Compiler Construction
Lent Term 2014
Lectures 6 and 7 (of 16)

- A closer look at static links
- Functions as "first class" values
  - Heap allocation of closures
- A few simple optimizations:
  - Inline expansion
  - Constant folding
  - Eliminating tail recursion

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

Nesting depth

```
fun b(z) = e

 fun g(x1) =
    fun h(x2) =
       fun f(x3) = e3(x1, x2, x3, b, g h, f)
       in
           e2(x1, x2, b, g, h, f)
        end
    in
        e1(x1, b, g, h)
   end
...
b(g(17))
...
```

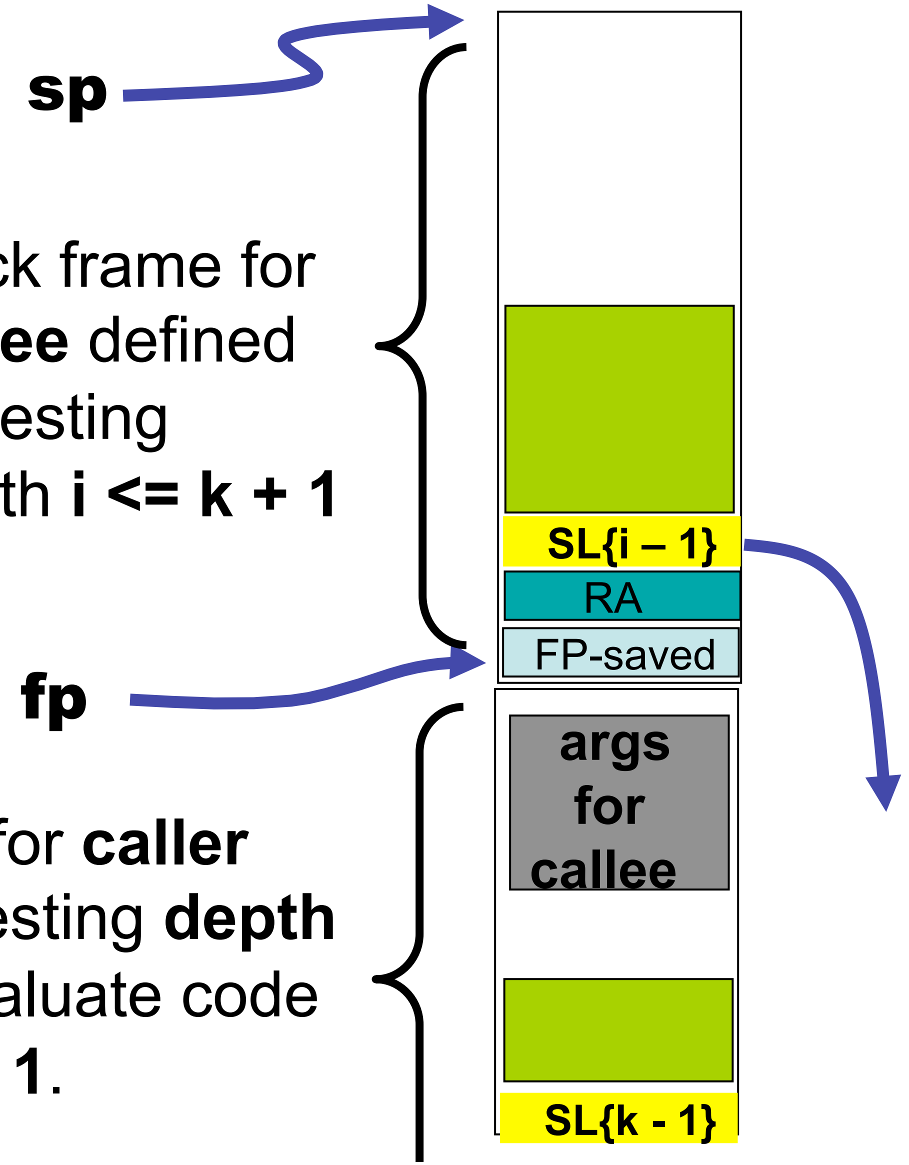
Nesting depth

code in big box is at nesting depth k

```
fun b(z) = e  nesting depth k + 1

 fun g(x1) =
   fun h(x2) =
     fun f(x3) = e3(x1, x2, x3, b, g h, f)        nesting depth k + 3
     in
        e2(x1, x2, b, g, h, f)
      end                                      nesting depth k + 2
   in
      e1(x1, b, g, h)
  end                                        nesting depth k + 1
...
b(g(17))
...
```

Function g is the **definer** of h. Functions g and b must share a definer defined at depth k-1

Recall : call stack augmented with Static Links (here SL{d} means a static link pointing at most recent frame of the definer at depth d)

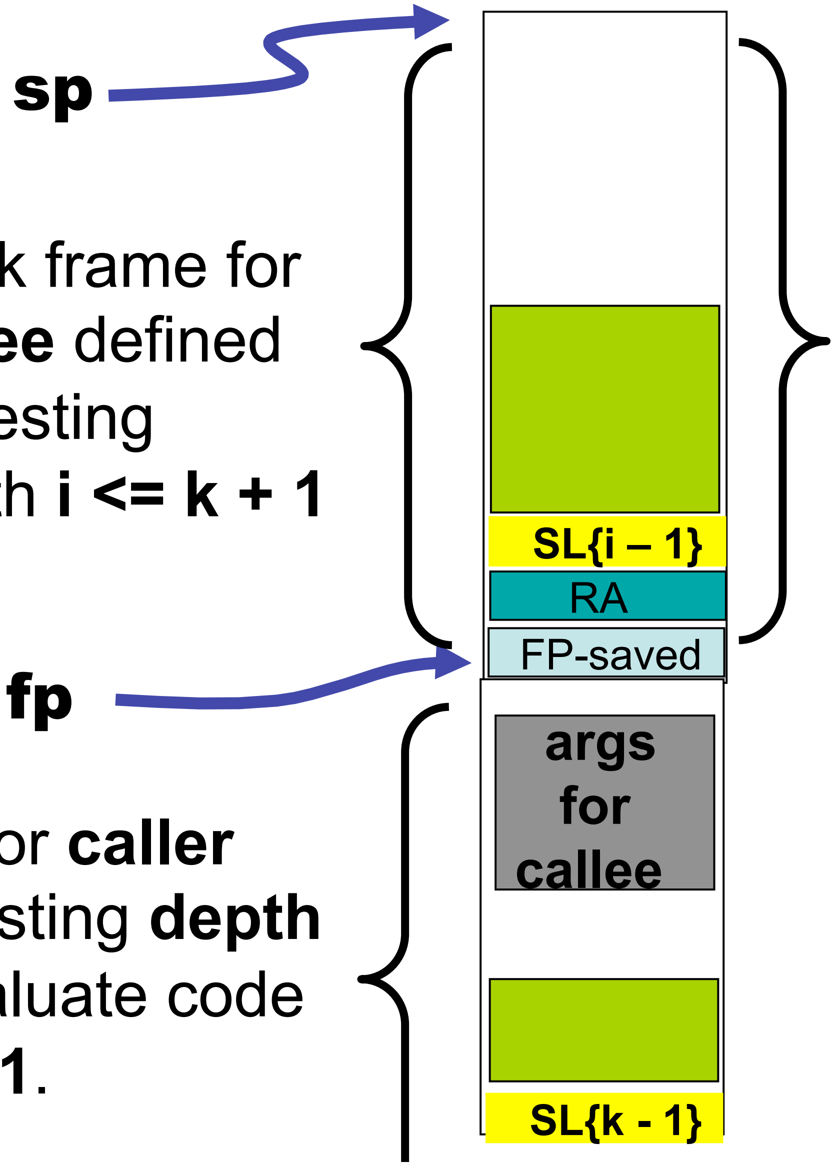


stack frame for **callee** defined at nesting depth **i <= k + 1**

The static link points down to the closest frame **of definer** at nesting depth **i - 1**

stack frame for **caller** defined at nesting **depth k** used to evaluate code at **depth k + 1**.

Finding argument values



stack frame for **callee** defined at nesting depth **i <= k + 1**

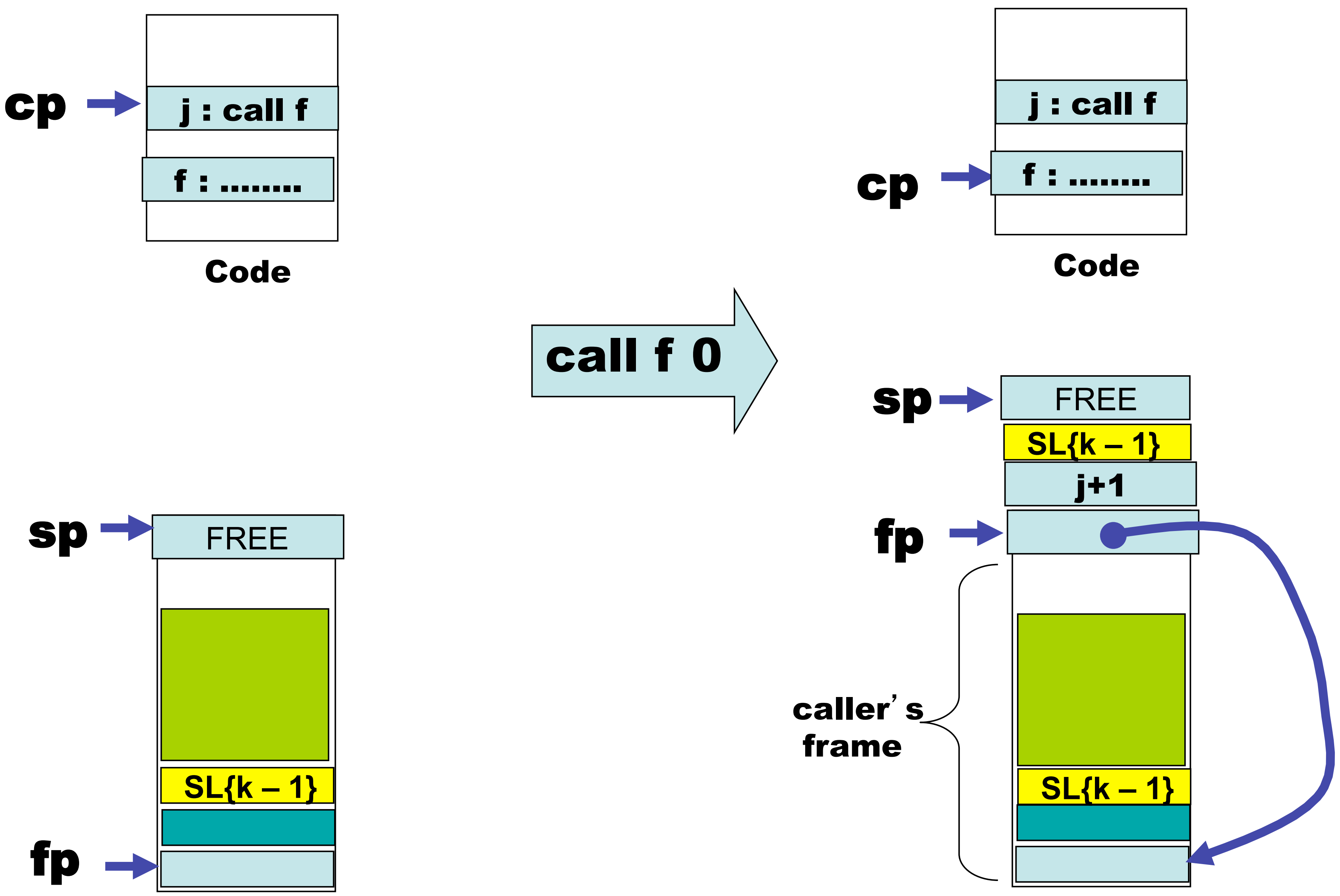
stack frame for **caller** defined at nesting **depth k** used to evaluate code at **depth k + 1**.

The code using this stack frame is defined at depth **i + 1.** Each program variable is associated with two values : (static distance, frame offset).

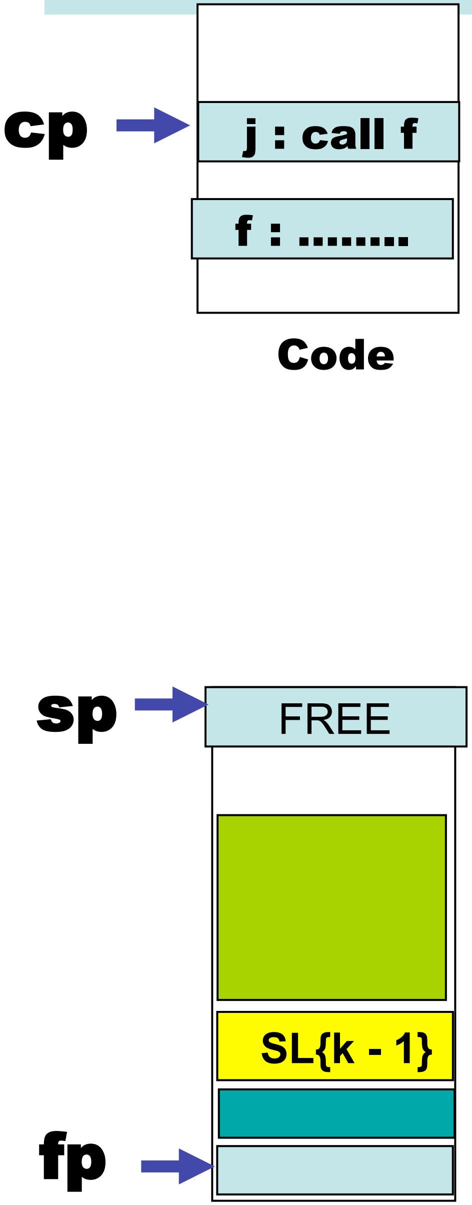
These args are at static distance 0. That is,

SL{i} = fp

caller and callee at same nesting depth k

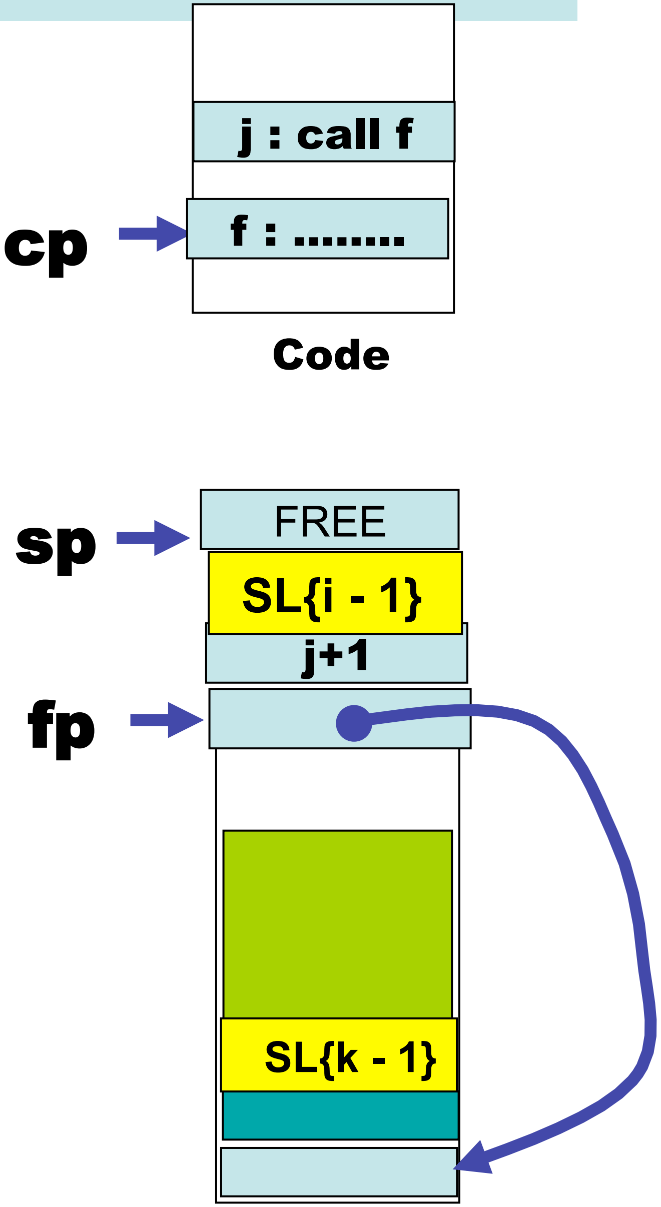


caller at depth k and callee at depth i < k

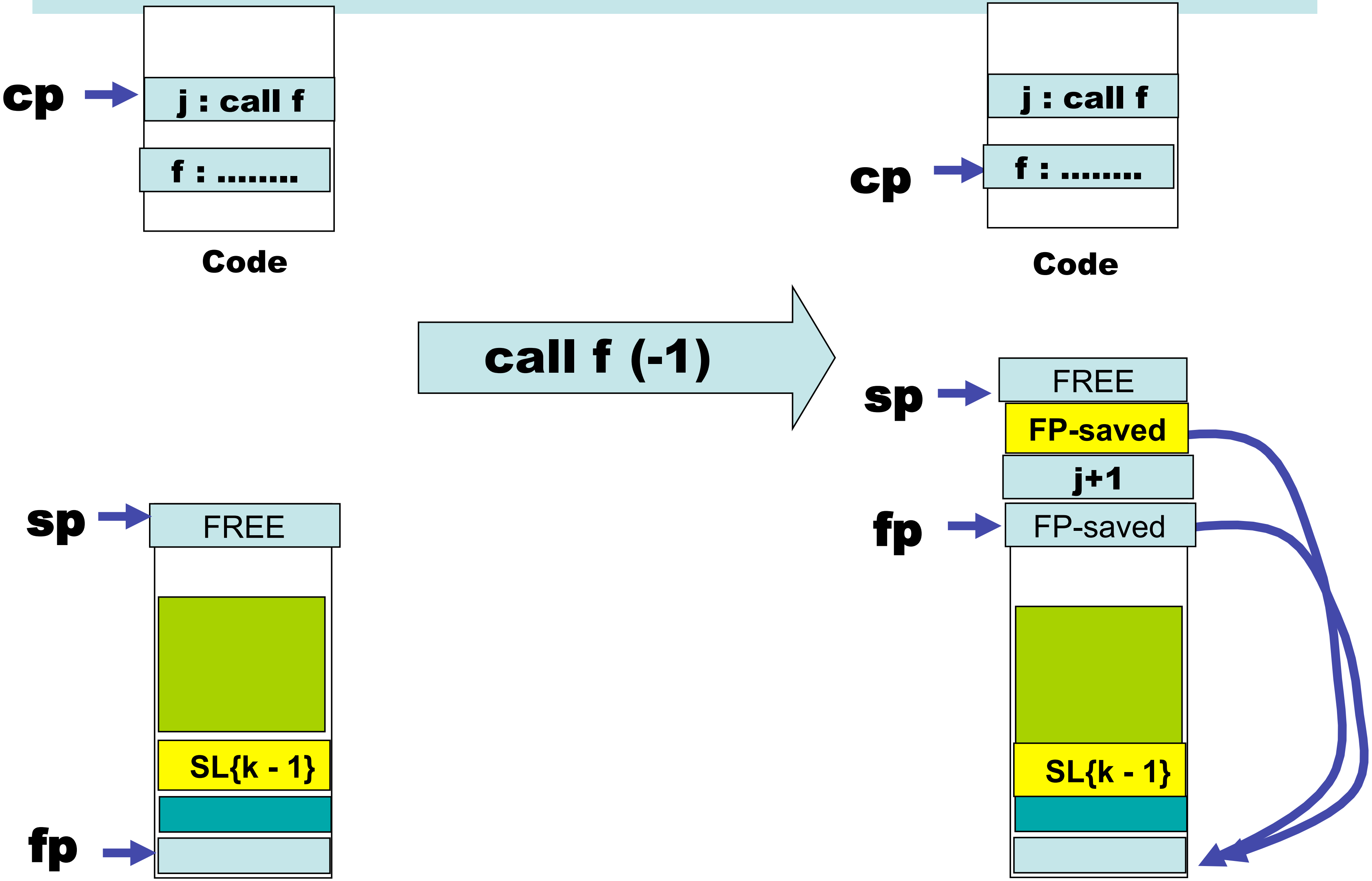


call f (k - i)

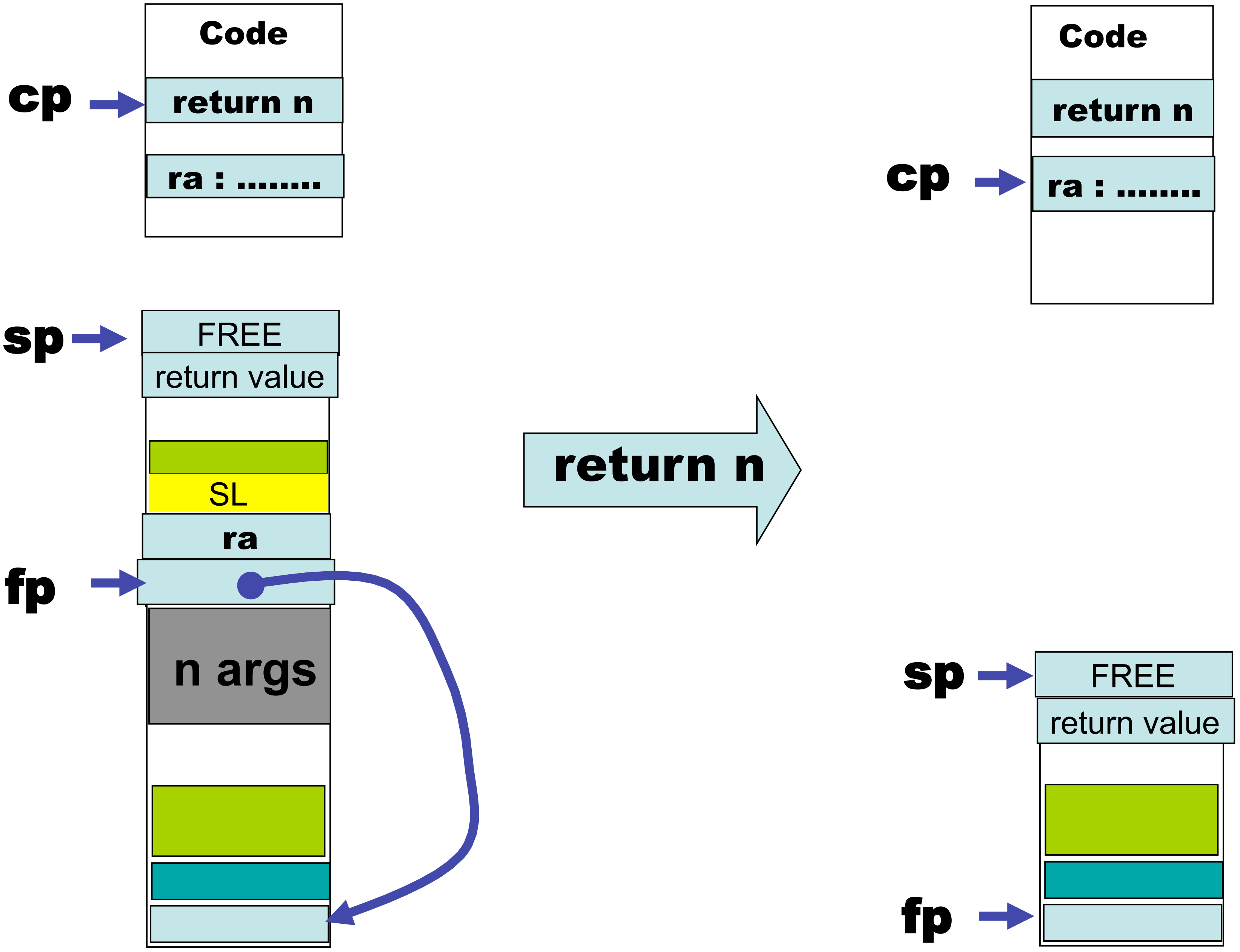
```
p := !(fp + 2);
for c = 1 to k – i
{
    p := !(p + 2);
}
SL{i-1} := p;
```



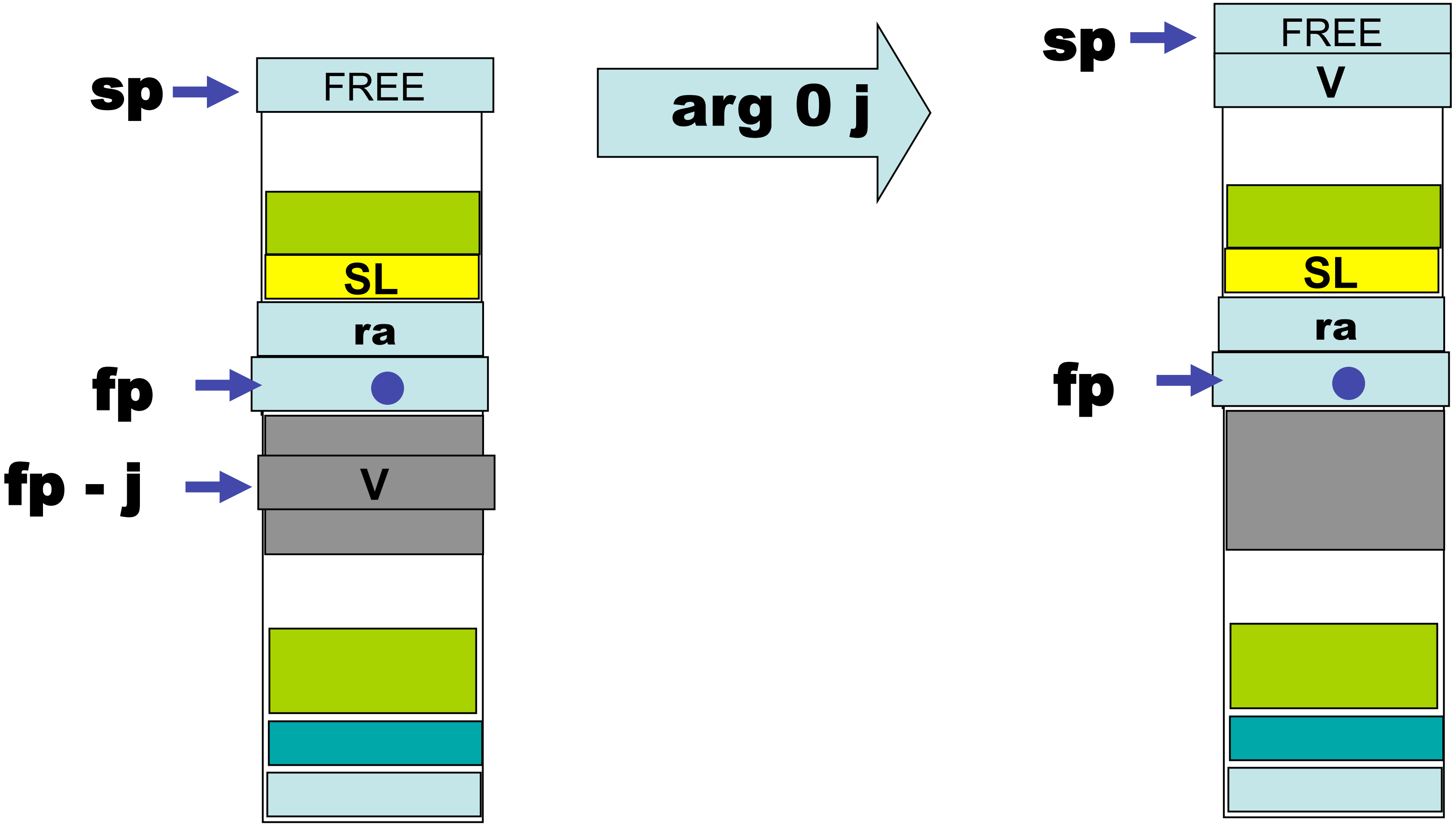
caller at depth k and callee at depth k + 1 (example from slide 3: caller = g, callee = h)



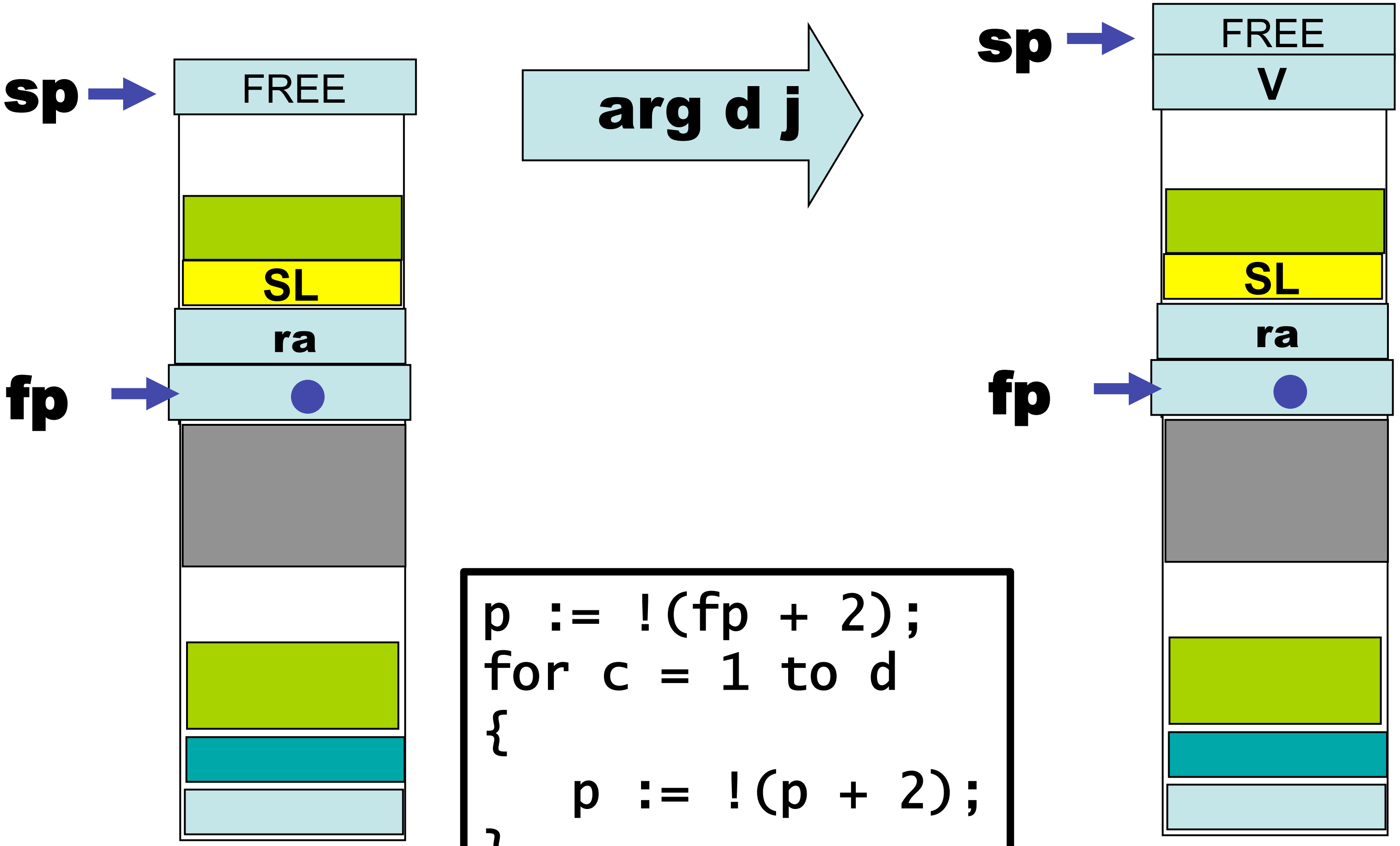
No change to return



Access to argument values at static distance 0



Access to argument values at static distance d, 0 < d



```
p := !(fp + 2);
for c = 1 to d
{
   p := !(p + 2);
}
v := !(p - j);
```

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” ….

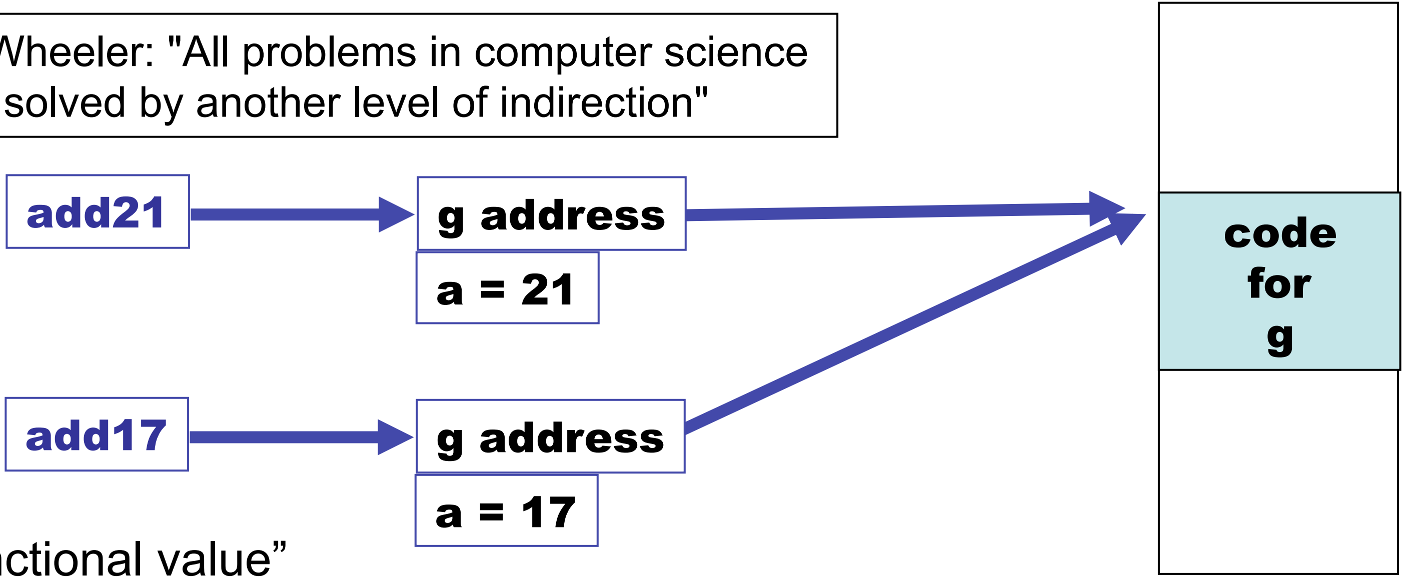
Idea --- a functional value is a pointer to a “closure”

```
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)
```

Problem: in the simple call stack the argument “a” (needed in body of g) does not survive the destruction of f’s activation record.

A closure is a record containing the address of a function AND the values of its free variables

David Wheeler: "All problems in computer science can be solved by another level of indirection"



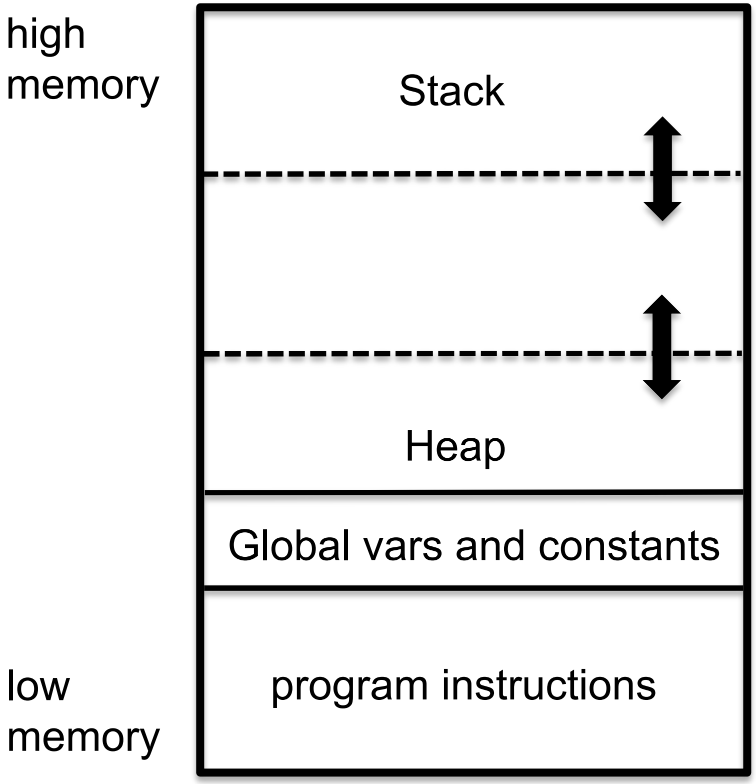
A “functional value” is a pointers to a closure.

Where should these closures be stored??

Code array

The Heap

Rough schematic of traditional layout in (virtual) memory.



The heap is used for dynamically allocating memory. Typically either for very large objects or for those objects that are returned by functions/procedures and must outlive the associated activation record.

In languages like Java and ML, the heap must be managed automatically ("garbage collection")

Return to example: How do functional values find their free-var 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)
```

```
fun g(x, c) {return !(c+1) + x;}

fun f(a) {return ALLOCATE_CLOSURE (g, [a]);}

let add21 = f(21);
let add17 = f(17);

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

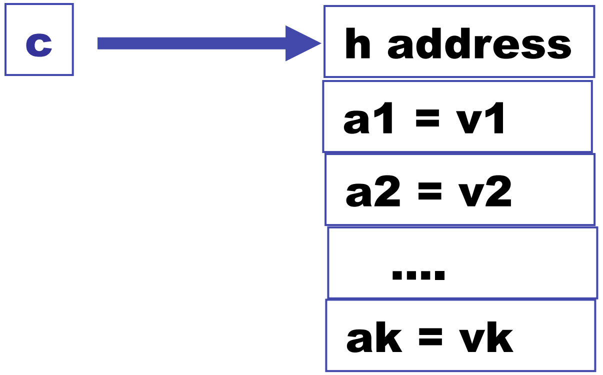
A possible intermediate representation

ALLOCATE_CLOSURE returns a pointer to the heap.

INVOKE_CLOSURE ?

Return to example: How do functional values find their free-var values?

```
fun g(x, c) {return !(c+1) + x;}

fun f(a) {return ALLOCATE_CLOSURE (g, [a]);}

let add21 = f(21);
let add17 : = f(17);

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



c

h address

a1 = v1

a2 = v2

....

ak = vk

`INVOKE_CLOSURE(c, u1, …, un)`

- Push arguments ui on stack
- Push c on stack
- Call h:
  - Build activation record for h
  - Body of h must access non-local vars using indirection through c.

Another example

```
fun f(a : int) : int -> int
{
   fun g(x :int) : int {return a + x;}
   fun h(x :int) : int {return a * x;}
   if a < 20 then return g else return h;
}

let f21 : int -> int  = f(21);
let f17 : int -> int  = f(17);

f17(3) + f21(-1)
```

Closure conversion (similar to “lambda lifting”)

```
fun f(a)
{
   fun g(x) {return a + x;}
   fun h(x) {return a * x;}
   if a < 20 then return g else return h;
}
```

```
fun g(x, c) {return !(c+1) + x;}
fun h(x, c) {return !(c+1) * x;}
fun f(a) {
 if a < 20
 then return ALLOCATE_CLOSURE (g, [a])
 else return ALLOCATE_CLOSURE (h, [a]);
}
```

A simple optimization with functions : Inline expansion

```
fun f(x) = x + 1
fun g(x) = x - 1
…
…
fun h(x) = f(x) + g(x)
```

inline f and g

```
fun f(x) = x + 1
fun g(x) = x - 1
…
…
fun h(x) = (x+1) + (x-1)
```

(+) Avoid building activation records at runtime

(-) May lead to “code bloat” (apply only to functions with “small” bodies?)

Question: if we inline all occurrences of a function, can we delete its definition from the code?

What if it is needed at link time?

Be careful with variable scope

Inline g in h

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

NO

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

YES

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

Constant propagation and constant folding

```
let x = 2
let y = x - 1
let z = y * 17
```

```
let x = 2
let y = 2 - 1
let z = y * 17
```

```
let x = 2
let y = 1
let z = y * 17
```

```
let x = 2
let y = 1
let z = 1 * 17
```

```
let x = 2
let y = 1
let z = 17
```

Propagate constants and evaluate simple expressions at compile-time

Note : opportunities are often exposed after inline expansion!

David Gries : “Never put off till run-time what you can do at compile-time.”

But be careful

How about this?

Replace

x * 0

with

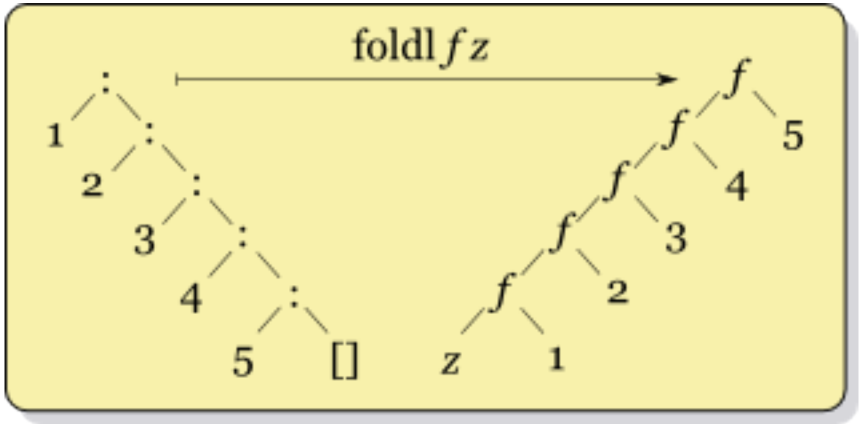
0

OOPS, not if x has type float!

NAN*0 = NAN,

Tail recursion

A recursive function exhibits tail recursion if on all recursive branches the last thing it does is call itself.



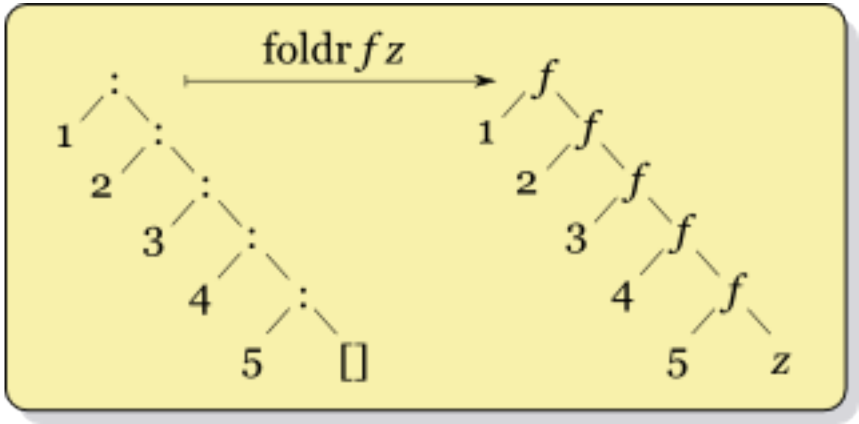
```
fun foldl f e []       = e
  | foldl f e (x::xr) = foldl f (f(x, e)) xr
```

We should be able to compile this to a LOOP in order to avoid constructing many activation records at runtime.
Exercise : How?

The ultimate tail-recursive function

```
fun while (c, b) =
    if c()
    then while (c, b ())
    else ()
```

Of course not all recursive functions are tail recursive...



```
fun foldr f e []       = e
  | foldr f e (x::xr) = f(x, foldr f e xr)
```

The “last thing” this function does is call **f**, not **foldr**

Sometimes recursive functions can be rewritten to tail recursive versions

```
fun sum_list [] = 0
  | sum_list (x::rest) = x + (sum_list rest)
```

```
fun sum_list il =
  let fun auxiliary carry [] = carry
        | auxiliary carry (x :: rest) =
            auxiliary (x + carry) rest
  in auxiliary 0 il end
```

Exercise : Think about trying to automate this kind of transformation in a compiler.