CS312

Lecture 21-22:

The Environment Model

Fall 2004

Bindings

- A binding associates an identifier with a value.
- We represent bindings as "equalities."

x = 2

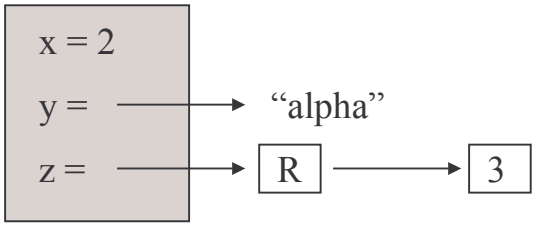
y = → "alpha"

z = → R → 3

z = → 4 | → R → 3

Frames

- A frame is a group of one or more bindings.
- We represent frames as rectangles that contain the bindings.

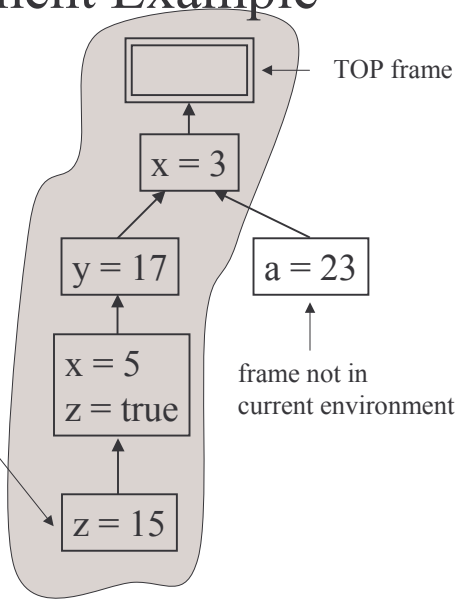


Environment

- An environment is a collection of frames.
- It is organized hierarchically; each frame points to a previous frame.
- Exception: the TOP frame has no parent.
- An environment is uniquely and completely identified if we specify its "bottom" frame.
- In SML, there is always exactly one current ("active") environment.

Environment Example

Values in current environment:
x -> 5
y -> 17
z -> 15
a -> **UNBOUND**



current environment (includes all frames on the path to TOP, including TOP)

TOP Environment

- The environment consisting of the TOP frame only is called the TOP or TOP-level environment.
- The TOP frame is populated by predefined bindings. It is here that the system defines functions like **hd** and **foldr** (at least, this is what we will assume).

Variables

- The value of an identifier is determined by the environment in which we look it up.
- We start at the bottom frame and look for a binding that binds the value of the variable.
- If the frame at hand does not have a suitable binding, we visit the parent frame.
- If we reach the TOP frame but we still do not find a binding, the identifier is unbound.

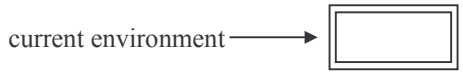
Let Expressions

```
let val x = e1 in e2
```

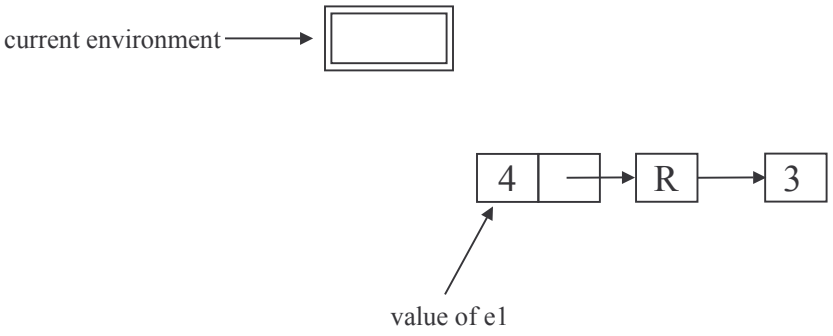
1. Evaluate e1 in the current environment.
2. Extend environment with a new frame; bind x to value(e1) in this frame.
3. Evaluate e2 in the extended environment
4. Restore the old environment (remove the frame you have added in step 2)
5. Return the value of e2.

Let Example: Before Step 1

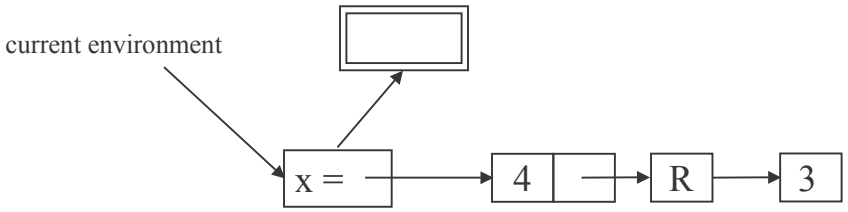
```
let val x = (4, ref 3) in #1 x end
```



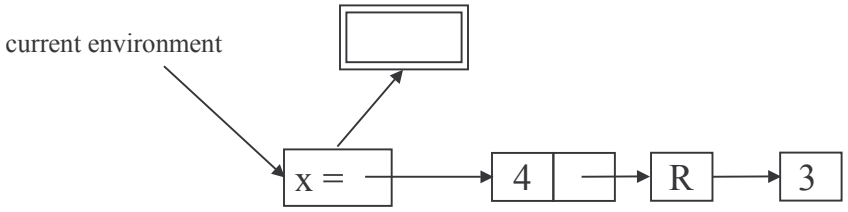
Let Example: Step 1



Let Example: Step 2

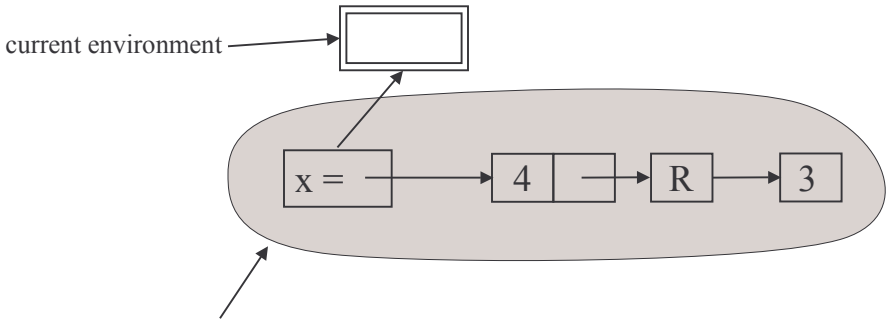


Let Example: Step 3



We evaluate #1 x in the current environment; the result is equal to 4.

Let Example: Step 4

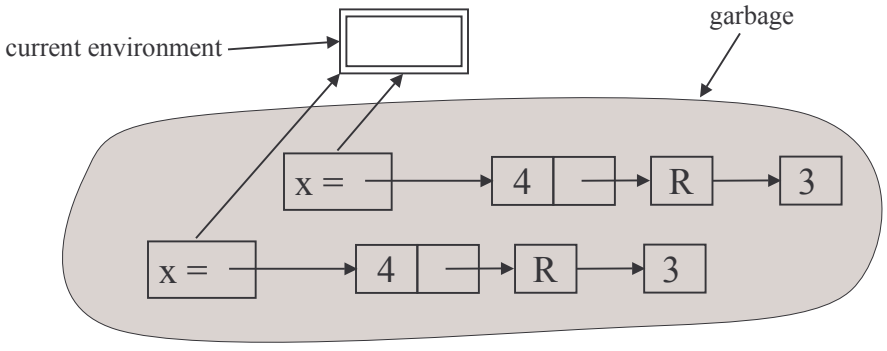


Garbage: these data structures are not reachable anymore from SML. Accumulating garbage will fill all available memory sooner or later; when this happens, we must perform garbage collection.

Repeated Executions

Repeated executions of the same statement do **not** lead to the reuse of the data structures created during the first execution. Here is the memory layout after the **let** statement below has been executed twice:

```
let val x = e1 in e2
```



Multiple Declarations

```
let
  val x = (2, 3)
  val y = ("alpha", ref x)
in
  #1(!(#2 y)
end
```

rewrite

```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

Subtle issue: declaration **n+1** is evaluated in the environment current **after** declaration **n** (sequential evaluation). Parallel evaluation is an alternative: evaluate all expressions in parallel, in the **same** environment.

Sequential vs. Parallel Evaluation

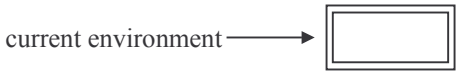
```
let
  val x = 2
in
  let
    val x = x + 2
    val y = x + 2
  in
    (x, y)
  end
end
```

Sequential evaluation:
(x, y) = (4, 6)
This is what SML does.

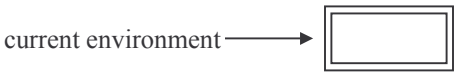
Parallel evaluation:
(x, y) = (4, 4)
This is **not** what SML does, but it is a model that could have been adopted.

Nested Let: Before Step 1

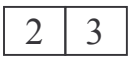
```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```



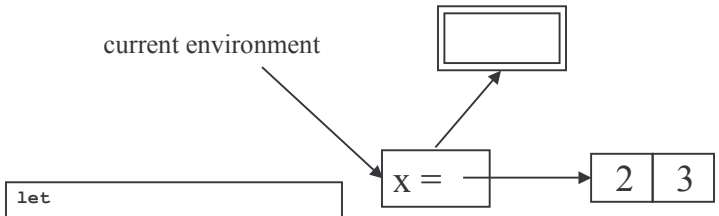
Nested Let Example: Step 1




```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

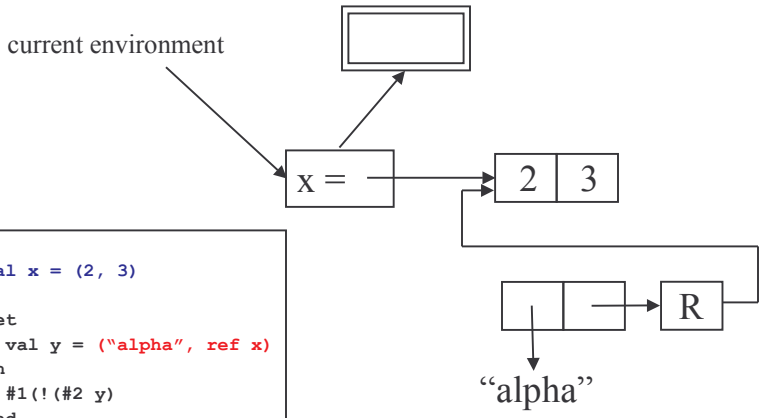


Nested Let Example: Step 2



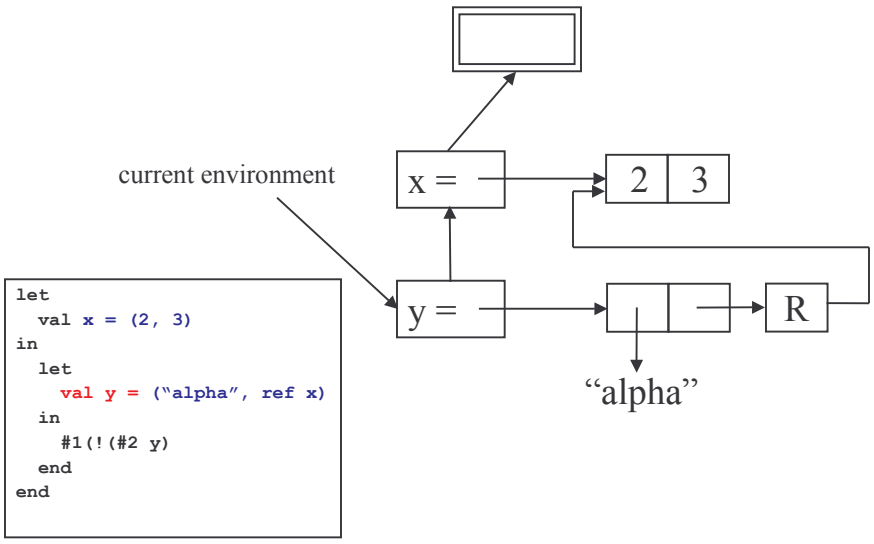

```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

Nested Let Example: Step 3.1



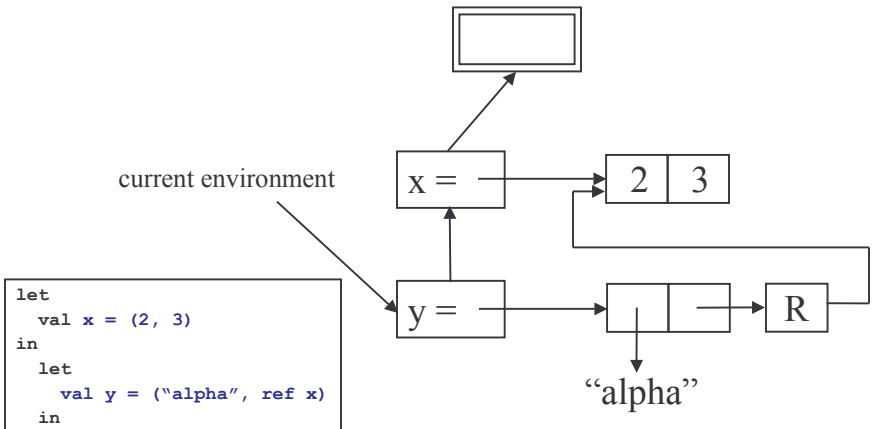

```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

Nested Let Example: Step 3.2




```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

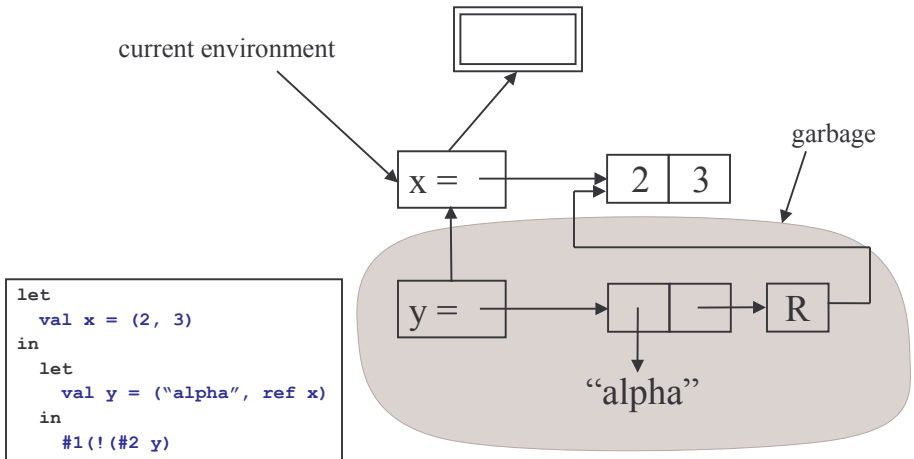
Nested Let Example: Step 3.3




```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

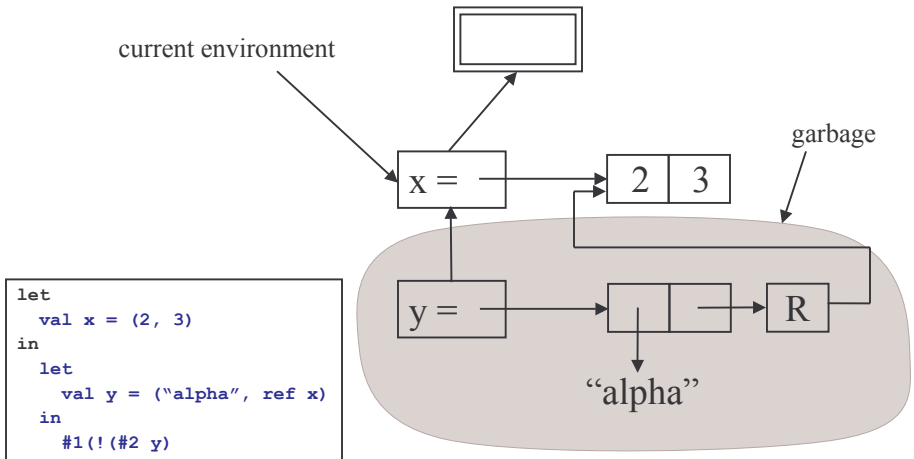
The value of expression `#1(!(#2 y)` is 2.

Nested Let Example: Step 3.4




```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

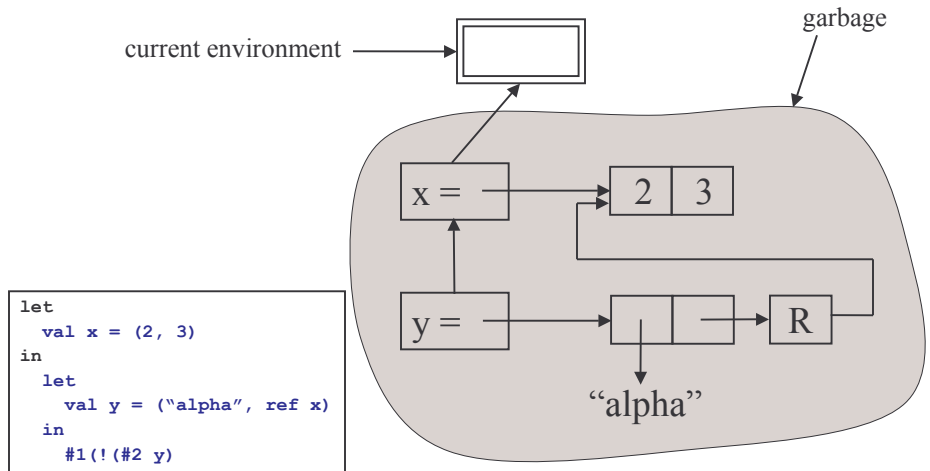
Nested Let Example: Step 3.5




```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

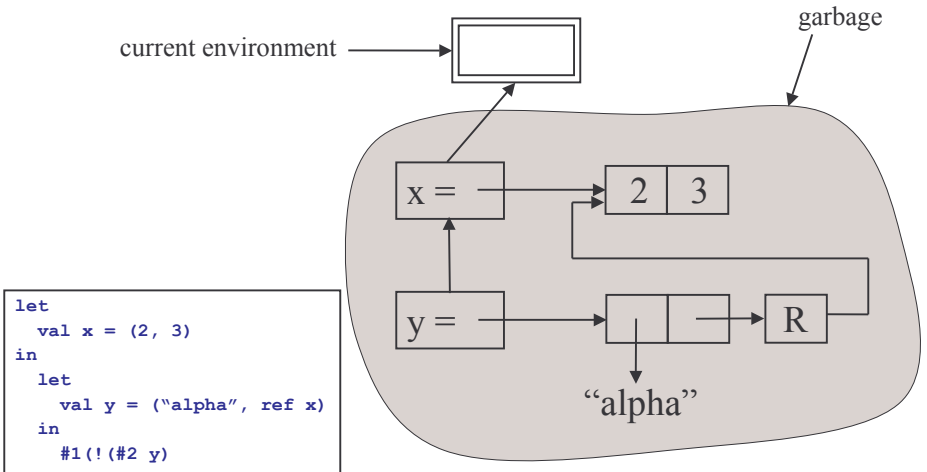
The value of the inner **let** is 2.

Nested Let Example: Step 4




```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

Nested Let Example: Step 5




```
let
  val x = (2, 3)
in
  let
    val y = ("alpha", ref x)
  in
    #1(!(#2 y)
  end
end
```

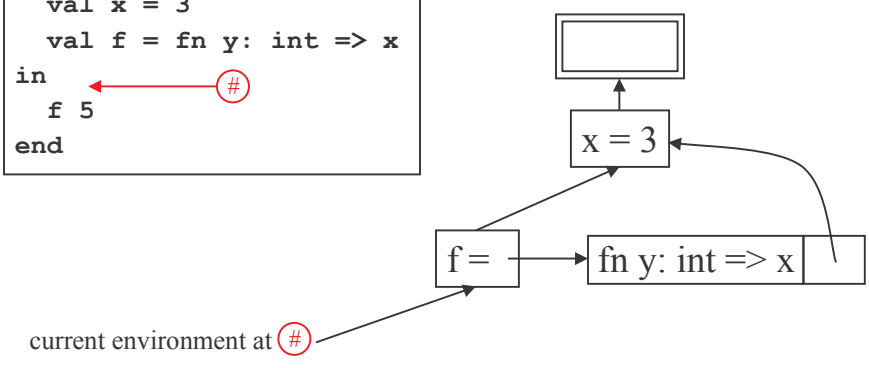
The value of the outer **let** is 2.

Functions

- Functions are represented as **closures**.
- A closure is a representation of the function (including the number and type of arguments, and the function body), together with a pointer to the **environment current at the time of the function's definition**.

Function Example

```
let
  val x = 3
  val f = fn y: int => x
in
  f 5
end
```



Function Calls

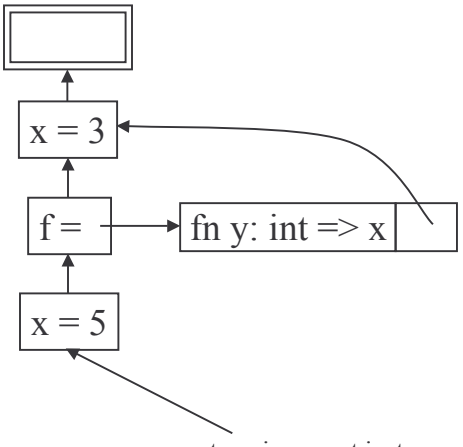
Steps to evaluate **e1(e2)**:

1. Evalute **e1**; must get a closure.
2. Evaluate **e2**; must get a value.
3. Save current environment;
4. Retrieve environment from closure; extend it with bindings of formal arguments to actual arguments.
5. Evaluate function body in extended environment.
6. Restore environment saved in step 3.
7. Return the result.

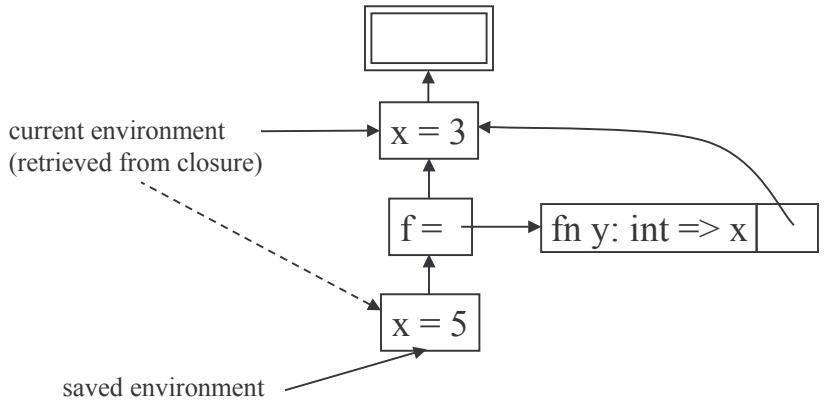
```
Note: Closures are
values, but not all
values are closures…
```

Function Call Example

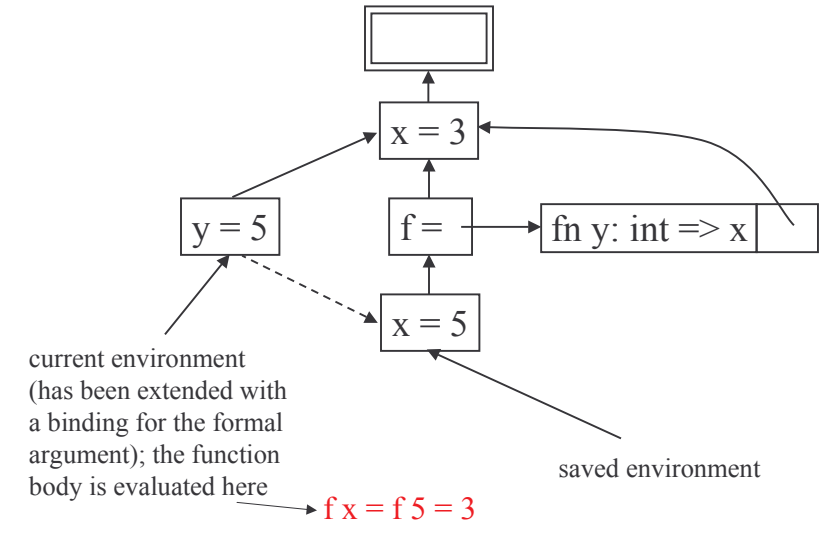
```
let
  val x = 3
  val f = fn y: int => x
  val x = 5
in
  f x
end
```



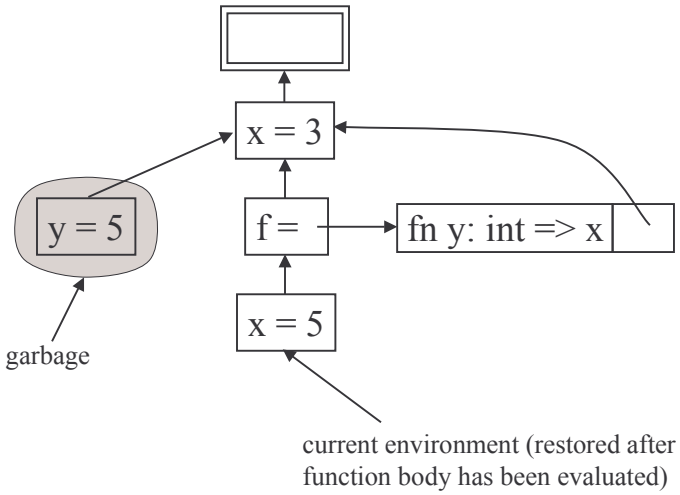
Function Call Example



Function Call Example



Function Call Example



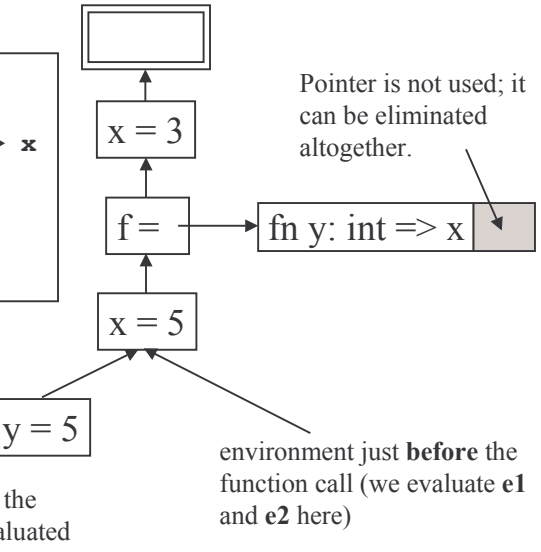
Static vs. Dynamic Scoping

- Assume that in step (4) we do not retrieve the current environment from the closure, but we extend the current environment with bindings for the function's formal arguments. What do we get? Dynamic scoping (see next slide)!
- For dynamic scoping, we do not need to store any pointer in the closure.

Dynamic Scoping

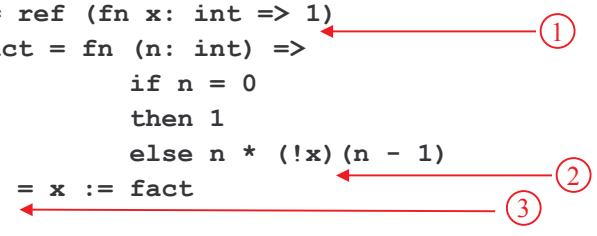
```
let
  val x = 3
  val f = fn y: int => x
  val x = 5
in
  f x
end
```

The value of f(x) is 5!

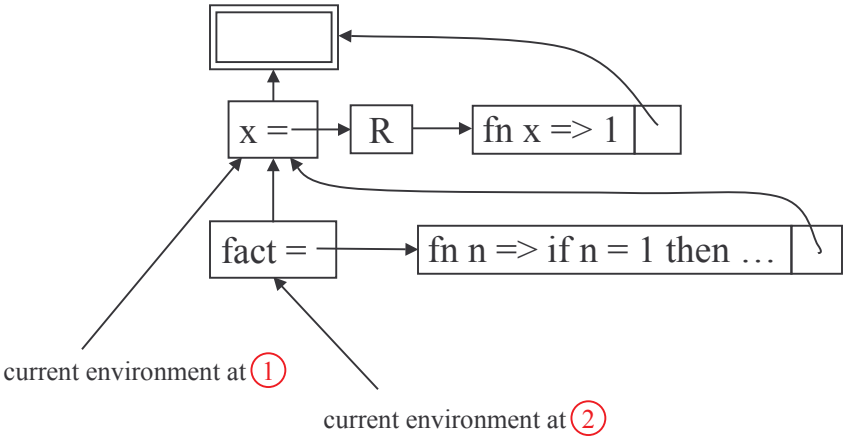


Recursion Using References

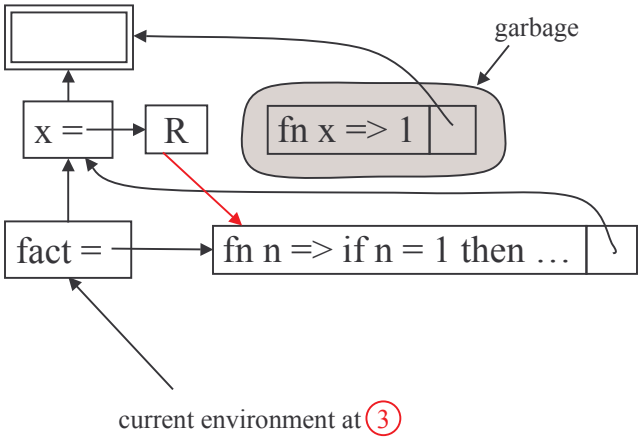
```
let
  val x= ref (fn x: int => 1)
  val fact = fn (n: int) =>
                if n = 0
                then 1
                else n * (!x)(n - 1)
  val () = x := fact
in
  fact 3
end
```



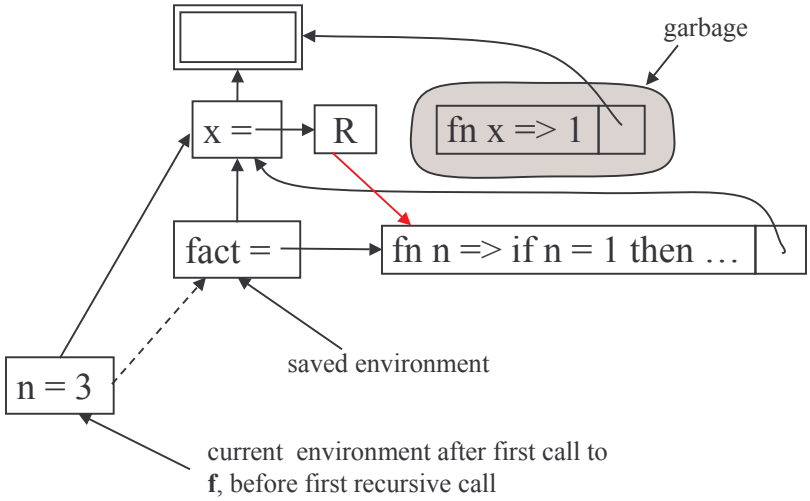
Recursion Using References



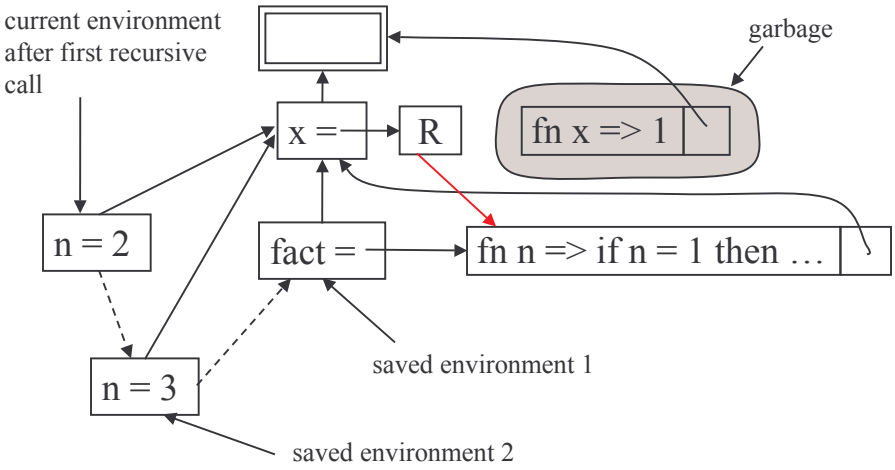
Recursion Using References



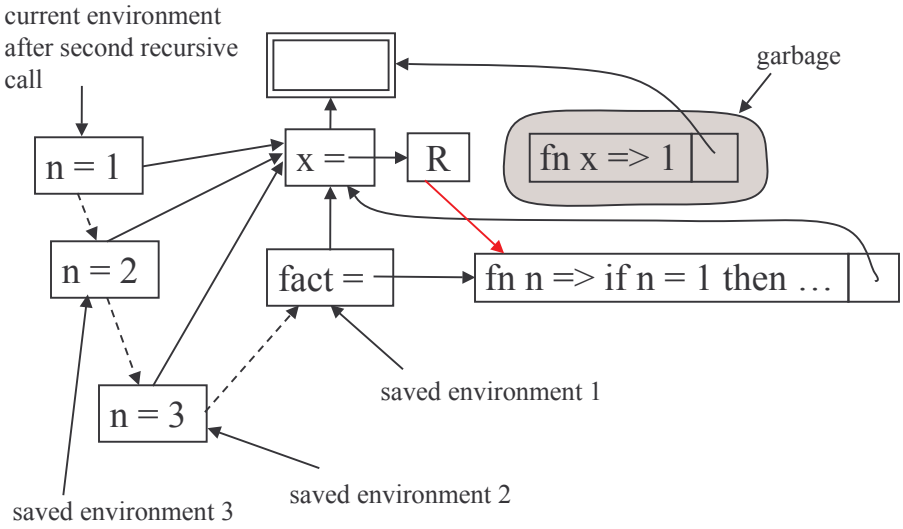
Recursion Using References



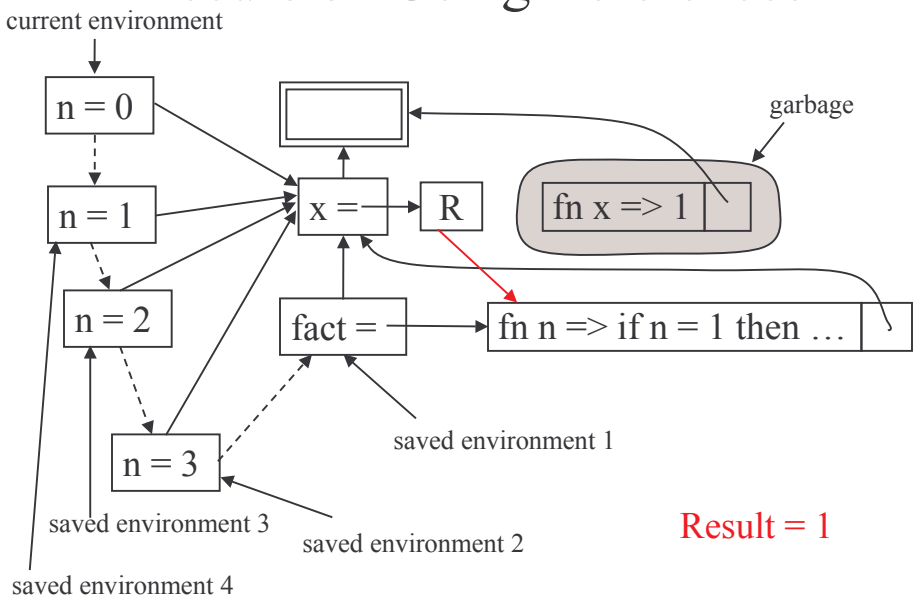
Recursion Using References



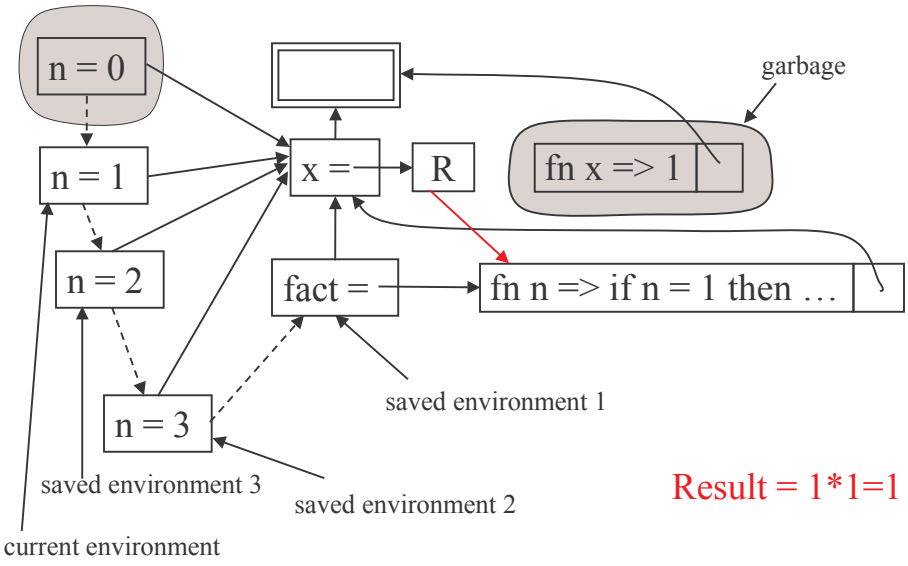
Recursion Using References



Recursion Using References

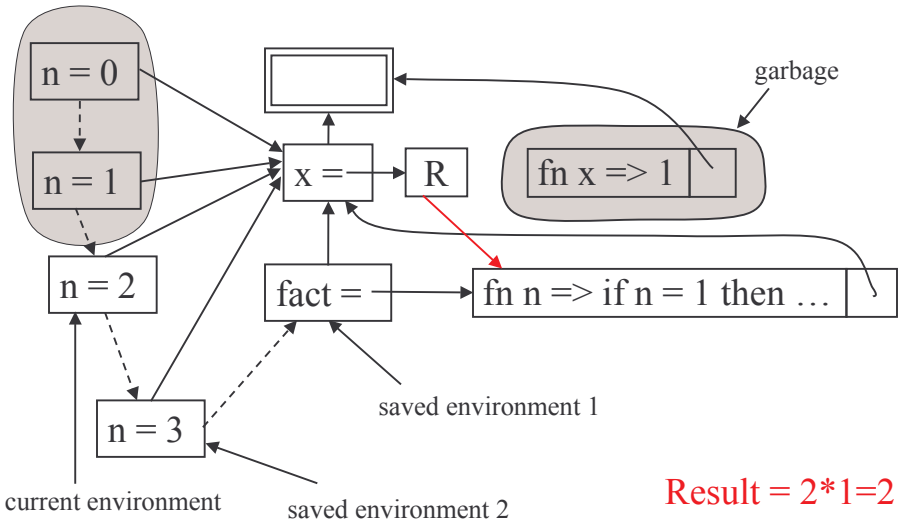


Recursion Using References



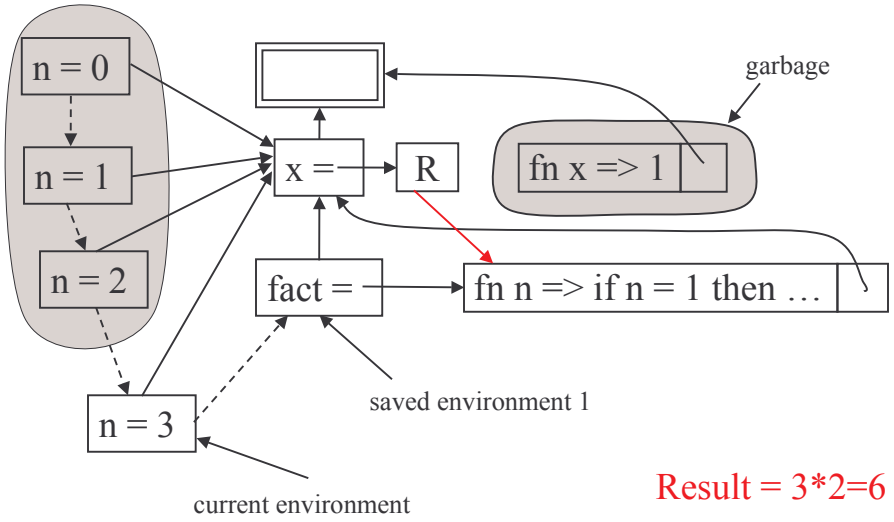
Result = 1*1=1

Recursion Using References



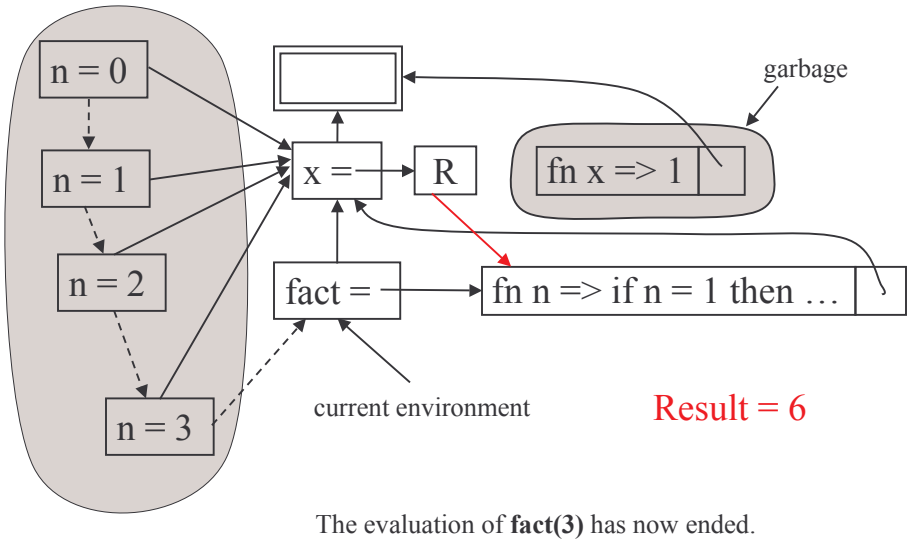
Result = 2*1=2

Recursion Using References



Result = 3*2=6

Recursion Using References



Result = 6

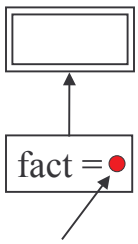
The evaluation of **fact(3)** has now ended.

"True" Recursion

- We declare recursive functions using **fun** or **val rec**.
- Must include the binding to function we are declaring in the environment stored in the closure.
- Solution: create binding for function name **before creating the closure**, extend environment with the binding, create closure and set pointer to extended environment, set the value in the binding.

Recursion

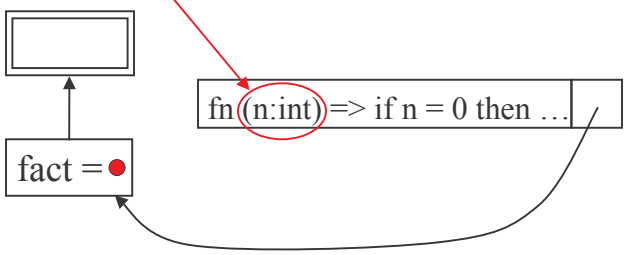
```
let
  fun fact(n:int) = if n = 0
                    then 1
                    else n * fact(n - 1)
in
  fact 3
end
```



A new binding has been created, but there is no value associated with it yet.

Recursion

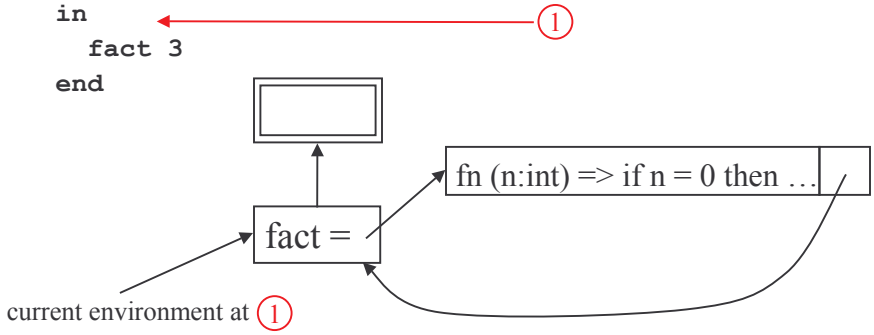
```
let
  fun fact(n:int) = if n = 0
                    then 1
                    else n * fact(n - 1)
in
  fact 3
end
```



The closure has been set up, including the pointer to the still incomplete environment.

Recursion

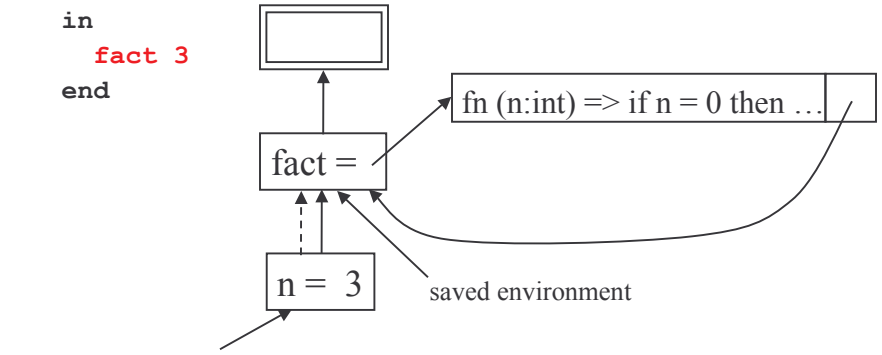
```
let
  fun fact(n:int) = if n = 0
                    then 1
                    else n * fact(n - 1)
in
  fact 3
end
```



We finished defining the function; the closure points to an environment that includes a binding for the function name.

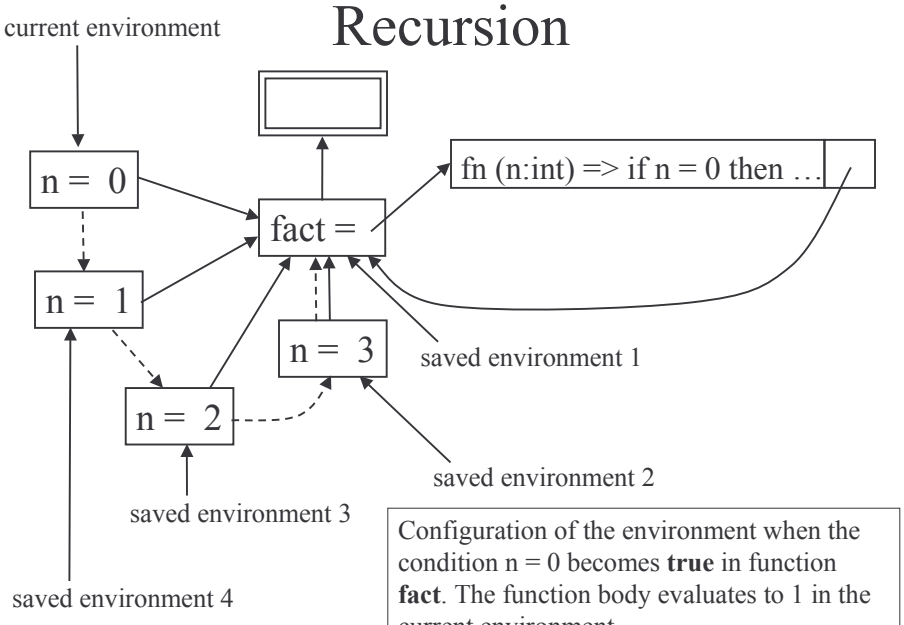
Recursion

```
let
  fun fact(n:int) = if n = 0
                    then 1
                    else n * fact(n - 1)
in
  fact 3
end
```



current environment in which the body of fact is evaluated during the first function call

Recursion



Configuration of the environment when the condition n = 0 becomes **true** in function **fact**. The function body evaluates to 1 in the current environment.

Remember Streams?

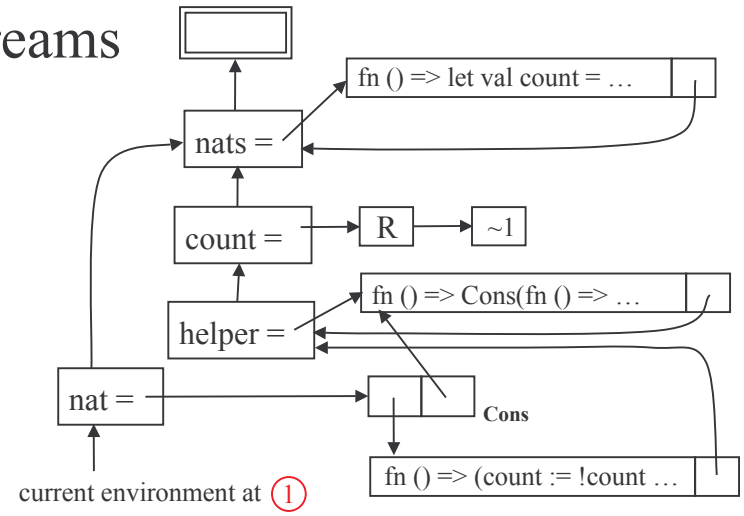
```
datatype 'a stream = Null | Cons of (unit -> 'a) * (unit -> 'a stream)
let
  fun nats(): int stream =
  let
    val count: int ref = ref ~1;
    fun helper(): int stream =
      Cons(fn () => (count := (!count) + 1;!count),
           helper)
  in
    helper()
  end
  val nat = nats()
in
    ...
end
```

① ←

The stream of natural numbers. Each stream is independent of every other stream.

This is a complex example on which you can test your understanding of the environment model. You can represent **Cons(a, b)**, as if it were tuple **(a, b)**. Mark the tuple with the subscript **Cons** to specify its true nature.

Streams



To simplify the diagram, we assumed that there is no frame added to the environment stored in the closure when a function's argument is unit. The alternative would be to add an empty frame. Note how **count** and **helper** are only accessible through **nat**; they "belong" to this stream only.

Models Are Simplifications…

It is important to remember that models are simplifications. We have ignored many details that would be important for any real implementation of the environment model. Some aspects of the model could have been defined slightly differently, without affecting its usability. For example, we decided to keep around bindings that become garbage, rather then eliminating them immediately. This helps us better illustrate the history of the computation. However, with some care, we could modify the model so that these bindings are removed as early as possible (but not the values themselves, which will wait for a garbage collection round).