + (2 * z);
```

Target Code:

```
push y
push 2
push z
mult
add
pop x
```



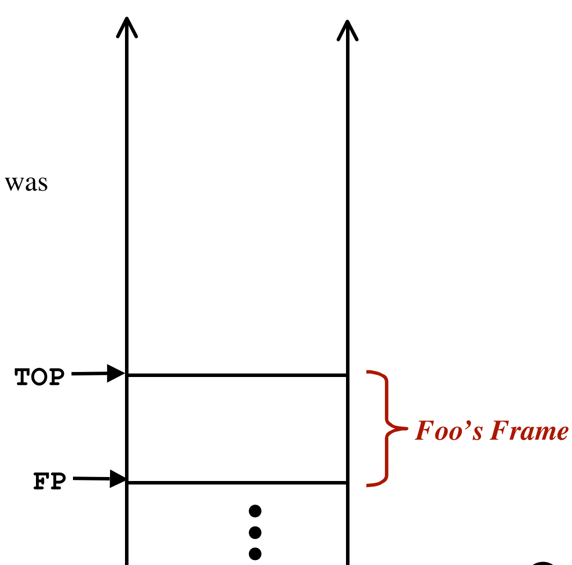
Using a Stack for Argument Evaluation and Parameter Passing

Calling Sequence:

Push args onto the stack
Save FP
FP = TOP
TOP = TOP + FrameSize

Return Sequence:

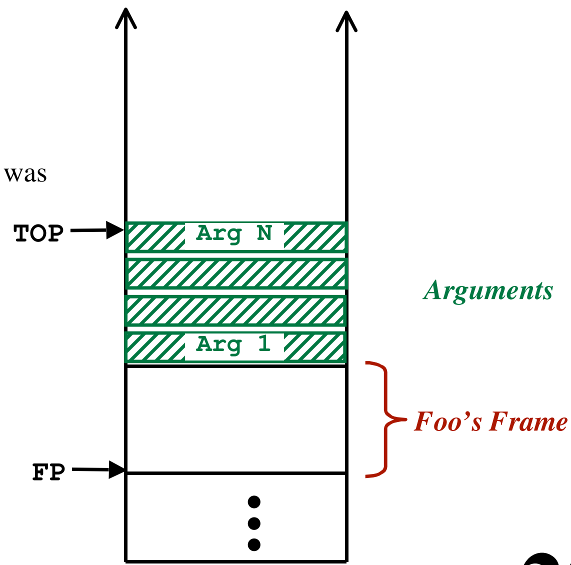
Move return value to where arg 1 was
Restore TOP, FP
Pop stack top into...



Using a Stack for
Argument Evaluation and Parameter Passing

Calling Sequence:
Push args onto the stack
Save FP
FP = TOP
TOP = TOP + FrameSize

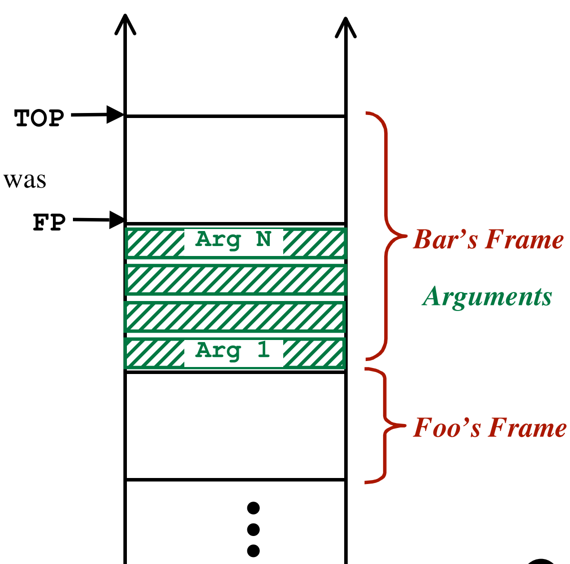
Return Sequence:
Move return value to where arg 1 was
Restore TOP, FP
Pop stack top into...



Using a Stack for
Argument Evaluation and Parameter Passing

Calling Sequence:
Push args onto the stack
Save FP
FP = TOP
TOP = TOP + FrameSize

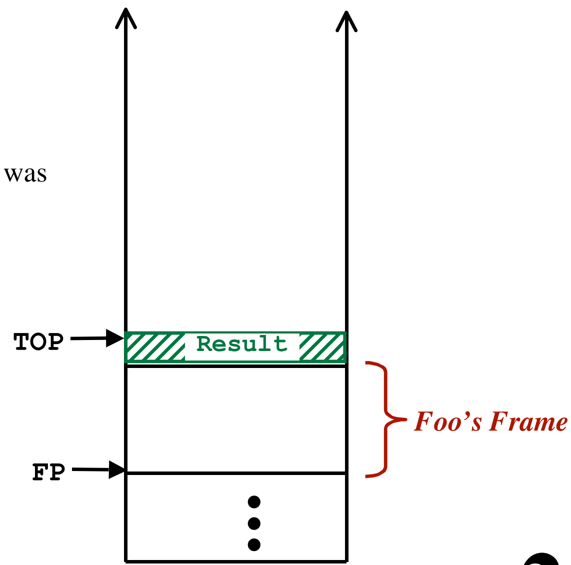
Return Sequence:
Move return value to where arg 1 was
Restore TOP, FP
Pop stack top into...



Using a Stack for Argument Evaluation and Parameter Passing

Calling Sequence:
- Push args onto the stack
- Save FP
- FP = TOP
- TOP = TOP + FrameSize

Return Sequence:
- Move return value to where arg 1 was
- Restore TOP, FP
- Pop stack top into...



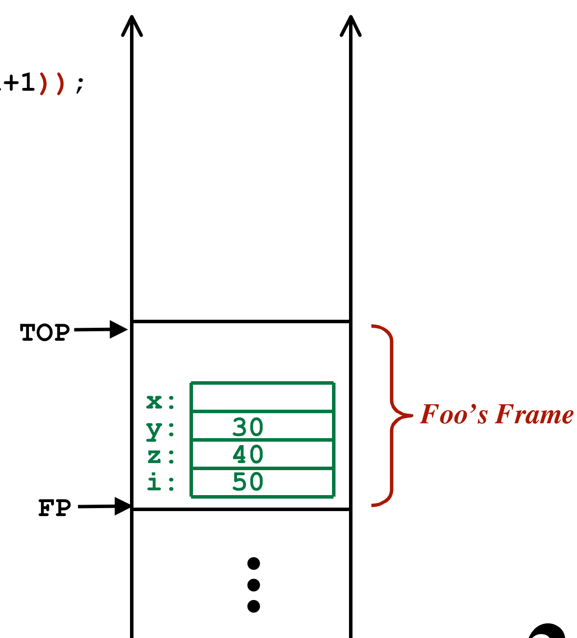
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



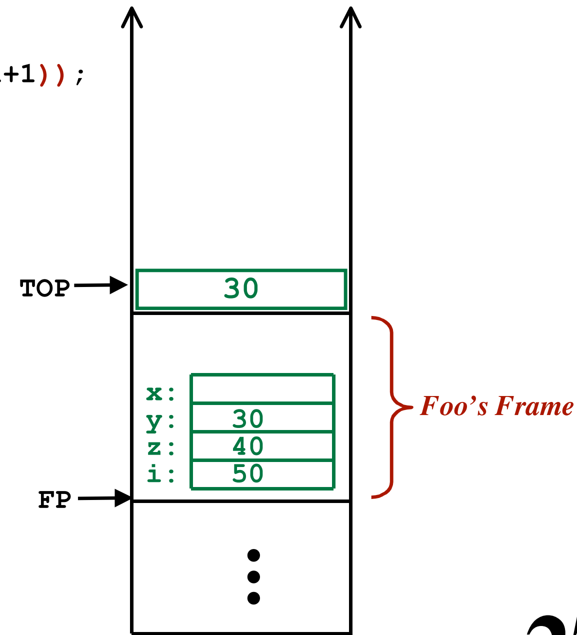
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



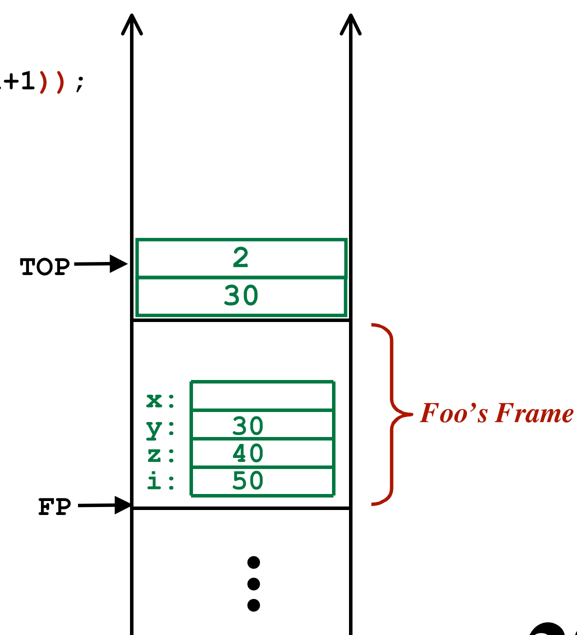
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



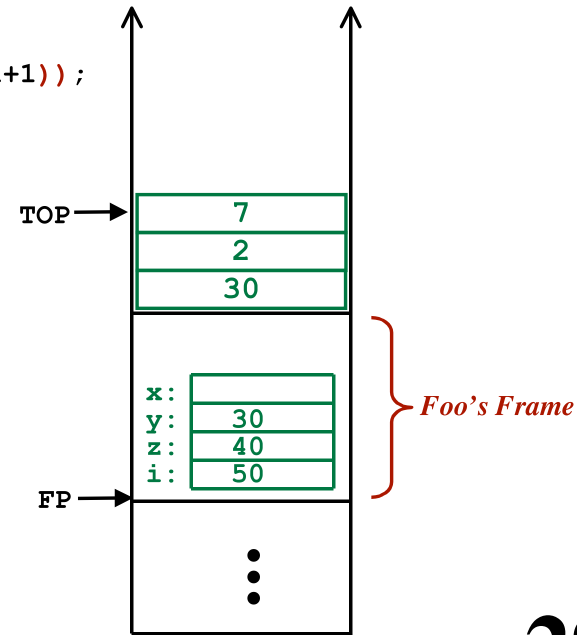
Using a Stack for
Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



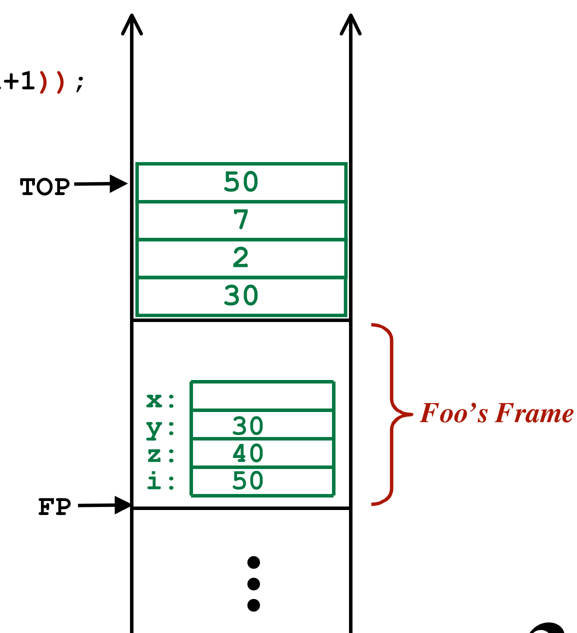
Using a Stack for
Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



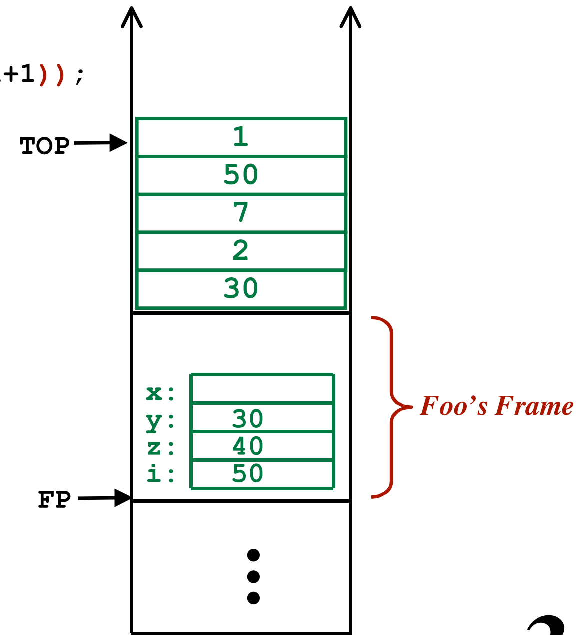
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



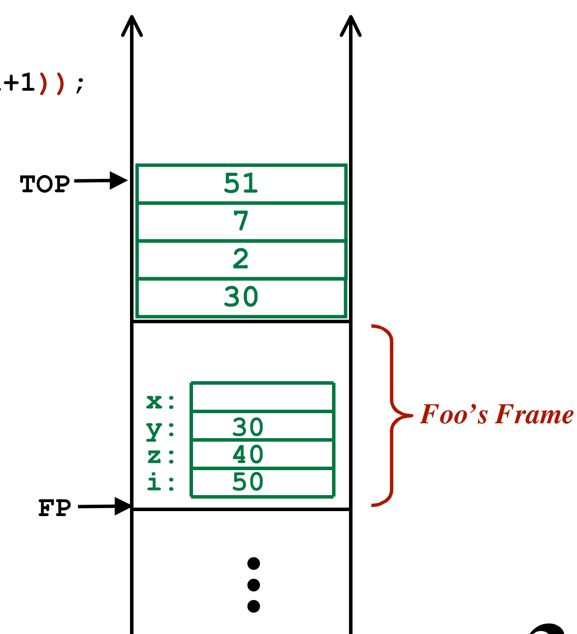
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



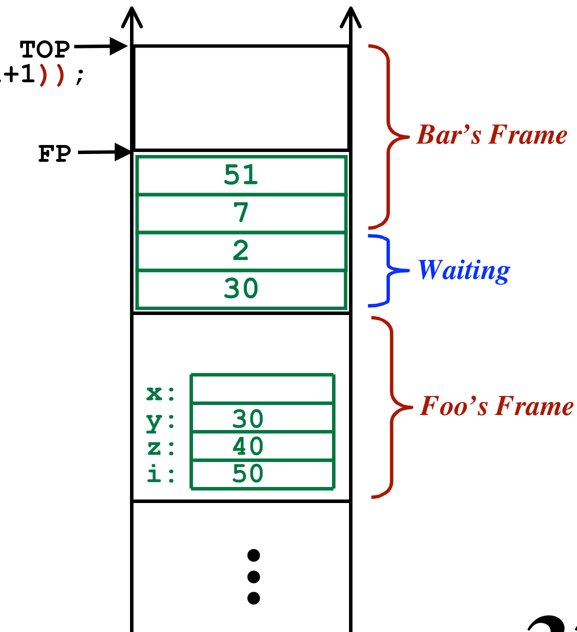
Using a Stack for
Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



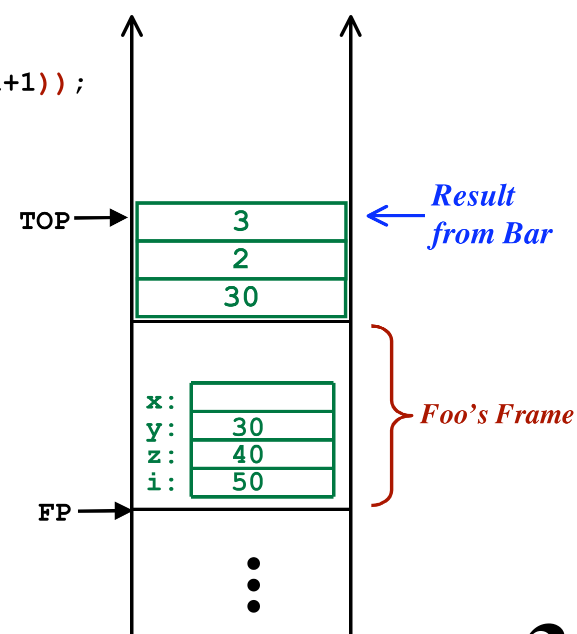
Using a Stack for
Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



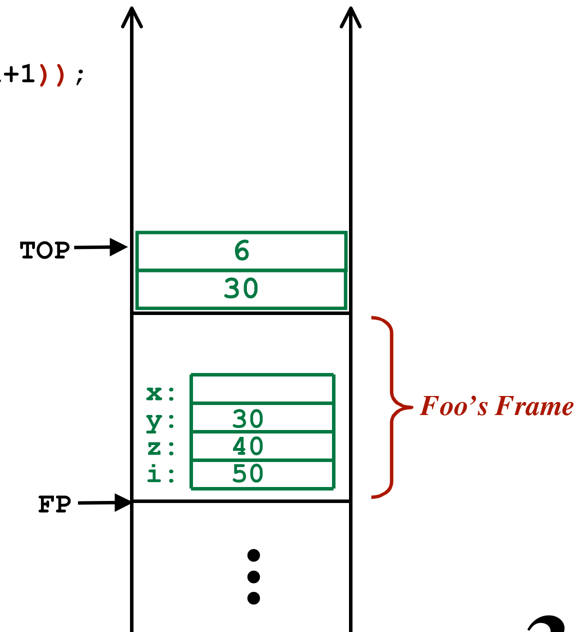
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



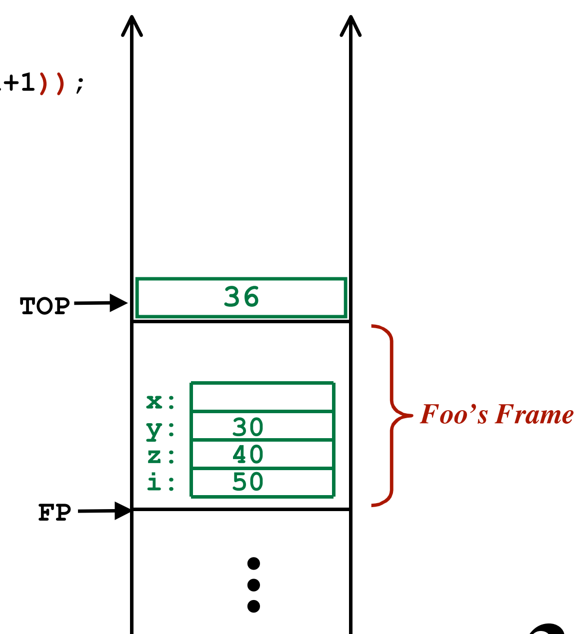
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



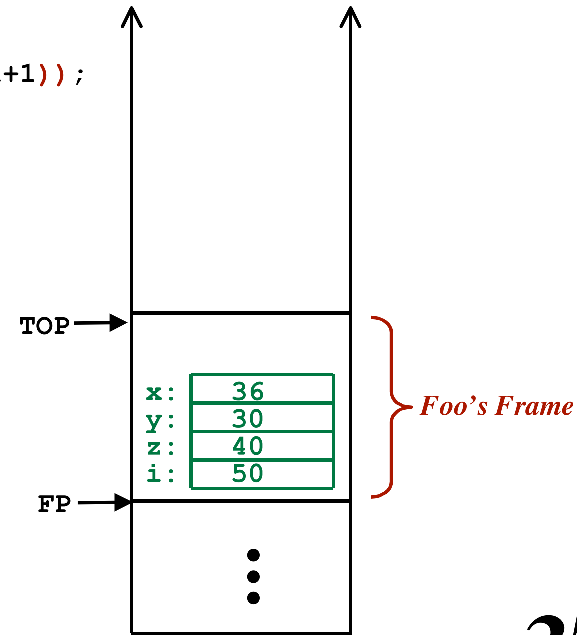
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:

```
x := y + 2 * (bar(7,i+1));
```

Target Code:

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```



Variable-Length Local Variables

Goal:

Allow a routine to have variable-length data
(i.e., dynamically-sized arrays) as local data in frame

Variable-Length Local Variables

Goal:

Allow a routine to have variable-length data
(i.e., dynamically-sized arrays) as local data in frame

Option 1:

Allocate the variable on the heap
Work with pointers to the data
PCAT: Hide the pointers from the programmer
Programmer codes:
a[i]
Compiler produces code like this:
*(a + 4*i)
Auto free the data when the routine returns?

Variable-Length Local Variables

Goal:

Allow a routine to have variable-length data
(i.e., dynamically-sized arrays) as local data in frame

Option 1:

Allocate the variable on the heap
Work with pointers to the data
PCAT: Hide the pointers from the programmer
Programmer codes:
a[i]
Compiler produces code like this:
*(a + 4*i)
Auto free the data when the routine returns?

Option 2:

Create the variable on the stack, dynamically
Effectively: Enlarge the frame as necessary
Still need to work with pointers

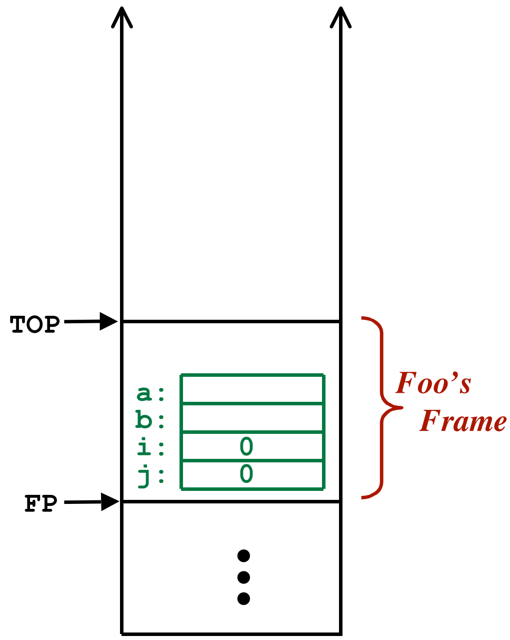
Variable-Length Local Variables

We must have two pointers:
  Stack Top
  FP

Local Variables
  at fixed offsets from FP

Dynamically sized variables
  use hidden pointers

All references to “a” and “b”
  will be indirect
    through hidden pointers



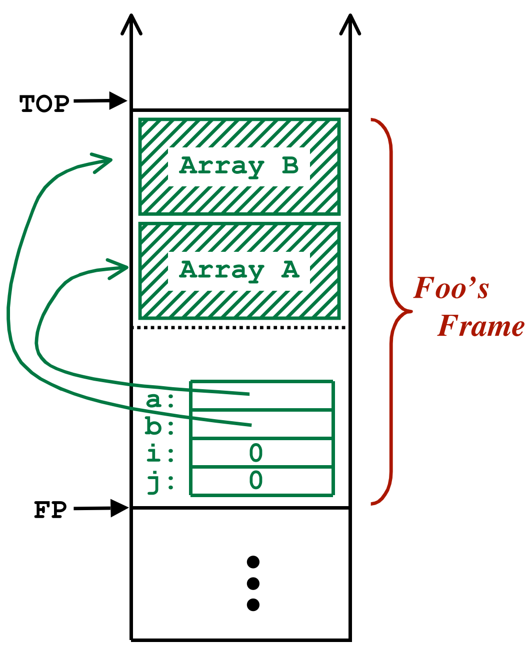
Variable-Length Local Variables

We must have two pointers:
  Stack Top
  FP

Local Variables
  at fixed offsets from FP

Dynamically sized variables
  use hidden pointers

All references to “a” and “b”
  will be indirect
    through hidden pointers



Local / Non-Local Variables

```
procedure main() {
  int y;
  procedure foo1() {
    int x;
    procedure foo2() {
      ...x...
      call foo3();
      ...y...
    }
    procedure foo3() {
      int x;

      ...x...
      call foo1 / call foo2
    }
    call foo1 / call foo2
    ...x...
  }
  call foo1
  ...y...
}
```


Local / Non-Local Variables

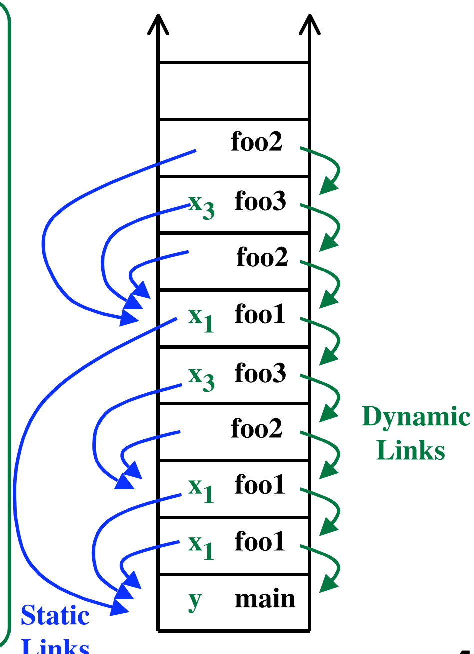
```
procedure main() {                          0
  int y;
  procedure foo1() {                      1
    int x1;
    procedure foo2() {                  2
      ...x...
      call foo3();
      ...y...
    }
    procedure foo3() {                  2
      int x3;

      ...x...
      call foo1 / call foo2
    }
    call foo1 / call foo2
    ...x...
  }
  call foo1
  ...y...
}
```

Local / Non-Local Variables

```
procedure main() {                              0
  int y;
  procedure foo1() {                            1
    int x1;
    procedure foo2() {                          2
      ..x..        Non-Local (1 level)
      call foo3();
      ..y..
    }              Non-Local (2 levels)
    procedure foo3() {                          2
      int x3;
                   Local (0 levels)
      ..x..
      call foo1 / call foo2
    }
    call foo1 / call foo2
    ...x...
  }
  call foo1
  ...y...
}
```


Local / Non-Local Variables

```
procedure main() {                              0
  int y;
  procedure foo1() {                            1
    int x1;
    procedure foo2() {                          2
      ..x..        Non-Local (1 level)
      call foo3();
      ..y..
    }              Non-Local (2 levels)
    procedure foo3() {                          2
      int x3;
                   Local (0 levels)
      ..x..
      call foo1 / call foo2
    }
    call foo1 / call foo2
    ...x...
  }
  call foo1
  ...y...
}
```



Static Scoping Rules

"Lexical Scoping"

The "norm" for most languages

For non-local variables...
- Look in syntactically enclosing routine
- Look in next enclosing routine
- ...etc...

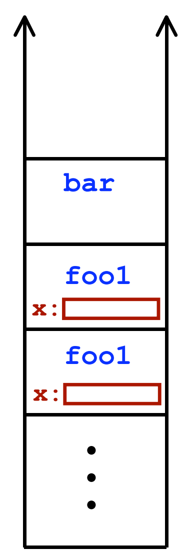
```
procedure foo2
  var x: integer
  procedure foo3
    var ...
    procedure foo4
      var ...

          ...x...
```


Dynamic Scoping Rules

For non-local variables...
- Search the calling stack at runtime to locate the right variable.

Uncommon.

```
procedure bar ()
  begin
    ...x...
  end
procedure foo1 ()
  var x: integer
  begin
    ...call bar()...
  end
procedure foo2 ()
  var x: integer
  begin
    ...call bar()...
  end
```



Statement Blocks

Syntax:
```
begin ... end;
{ ... }
```

```
procedure foo () is
  var x,y: ...;
    ...
    begin
      var temp: ...;
        temp := x;
        x := y;
        y := temp;
    end;
    ...
    begin
      var y,z: ...;
        ...
        begin
          ...
        end;
        ...
    end;
    ...
endProcedure;
```


Statement Blocks

Blocks are entered and exited in nested order.

Idea:

Create a new frame for each block.
Push on stack.

Statement Blocks

Blocks are entered and exited in nested order.

Idea:

Create a new frame for each block.
Push on stack.

But:

No parameters
No recursion
All calls are inline
⇒ Overhead!

So:

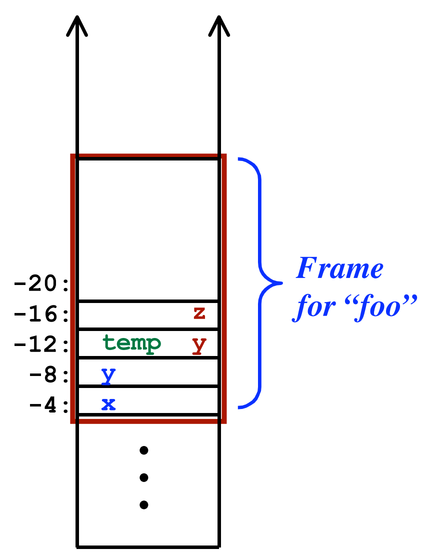
Just put variables in frame of surrounding routine!

Statement Blocks

Syntax:

```
begin ... end;
  { ... }
```

```
procedure foo () is
  var x,y: ...;
    ...
    begin
      var temp: ...;
        temp := x;
        x := y;
        y := temp;
    end;
    ...
    begin
      var y,z: ...;
        ...
        begin
          ...
        end;
        ...
    end;
    ...
endProcedure;
```



Function Contexts

Consider a language with:
- Functions as objects
- Non-local variable accesses

```
procedure main()
    var p: function
    ...
    p = ...
    ...
    call p()
    ...
endProcedure
```


Function Contexts

Consider a language with:
- Functions as objects
- Non-local variable accesses

- **Bar is called**
  - **Bar's frame is created**
    - **"x" is created**
- **Bar sets "p" to point to function "foo"**
- **Bar returns**
  - **Bar's frame is popped**
    - **"x" is destroyed**
- **foo is invoked**
  - **foo accesses variable "x"**

```
procedure main()
    var p: function
    procedure bar()
        var x: ...
        procedure foo()
            ... x ...
        endProcedure
        ...
        p = foo
        ...
    endProcedure
    ...
    call bar()
    ...
    call p()
    ...
endProcedure
```

Function Contexts

Consider a language with:
- Functions as objects
- Non-local variable accesses

- Bar is called
  - Bar's frame is created
    - "x" is created
- Bar sets "p" to point to function "foo"
- Bar returns
  - Bar's frame is popped
    - "x" is destroyed
- foo is invoked
  - foo accesses variable "x"

Solution:
- Do not free bar's frame
  - ... until it is no longer needed
- Put bar's frame on the heap
  - Automatic garbage collection

```
procedure main()
    var p: function
    procedure bar()
        var x: ...
        procedure foo()
            ... x ...
        endProcedure
        ...
        p = foo
        ...
    endProcedure
    ...
    call bar()
    ...
    call p()
    ...
endProcedure
```


The "C" Solution

"C" allows non-locals
to be used within a function

However...
- Functions may not be nested
- Variables are either
  - Local
  - Global (i.e., static)

```
static int x;

void foo() {
    ... x ...
}

void bar () {
    ...
    p = &foo;
}

void main () {
    ...
    bar();
    ...
    (*p) ();
    ...
}
```

The "Static" Link

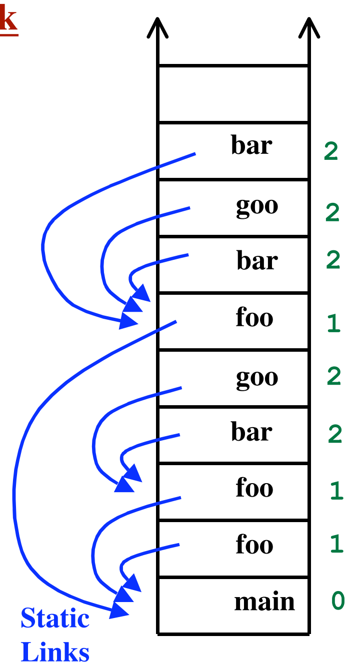
"Nesting Depth: A routine's lexical level...
    Main body ⇒ 0
    Nested routines ⇒ Add one
Each frame contains a "static" link
Points to the frame of
    the most recently invoked activation
        of the lexically surrounding routine.

```
procedure main() {                        0
  procedure foo() {                      1
    procedure bar() {                   2
       . . .
    }
    procedure goo() {                   2
       . . .
    }
    . . .
  }
  . . .
}
```


The "Static" Link

"Nesting Depth: A routine's lexical level...
    Main body ⇒ 0
    Nested routines ⇒ Add one
Each frame contains a "static" link
Points to the frame of
    the most recently invoked activation
        of the lexically surrounding routine.

```
procedure main() {                        0
  procedure foo() {                      1
    procedure bar() {                   2
       . . .
    }
    procedure goo() {                   2
       . . .
    }
    . . .
  }
  . . .
}
```

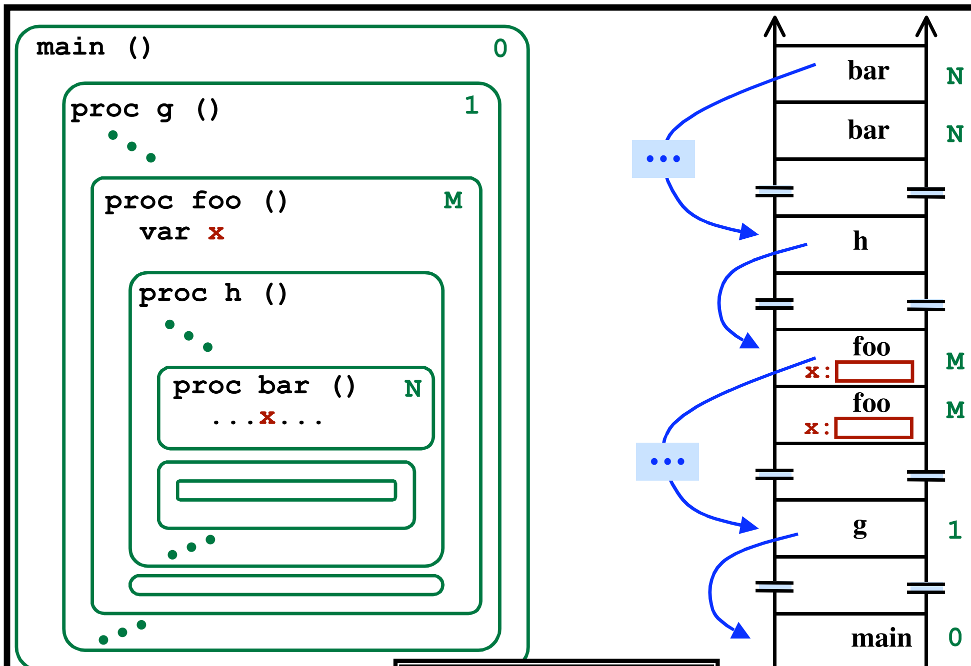


Given a variable usage...
How do we find the frame containing the right variable?

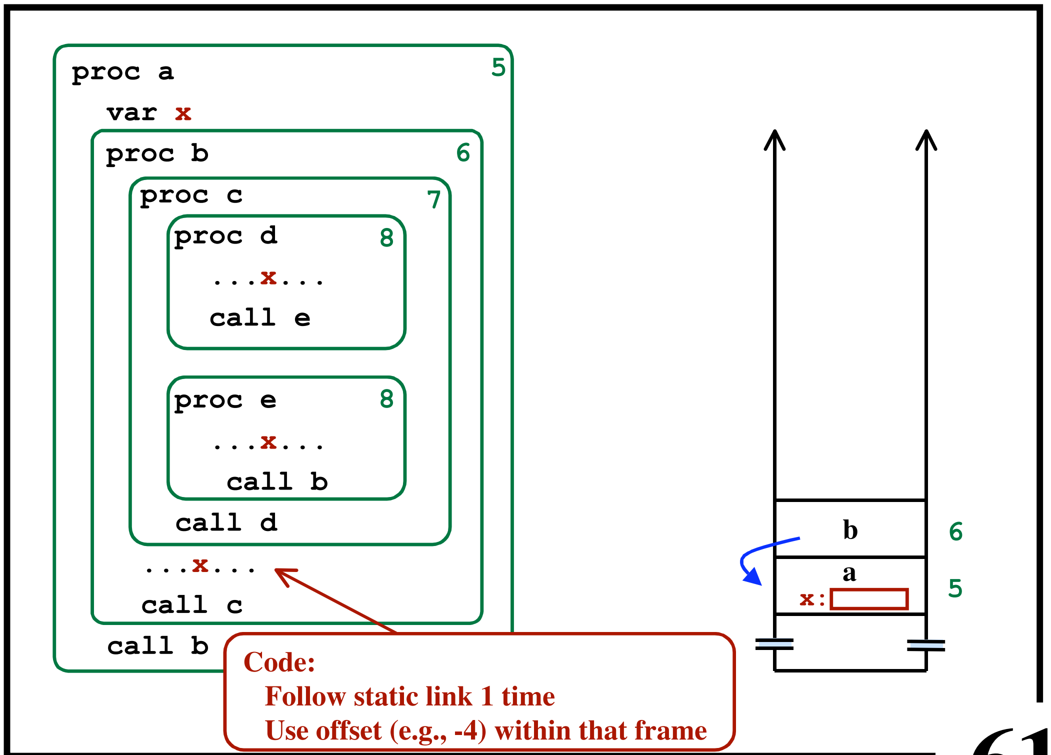
Assume that x is declared at lexical level M.

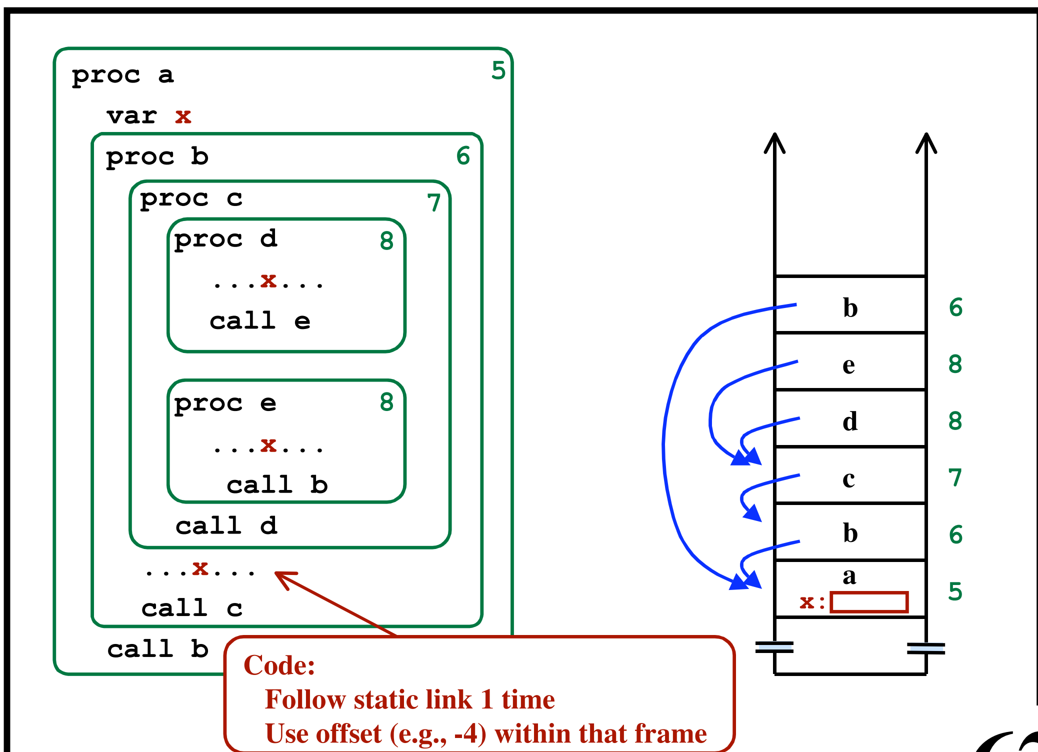
Assume that x is used at lexical level N.
(We must have N ≥ M)

At runtime...
Follow N-M static links to find the right frame.
Use the offset of x within that frame.



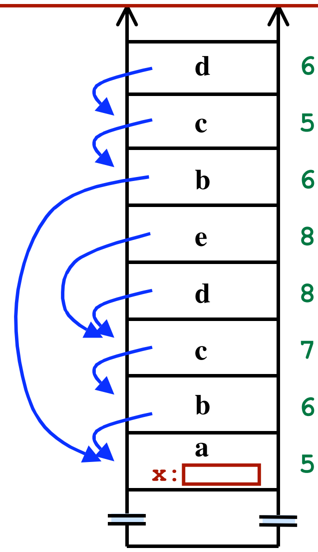
“x” declared at level M
“x” used at level N
Follow N-M static links


proc a
5
var x
proc b
6
proc c
7
proc d
8
...x...
call e
proc e
8
...x...
call b
call d
...x...
call c
call b
b
6
a
5
x:
Code:
Follow static link 1 time
Use offset (e.g., -4) within that frame


proc a
5
var x
proc b
6
proc c
7
proc d
8
...x...
call e
proc e
8
...x...
call b
call d
...x...
call c
call b
b
6
e
8
d
8
c
7
b
6
a
5
x:
Code:
Follow static link 1 time
Use offset (e.g., -4) within that frame

Code:
Follow static link 3 times
Use offset (e.g., -4) within that frame

```
proc a                              5
  var x
  proc b                            6
    proc c                          7
      proc d                        8
        ...x...
        call e
      proc e                        8
        ...x...
         call b
      call d
    ...x...
    call c
  call b
```



How do we set up the Static Links?

Whenever we call a routine...
    must initialize static link in new frame.

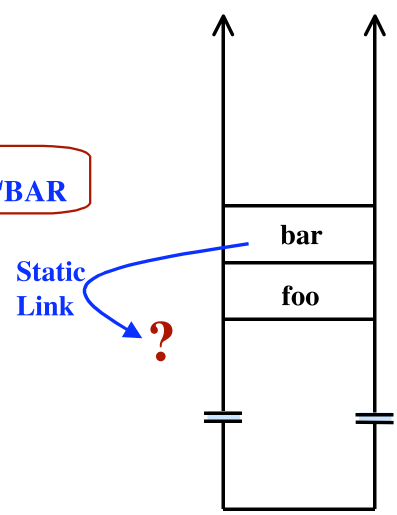
Assume "foo" calls "bar"

$Level = L_{BAR}$

$Level = L_{FOO}$

Want to initialize bar's static link.

What frame should it point to???



Initializing the Static Link

foo calls bar

Goal:

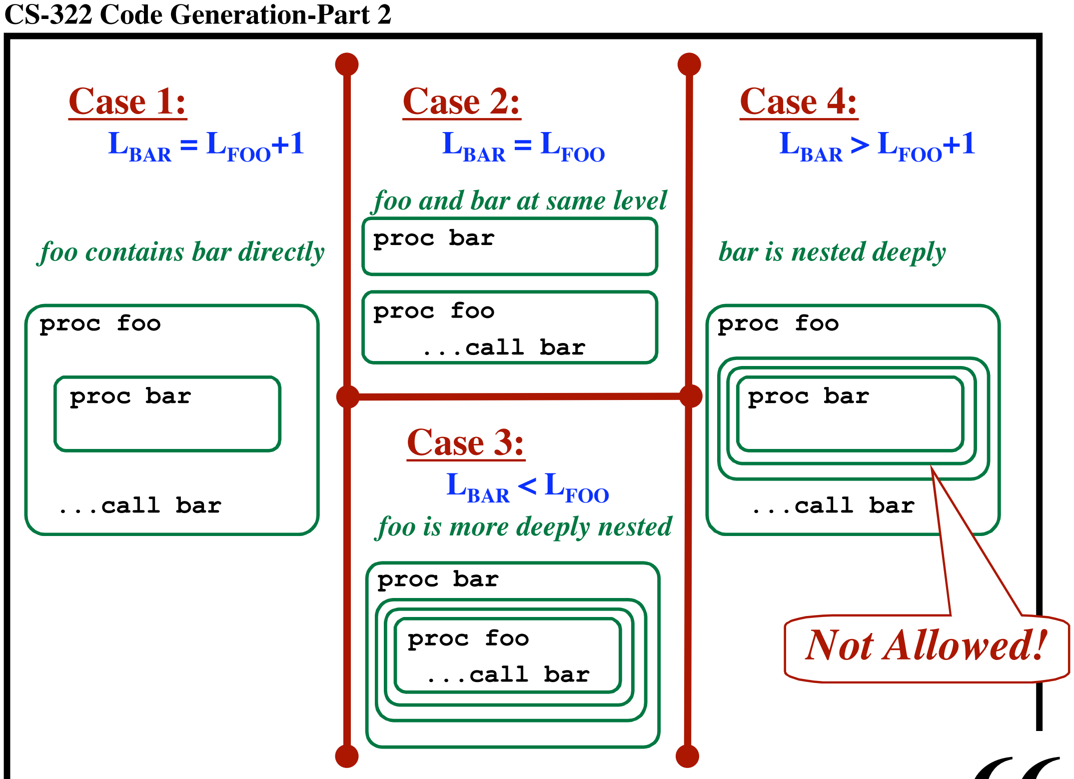
Find the frame of the routine that lexically encloses bar
Set bar's static link to point to it.

Given:

foo's frame is on the stack,
directly below the newly allocated frame for bar.

Approach:

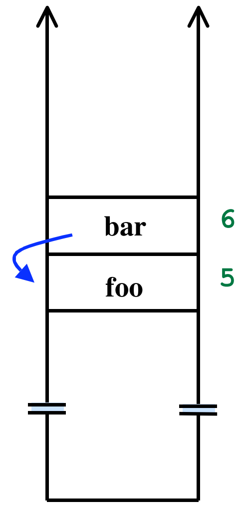
Use the static link in foo's frame.
Follow **L_{FOO} - L_{BAR} + 1** static links from foo's frame.
This will be the frame of the routine that lexically encloses bar!!!
Make bar's static link point to it.



Case 1: $L_{BAR} = L_{FOO}+1$

Foo statically contains bar directly.

```
proc foo                5
  proc bar          6

  ...call bar
```



$L_{FOO} - L_{BAR} + 1$

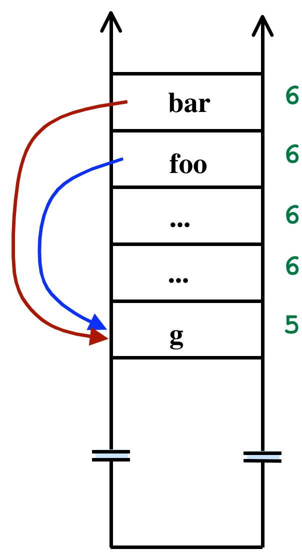
5 - 6 + 1 = 0

From foo's frame...
follow 0 links!
Just make bar's static link point to foo's frame.

Case 2: $L_{BAR} = L_{FOO}$

Foo and bar at same level.

```
proc g                  5
  proc bar          6
  proc foo          6
    ...call bar
```



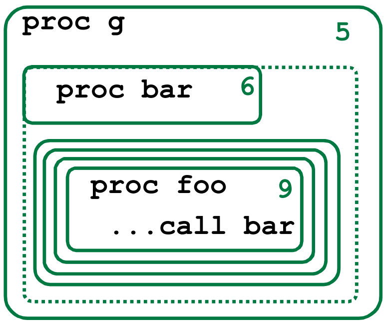
$L_{FOO} - L_{BAR} + 1$

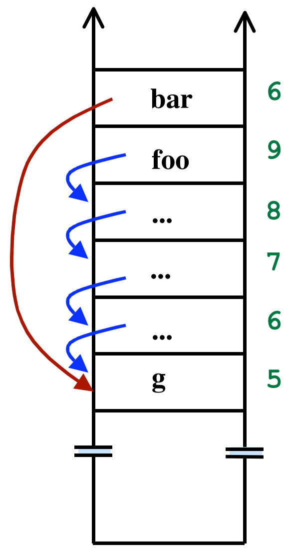
6 - 6 + 1 = 1

From foo's frame...
follow 1 link!

Case 3: $L_{BAR} < L_{FOO}$

foo is more deeply nested
(within bar or one of bar's siblings)





$L_{FOO} - L_{BAR} + 1$

9 - 6 + 1 = 4

From foo's frame
follow 4 links!

CS-322 Code Generation-Part 2

Display Registers

The Idea:

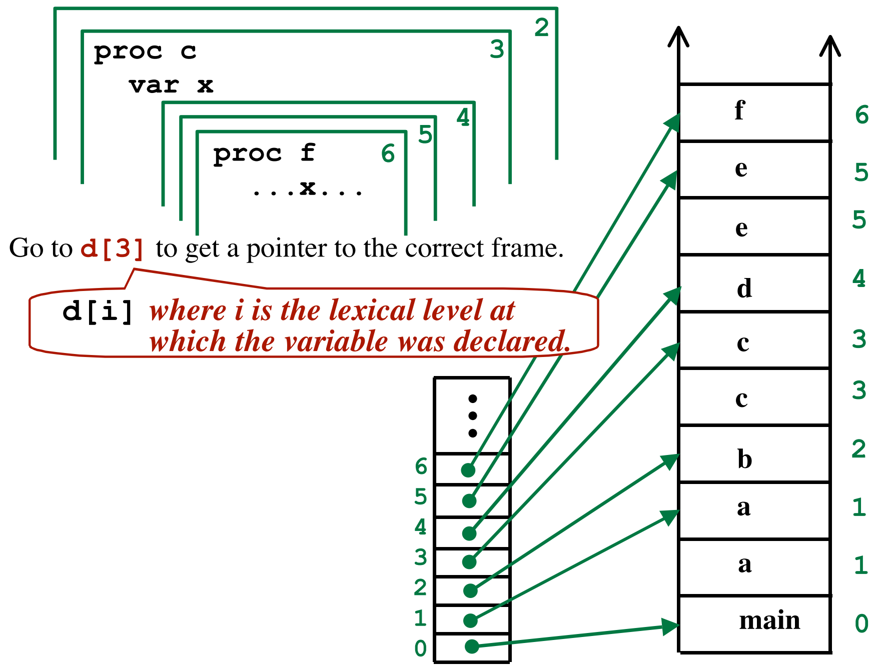
In static storage...
maintain an array of pointers
`d[...]`
The i-th element will be a pointer to an activation record on the stack...
...whose lexical level is i.

Assume we are currently executing in a routine ("f") at lexical level 6...

`d[6]` points to the top frame
(i.e., the currently executing frame, for "f")
`d[5]` points to the most recent activation of the routine
that lexically encloses "f".
(a routine at level 5, call it "g")
`d[4]` points to the most recent activation of the routine
that lexically encloses "g"
(a routine at level 4, call it "h")
⋮
`d[0]` points to the most recent activation of a routine
at level 0, call it "main"

© Harry H. Porter, 2006

To access a non-local variable...



How to Maintain the Display Registers?

During "call" and "return" sequences...

Each activation record will have a word in which to save
an old value of a display register.
"*display register save area*"

When calling a routine at lexical level "i"...

```
Allocate a new frame on the stack
Save old value of d[i] in that word in the new frame
d[i] := ptr to the new frame
```

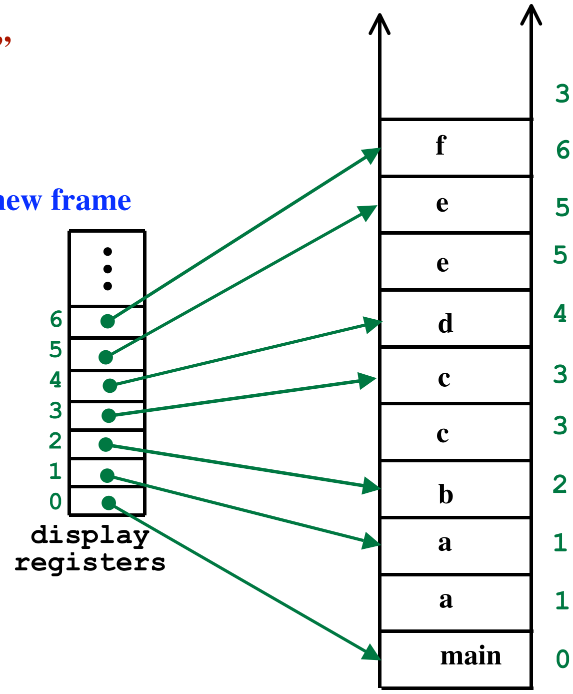
When returning...

```
d[i] = the saved value
```

Note: The entire array of display registers will always be restored to its previous value after any sequence of calls and matching returns!

Example: "f" calls "c"

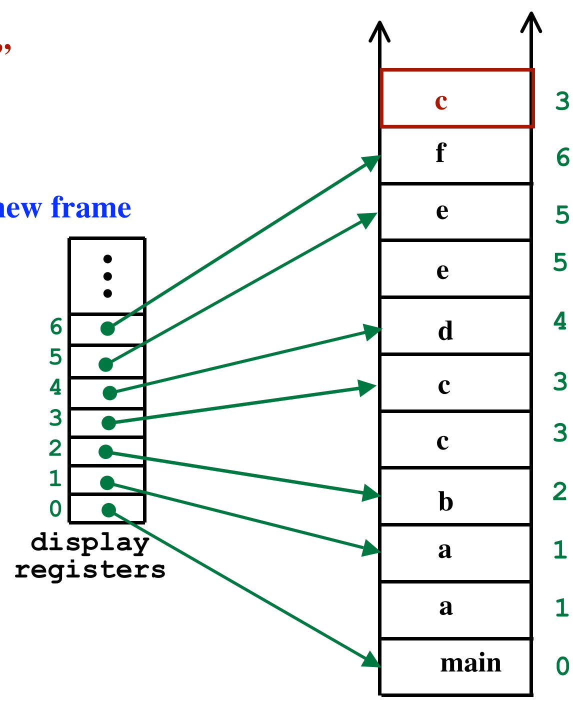
- Allocate new frame
- Save old d[3]
- Set d[3] to point to new frame



CS-322 Code Generation-Part 2

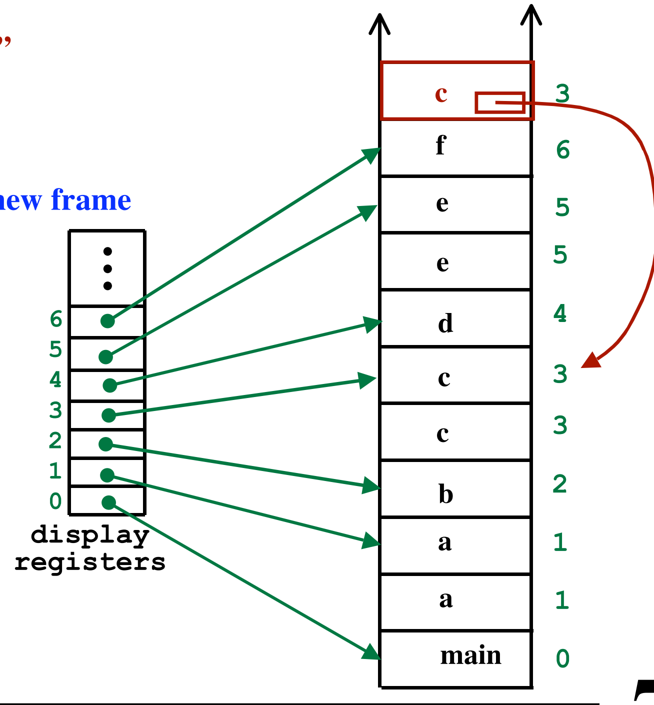
Example: "f" calls "c"

- Allocate new frame
- Save old d[3]
- Set d[3] to point to new frame



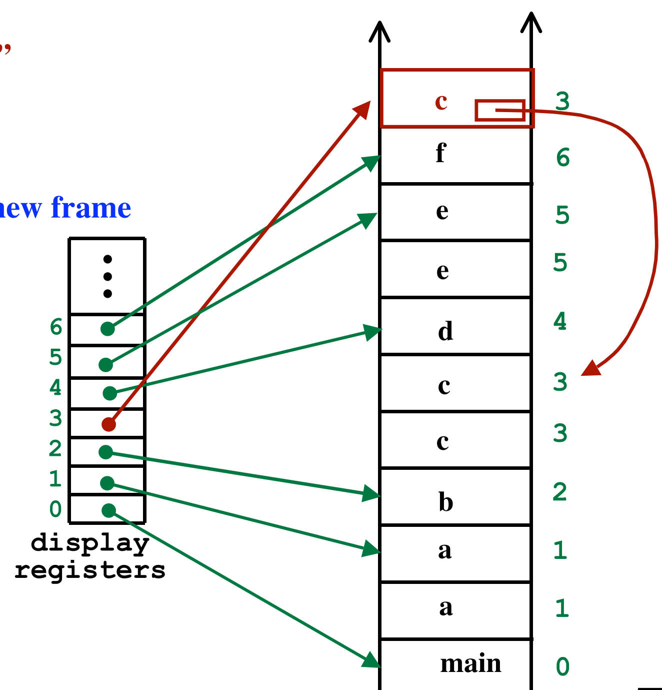
Example: "f" calls "c"

- Allocate new frame
- Save old d[3]
- Set d[3] to point to new frame

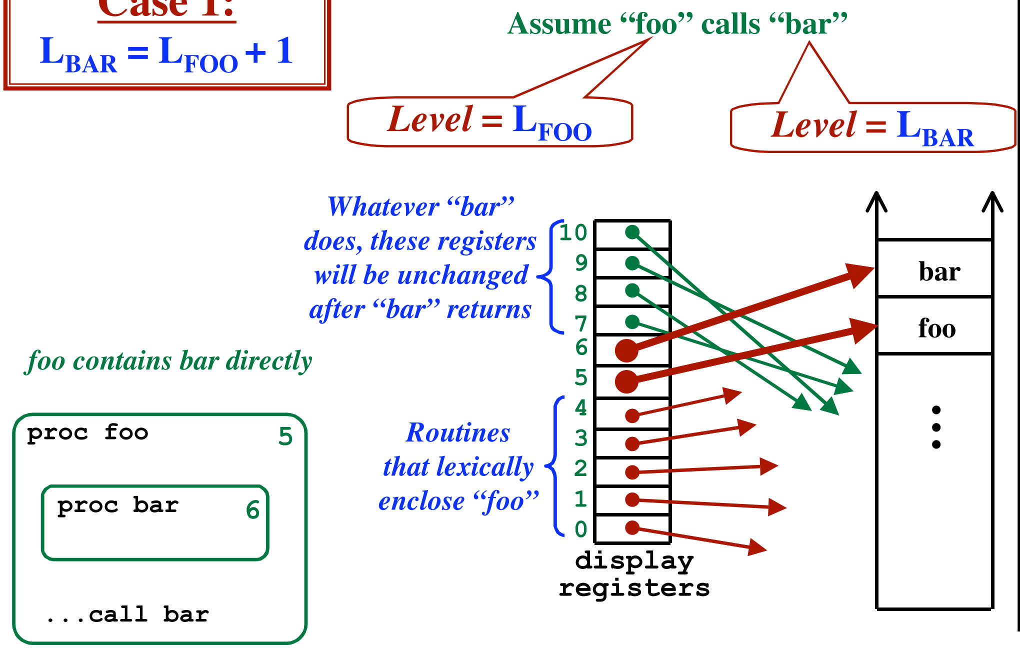


Example: "f" calls "c"

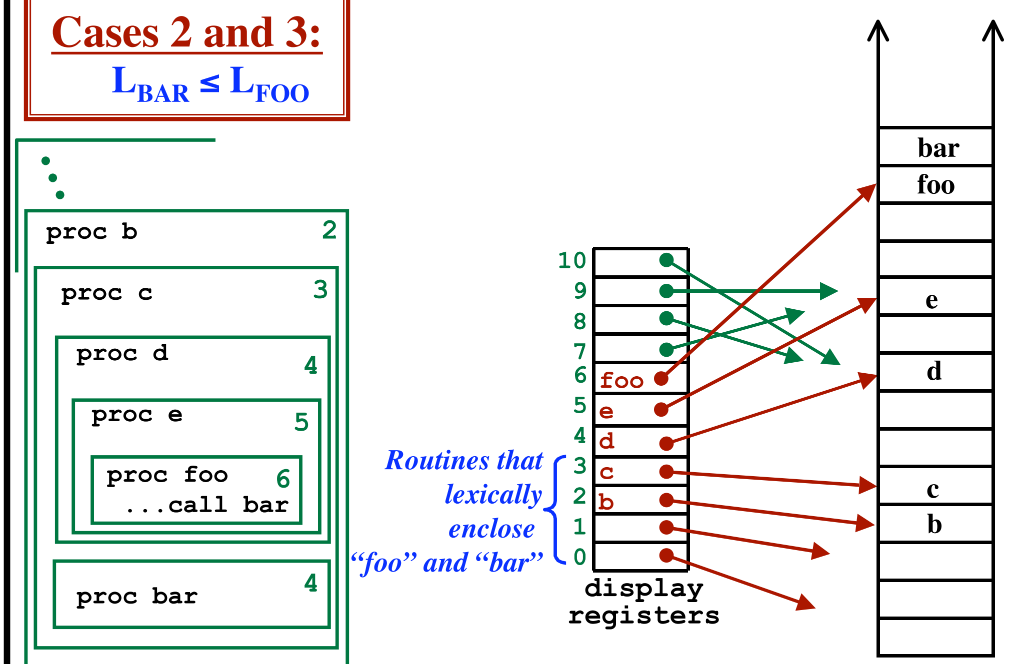
- Allocate new frame
- Save old d[3]
- Set d[3] to point to new frame



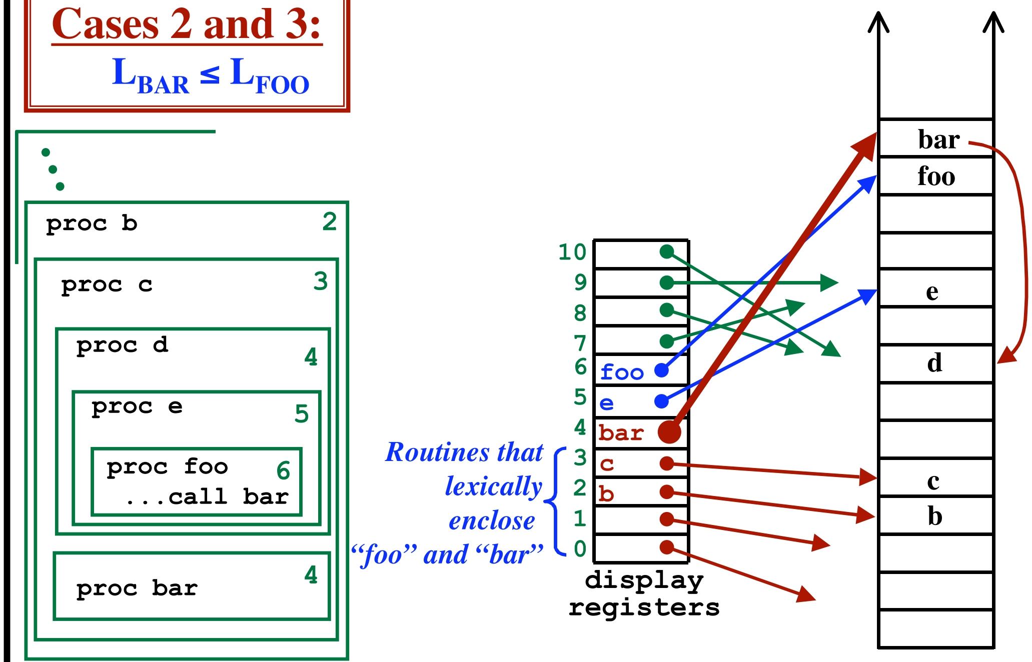
Case 1:
$L_{BAR} = L_{FOO} + 1$



Cases 2 and 3:
$L_{BAR} \leq L_{FOO}$



Cases 2 and 3:
$L_{BAR} \leq L_{FOO}$



Producing Target Code

```
            .data
display0:   .word    0
display1:   .word    0
display2:   .word    0
  ...
display10:  .word    0
```

At the beginning of the the program

We'll need to compute the <u>maximum</u> lexical level!

```
bar:        save     %fp,???,%fp
            set      display4,%l1
            ld       [%l1],%l0
            st       %l0,[%sp+64]
            st       %fp,[%l1]
```

procEntry bar,lexLevel=4

```
            ld       [%sp+64],%l0
            set      display4,%l1
            st       %l0,[%l1]
            ret
            restore
```

returnVoid