Compiler Construction
Lent Term 2013

Lecture 5 (of 16)

Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge

LECTURES 5
First-order functions/procedures

- Block structure
- Very simple functions
- The call stack, stack frames
- Caller and Callee
- Need for ***calling conventions***
- A simple stack-oriented VM model
- Nested functions and possible modifications needed

(Nested) Block Structure

```
{ x : int;
  y : bool;
  ….
  If (e1) {
     z :int  = x + y;
     w :string = “hello”;
     if (e2) {
        u : int = size (w);
        v : int = u + z + x;

        … visible : x y z w u v
     }

   … visible: x y z w
  }

   … visible: x y
}
visible:
```

We need to implement this in a world with one large “flat” scope.

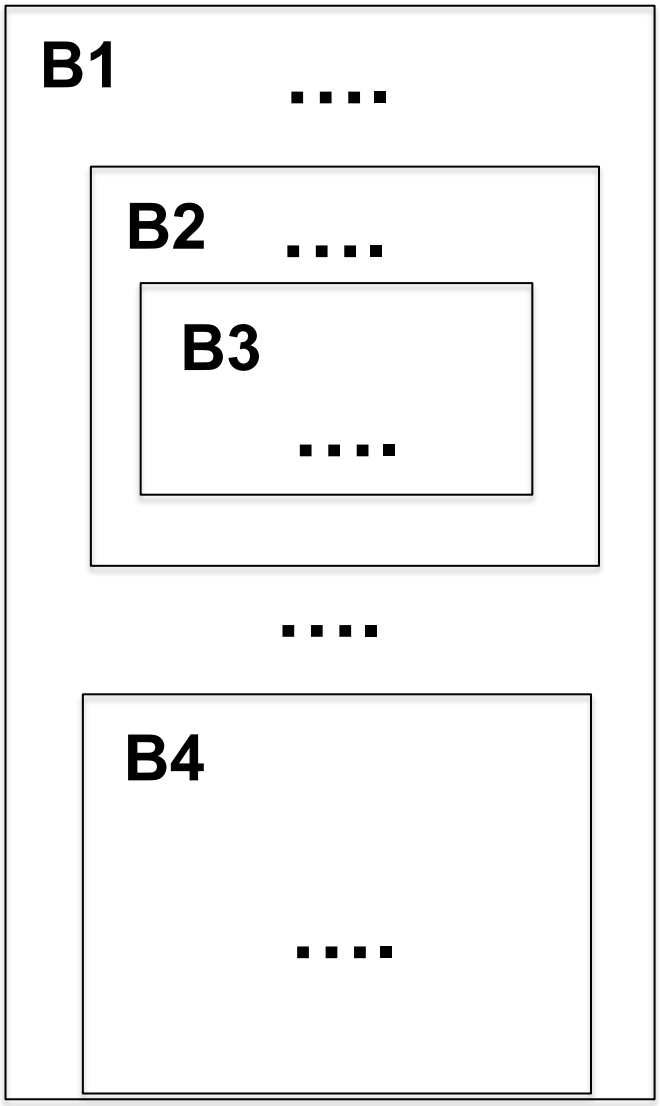
How do we allocate space for the values associated with the variables, and how do we find these values at run-time?

(Nested) Block Structure (2)

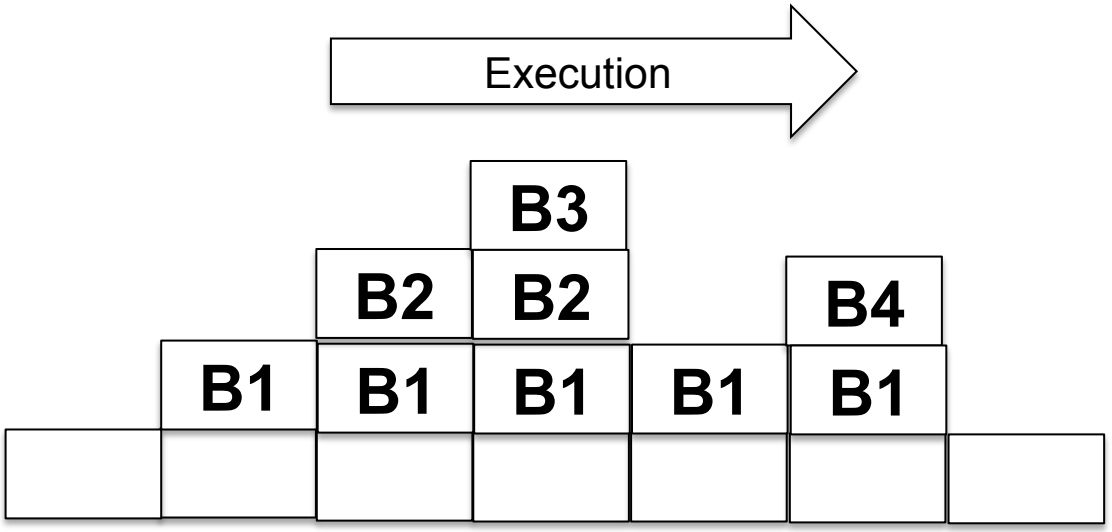
```
{ x : int;
  y : bool;
  ….
  If (e1) {
     z :int  = x + y;
     y :string = “hello”;
     if (e2) {
        u : int = size (y);
        v : int = u + z + x;

        … visible : x z y u v
     }

   … visible: x z y
  }

 … visible: x y
}
visible:
```

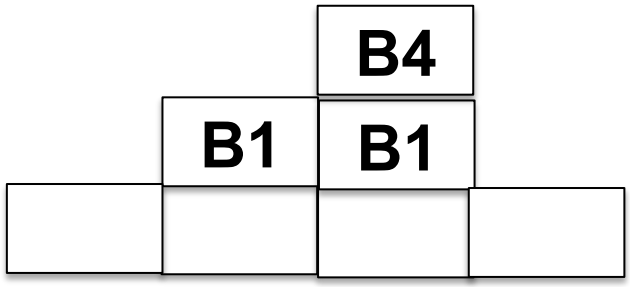
And we must
Correctly implement
Name binding rules.

Smells like LIFO, so use a stack



Possible run-time “activations” of these blocks

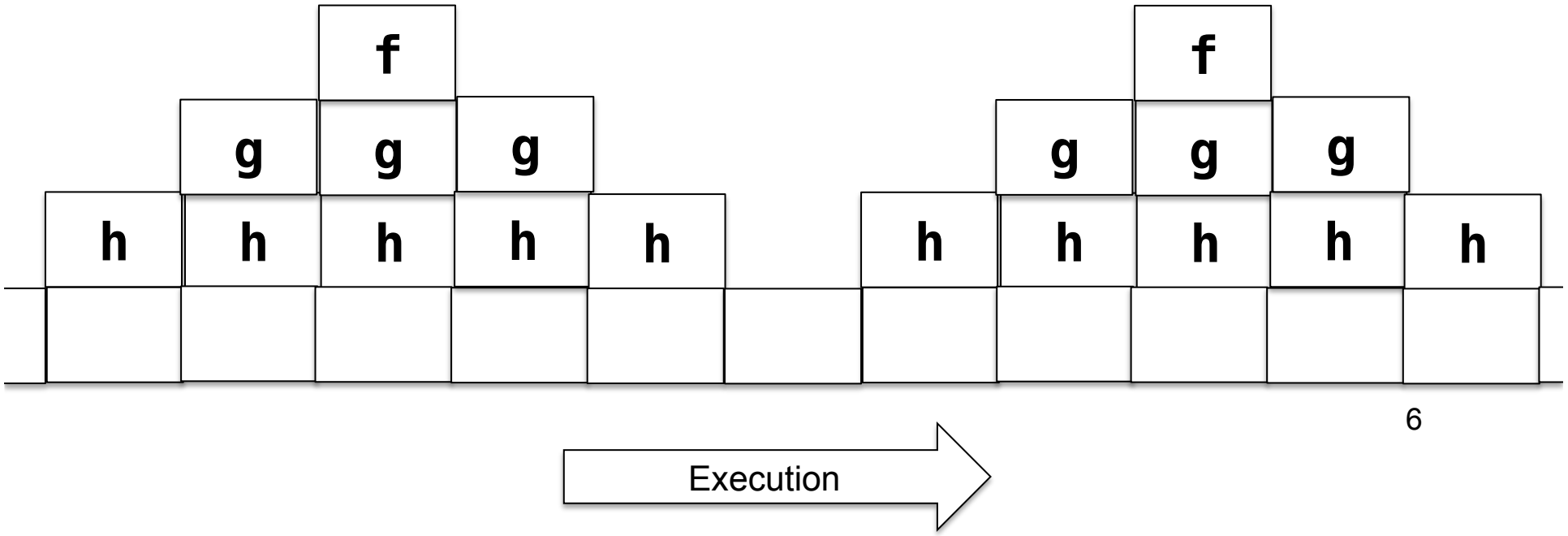




The same is seen function calls

```
let fun f (x) = x + 1
    fun g(y) = f(y+2)+2
    fun h(w) = g(w+1)+3
in
   h(h(17))
end
```

The run-time data structure is the call stack containing an activation record for each function invocation.



Caller and Callee

```
fun f (x, y) = e1

…

fun g(w, v) =
    … f(e2, e3) …
```

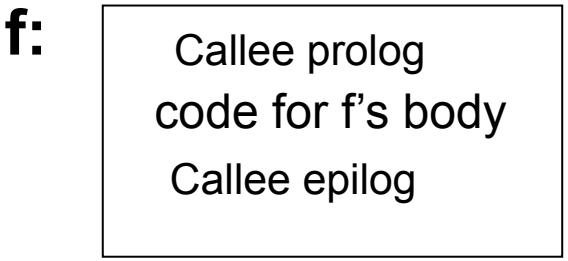
With respect to f, we say that g is the caller while f is the callee

Recursive functions can play both roles at the same time!

Caller and Callee need a contract

code generated for caller

code generated for callee



Sample Questions

- **Where are argument values placed?**
- **How does the body of callee find them?**
- **How does code for callee jump back to m+1 to resume execution of caller code?**
- **Does the caller or callee build an activation record?**
- **Does the caller or callee restore the stack or any other temporary locations used?**

Who, What, Where?

“Calling conventions” answer those questions

Calling conventions can come in many flavors and in many levels of abstraction

- Platform (OS+ISA) specific
- ISA specific
- Language specific
- Compiler specific
- …

A language-level calling convention, rather informal

The C Language Calling Sequence
S. C. Johnson and D. M. Ritchie
Bell Laboratories, September, 1981

http://cm.bell-labs.com/cm/cs/who/dmr/clcs.html

The Basic Call/Return Process

The following things happen during a C call and return:

1. The arguments to the call are evaluated and put in an agreed place.
2. The return address is saved on the stack.
3. Control passes to the called procedure.
4. The bookkeeping registers and register variables of the calling procedure are saved so that their values can be restored on return.
5. The called procedure obtains stack space for its local variables, and temporaries.
6. The bookkeeping registers for the called procedure are appropriately initialized. By now, the called procedure must be able to find the argument values.
7. The body of the called procedure is executed.
8. The returned value, if any, is put in a safe place while the stack space is freed, the calling procedure's register variables and bookkeeping registers are restored, and the return address is obtained and transferred to.

Some machine architectures will make certain of these jobs trivial, and make others difficult. These tradeoffs will be the subject of the next several sections.

A simple scenario

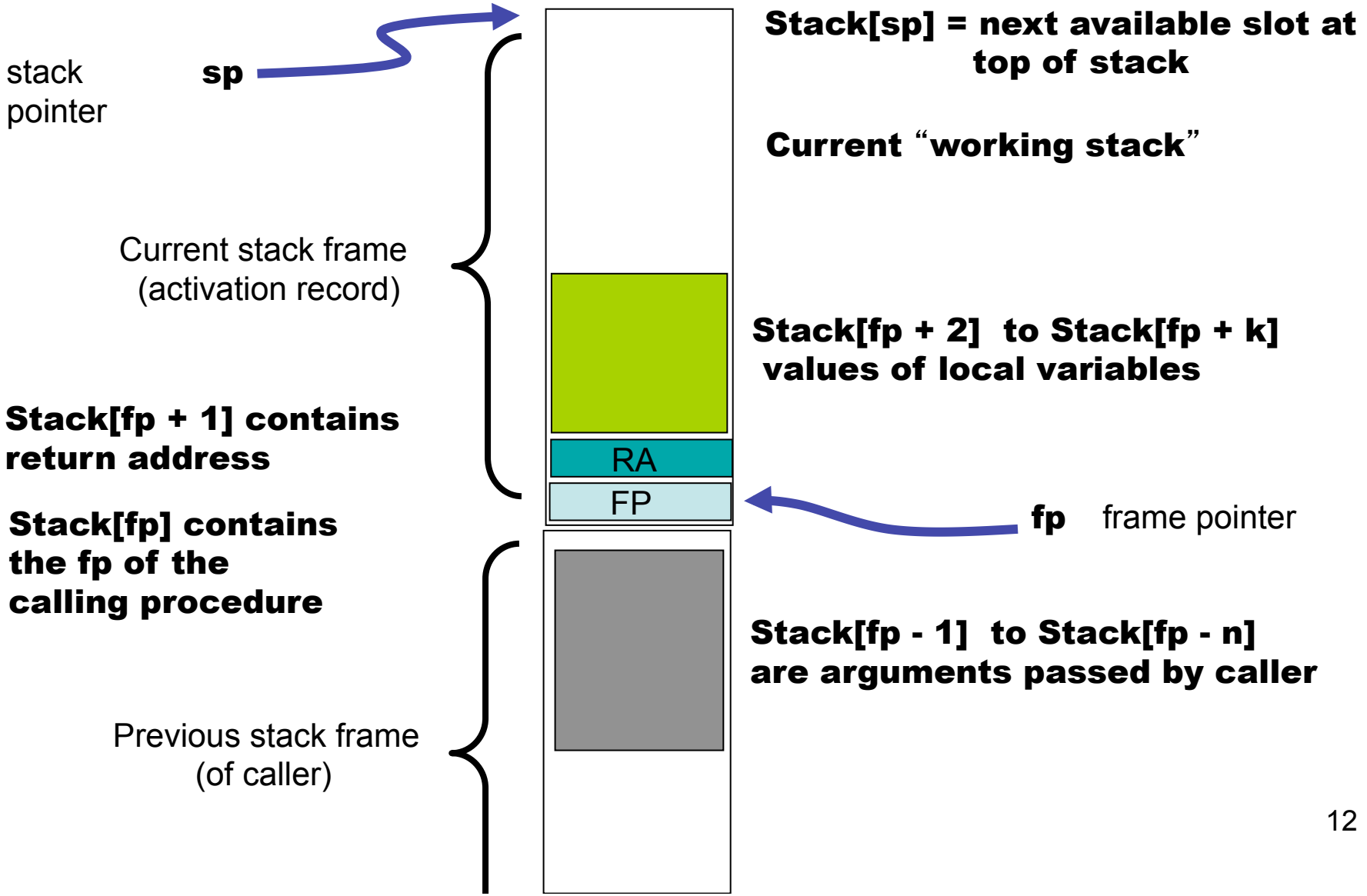
On our first pass we will assume that
(1) functions cannot return functions
as results or take functions as arguments and
(2) functions are not nested.

On our first pass we will use a simple stack-oriented machine abstraction.

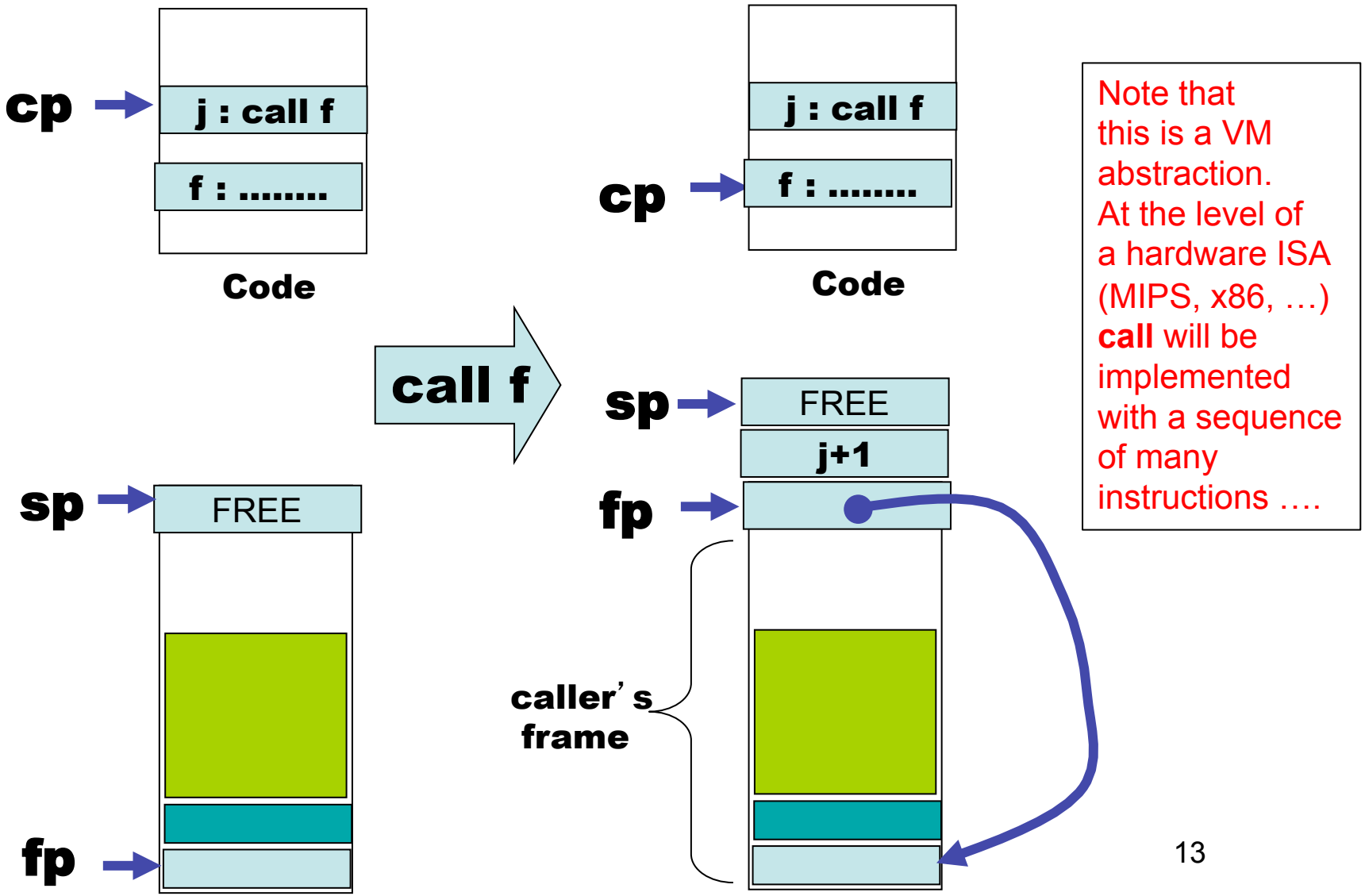
Note that both stack-oriented and register-oriented machines use a call stack with activation records (aka “stack frames”).

Call stack vs. operand stack. Conceptually distinct but in a stack-oriented machine may want to use same stack for both roles…

Simple Call Stack
(we will implement this in the VSM)



Our “call” operation



Note that this is a VM abstraction. At the level of a hardware ISA (MIPS, x86, …) **call** will be implemented with a sequence of many instructions ….

Hardware implementation of “call”?

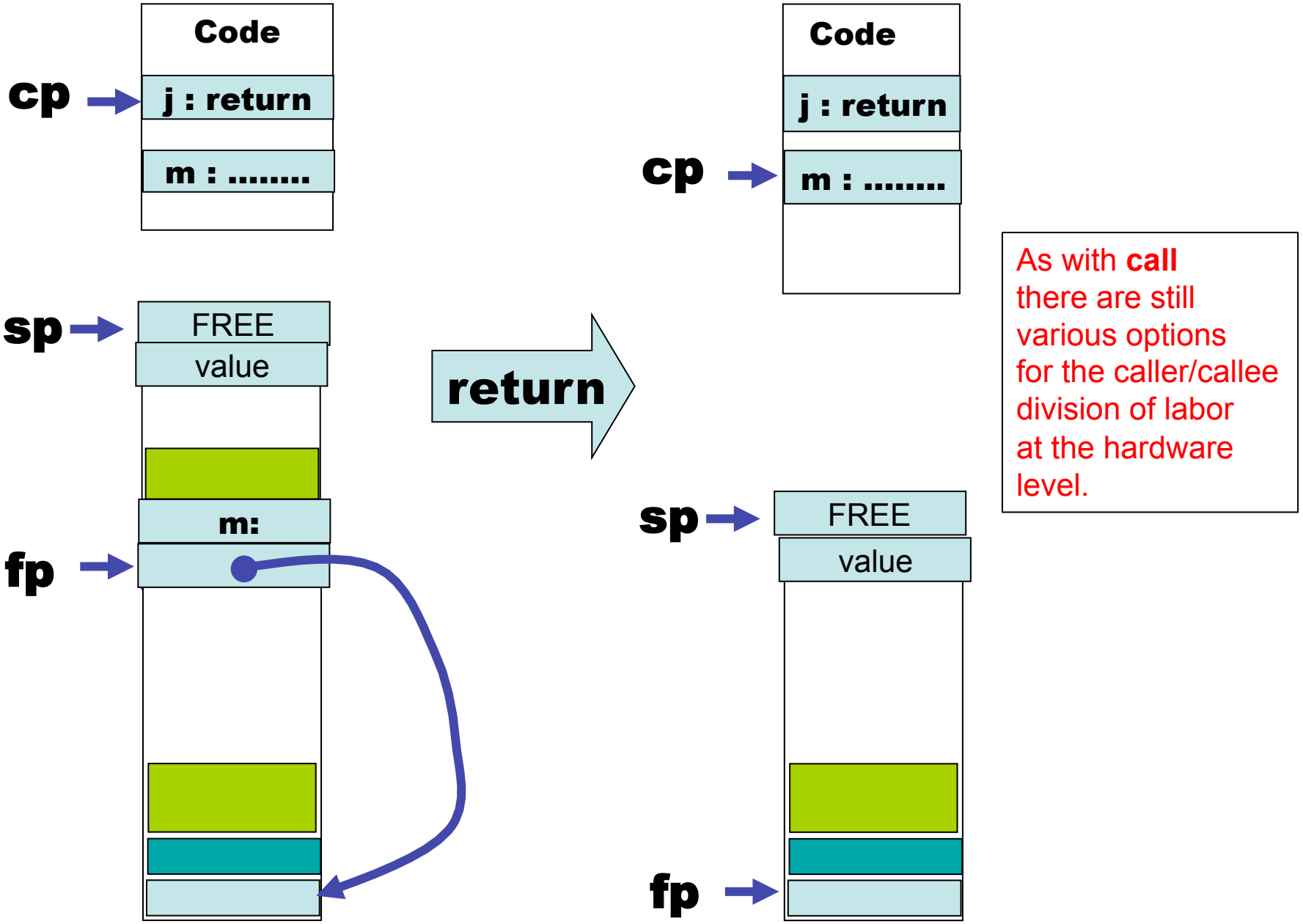
We need to
(1) push fp of caller on stack
(2) set current fp
(3) push return address on stack
(4) jump to address of caller

By using a VM we have “kicked the can down the road” for dividing the work between caller and callee.

It seems natural that 1-4 all be done by the caller.

But it is possible to imagine 1-3 being done by callee if it could find the return address somewhere (perhaps the hardware has a special kind of jump that saves the address of the next instruction in a special register ….)

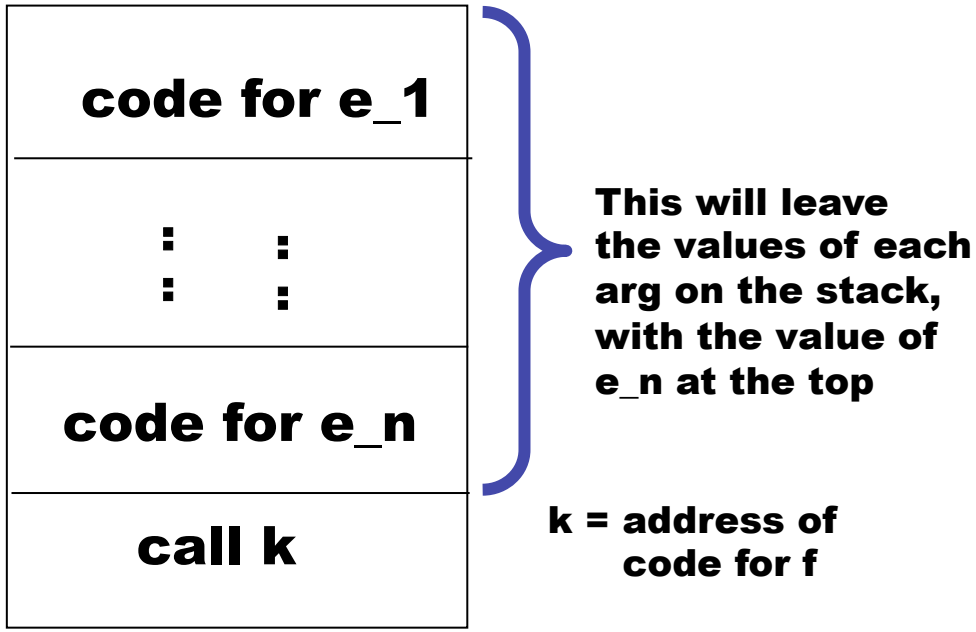
return



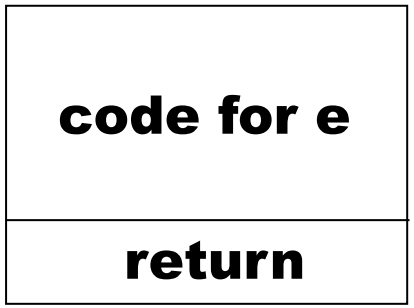
As with **call** there are still various options for the caller/callee division of labor at the hardware level.

Translation of function call

f(e_1, ..., e_n)

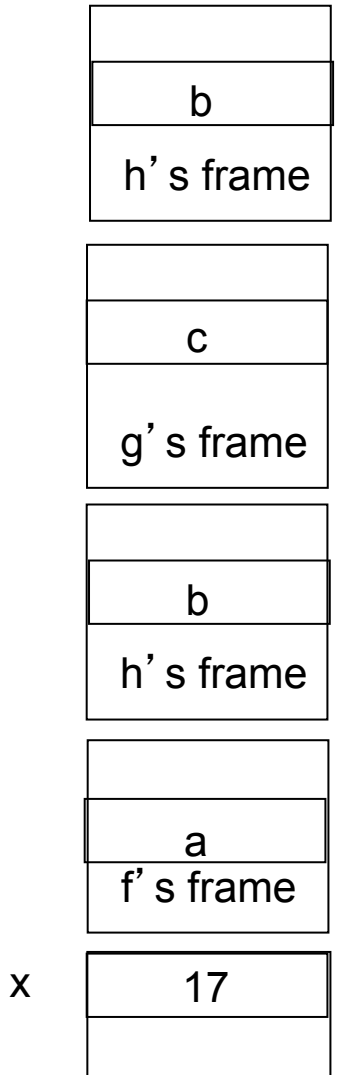


return e;



First step beyond simple scenario: Nested functions

```
fun f(x) {
   let a = …;
   fun h(y) {
     let b = …;
     fun g(w) {
        let c = …;
        if ..
        then return a;
        else return h(c)
     }
     return b + g(y);
   }
   return x + h(a);
}

f(17)
```



How this call to h access the value of a and x?

But first, a word about "dynamic binding" --- IT IS A VERY BAD IDEA

```
let val x = 1
    fun g(y) = x + y
    fun h(x) = g(x) + 1
in
   h(17)
end
```

With good old static binding we get 19.

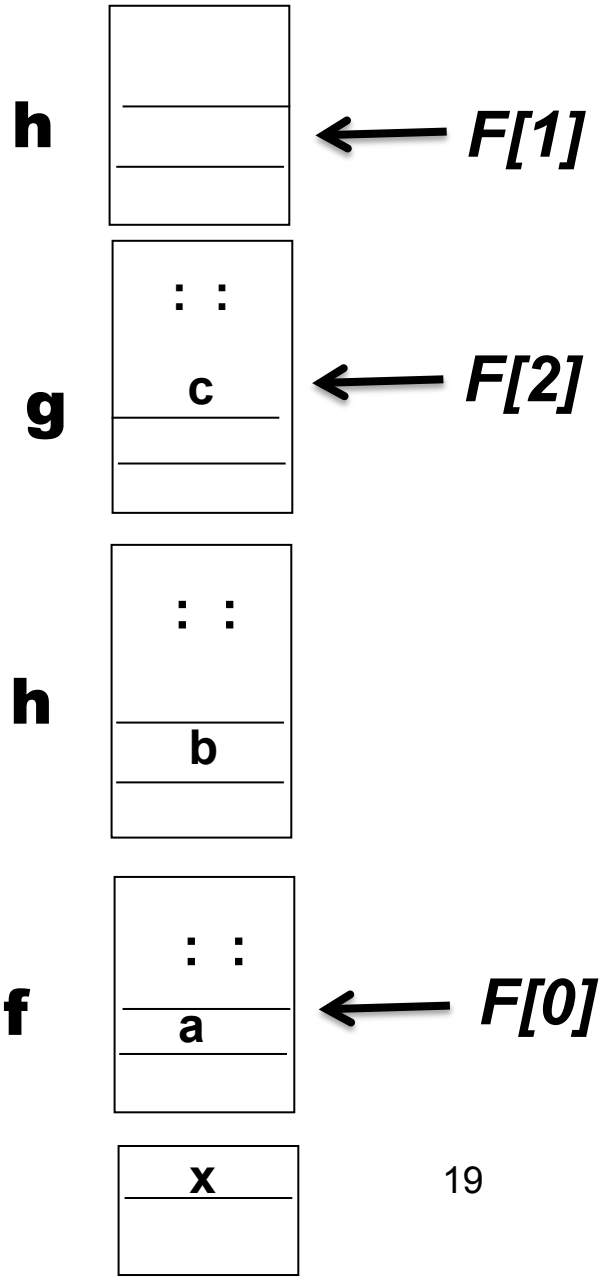
With insane dynamic binding we get 35.

We will stick to static binding …

Alternative 1: Dijkstra Displays

```
fun f(x) {                          Depth 0
   let a = …;
   fun h(y) {                       Depth 1
     let b = …;
     fun g(w) {                     Depth 2
        let c = …;
        if ..
        then return a;
        else return h(c)
     }
     return b + g(y);
   }
   return x + h(a);
}
```

Use an array **F[d]** to point at the most recent activation record at nesting depth d.



(+) at run-time only need a fixed number of indirections to find the value of a non-local variable

(-) slows down non-local variable access

Where to store The array **F**?

How is it managed?

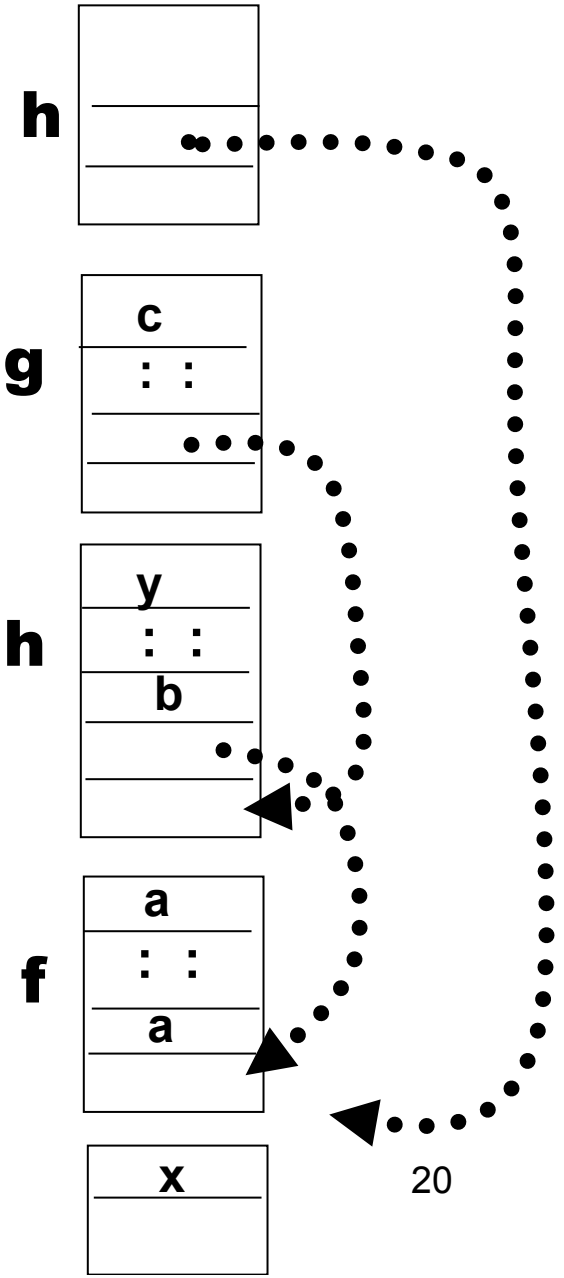
Alternative 2: Single Static Link per Frame

```
fun f(x) {                          Depth 0
   let a = …;
   fun h(y) {                       Depth 1
     let b = …;
     fun g(w) {                     Depth 2
        let c = …;
        if ..
        then return a;
        else return h(c)
     }
     return b + g(y);
   }
   return x + h(a);
}
```

Static frame pointer

(+) takes less time to set up and tear down.

(-) At run-time, need to "chase pointers" to find the value of a non-local variable.



If function *g* is at static nesting depth *i*, then use a single static link to the most recent frame for nesting depth *i-1*.

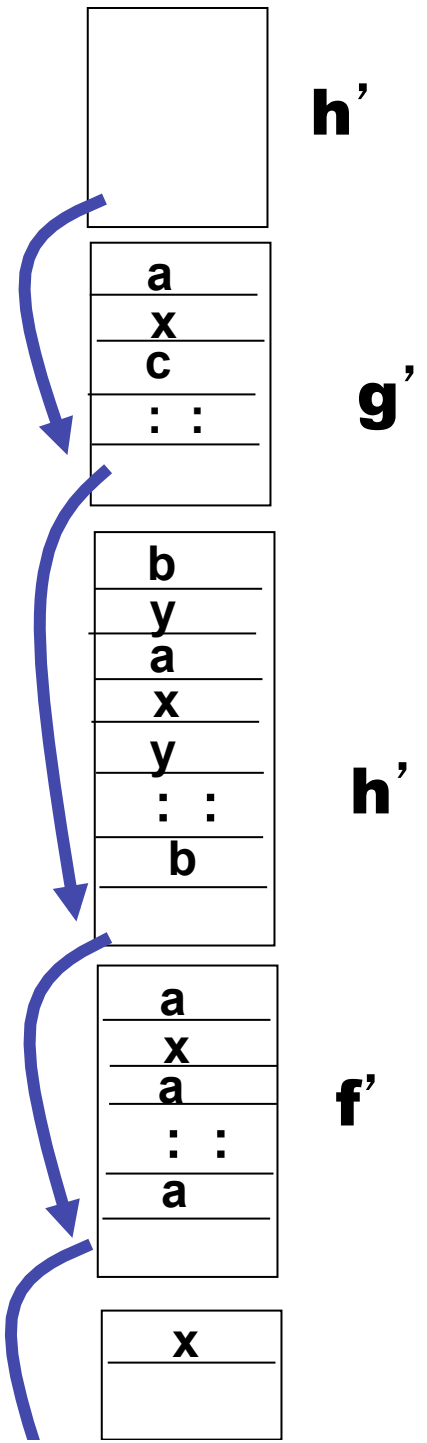
Alternative 3: "Lambda Lifting"

```
fun f(x) {
   let a = …;
   fun h(y) {
     let b = …;
     fun g(w) {
        let c = …;
        if ..
        then return a;
        else return h(c)
     }
     return b + g(y);
   }
   return x + h(a);
}

f(17)
```

```
fun g'(w, x, a, y, b) {
   let c = …;
   if ..
   then return a;
   else return h'(c, x, a )
}
fun h'(y, x, a) {
   let b = …;
   return b + g'(y, x, a, y, b)
}

fun f'(x) {
   let a = …;
   return x + h'(a, x, a);
}

f'(17)
```

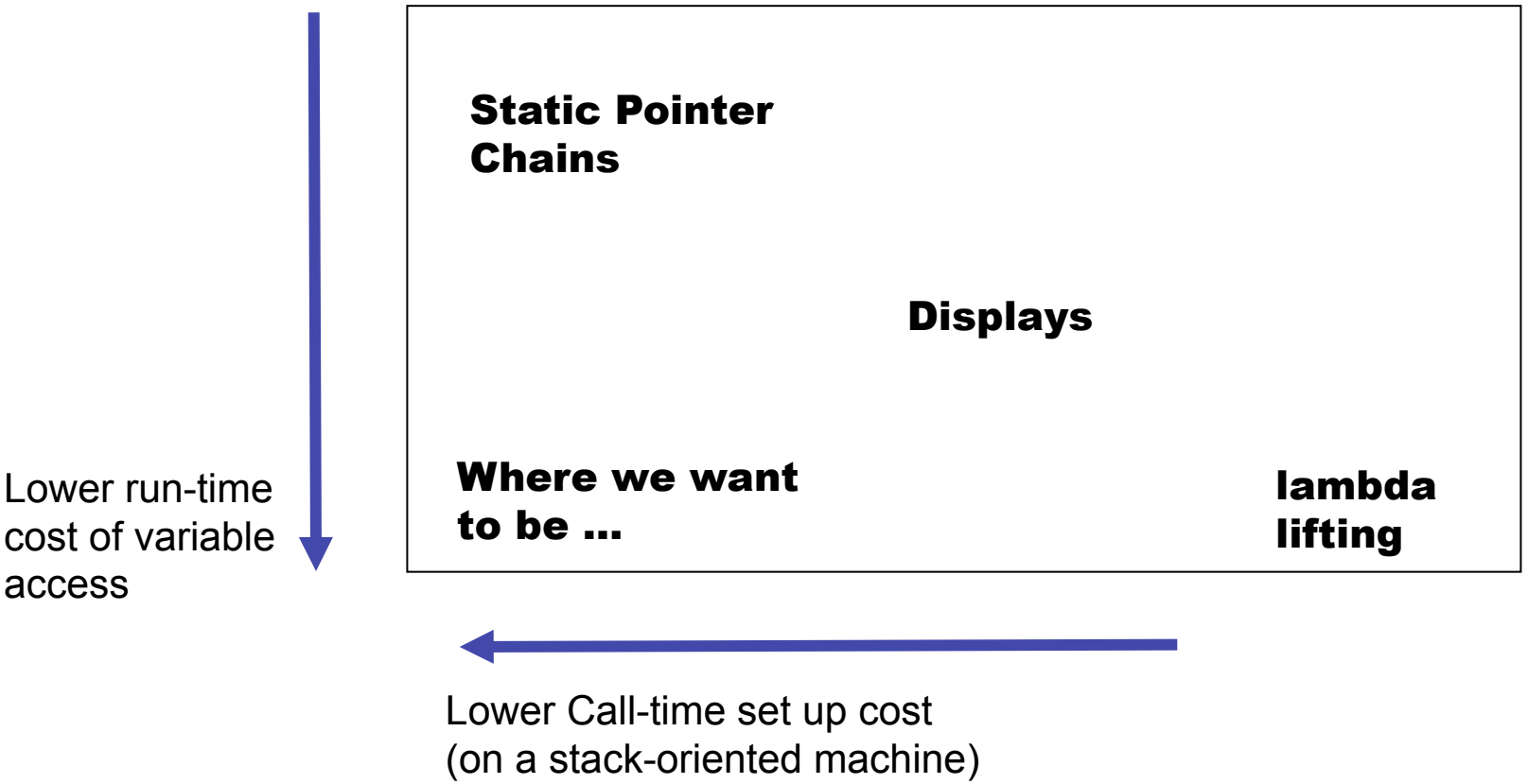
Stack Evaluation

```
fun g'(w, x, a, y, b) {
    let c = …;
    if ..
    then return a;
    else return h'(c, x, a)
}
fun h'(y, x, a) {
    let b = …;
    return b + g'(y, x, a, y, b)
}

fun f'(x) {
    let a = …;
    return x + h'(a, x, a);
}

f'(17)
```



Problem : can lead to a lot of duplication on the stack ….

A Classic Trade Off



What about functions-as-values?

```
fun f(a : int) : int -> int
{
   fun g(x :int) : int {return a + x;}
   return g;
}

let add21 : int -> int  = f(21);
let add17 : int -> int  = f(17);

add17(3) + add21(-1)
```

Oh NO! Our previous approaches no longer work!

The values associated with "a" have to outlive f's activation records!

Similar problem with the lifetime of reference cells

```
fun f(a : int) : int ref
{
   let b : int ref := a;
   return b;
}

let z : int ref = f(17);

!z
```

We need some way to store data that outlives the activation record in which it is created.

Solution: The “Heap” ….