

Lecture #16: Introduction to Runtime Organization

Administrivia

- Homework for next Friday will be posted tomorrow.

Status

- Lexical analysis
 - Produces tokens
 - Detects & eliminates illegal tokens
- Parsing
 - Produces trees
 - Detects & eliminates ill-formed parse trees
- Static semantic analysis
 - Produces *decorated tree* with additional information attached
 - Detects & eliminates remaining static errors
- Next are the dynamic "back-end" phases: \Leftarrow *we are here*
 - Code generation (at various semantic levels)
 - Optimization

Run-time environments

Before discussing code generation, we need to understand what we are trying to generate.

- We'll use the term *virtual machine* to refer to the compiler's target.
- Can be just a bare hardware architecture (small embedded systems).
- Can be an interpreter, as for Java, or an interpreter that does additional compilation at execution, as in modern Java JITs
- For now, we'll stick to hardware + conventions for using it (the *API: application programmer's interface*) + some *runtime-support library*.

Code Generation Goals and Considerations

- *Correctness*: execution of generated code must be consistent with the programs' specified dynamic semantics.
- In general, however, these semantics do not completely specify behavior, often to allow compiler to accomplish other goals, such as...
- *Speed*: produce code that executes as quickly as possible, or reliably meets certain timing constraints (as in real-time systems).
- *Size*: minimize size of generated program or of runtime data structures.
- Speed and size optimization can be conflicting goals. Why?
- *Compilation speed*: especially during development or when using JITs.
- Most complications in code generation come from trying to be fast as well as correct, because this requires attention to special cases.

Subgoals and Constraints

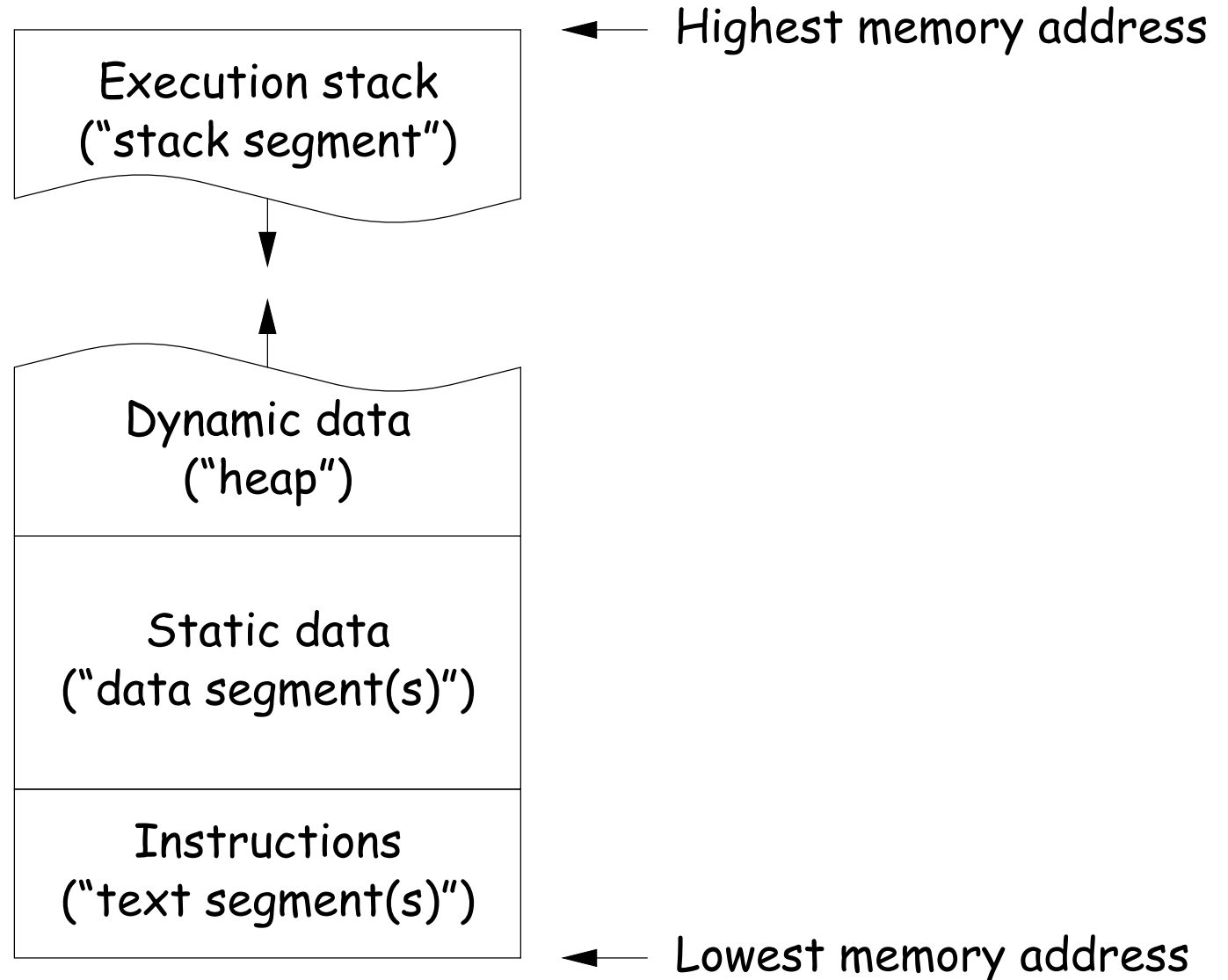
- Subgoals for improving speed and size:
 - Minimize instruction counts.
 - Keep data structure static, known at compilation (e.g., known constant offsets to fields). Contrast Java and Python.
 - Maximize use of registers ("top of the memory hierarchy").
- Subgoals for improving compilation speed:
 - Try to keep analyses as *local* as possible (single statement, block, procedure), because their compilation-time cost tends to be non-linear.
 - Simplify assumptions about control flow: procedure calls "always" return, statements generally execute in sequence. (Where are these violated?)

Activations and Lifetimes (Extents)

- An invocation of procedure P is an *activation* of P .
- The *lifetime of an activation* of P is all the steps to execute P , including all the steps in procedures P calls.
- The *lifetime (extent) of a variable* is the portion of execution during which that variable exists (whether or not the code currently executing can reference it).
- Lifetime is a dynamic (run-time) concept, as opposed to scope, which is static.
- Lifetimes of procedure activations and local variables properly nest (in a single thread), suggesting a *stack* data structure for maintaining their runtime state.
- Other variables have extents that are not coordinated with procedure calls and returns.

Memory Layout

Characteristics of procedure activations and variables give rise to the following typical data layout for a (single-threaded) program:



Activation Records

- The information needed to manage one procedure activation is called an *activation record (AR)* or *(stack) frame*.
- If procedure F (the *caller*) calls G (the *callee*, typically G 's activation record contains a mix of data about F and G):
 - *Return address* to instructions in F .
 - *Dynamic link* to the AR for F .
 - Space to save registers needed by F .
 - Space for G 's local variables.
 - Information needed to find non-local variables needed by G .
 - Temporary space for intermediate results, arguments to and return values from functions that G calls.
 - Assorted machine status needed to restore F 's context (signal masks, floating-point unit parameters).
- Depending on architecture and compiler, registers typically hold part of AR (at times), especially parameters, return values, locals, and pointers to the current stack top and frame.

Calling Conventions

- Many variations are possible:
 - Can rearrange order of frame elements.
 - Can divide caller/callee responsibilities differently.
 - Don't need to use an array-like implementation of the stack: can use a linked list of ARs.
- An organization is better if it improves execution speed or simplifies code generation
- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record.
- Furthermore, it is common to compile procedures separately and without access of each other's details, which motivates the the imposition of *calling conventions*.

Static Storage

- Here, "static storage" refers to variables whose extent is an entire execution and whose size is typically fixed before execution.
- Not generally stored in an activation record, but assigned a fixed address once.
- In C/C++ variables with file scope (declared `static` in C) and with external linkage ("global") are in static storage.
- Java's "static" variables are an odd case: they don't really fit this picture (why?)

Heap Storage

- Variables whose extent is greater than that of the AR in which they are created can't be kept there:

```
Bar foo() { return new Bar(); }
```

- Call such storage *dynamically allocated*.
- Typically allocated out of an area called the *heap* (confusingly, not the same as the heap used for priority queues!)

Achieving Runtime Effects—Functions

- Language design and runtime design interact. Semantics of functions make good example.
- Levels of function features:
 1. Plain: no recursion, no nesting, fixed-sized data with size known by compiler.
 2. Add recursion.
 3. Add variable-sized unboxed data.
 4. Allow nesting of functions, up-level addressing.
 5. Allow function values w/ properly nested accesses only.
 6. Allow general closures.
 7. Allow continuations.
- Tension between these effects and structure of machines:
 - Machine languages typically only make it easy to access things at addresses like $R + C$, where R is an address in a register and C is a relatively small integer constant.
 - Therefore, fixed offsets **good**, data-dependent offsets **bad**.

1: No recursion, no nesting, fixed-sized data

- Total amount of data is bounded, and there is only one instantiation of a function at a time.
- So all variables, return addresses, and return values can go in fixed locations.
- No stack needed at all.
- Characterized FORTRAN programs in the early days.
- In fact, can dispense with call instructions altogether: expand function calls in-line. E.g.,

```
def f (x):  
    x *= 42  
    y = 9 + x;  
    g (x, y)  
  
f (3)
```

\implies becomes \implies

```
x_1 = 3  
x_1 *= 42  
y_1 = 9 + x_1  
g (x_1, y_1)
```

- However, program may get bigger than you want. Typically, one in-lines only small, frequently executed functions.

1: Calling conventions

- If we don't use function inlining, will need to save return address, parameters.
- There are many options. Here's one example, from the IBM 360, of calling function F from G and passing values 3 and 4:

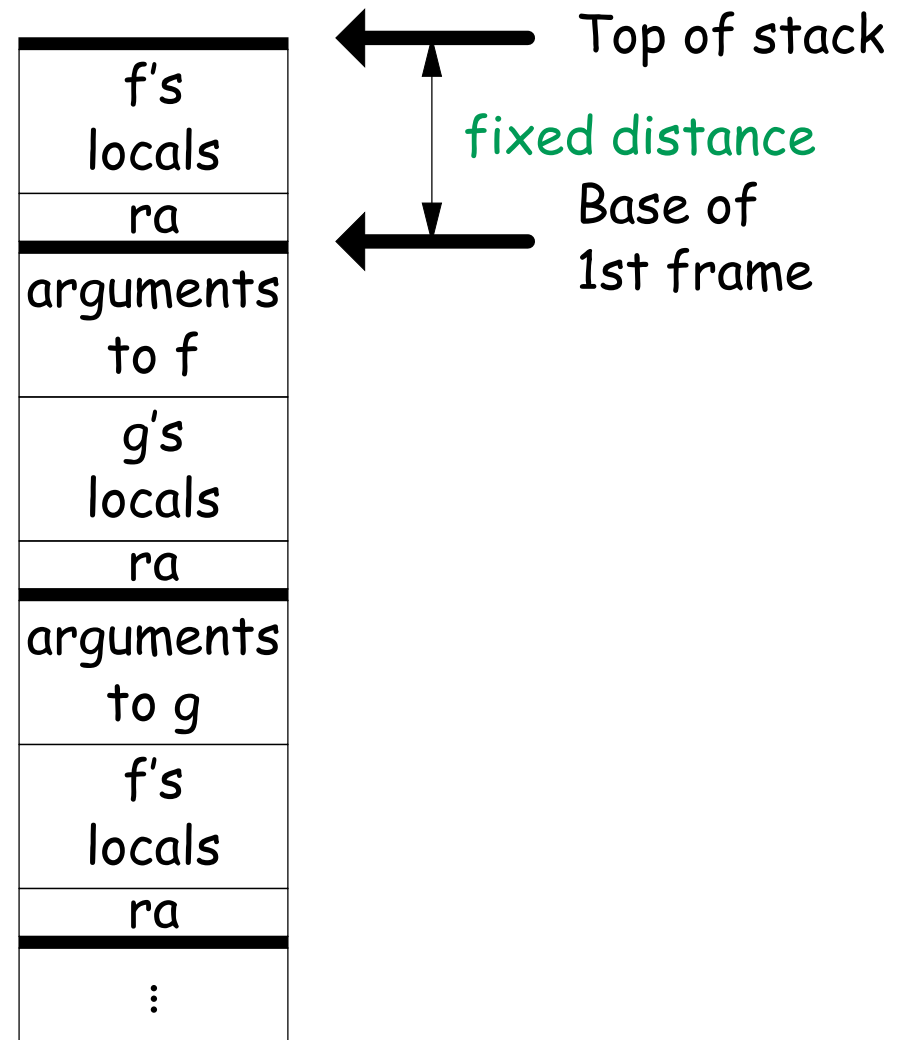
```
GArgs DS 2F          Reserve 2 4-byte words of static storage */
      ...
      ENTRY G
G      ...
      LA R1,GArgs    Load Address of arguments into register 1
      LA R0,3        Store 3 and 4 in GArgs+0 and GArgs+4
      ST R0,GArgs
      LA R0,4
      ST R0,GArgs+4
      BAL R14,F      Call ("Branch and Link") to F, R14 gets return point
```

and F might contain

```
FRet DS F
      ENTRY F
F      ST R14,FRet   Save return address
      L R2,0(R1)    Load first argument.
      ...
      L R14,FRet    Get return address
      BR R14        Branch to it
```

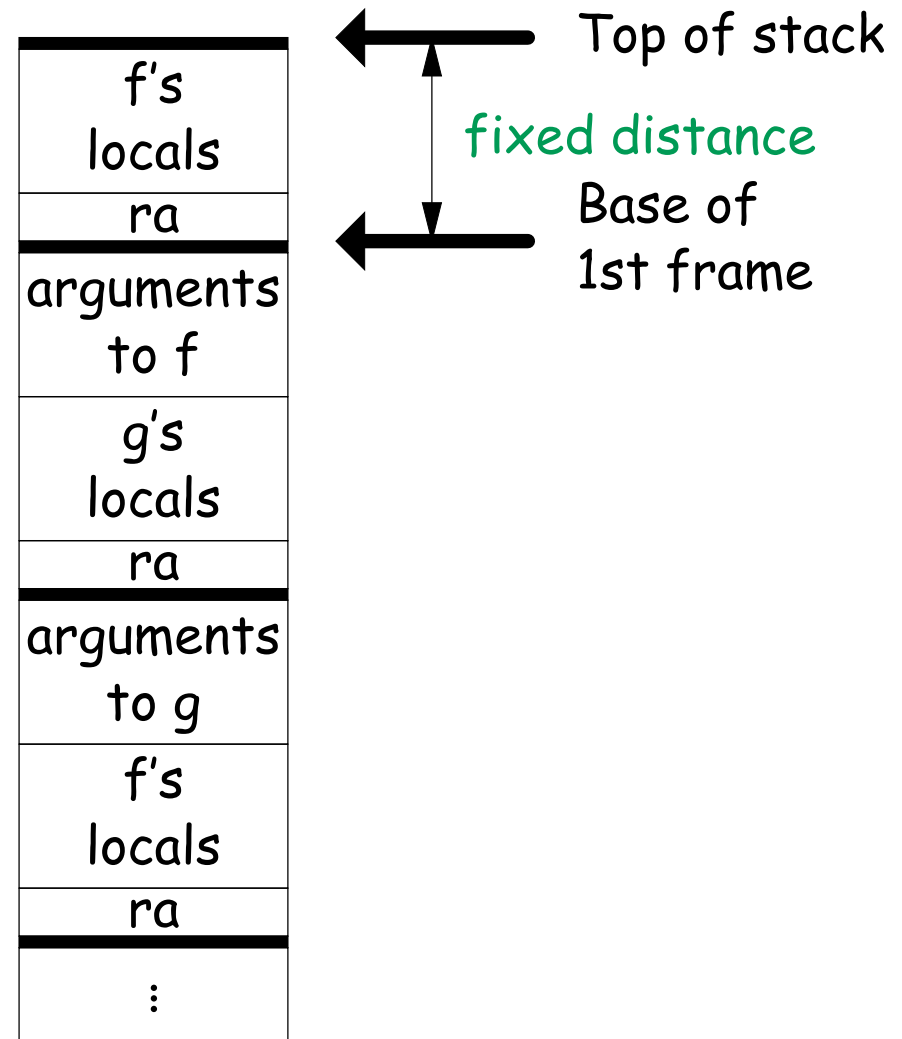
2: Add recursion

- Now, total amount of data is unbounded, and several instantiations of a function can be active simultaneously.
- Calls for some kind of expandable data structure: a stack.
- However, variable sizes still fixed, so size of each activation record (stack frame) is fixed.
- All local-variable addresses and the value of dynamic link are known offsets from stack pointer, which is typically in a register.



2: Calling Sequence when Frame Size is Fixed

- So dynamic links not really needed.
- Suppose f calls g calls f , as at right.
- When called, the initial code of g (its *prologue*) decrements the stack pointer by the size of g 's activation record.
- g 's exit code (its *epilogue*):
 - increments the stack pointer by this same size,
 - pops off the return address, and
 - branches to address just popped.



2: Calling sequence from ia32

Assembly excerpt (GNU operand order):

C code:

```
int
f (int x, int y)
{
    int s;
    s = 1;
    while (y > 0) {
        s *= x;
        y -= 1;
    }
    return s;
}

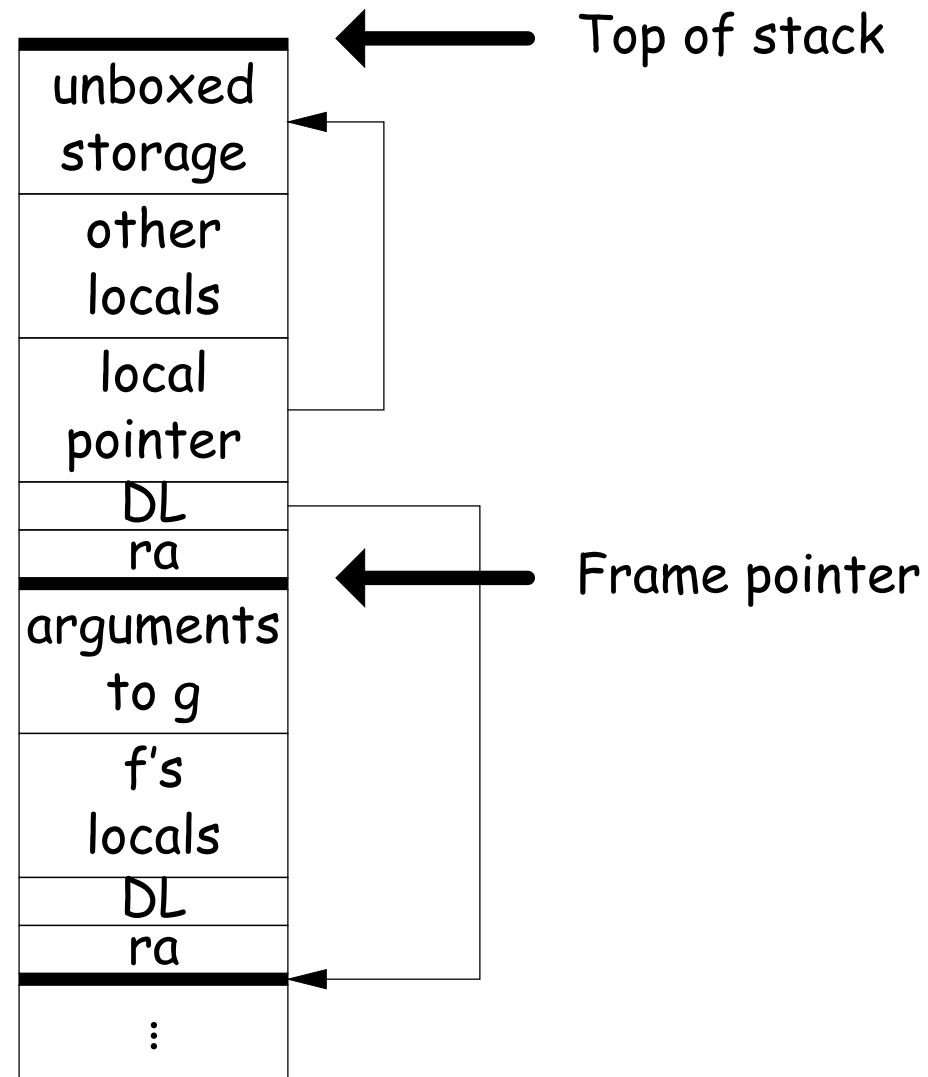
int
g(int q)
{
    return f(q, 5);
}
```

```
f:          / Return address (RA) at SP, x at SP+4, y at SP+8
            subl  $4, %esp          / Decrement SP to make space for s
            movl  $1, (%esp)        / s = 1
.L2:
            cmpl  $0, 12(%esp)      / compare 0 with y (at SP+12)
            jle   .L3
            movl  (%esp), %eax       / tmp = s
            imull 8(%esp), %eax      / tmp *= x
            movl  %eax, (%esp)       / s = tmp
            leal  12(%esp), %eax     / tmp = &y
            decl  (%eax)             / *tmp -= 1
            jmp   .L2
.L3:
            movl  (%esp), %eax       / return s in EAX
            addl  $4, %esp          / Restore stack pointer so RA on top,
            ret                    / then pop RA and return.

g:  ...
            movl  12(%esp), %eax     / tmp = q
            movl  %eax, (%esp)       / top of stack = q
            call  f                   / branch to f and push address of next.
next:
```

3: Add Variable-Sized Unboxed Data

- “Unboxed” means “not on heap.”
- Boxing allows all quantities on stack to have fixed size.
- So Java implementations have fixed-size stack frames.
- But does cost heap allocation, so some languages also provide for placing variable-sized data directly on stack (“heap allocation on the stack”)
- `alloca` in *C*, e.g.
- Now we do need dynamic link (DL).
- But can still insure fixed offsets of data from frame base (*frame pointer*) using pointers.
- To right, *f* calls *g*, which has variable-sized unboxed array (see right).



Other Uses of the Dynamic Link

- Often use dynamic link even when size of AR is fixed.
- Allows use of same strategy for all ARs, simplifies code generation.
- Makes it easier to write general functions that *unwind* the stack (i.e., pop ARs off, thus returning).

3: Calling sequence for the ia32

Assembly excerpt (GNU operand order):

C code:

```
int
f (int x, int y)
{
    int s;
    s = 1;
    while (y > 0) {
        s *= x;
        y -= 1;
    }
    return s;
}

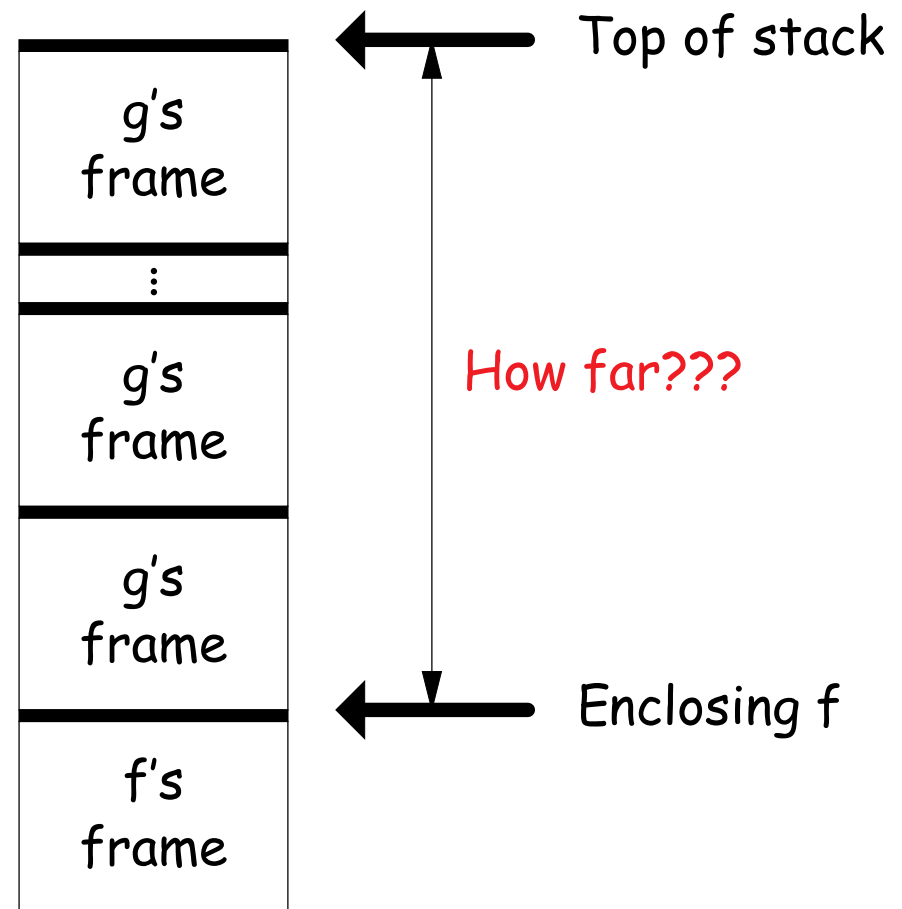
int
g(int q)
{
    return f(q, 5);
}

f:          / Return address (RA) at SP, x at SP+4, y at SP+8
    pushl %ebp          / Save old dynamic link.
    movl  %esp, %ebp    / Set ebp to current frame base.
    subl  $4, %esp      / Decrement SP to make space for s
    movl  $1, -4(%ebp)  / s = 1
.L2:
    cmpl  $0, 12(%ebp)  / compare 0 with y (at BP+12)
    jle  .L3
    movl  -4(%ebp), %eax / tmp = s
    imull 8(%ebp), %eax / tmp *= x
    movl  %eax, -4(%ebp) / s = tmp
    leal  12(%ebp), %eax / tmp = &y
    decl  (%eax)         / *tmp -= 1
    jmp  .L2
.L3:
    movl  -4(%ebp), %eax / return s
    leave          / Restore %esp to %ebp+4 and %ebp to 0(%ebp)
    ret
g:  ...
    movl  8(%ebp), %eax  / tmp = q
    movl  %eax, (%esp)   / top of stack = q
    call  f              / branch to f and push address of next.
next:
```

4: Allow Nesting of Functions, Up-Level Addressing

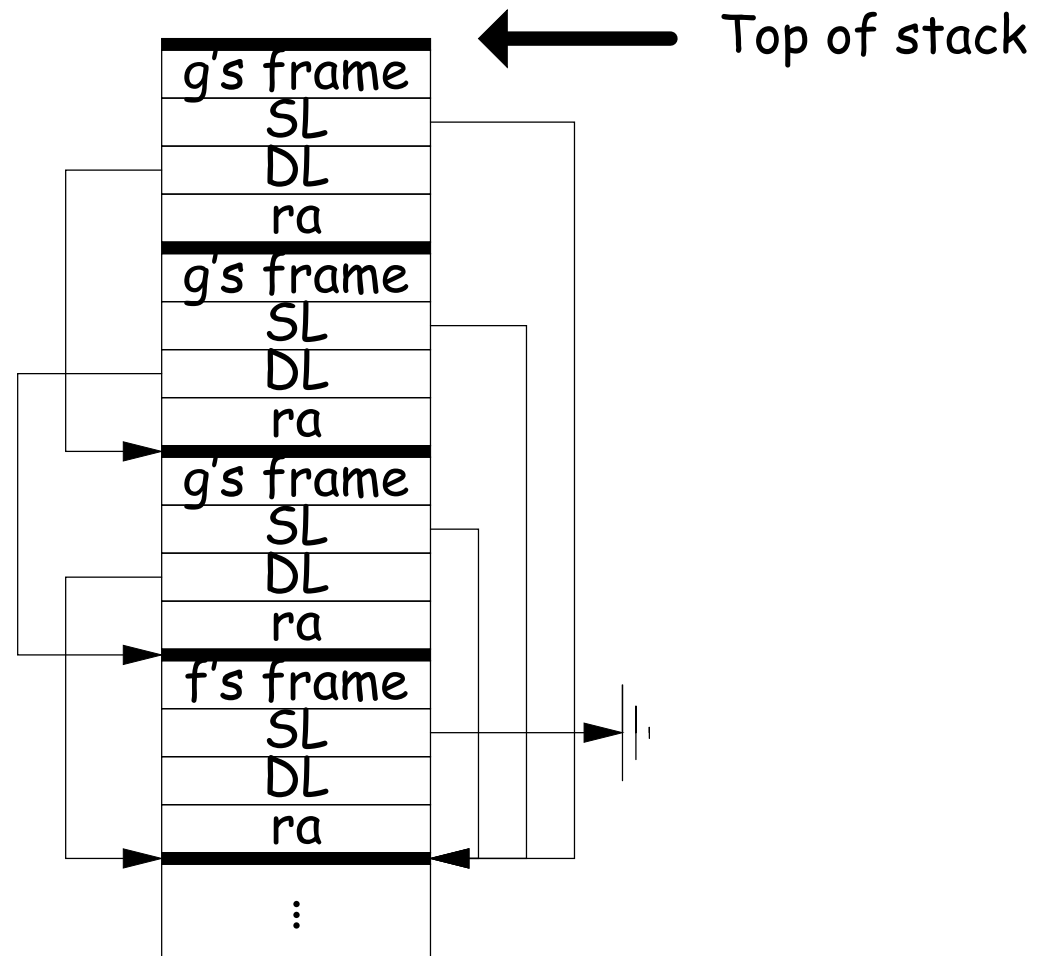
- When functions can be nested, there are three classes of variable:
 - a. Local to function.
 - b. Local to enclosing function.
 - c. Global
- Accessing (a) or (c) is easy. It's (b) that's interesting.
- Consider (in Python):

```
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.

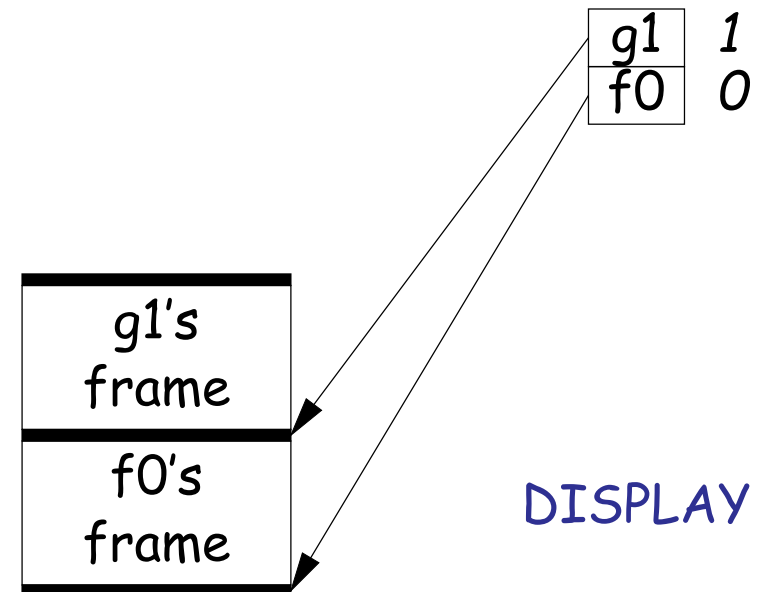


The Global Display

- 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 () ...  
    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

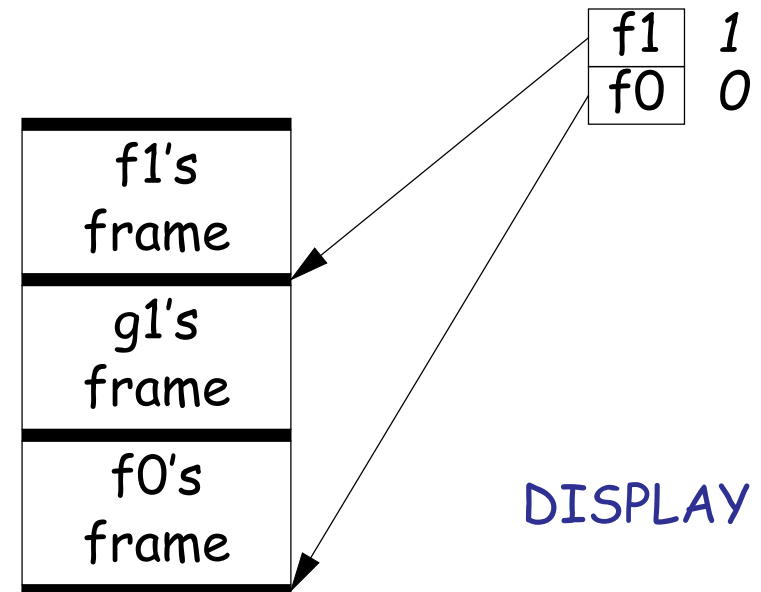


The Global Display

- 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 () ...  
    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

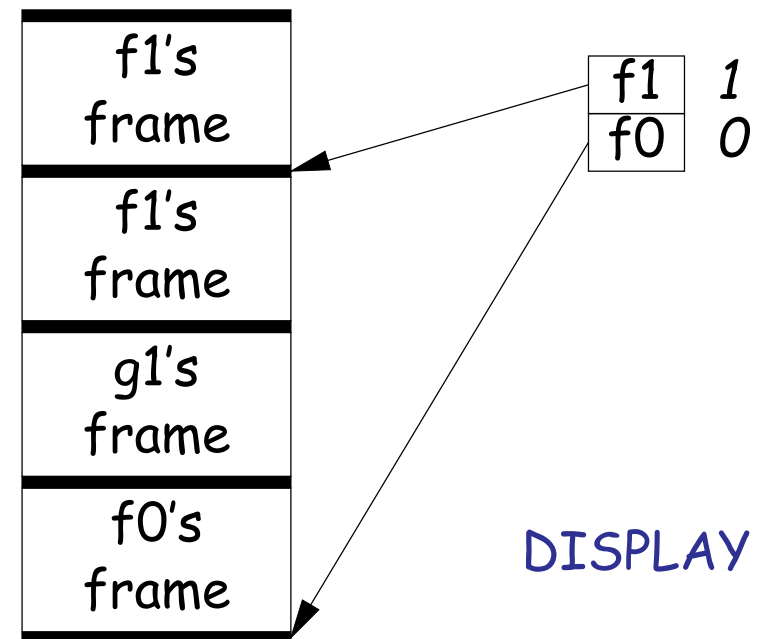


The Global Display

- 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 () ...  
    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

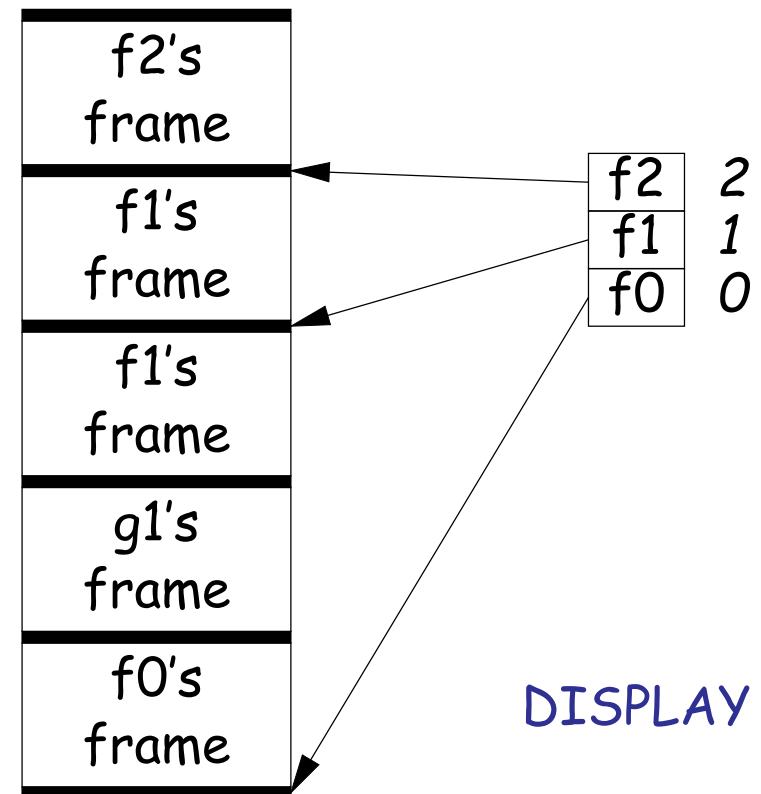


The Global Display

- 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 () ...  
    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

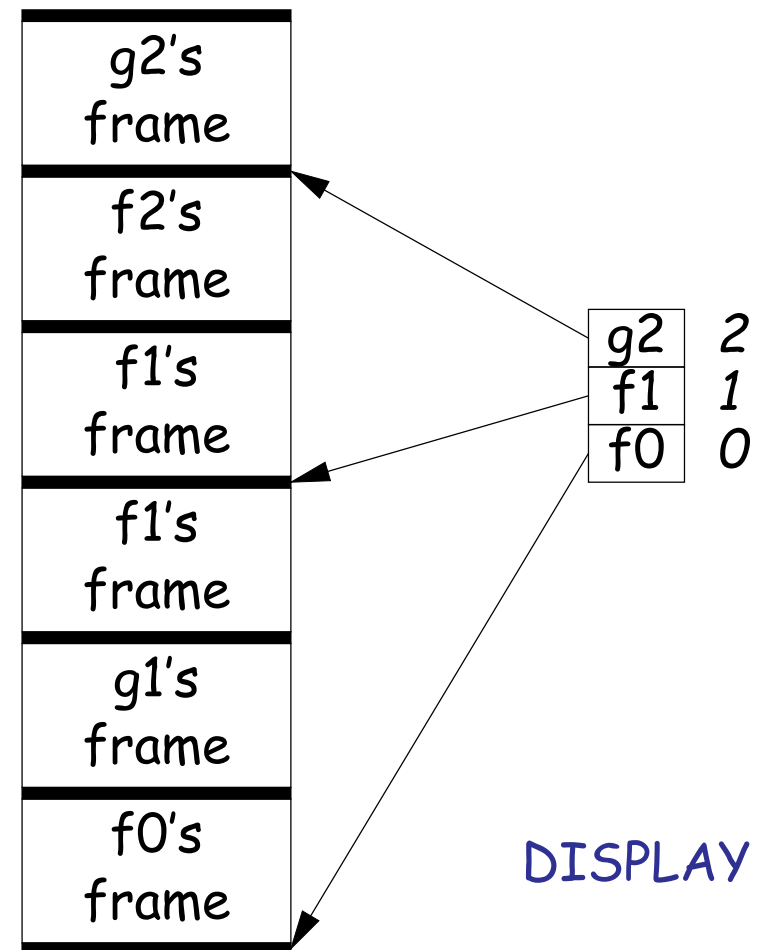


The Global Display

- 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 () ...  
    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

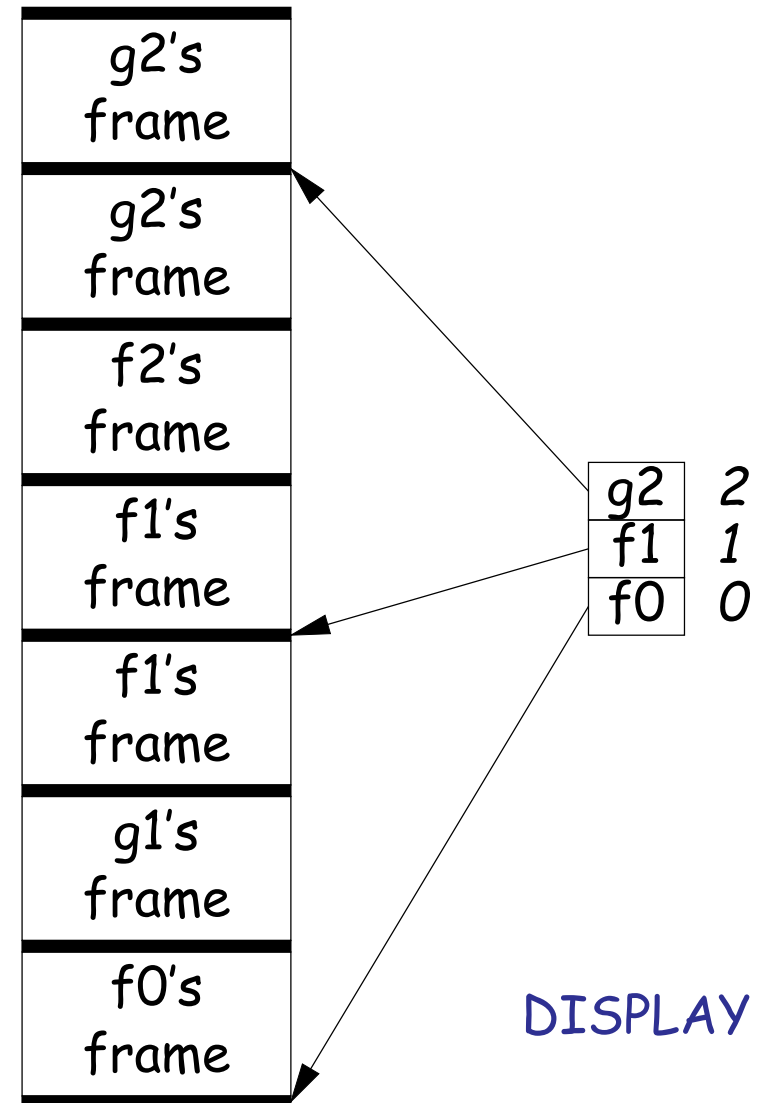


The Global Display

- 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 () ...  
    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

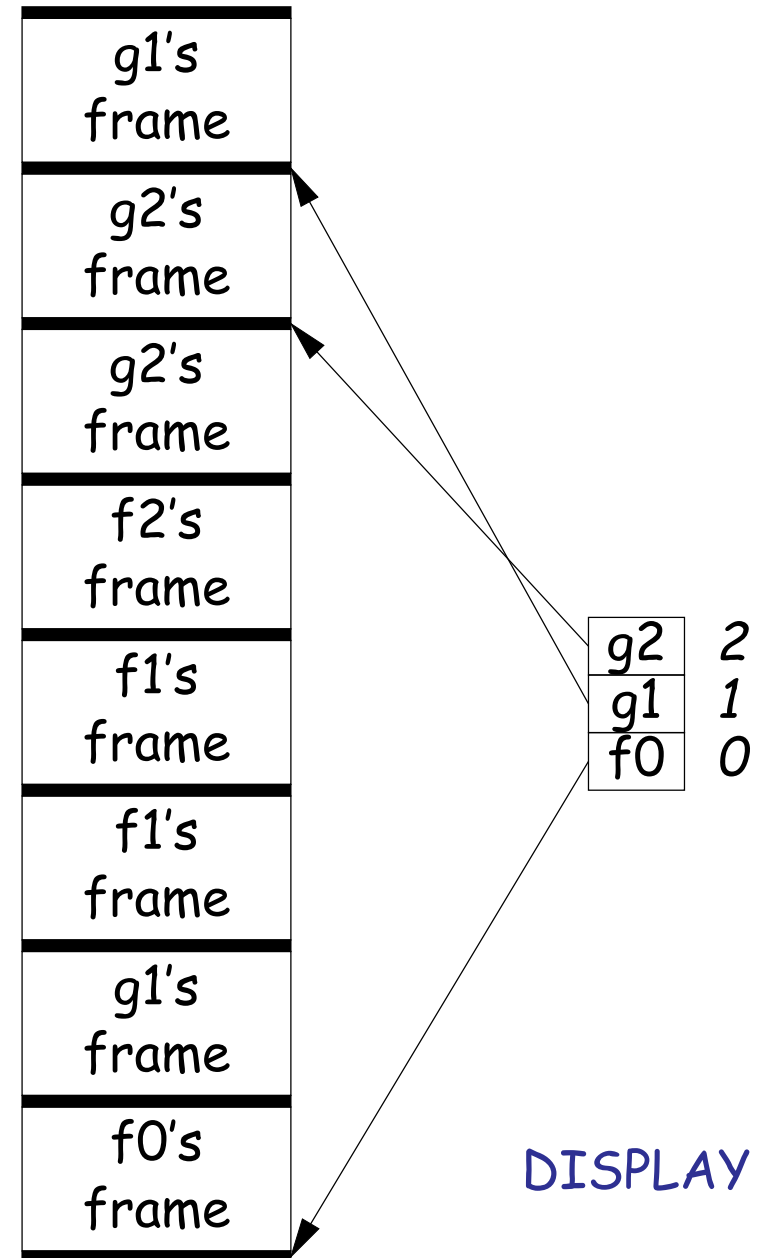


The Global Display

- 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 () ...
    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



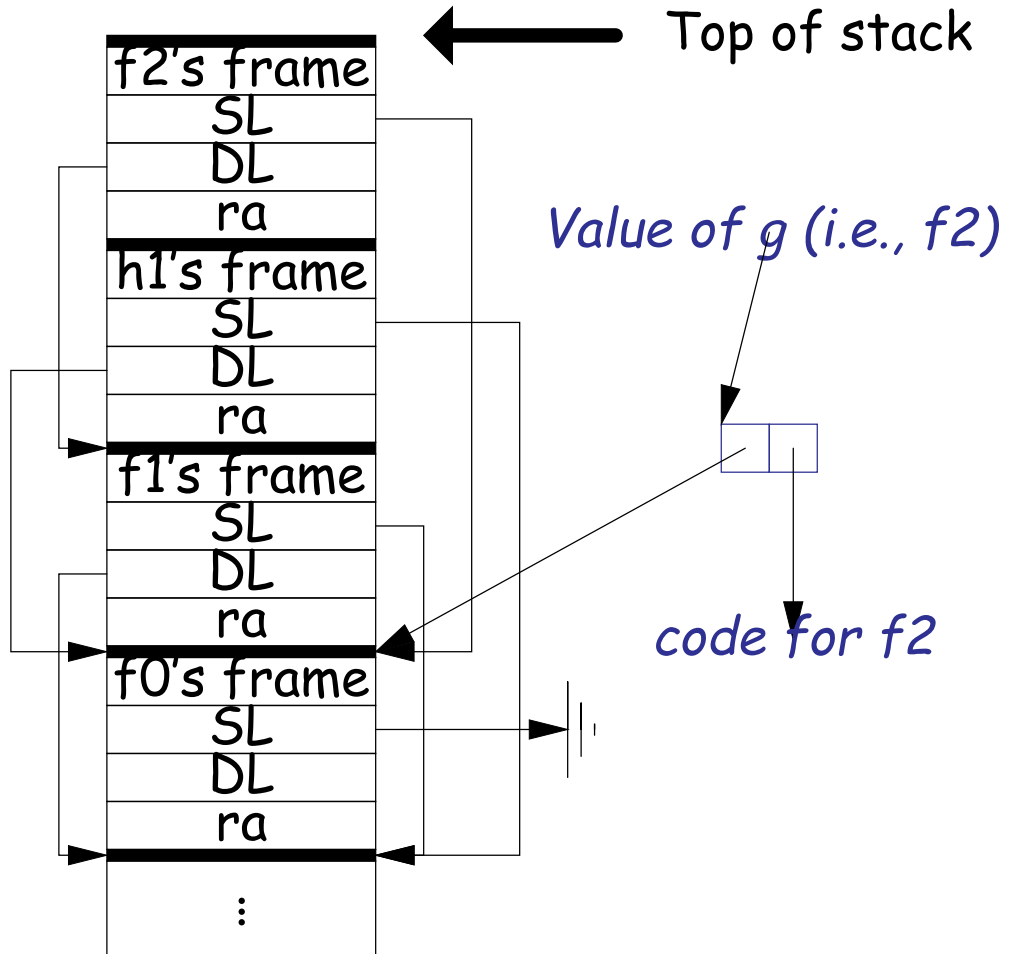
5: Allow Function Values, Properly Nested Access

- 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?]

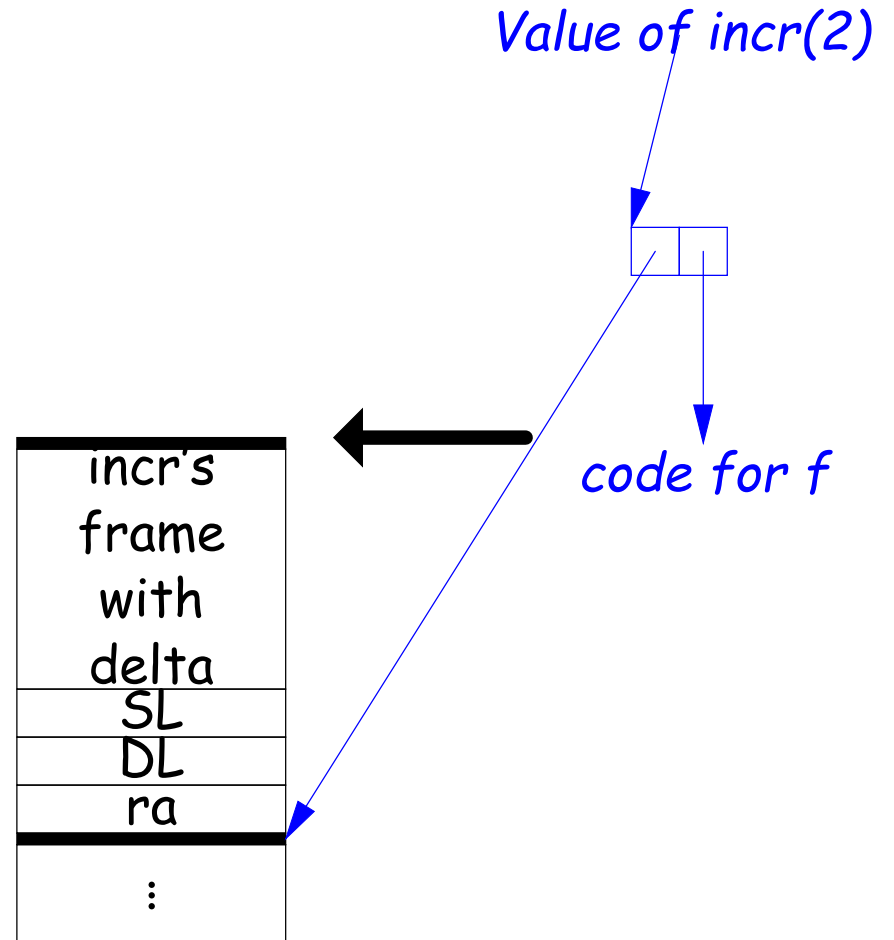


6: General Closures

- 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)
```



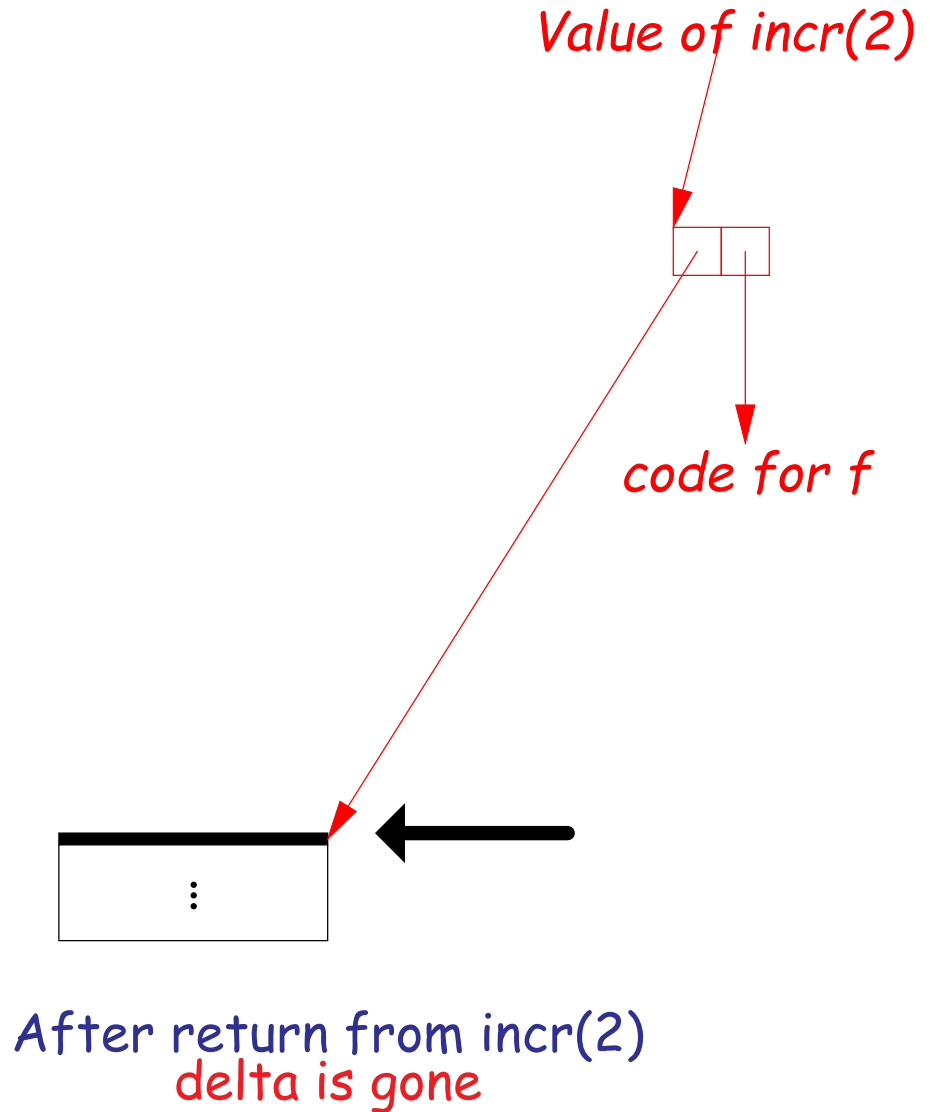
During execution of `incr(2)`

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:

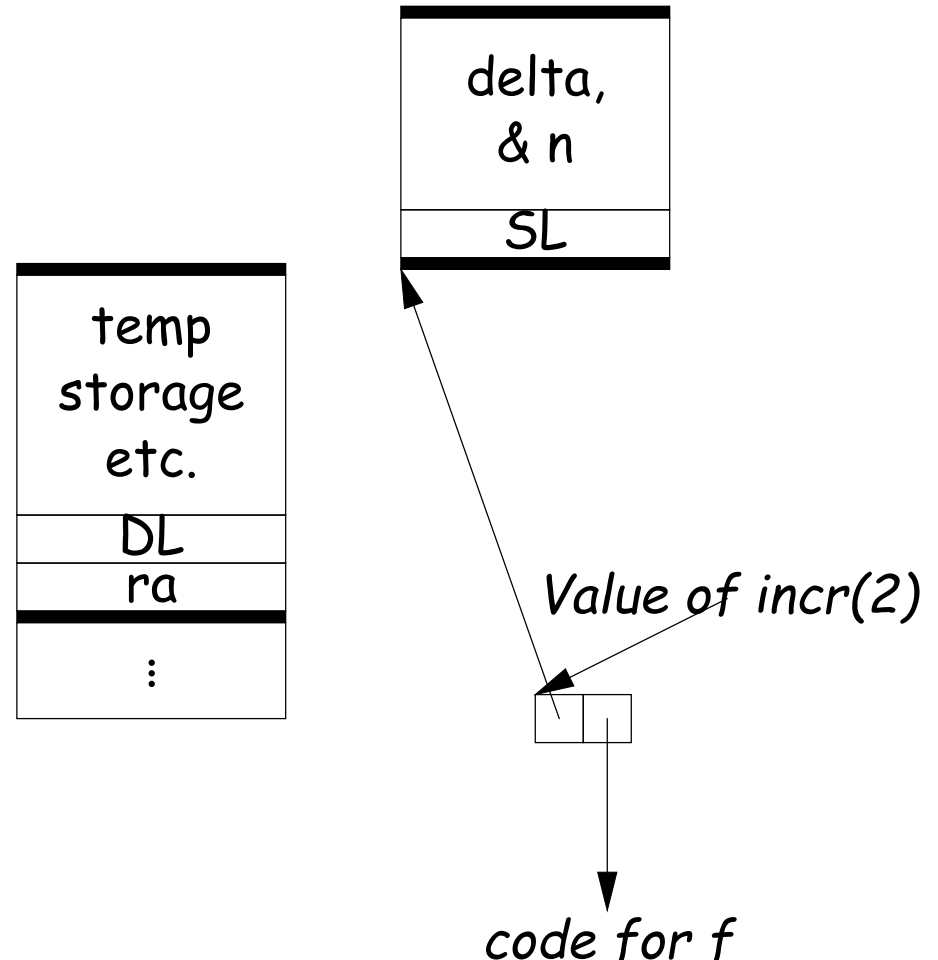
```
def incr (n):  
    delta = n  
    def f (x):  
        return delta + x  
    return f
```

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



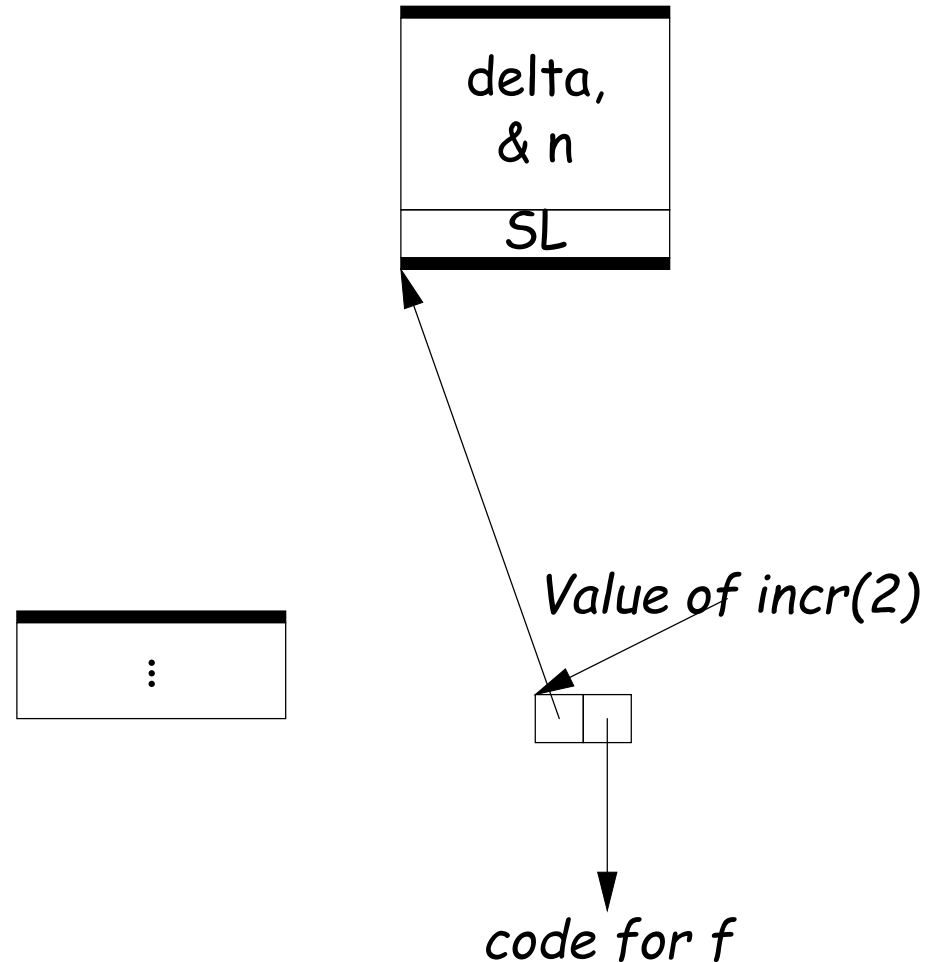
Representing Closures

- 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 return from *f*."
- 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.