Compiler Construction
Lent Term 2015
Lecture 6 (of 16)

- Alternatives for managing access to non-local variables
  - Lambda lifting
  - Static links
  - Heap allocated closures

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

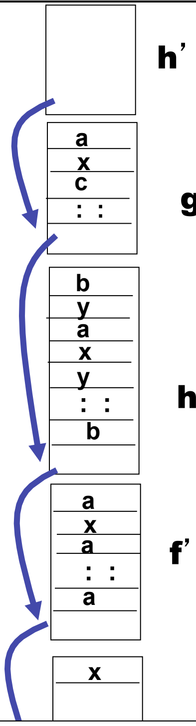
Alternative 1: “Lambda Lifting”

```
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)
```

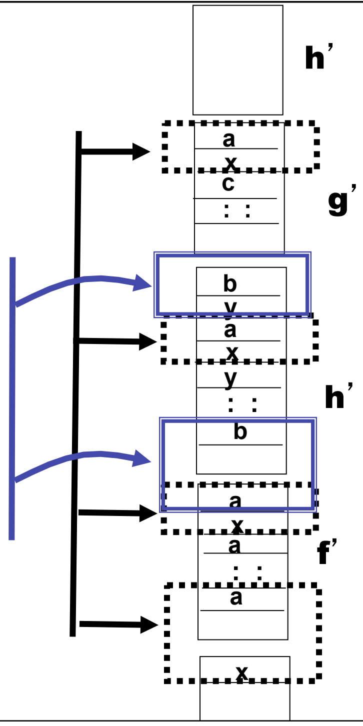
Stack Evaluation

```
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: a lot of Duplication!

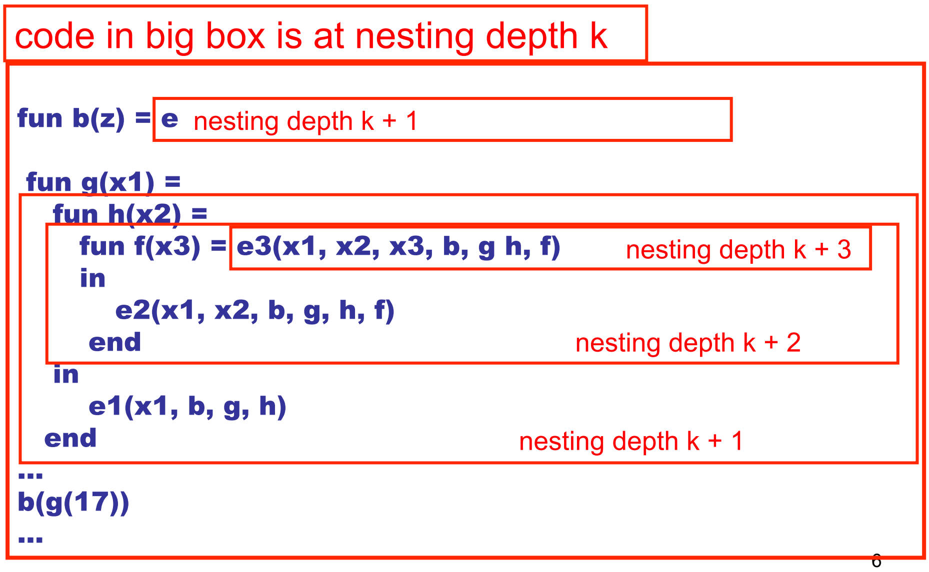
```
fun g’(w, x, a, y, b) {
   let c = …;
   if ..
   then return a;
   else return h’(c, x, a)
}
fun h’(y, x, a) {
   let b = …;
   return b + g’(y, x, a, y, b)
}

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

f’(17)
```



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

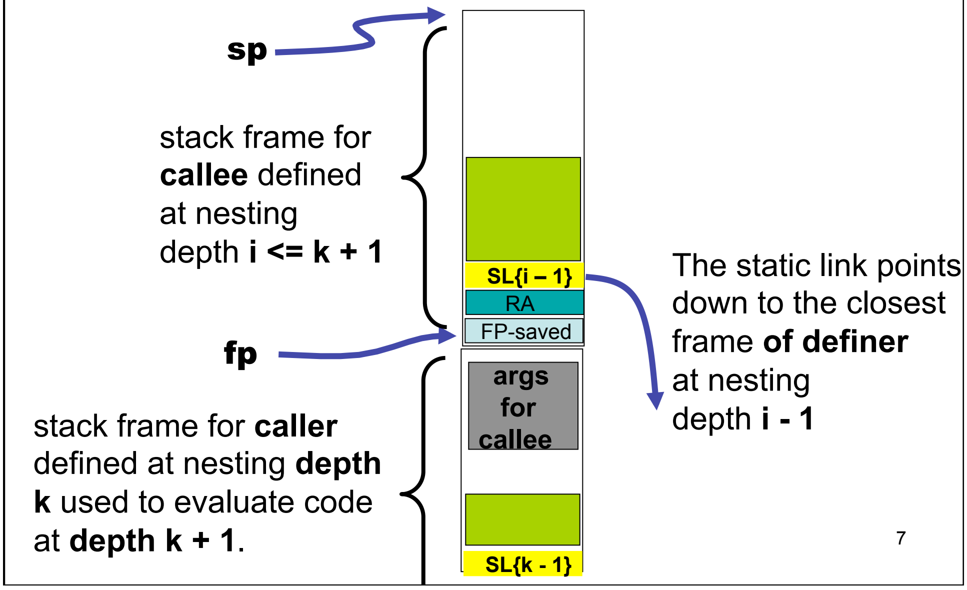


code in big box is at nesting depth k

```
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

Alternative 2: Augment stack frames with Static Links (here SL{d} means a static link pointing at most recent frame of the definer at depth d)

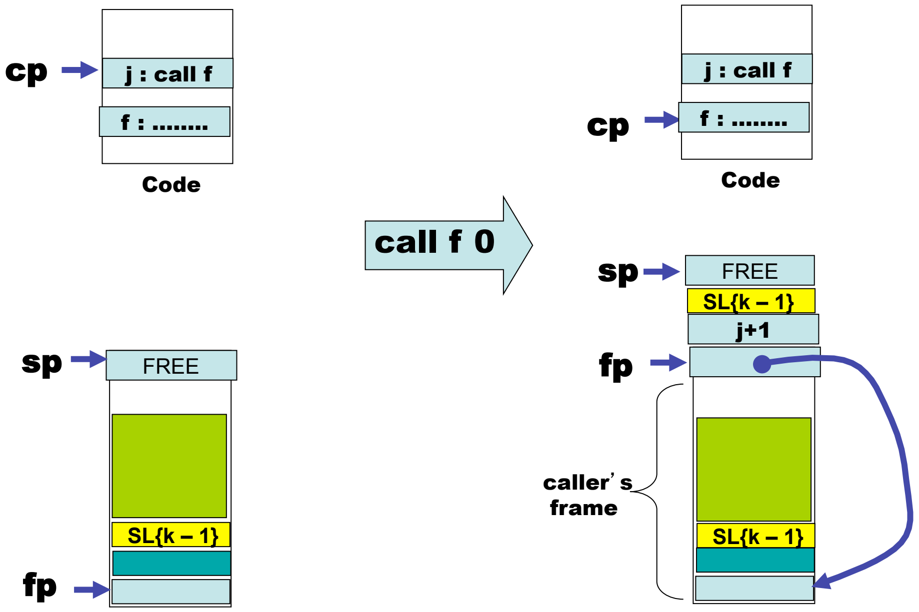


stack frame for **callee** defined at nesting depth **i <= k + 1**

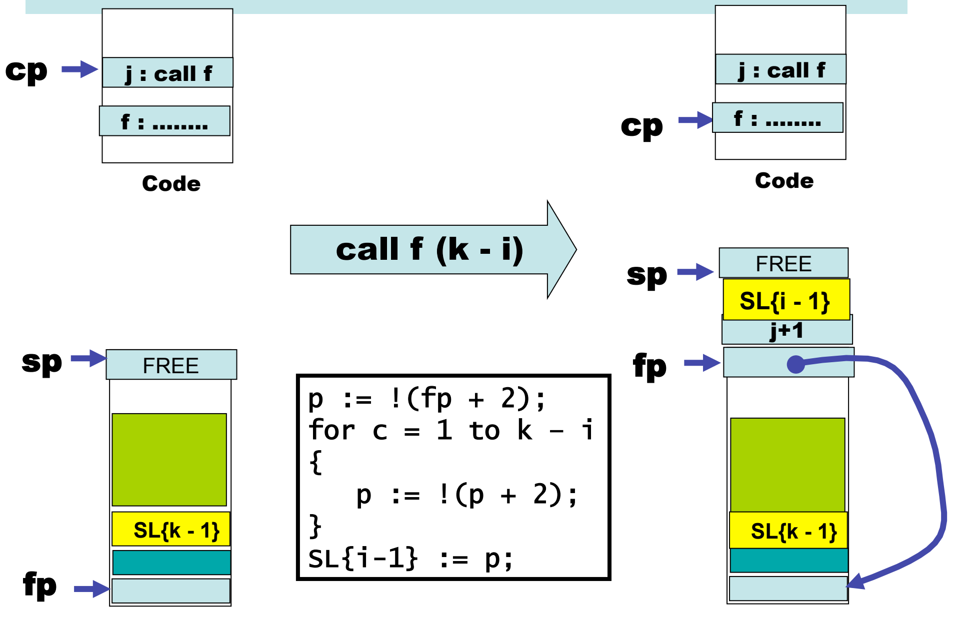
stack frame for **caller** defined at nesting **depth k** used to evaluate code at **depth k + 1**.

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

caller and callee at same nesting depth k

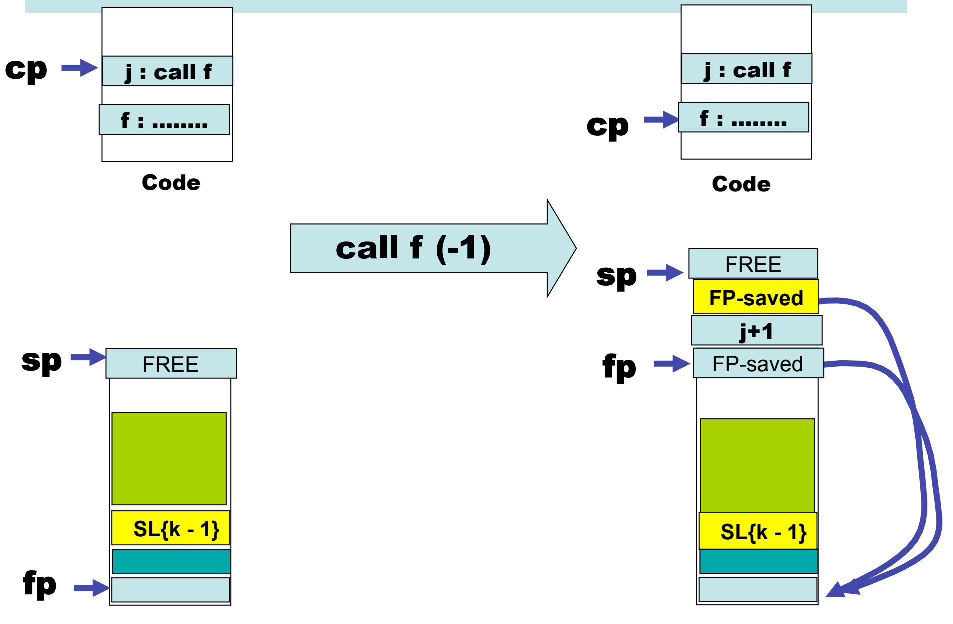


caller at depth k and callee at depth i < k

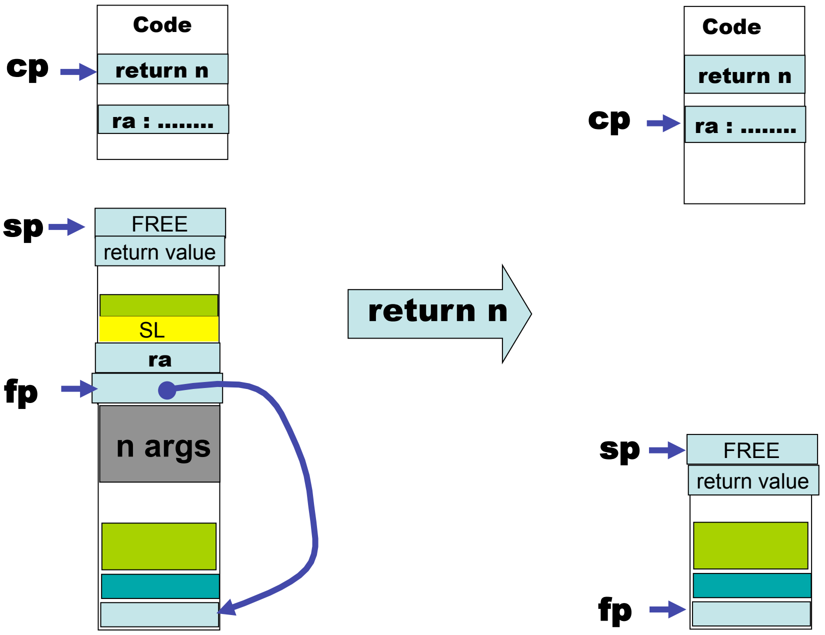


```
p := !(fp + 2);
for c = 1 to k - i
{
    p := !(p + 2);
}
SL{i-1} := p;
```

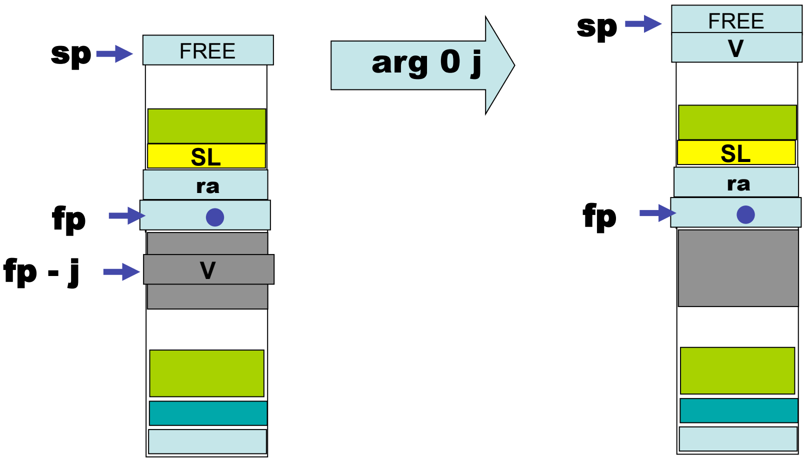
caller at depth k and callee at depth k + 1



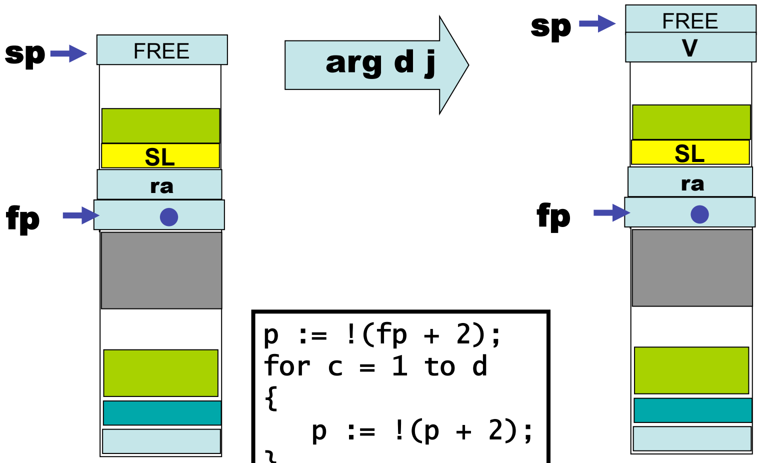
No change to return



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);
```

Approach 3 : Closures

Idea : represent the dynamic value of a function/procedure with free variables as a record.

```
let f(y : int) : int =
   let g(x :int) : int = x + y
   in
       g(y * y)
   end
in
    f(17) + f(21)
end
```

Note that the two calls to f are associated with two variants of g --- one with free variable y bound to 17, the other with y bound to 21.

First record : `{ address := g,  y := 17 }`

Second record : `{ address := g,  y := 21 }`

Now pass closure record to the function itself

```
let g(c, x) = x + c.y

let f(y : int) : int =
   let c = { address := g, y := y }
   in
       g(c, y * y)
   end
in
    f(17) + f(21)
end
```

This looks a lot like lambda lifting, but here we package all values for free variables into a single record, together with the function's address.

Why add g's address to the closure record?

This is not really required for this example, but see next slide …

Closures work for functions-as-values!

```
let f(y : int) : int -> int =
  let g(x :int) : int = y + x
  in g end
in
  let add21 : int -> int  = f(21)
  and add17 : int -> int  = f(17)
  in
     add17(3) + add21(-1)
  end
end
```

NOTE: Neither lambda lifting nor static links can implement this example.
WHY?
The values associated with y have to outlive f's activation records!

A possible intermediate representation

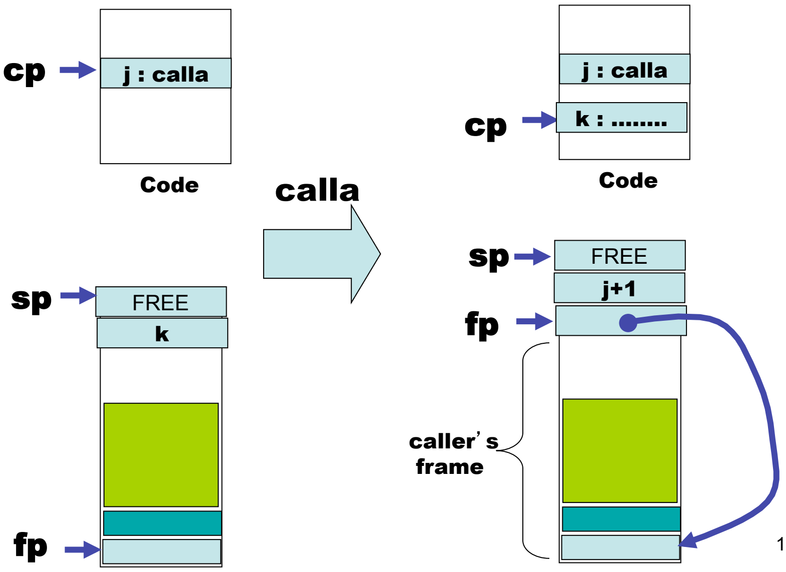
```
Let g(c, x) = x + c.y

let f(y : int) : int -> int = {address := g, y := y}

let add21 = f(21)

let add17 = f(17)

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

Where, in pseudo-code, we have

```
apply_closure(c, v_1, v_2, …, v_k)

          = c.address(c, v_1, v_2, …, v_k)
```

---

calla : gets address from stack top



Another example

```
let f(y : int) : int -> int =
  let g(x :int) : int = y + x
  and h(x :int) : int = y * x
  in
       if y < 17 then g else h
  end
in
     map f l
end
```

This example may make it clearer why
a closure contains the address of the function.

Here the functions address (either g’s or h’s) is
determined dynamically.

A possible intermediate representation

```
let g(c, x) = c.y + x

let h(c, x) = c.y * x

let f(y : int) : int -> int =
    if y < 17
    then { address := g, y := y }
    else { address := h, y := y }
```

We may want to make a distinction between
functions that are called directly

```
f(17)
```

And those called indirectly

```
apply_closure(f(17), 21)
```

Oh, no! What have we done?

We have just implemented a higher level feature (nested functions, first-class functions) using another higher level feature (records).

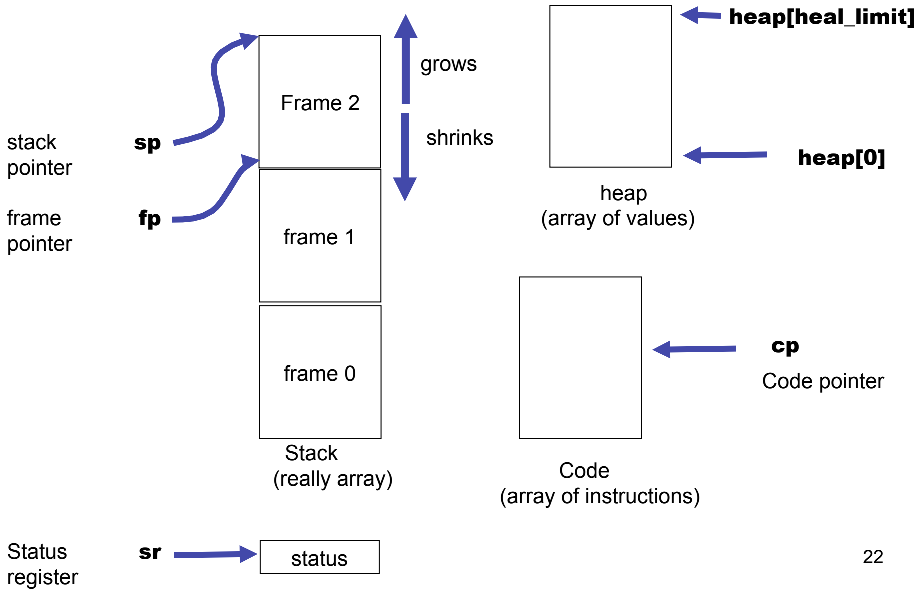
OK, perhaps records are not so high level …

But how do we allocate space for records at run-time?

ANSWER : need a region of storage for “long lived” and “large” data structures (not just closures!)

This is normally called THE HEAP.

Jargon Virtual Machine (v0.2)



heap[heal_limit]

grows

shrinks

Frame 2

stack pointer **sp**

heap[0]

frame pointer **fp**

frame 1

heap
(array of values)

frame 0

cp
Code pointer

Stack
(really array)

Code
(array of instructions)

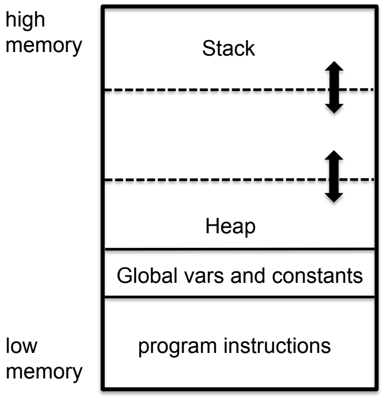
Status register **sr**

status

Typical (Low-Level) Memory Layout (UNIX)

Rough schematic of traditional layout in (virtual) memory.

Dealing with Virtual Machines allows us to ignore some of the low-level details….



high memory

Stack

Heap

Global vars and constants

low memory

program instructions

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”)

Similar situation 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” ….