Lecture #16: Runtime Support for Functions (contd)

4: Allow Nesting of Functions, Up-Level Addressing

- When functions can be nested, there are three classes of variable:

  a. Local to function.

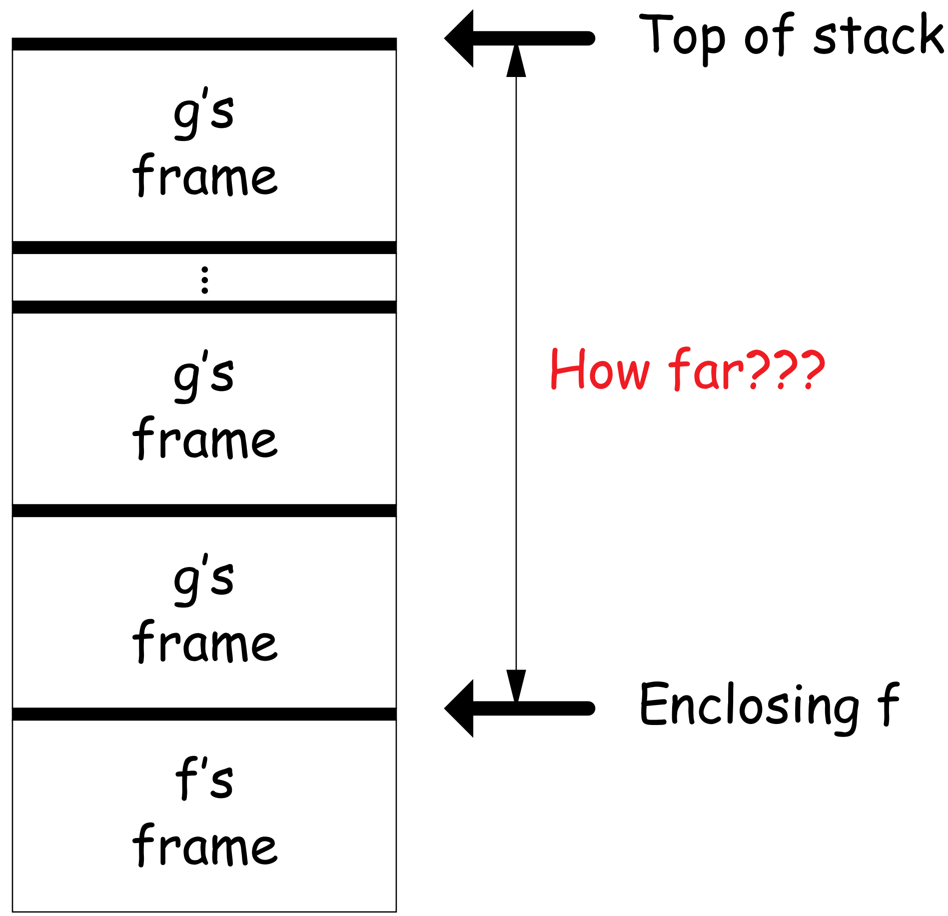
  b. Local to enclosing function.

  c. Global

- Accessing (a) or (c) is easy. It's (b) that's interesting.
- Consider (in Python):

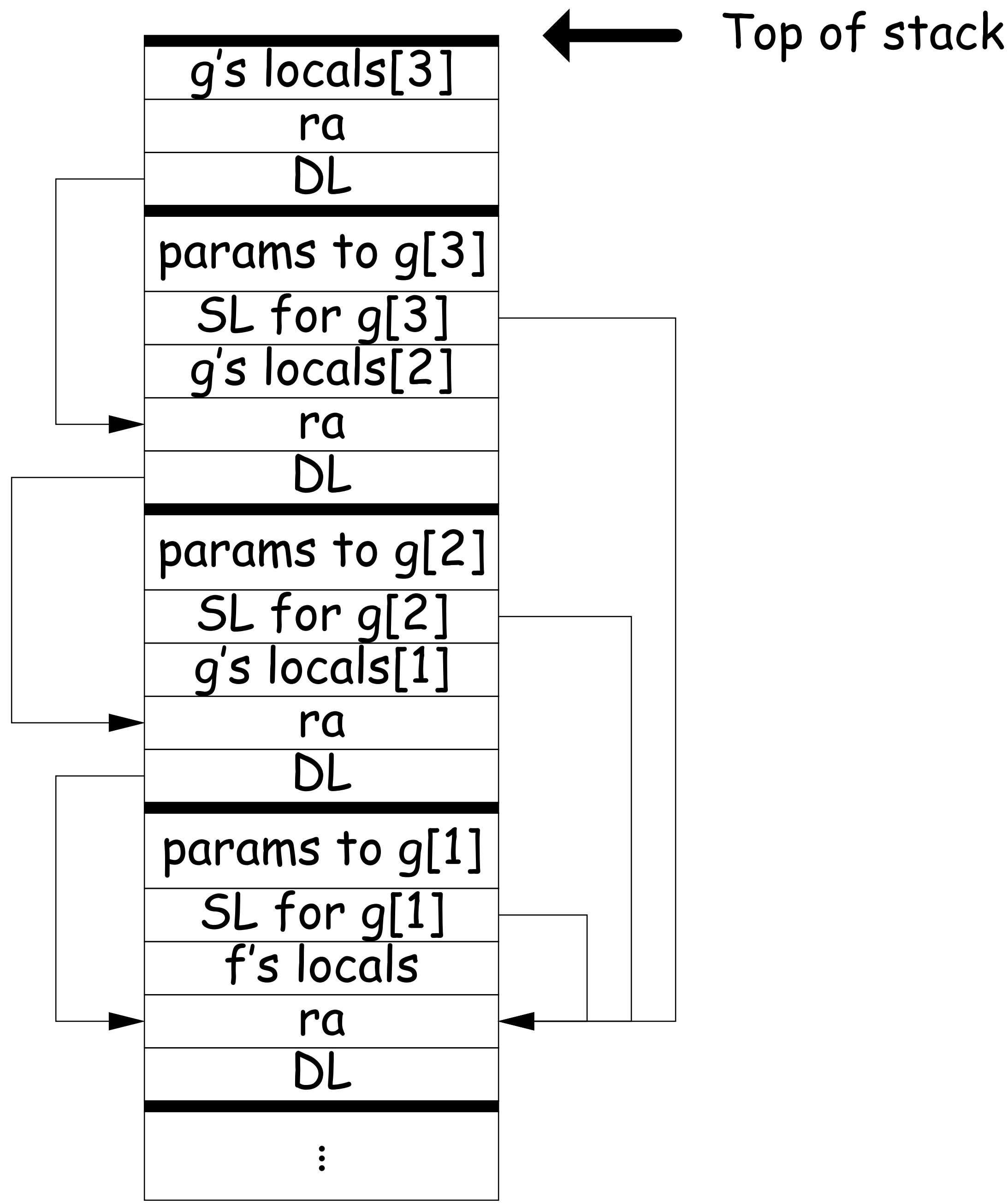
```
def f ():
  y = 42   # Local to f
  def g (n, q):
     if n == 0: return q+y
     else: return g (n-1, q*2)
```

- Here, y can be any distance away from top of stack.



Static Links

- To overcome this problem, go back to environment diagrams!
- Each diagram had a pointer to *lexically enclosing environment*
- In Python example from last slide, each 'g' frame contains a pointer to the 'f' frame where that 'g' was defined: the *static link* (SL)
- To access local variable, use frame-base pointer (or maybe stack pointer).
- To access global, use absolute address.
- To access local of nesting function, follow static link once per difference in levels of nesting.



Calling sequence for RISC V: f0

C code:

```
int
f0 (int n0)
{
  int s = -n0;
  int g1 () { return s; }
  int f1 (int n1) {
    int f2() {
      return n0 + s
             + n1 + g1();
    }
    return n0 + f2();
  }
  return f1(10);
}
```

Assembly excerpt for f0:

```
f0:
    sw fp, 0(sp)       # Save old frame pointer
    sw ra, -4(sp)      # Save return address
    addi sp, sp, -12   # Adjust SP to leave room for s, ra, DL
    addi fp, sp, 8     # FP now points to ra.
    lw t0, 8(fp)       # n0
    sub t0, zero, t0   # -n0
    sw t0, -4(fp)      # Set s
    sw fp, 0(sp)       # SL to f1
    li t0, 10          # argument to f1
    sw t0, -4(sp)
    addi sp, sp, -8    # Adjust for arguments
    jal f1
    addi sp, sp, 8
    addi sp, fp, 4
    lw ra, 0(fp)
    lw fp, 4(fp)
    jr ra
```

Calling sequence for RISC V: f1

C code:

```
int
f0 (int n0)
{
  int s = -n0;
  int g1 () { return s; }
  int f1 (int n1) {
    int f2() {
      return n0 + s
             + n1 + g1();
    }
    return n0 + f2();
  }
  return f1(10);
}
```

```
f1:
      sw fp, 0(sp)        # Save old frame pointer
      sw ra, -4(sp)       # Save return address
      addi sp, sp, -8     # Adjust SP to leave room for ra, DL
      addi fp, sp, 4      # FP now points to ra.
      lw t0, 12(fp)       # Load my static link (to f0)
      lw t2, 8(t0)        # n0
      sw t2, 0(sp)        # Save it for now.
      sw fp, -4(sp)       # Push f2's static link (my fp)
      addi sp, sp, -8     # Adjust sp
      jal f2
      addi sp, sp, 8
      lw t0, 0(sp)        # Saved n0 from before call
      add a0, t0, a0      # n0 + f2()
      addi sp, fp, 4      # Restore sp
      lw ra, 0(fp)        # Restore ra
      lw fp, 4(fp)        # Restore fp
      jr ra
```

Calling sequence for RISC V: f2

C code:

```
int
f0 (int n0)
{
  int s = -n0;
  int g1 () { return s; }
  int f1 (int n1) {
    int f2() {
      return n0 + s
             + n1 + g1();
    }
    return n0 + f2();
  }
  return f1(10);
}
```

```
f2:
      sw fp, 0(sp)        # Save old frame pointer
      sw ra, -4(sp)       # Save return address
      addi sp, sp, -8     # Adjust SP to leave room for ra, DL
      addi fp, sp, 4      # FP now points to ra.
      lw t0, 8(fp)        # Load my static link (to f1)
      lw t1, 12(t0)       # Load f1's static link (to f0)
      lw t2, 8(t1)        # n0
      lw t3, -4(t1)       # s
      add t2, t2, t3      # n0 + s
      lw t3, 8(t0)        # n1
      add t2, t2, t3      # n0 + s + n1
      sw t2, 0(sp)        # Save
      sw t1, -4(sp)       # SL for g1 (to f0, same as f1)
      addi sp, sp, -8     # Adjust stack
      jal g1
      addi sp, sp, 8
      lw t0, 0(sp)        # Saved n0 + s + n1
      add a0, t0, a0      # n0 + s + n1 + g1()
      addi sp, fp, 4      # Restore sp
      lw ra, 0(fp)        # Restore ra
      lw fp, 4(fp)        # Restore fp
      jr ra
```

Calling sequence for the ia32: g1

C code:

```
int
f0 (int n0)
{
  int s = -n0;
  int g1 () { return s; }
  int f1 (int n1) {
    int f2 () {
      return n0 + n1
             + s + g1 ();
    }
    return n0 + f2();
  }
  f1 (10);
}
```

Assembly g1:

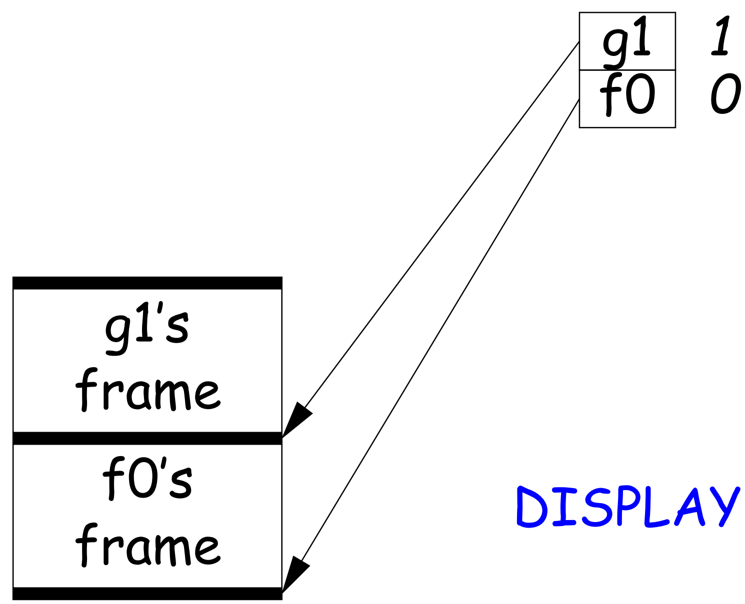
```
g1:  # Leaf procedure (optimized).
      lw t0, 4(sp)      # Load my static link (to f0)
      lw a0, -4(t0)     # s
      jr ra
```

The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```
def f0 ():
  q = 42; g1 ()
  def f1 ():
    def f2 (): ... g2 () ...
    def g2 (): ... g2 () ... g1 () ...
    ... f2 () ... f1 () ...
  def g1 (): ... f1 () ...
```

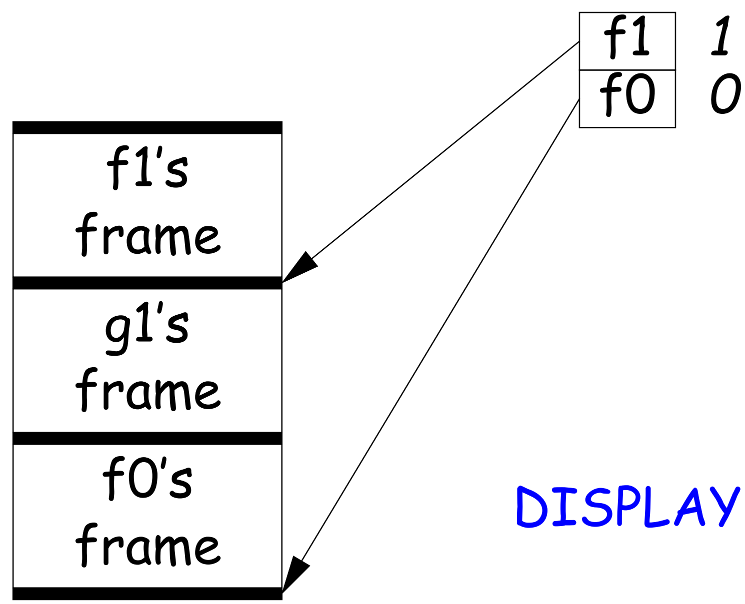
- Each time we enter a function at lexical level k (i.e., nested inside k functions), save pointer to its frame base in DISPLAY[k]; restore on exit.
- Access variable at lexical level k through DISPLAY[k].
- Relies heavily on scope rules and proper function-call nesting



The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```
def f0 ():
  q = 42; g1 ()
  def f1 ():
    def f2 (): ... g2 () ...
    def g2 (): ... g2 () ... g1 () ...
    ... f2 () ... f1 () ...
  def g1 (): ... f1 () ...
```



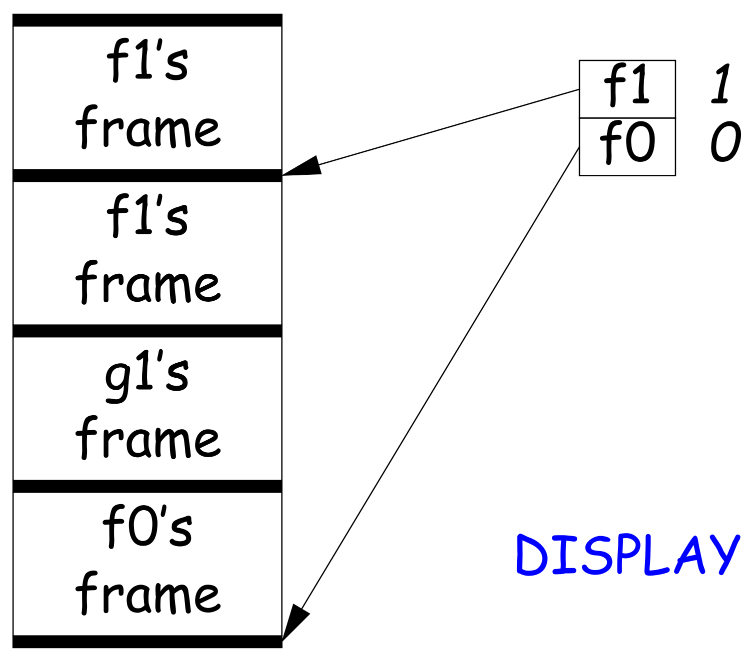
- Each time we enter a function at lexical level k (i.e., nested inside k functions), save pointer to its frame base in DISPLAY[k]; restore on exit.
- Access variable at lexical level k through DISPLAY[k].
- Relies heavily on scope rules and proper function-call nesting

The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```
def f0 ():
  q = 42; g1 ()
  def f1 ():
    def f2 (): ... g2 () ...
    def g2 (): ... g2 () ... g1 () ...
    ... f2 () ... f1 () ...
  def g1 (): ... f1 () ...
```

- Each time we enter a function at lexical level k (i.e., nested inside k functions), save pointer to its frame base in DISPLAY[k]; restore on exit.
- Access variable at lexical level k through DISPLAY[k].
- Relies heavily on scope rules and proper function-call nesting

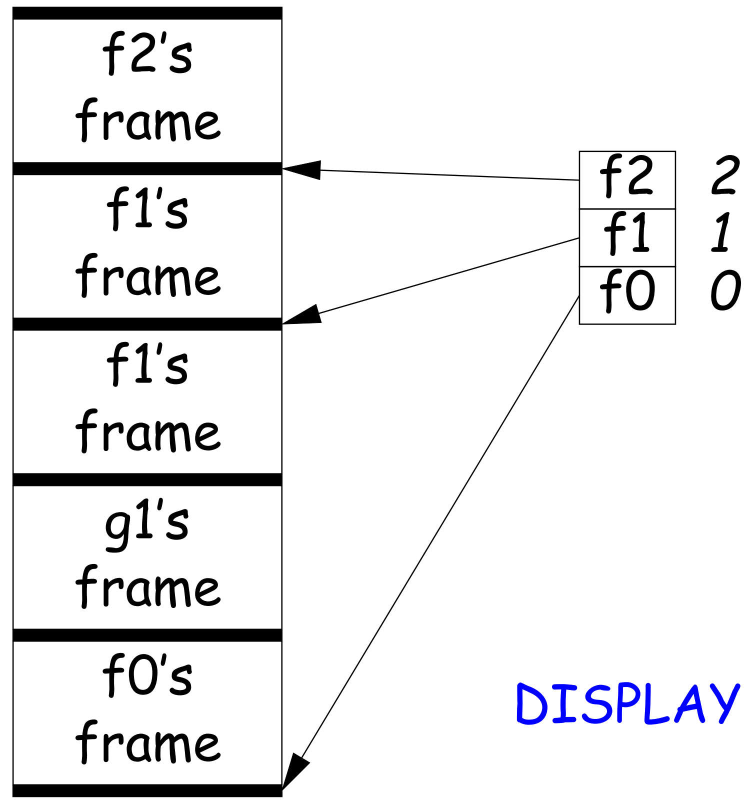


The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```
def f0 ():
  q = 42; g1 ()
  def f1 ():
    def f2 (): ... g2 () ...
    def g2 (): ... g2 () ... g1 () ...
    ... f2 () ... f1 () ...
  def g1 (): ... f1 () ...
```

- Each time we enter a function at lexical level k (i.e., nested inside k functions), save pointer to its frame base in DISPLAY[k]; restore on exit.
- Access variable at lexical level k through DISPLAY[k].
- Relies heavily on scope rules and proper function-call nesting

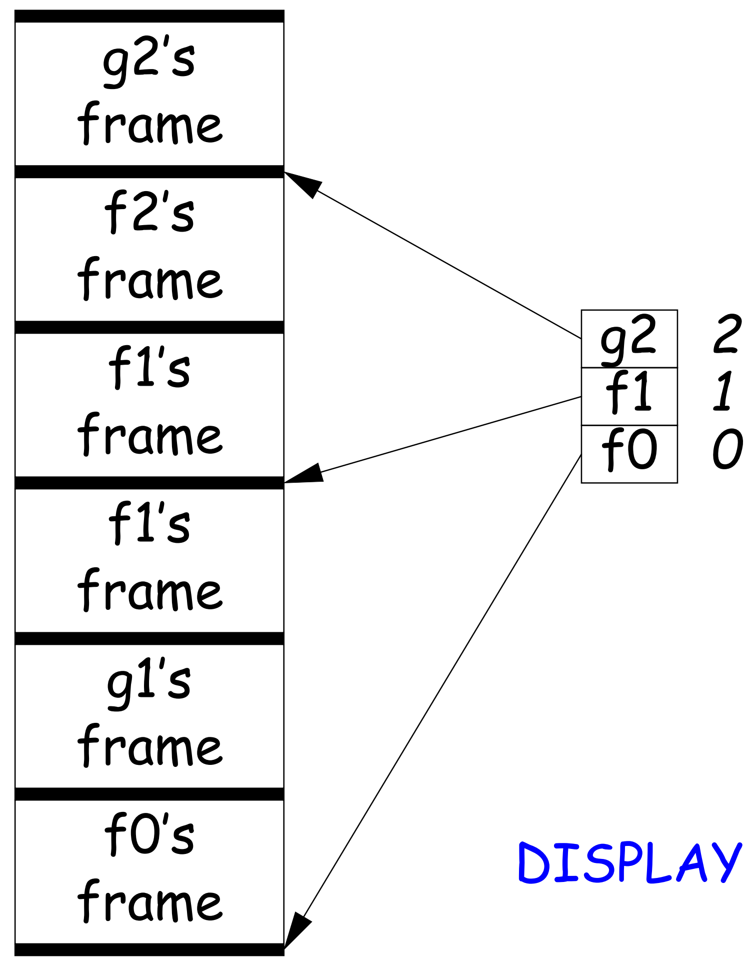


The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```
def f0 ():
    q = 42; g1 ()
    def f1 ():
        def f2 (): ... g2 () ...
        def g2 (): ... g2 () ... g1 () ...
        ... f2 () ... f1 () ...
    def g1 (): ... f1 () ...
```

- Each time we enter a function at lexical level k (i.e., nested inside k functions), save pointer to its frame base in DISPLAY[k]; restore on exit.
- Access variable at lexical level k through DISPLAY[k].
- Relies heavily on scope rules and proper function-call nesting

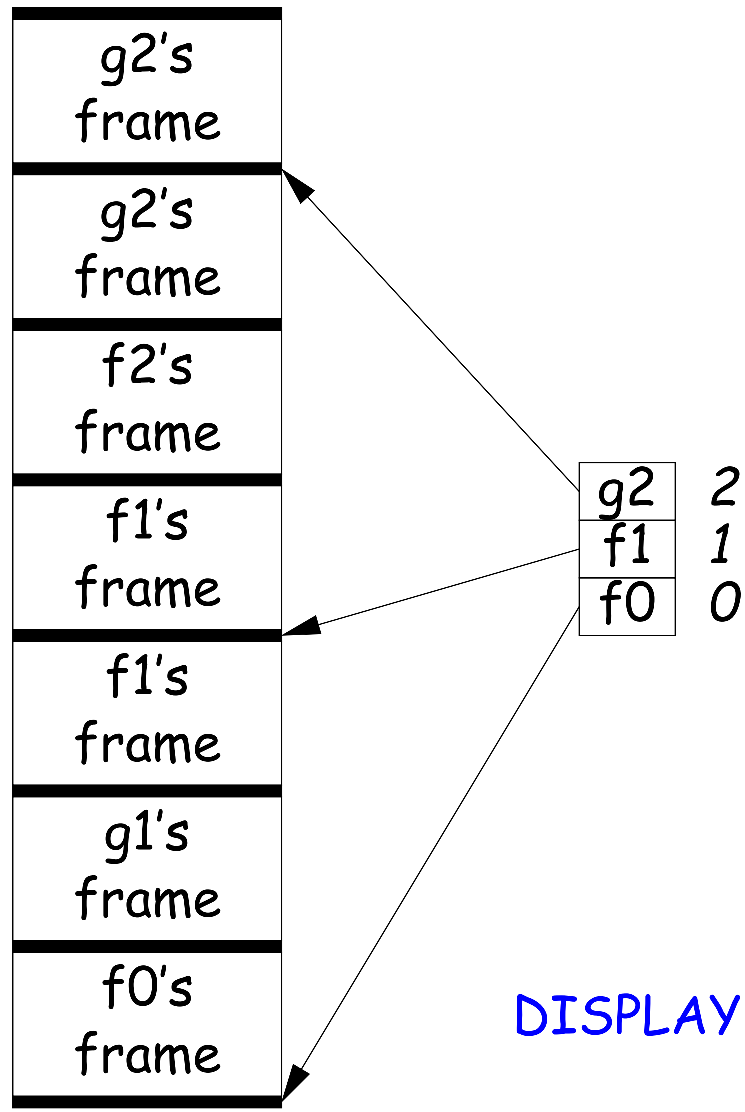


The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```
def f0 ():
  q = 42; g1 ()
  def f1 ():
    def f2 (): ... g2 () ...
    def g2 (): ... g2 () ... g1 () ...
    ... f2 () ... f1 () ...
  def g1 (): ... f1 () ...
```

- Each time we enter a function at lexical level k (i.e., nested inside k functions), save pointer to its frame base in DISPLAY[k]; restore on exit.
- Access variable at lexical level k through DISPLAY[k].
- Relies heavily on scope rules and proper function-call nesting

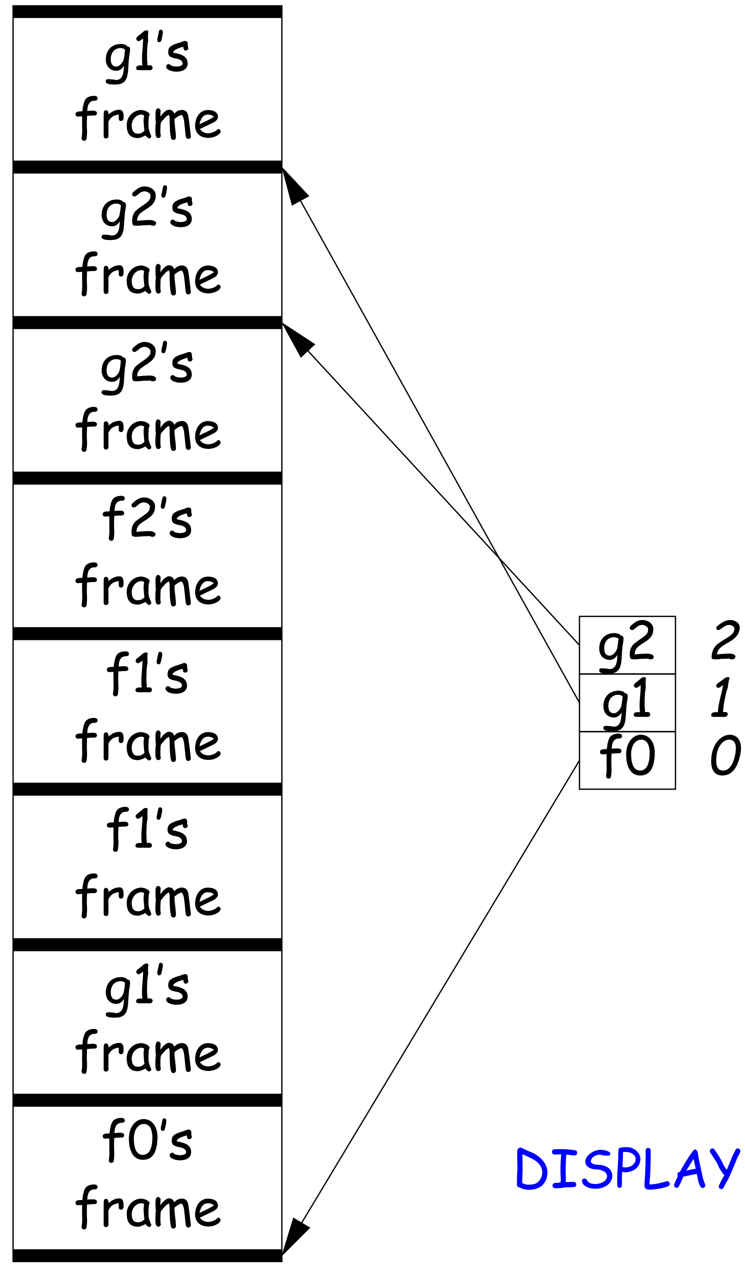


The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```
def f0 ():
  q = 42; g1 ()
  def f1 ():
    def f2 (): ... g2 () ...
    def g2 (): ... g2 () ... g1 () ...
    ... f2 () ... f1 () ...
  def g1 (): ... f1 () ...
```

- Each time we enter a function at lexical level k (i.e., nested inside k functions), save pointer to its frame base in DISPLAY[k]; restore on exit.
- Access variable at lexical level k through DISPLAY[k].
- Relies heavily on scope rules and proper function-call nesting



Using the global display (sketch)

C code:

```
int
f0 (int n0)
{
  int s = -n0;
  int g1 () { return s; }
  int f1 (int n1) {
    int f2 () {
      return n0 + n1
             + s + g1 ();
    }
    return f2 (s) + f1 (n0)
           + g1 ();
  }
  f1 (10);
}
```

```
f0:
    sw fp, 0(sp)      # Save old frame pointer
    sw ra, -4(sp)     # Save return address
    addi sp, sp, -16  # Adjust SP for s, ra, DL, old _DISPLAY[0]
    addi fp, sp, 12   # FP now points to ra.
    lw t0, _DISPLAY+0 # Save old _DISPLAY[0] ...
    sw t0, -8(fp)     # ... on stack
    sw fp, _DISPLAY+0 # And insert my FP in its place.
    ...
    lw t0, -8(fp)     # Restore old _DISPLAY[0]
    sw t0, _DISPLAY+0
    addi sp, fp, 4    # Restore sp
    etc.
f1: ...
    sw fp, 0(sp)      # Save old frame pointer
    sw ra, -4(sp)     # Save return address
    addi sp, sp, -12  # Adjust SP for ra, DL, old _DISPLAY[1]
    addi fp, sp, 8    # FP now points to ra.
    lw t0, _DISPLAY+4 # Save old _DISPLAY[1] ...
    sw t0, -4(fp)     # ... on stack
    sw fp, _DISPLAY+4 # And insert my FP in its place.
    ...
    lw t0, -4(fp)     # Restore old _DISPLAY[0]
    sw t0, _DISPLAY+4
    addi sp, fp, 4    # Restore sp
    etc.
f2 and g1: no extra code, since they have no nested functions.
```

Using the global display: accessing nonlocals

C code:

```
int
f0 (int n0)
{
  int s = -n0;
  int g1 () { return s; }
  int f1 (int n1) {
    int f2 () {
      return n0 + n1
             + s + g1 ();
    }
    return f2 (s) + f1 (n0)
           + g1 ();
  }
  f1 (10);
}
```

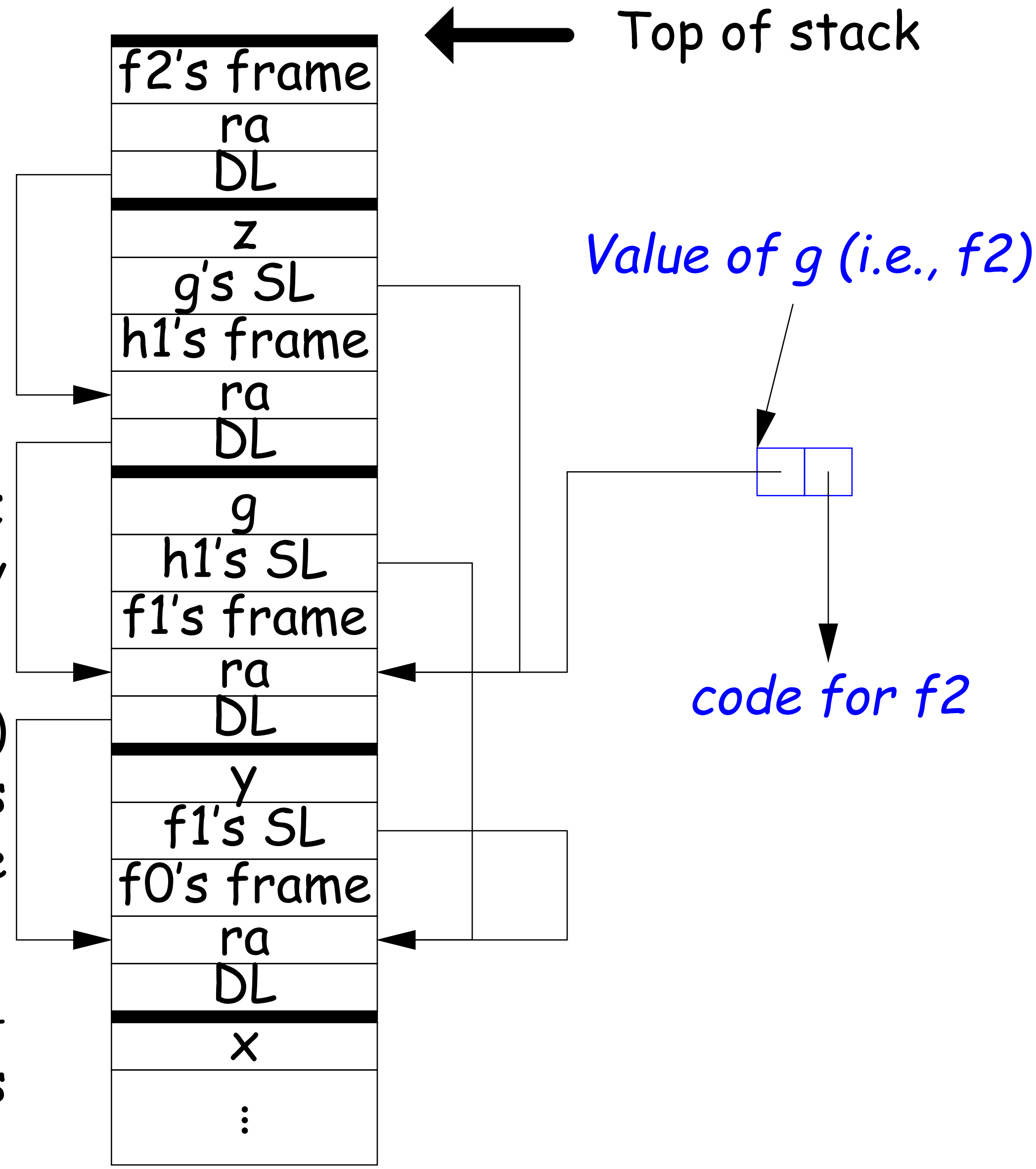
```
f2:
    ...
    lw t0, _DISPLAY+4 # Load my static link (to f1)
    lw t1, _DISPLAY+0 # Load f1's static link (to f0)
    lw t2, 8(t1)      # n0
    lw t3, -4(t1)     # s
    add t2, t2, t3    # n0 + s
    lw t3, 8(t0)      # n1
    add t2, t2, t3    # n0 + s + n1
    sw t2, 0(sp)      # Save
    # No need to pass static link to g1; it's in _DISPLAY[1]
    addi sp, sp, -4   # Adjust stack
    jal g1
    ...
```

5: Allow Function Values, Properly Nested Access

- In C, C++, no function nesting.
- So all non-local variables are global, and have fixed addresses.
- Thus, to represent a variable whose value is a function, need only to store the address of the function's code.
- But when nested functions possible, function value must contain more.
- When function is finally called, must be told what its static link is.
- Assume first that access is properly nested: variables accessed only during lifetime of their frame.
- So can represent function with address of code + the address of the frame that contains that function's definition.
- It's environment diagrams again!!

Function-Value Representation

```
def f0 (x):
   def f1 (y):
     def f2 (z):
       return x + y + z
     print h1 (f2)
   def h1 (g): g (3)
   f1 (42)
```



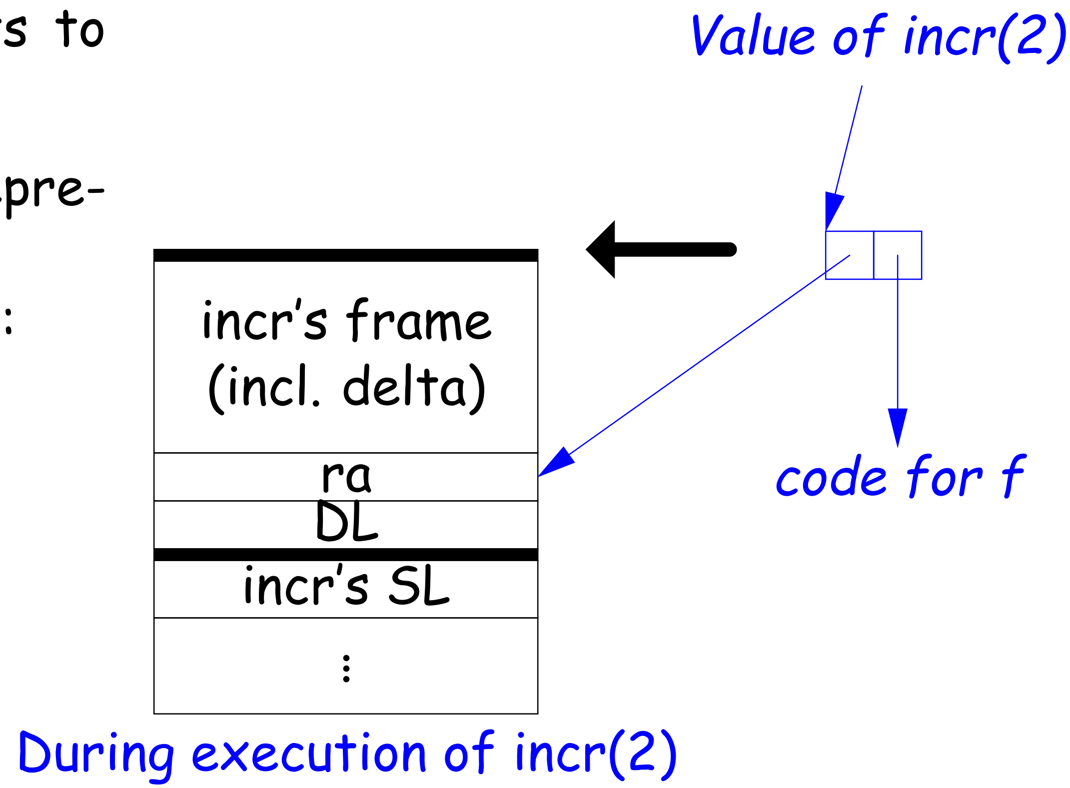
- Call `f0` from the main program; look at the stack when `f2` finally is called (see right).
- When `f2`'s value (as a function) is computed, current frame is that of `f1`. That is stored in the value passed to `h1`.
- Easy with static links; global display technique does not fare as well [why?]

6: General Closures

- What happens when the frame that a function value points to goes away?
- If we used the previous representation (#5), we'd get a *dangling pointer* in this case:

```
def incr (n):
   delta = n
   def f (x):
     return delta + x
   return f

p2 = incr(2)
print p2(3)
```

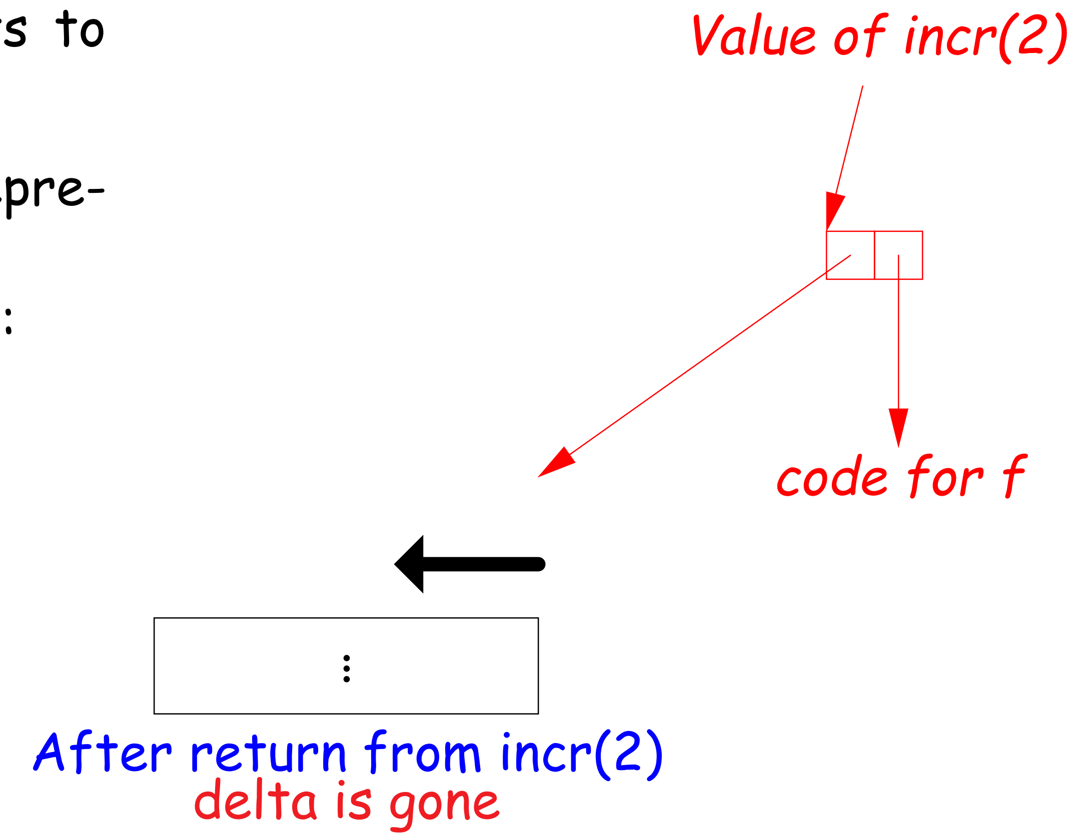


6: General Closures

- What happens when the frame that a function value points to goes away?
- If we used the previous representation (#5), we'd get a *dangling pointer* in this case:

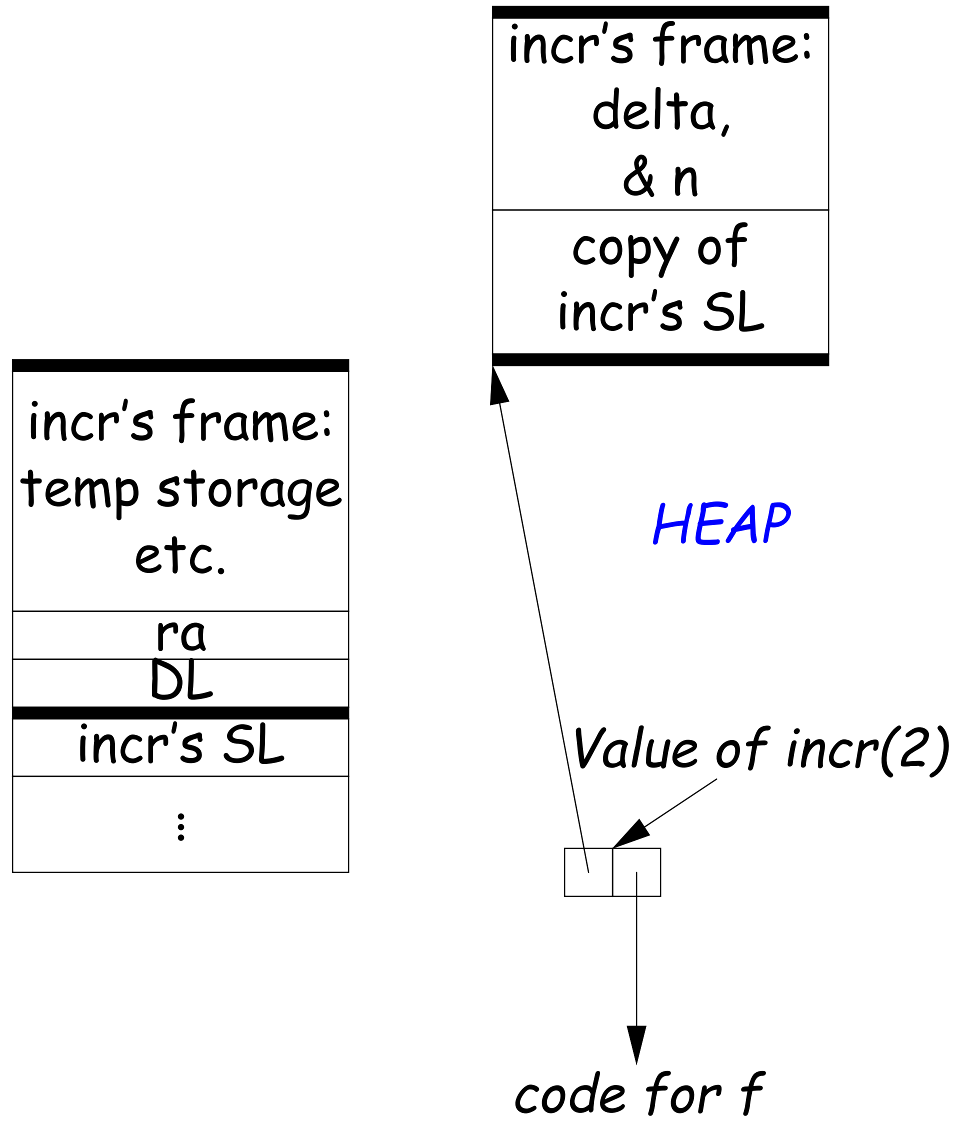
```
def incr (n):
   delta = n
   def f (x):
     return delta + x
   return f

p2 = incr(2)
print p2(3)
```



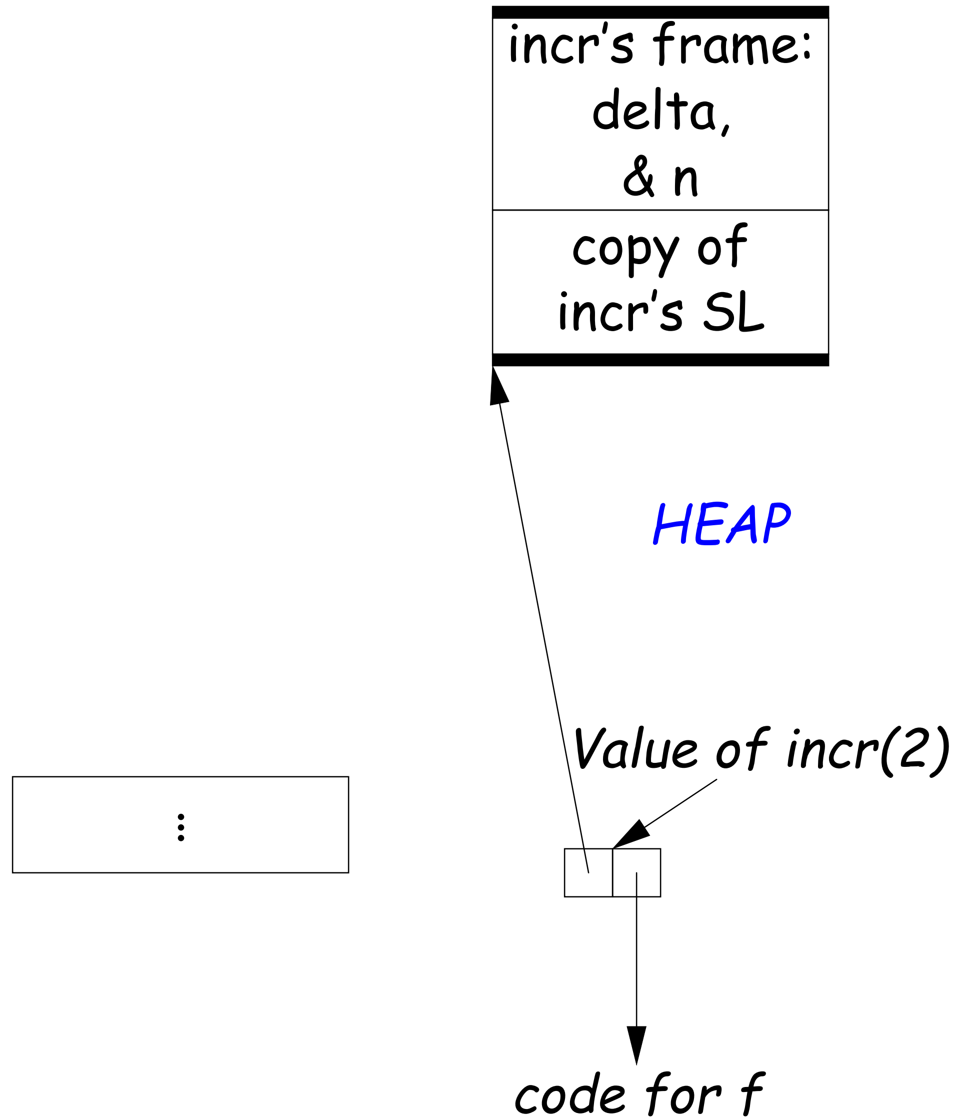
Representing Closures

- Could just forbid this case (as some languages do):
  - Algol 68 would not allow pointer to f (last slide) to be returned from incr.
  - Or, one could allow it, and do something random when f (i.e. via delta) is called.
- Scheme and Python allow it and do the right thing.
- But must in general put local variables (and a static link) in a record on the heap, instead of on the stack.



Representing Closures

- Could just forbid this case (as some languages do):
  - Algol 68 would not allow pointer to f (last slide) to be returned from incr.
  - Or, one could allow it, and do something random when f (i.e. via delta) is called.
- Scheme and Python allow it and do the right thing.
- But must in general put local variables (and a static link) in a record on the heap, instead of on the stack.
- Now frame can disappear harmlessly.



7: Continuations

- Suppose function return were not the end?

```
def f (cont): return cont
x = 1
def g (n):
  global x, c
  if n == 0:
     print "a", x, n,
     c = call_with_continuation (f)
     print "b", x, n,
  else: g(n-1); print "c", x, n,
g(2); x += 1; print; c()
```

```
# Prints:
#  a 1 0 b 1 0 c 1 1 c 1 2
#  b 2 0 c 2 1 c 2 2
#  b 3 0 c 3 1 c 3 2
...
```

- The *continuation*, c, passed to f is "the function that does whatever is supposed to happen after I returns from f (and exits program)."
- Can be used to implement exceptions, threads, co-routines.
- Implementation? Nothing much for it but to put all activation frames on the heap.
- Distributed cost.
- However, we can do better on special cases like exceptions.

Summary

Problem	Solution
1. Plain: no recursion, no nesting, fixed-sized data with size known by compiler, first-class function values.	Use inline expansion or use static variables to hold return addresses, locals, etc.
2. #1 + recursion	Need stack.
3. #2 + Add variable-sized unboxed data	Need to keep both stack pointer and frame pointer.
4. #3 – first-class function values + Nested functions, up-level addressing	Add static link or global display.
5. #4 + Function values w/ properly nested accesses: functions passed as parameters only.	Static link, function values contain their link. (Global display doesn't work so well)
6. #5 + General closures: first-class functions returned from functions or stored in variables	Store local variables and static link on heap.
7. #6 + Continuations	Put everything on the heap.