Compiler Construction
Lent Term 2014
Lecture 5

- Block structure, simple functions
- The call stack, stack frames
- Caller and Callee
- A simple stack-oriented VM model
- Nested functions and possible modifications required

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

Caller and Callee

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

…

fun g(w, v) =
    w + f(v, v)
```

For this invocation of the function f, we say that g is the caller while f is the callee

Recursive functions can play both roles at the same time …

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.

But might there be a place for dynamic binding? Is there dynamic binding of some kind behind the raise/handle exception mechanism?

Mind the gap. 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.

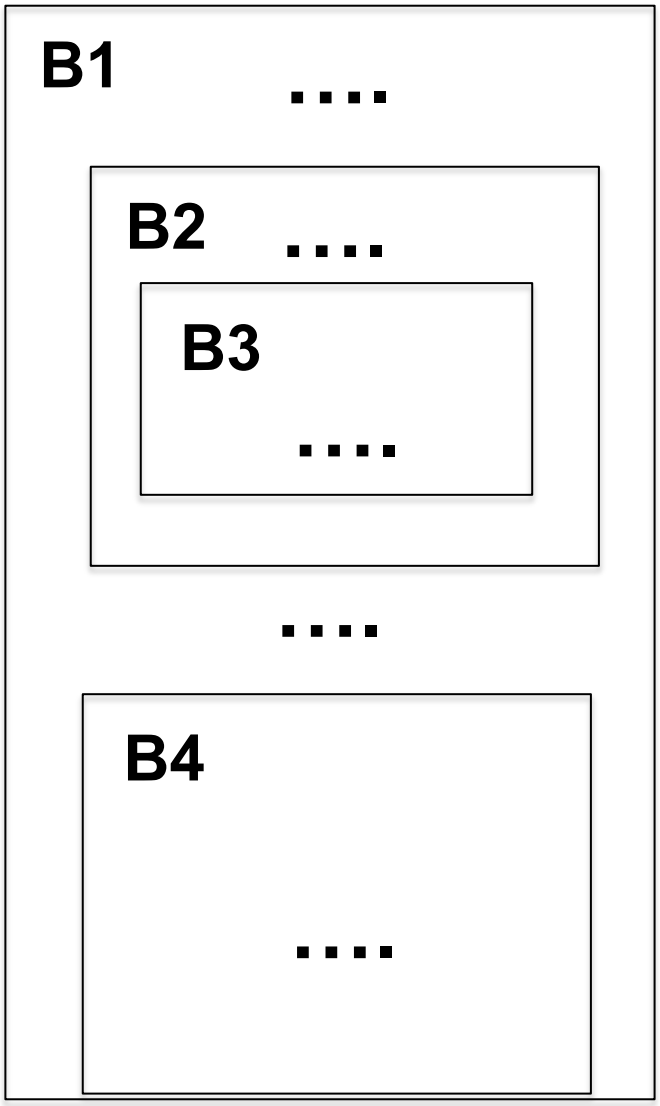
How do we allocate space for the values associated with the variables, and how do we find these values at run-time?

Block Structure (2)

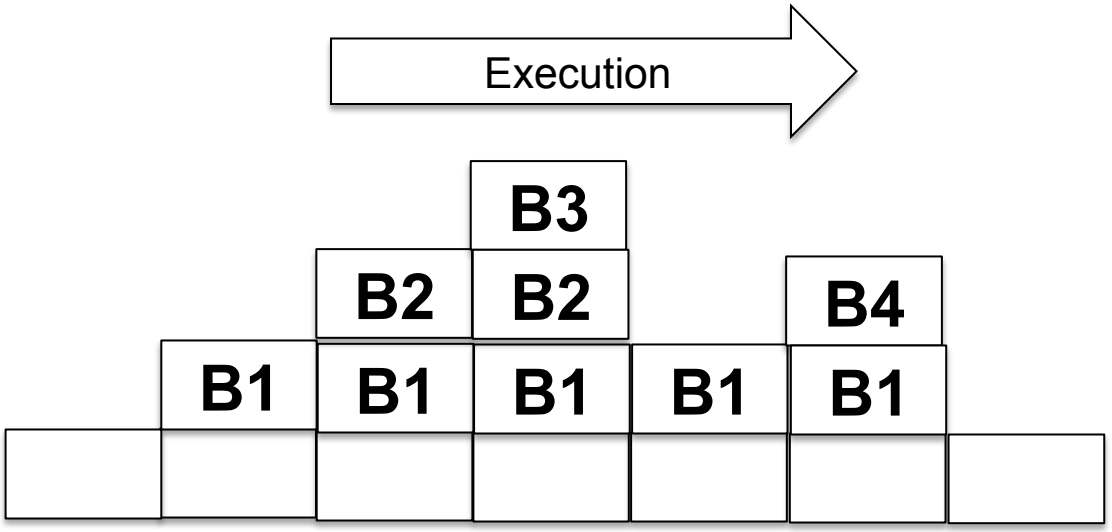
```
{ x : int;
  y : bool;
  ….
  if (e1) {
     z :int  = x + y;
     y :string = “hello”;
     if (e2) {
        u : int = size (y);
        v : int = u + z + x;

        … visible : x z y u v
     }

   … visible: x z y
  }

 … visible: x y
}
visible:
```

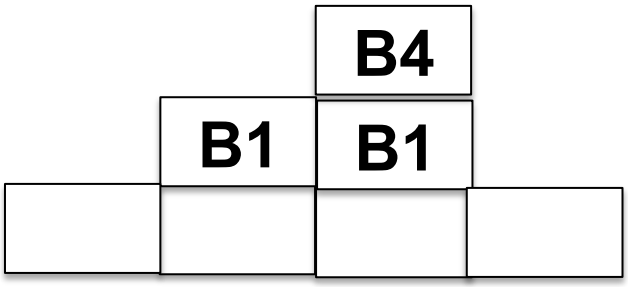
And we must
Correctly implement
Name binding rules.

Smells like LIFO, so use a stack



Possible run-time “activations” of these blocks

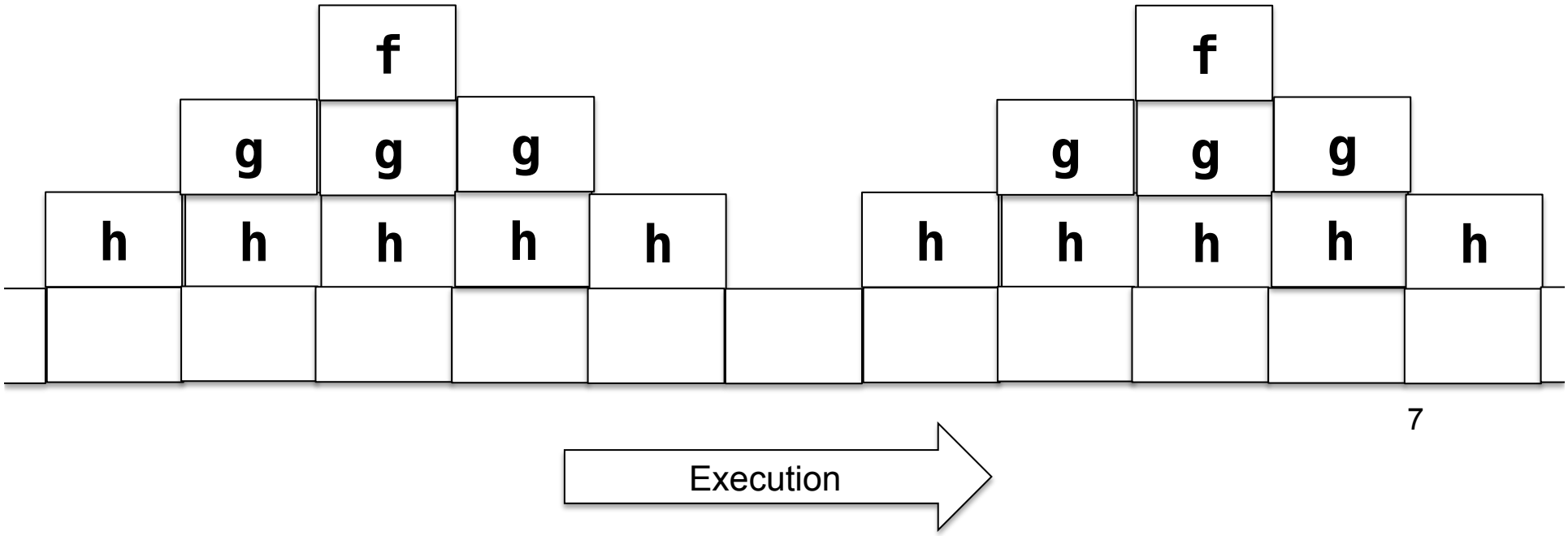




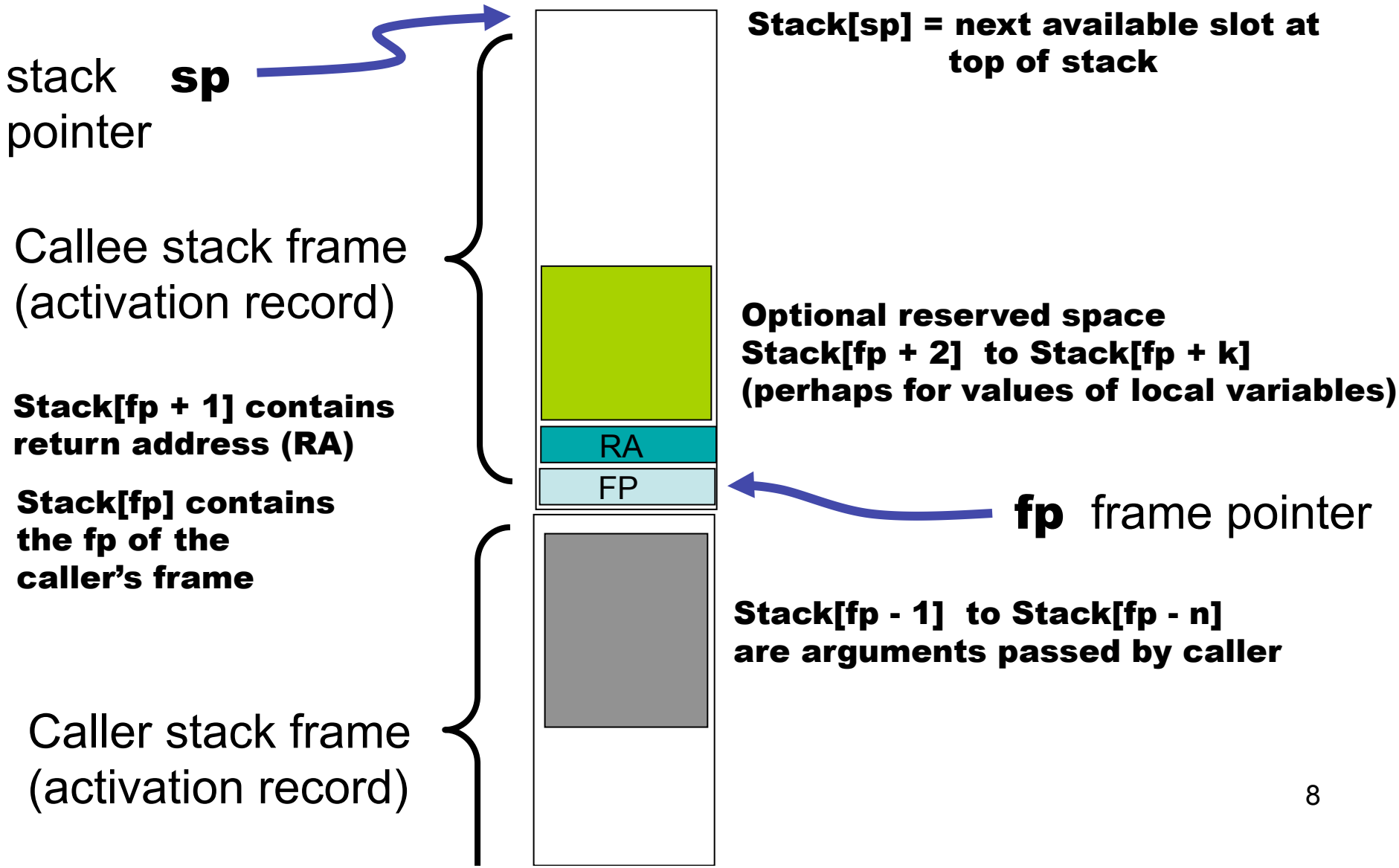
Same for calls to functions/procedures

```
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
```

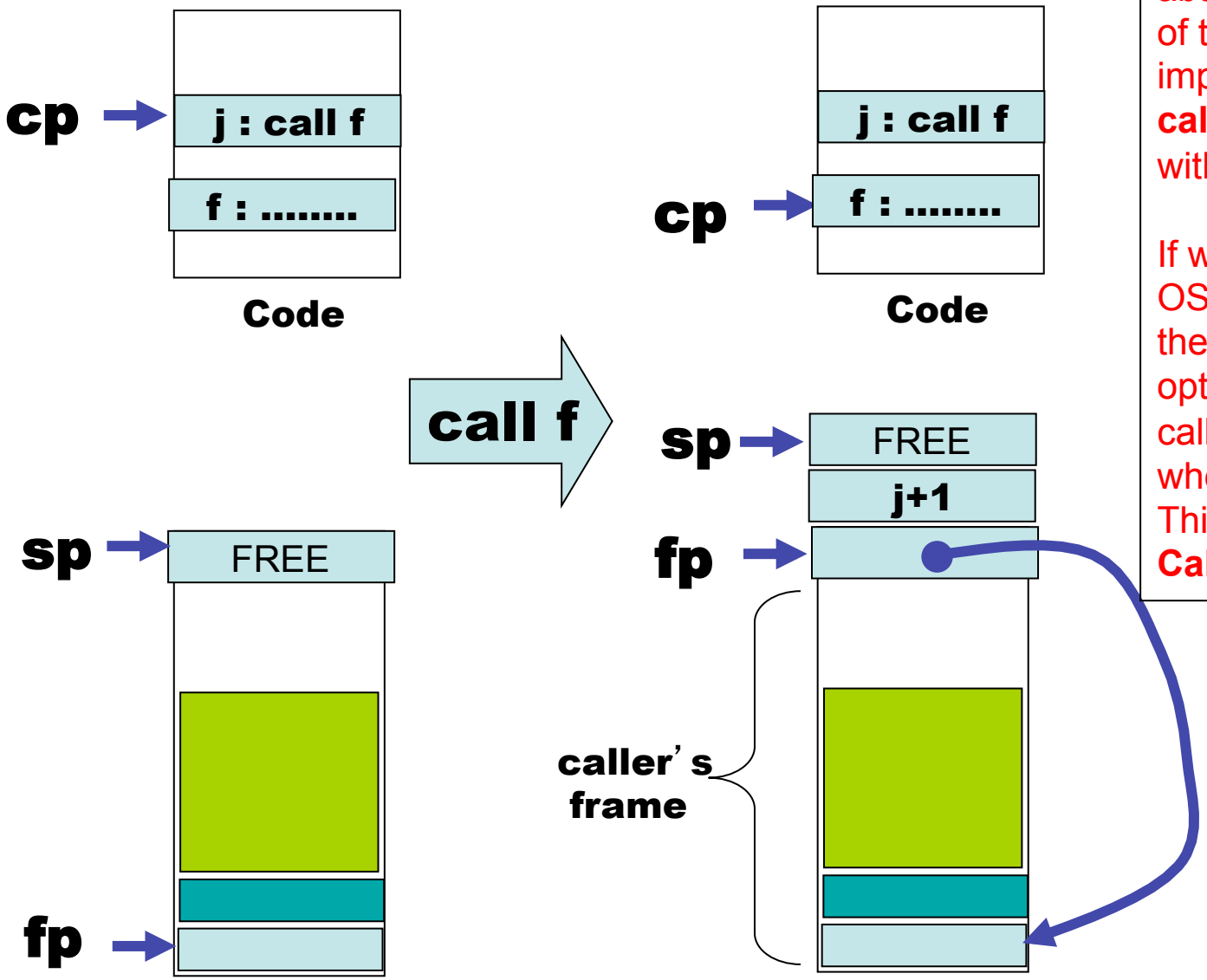
The run-time data structure is the call stack containing an activation record for each function invocation.



Structure of Our Simple VSM Call Stack



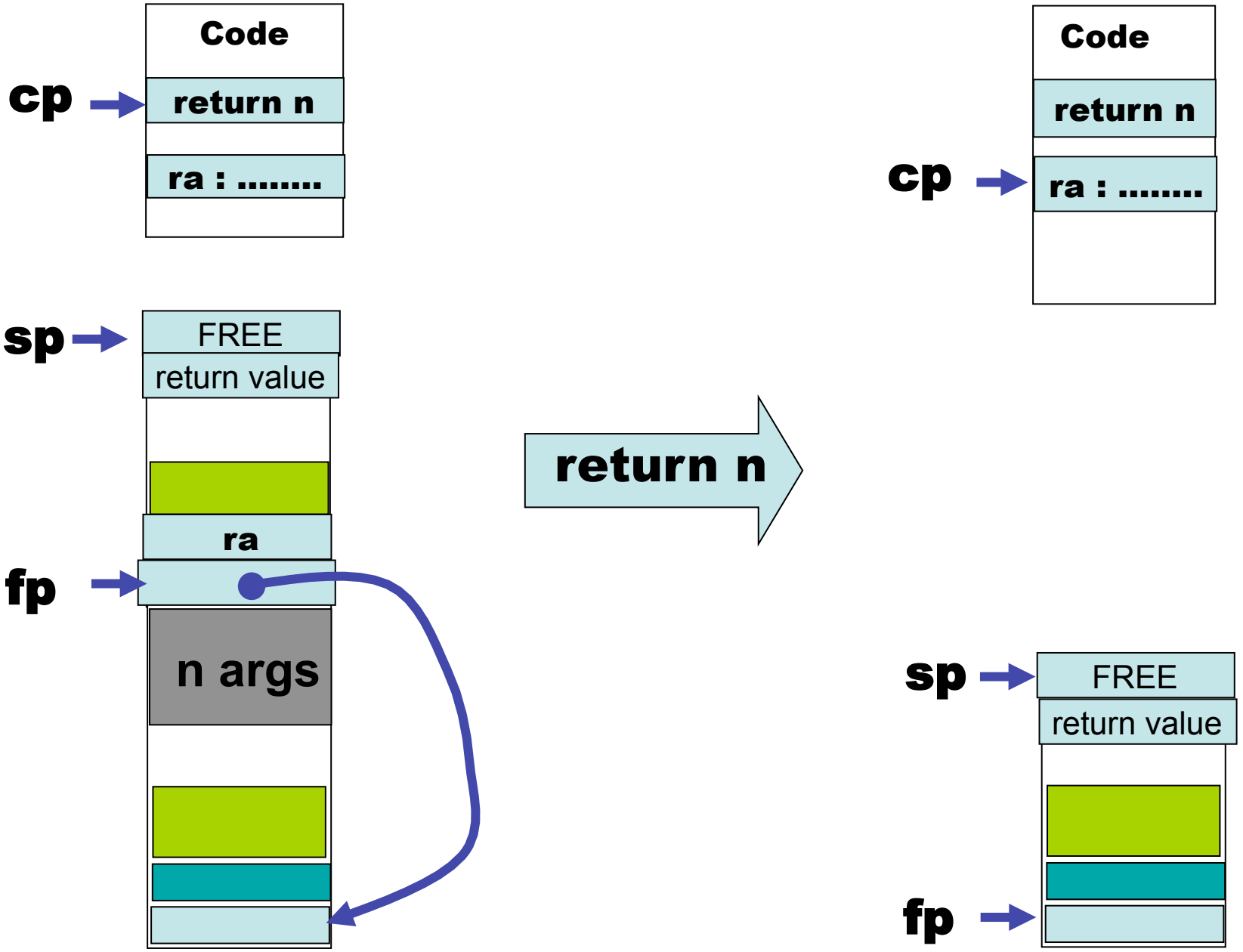
We can now design “high level” VSM commands



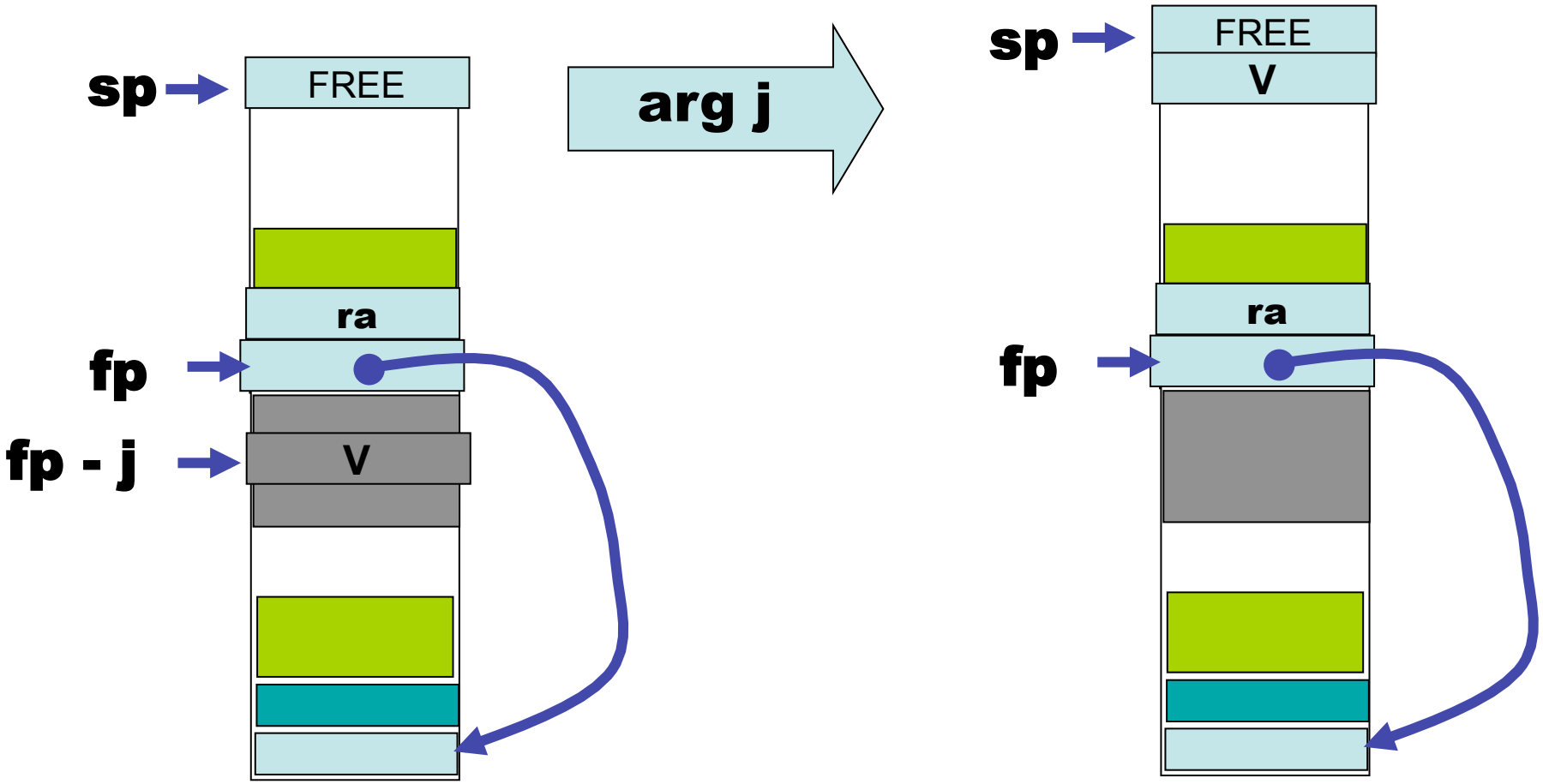
This is a VM-level abstraction. At the level of the program implementing the VM **call** is implemented with many instructions.

If we are targeting an OS/ISA pair (Unix/x86, …), then there are many more options as to who (caller or callee) does what when and where (registers or stack). This is captured in a **Calling Convention**.

Our “high level” VSM return

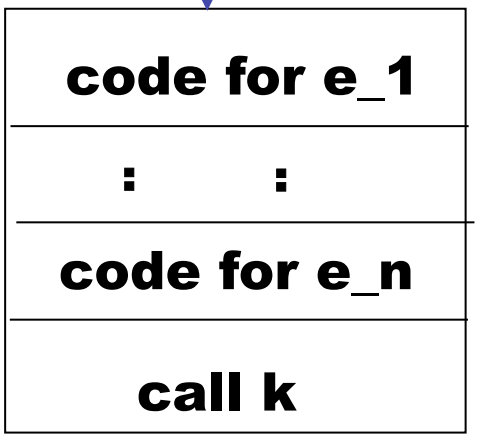


Access to argument values



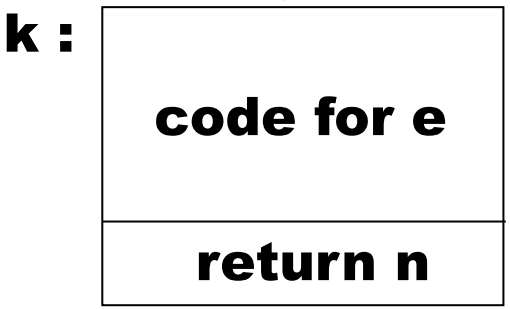
Translation of (call-by-value) functions

f(e_1, ..., e_n)



This will leave the values of each arg on the stack, with the value of e_n at the top. Here k is the address for the start of the code for f.

fun f(x_1, ..., x_n) = e

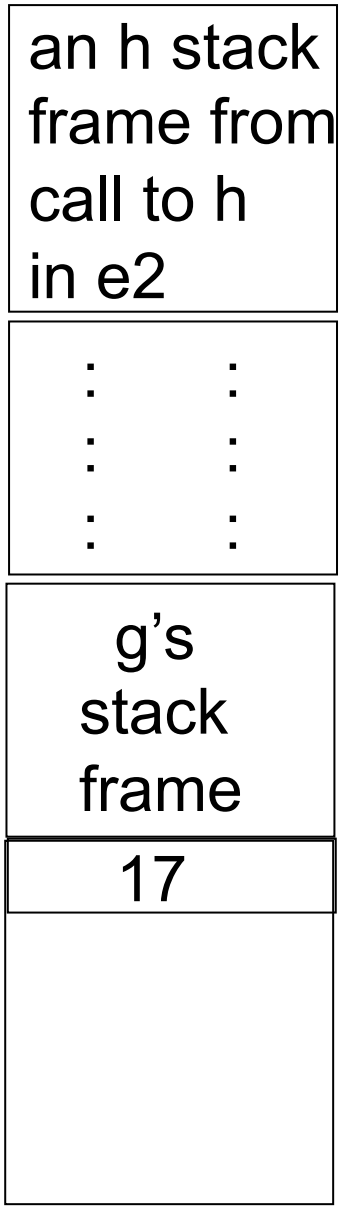


k is a location (address) where code for function f starts.

In code for e, access to variable x_i is translated to arg ((n – i) + 1).

What if we allow nested functions?

```
fun g(x) =
    fun h(y) = e1
    in e2 end
...
g(17)
...
```



How will the code generated from e1 find the value of x?

Approach 1: Lambda Lifting

```
fun g(x) =
    fun h(y) = e1
    in e2 end
...
g(17)
...
```

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

fun g(x) = e3
...
g(17)
...
```

Construct e3 from e2 by replacing each call h(e) with h(e, x)

(+) Keeps our VM simple
(+) Low variable access cost
(-) can duplicate many arg values on the stack

Local blocks ...

```
let x = e1 in e2 end
```

```
(fn x => e2) e1
```

OR

```
fun f(x) = e2 in f(e1) end
```

f is a fresh name

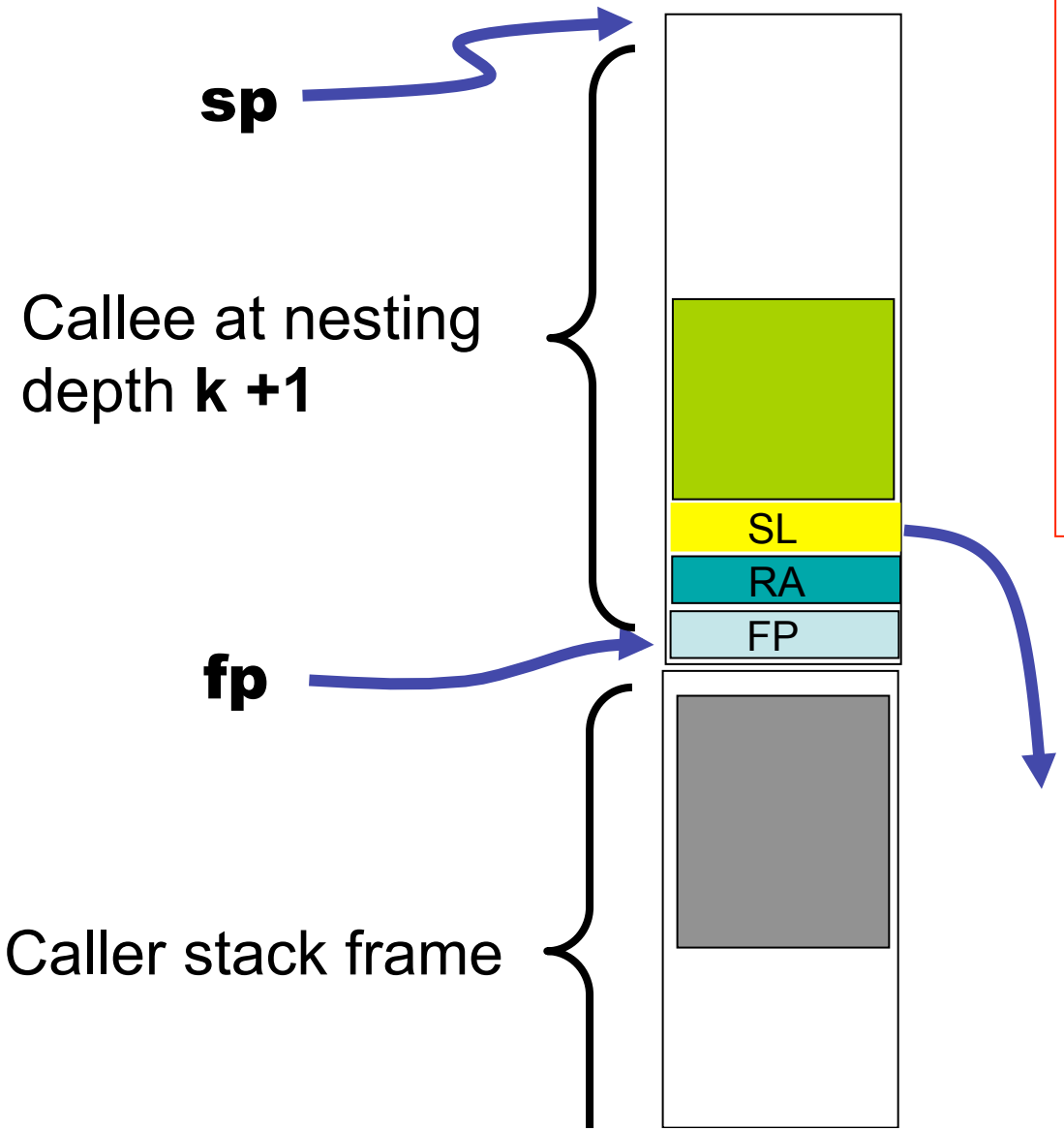
... can give rise to nested functions

```
fun g(x, y) =
    let z = e1(x, y)
    in e2(x, y, z) end
```

```
fun g(x,y) =
    fun f(z) = e2(x, y, z)
    in f(e1(x, y) ) end
```

f is a fresh name

Approach 2 : add Static Links to call stack



(+) takes less time to set up

(-) At run-time, need to "chase pointers" to find the value of a non-local variable. In the worst case a variable access at nesting depth j has to "chase" j static links to find the outer-most stack frame.

The static link points down to the closest frame of at nesting depth **k**

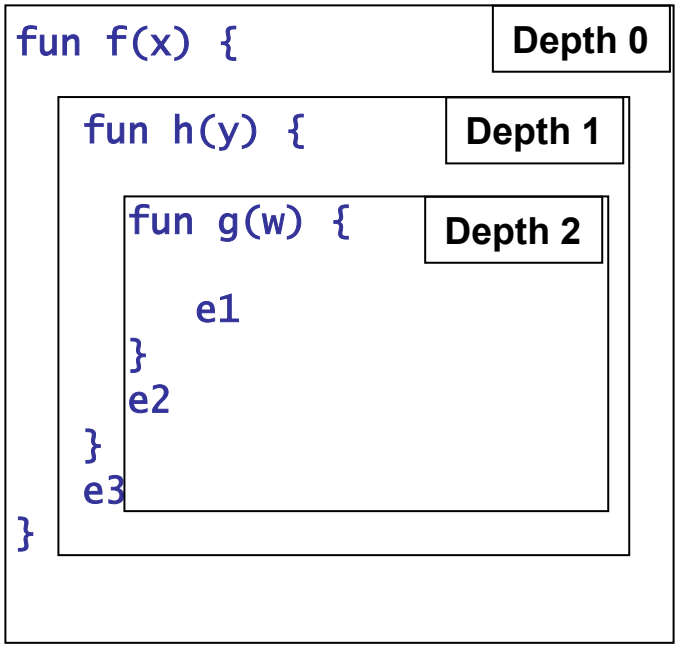
An exercise for you (to be resolved next lecture)

- What changes do static links require for the commands of our VSM?
- How do we change the compilation of these expressions?

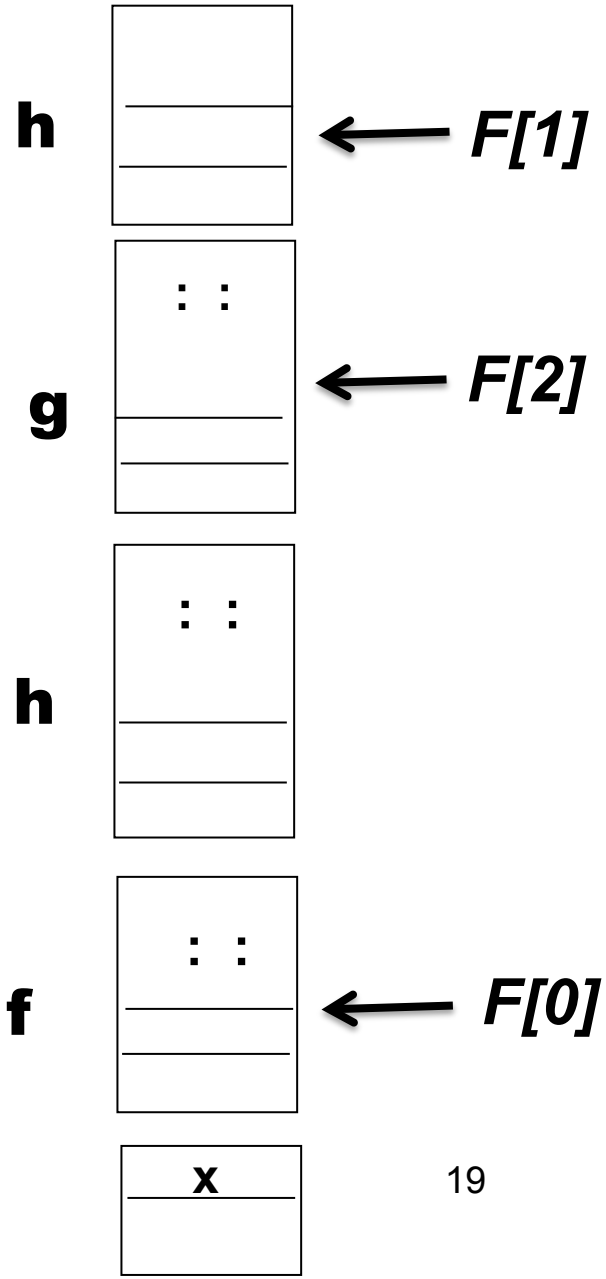
```
f(e_1, ..., e_n)
```

```
fun f(x_1, ..., x_n) = e
```

Approach 3: Dijkstra Displays



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

(-) maintaining display

Where to store **F**?

How is it managed?

A Classic Trade Off