CS 312
Spring 2004
Lecture 18
Environment Model

Substitution Model

- Represents computation as doing *substitutions* for *bound variables* at reduction of let, application:

$$\texttt{let val } x \texttt{ = } v \texttt{ in } e \rightarrow e\{v/x\}$$

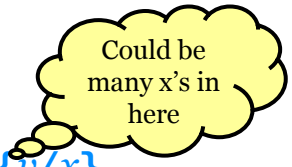
$$\texttt{(fn(}x\texttt{:}t\texttt{)=>}e\texttt{)(}v\texttt{)} \rightarrow e\{v/x\}$$

```
let val x = fn z:'a=>z in
  x(x(x))
end
```

→ `(fn z=>z)((fn z=>z)(fn z=>z))`

Problems

- Not a realistic implementation! Substitution is too slow.



$(\texttt{fn}(x\texttt{:}t)\texttt{=>}e)(v) \rightarrow e\{v/x\}$

- Idea: use *environment* to record substitutions, do them only as needed

$(\texttt{fn}(x\texttt{:}t)\texttt{=>}e)(v) \rightarrow e$

$\boxed{x = v}$

environment

Environment Model

- No substitution, realistic cost model
- Environment is a finite map from variables to values
- Example:

```
let val x = 2
    val y = "hello"
    val f = fn z:int=>x
in f(x + size(y)) end
```

Evaluate:

f(x + size(y))

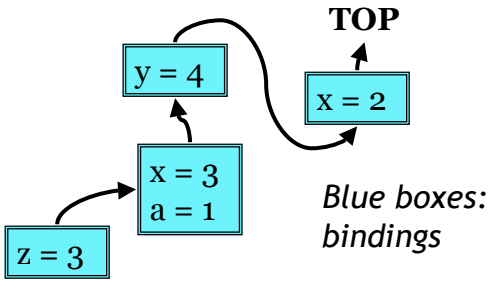
in environment:

```
x = 2
y = "hello"
f = fn z:int=>x
```

?

Variables

- To evaluate a variable, look it up in the environment. To look it up, we start with the last binding added to the environment and then work towards the TOP.
- Evaluating "x" in this environment yields 3:



Let expressions

To evaluate **`let val x = e1 in e2`**:

1. Evaluate **`e1`** in the current environment
2. Extend the current environment with a binding that maps **`x`** to the value of **`e1`**
3. Evaluate **`e2`** in the extended environment
4. Restore the old environment (i.e., remove the binding for **`x`**)
5. Return the value of **`e2`**

Let Example

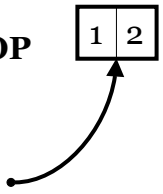
```
let val x = (1,2) in #1 x end
```

current env→**TOP**

Let Example

```
let val x = (1,2) in #1 x end
```

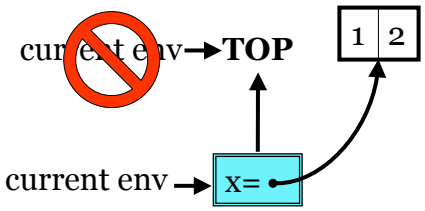
current env→**TOP**



1. Evaluating (1,2) yields a pointer to a tuple in memory.

Let Example

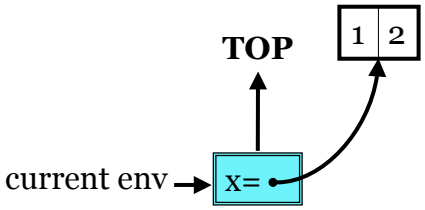
```
let val x = (1,2) in #1 x end
```



1. Evaluating (1,2) yields a pointer to a tuple in memory.
2. Extend the environment with a binding for x.

Let Example

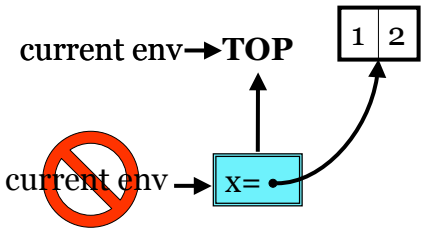
```
let val x = (1,2) in #1 x end
```



1. Evaluating (1,2) yields a pointer to a tuple in memory.
2. Extend the environment with a binding for x.
3. Evaluate the body of the let in the new environment.
   x evaluates to a pointer to the tuple, so #1 x evaluates to the first component, namely 1.

Let Example

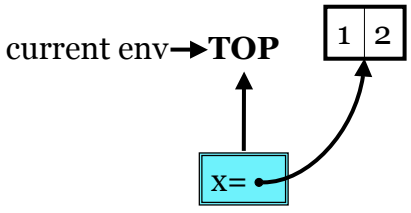
```
let val x = (1,2) in #1 x end
```



1. Evaluating (1,2) yields a pointer to a tuple in memory.
2. Extend the environment with a binding for x.
3. Evaluate the body of the let in the new environment.
   x evaluates to a pointer to the tuple, so #1 x evaluates to the first component, namely 1.
4. Restore the old environment.

Let Example

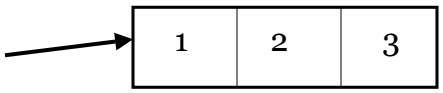
```
let val x = (1,2) in #1 x end
```



1. Evaluating (1,2) yields a pointer to a tuple in memory.
2. Extend the environment with a binding for x.
3. Evaluate the body of the let in the new environment.
   x evaluates to a pointer to the tuple, so #1 x evaluates to the first component, namely 1.
4. Restore the old environment.
5. Return the value we got: 1

Pictorial Overview:

- Primitive values like integers, reals, unit, or nil evaluate to themselves.
- A tuple value, such as (1,2,3) evaluates to a pointer to a box in memory containing the values of the sub-expressions:



Multiple Declarations

To evaluate:

```
let val x = e1
    val y = e2
    val z = e3
in
    e4
end
```

Do the same the same thing as you would for:

```
let val x = e1
in  let val y = e2
    in  let val z = e3
        in
            e4
        end
    end
end
```

Evaluation of Example

```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```

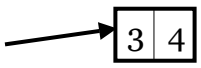
```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```

current env → TOP

Evaluation of Example

```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```

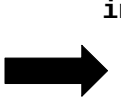
```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```



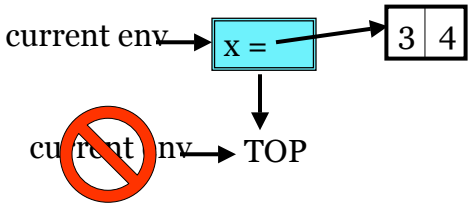
current env → TOP

Evaluation of Example

```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```



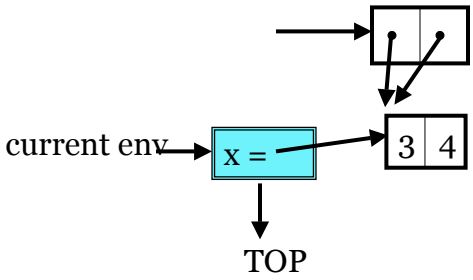
```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```



Evaluation of Example

```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```

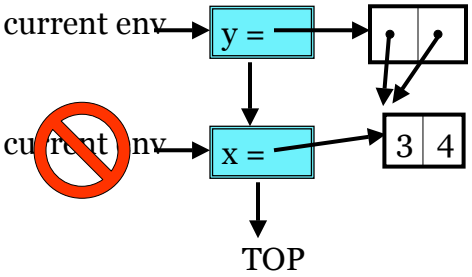
```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```



Evaluation of Example

```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```

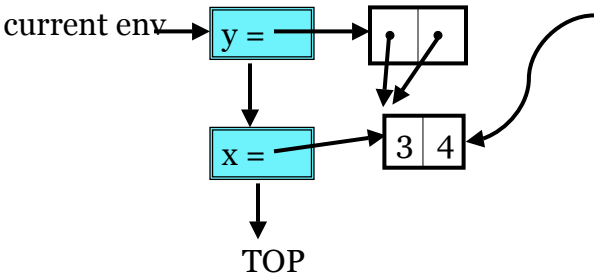
```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```



Evaluation of Example

```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```

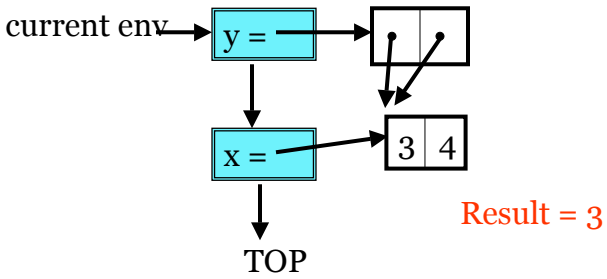
```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```



Evaluation of Example

```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```

```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```

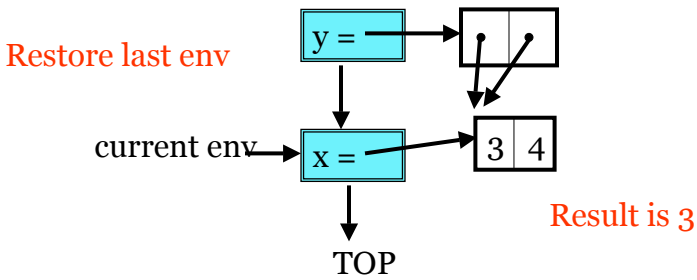


Result = 3

Evaluation of Example

```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```

```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```

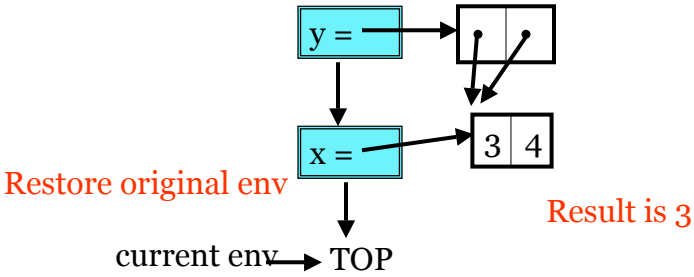


Result is 3

Evaluation of Example

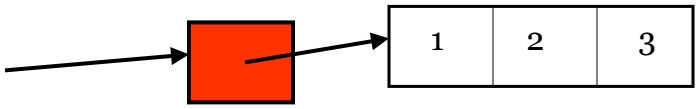
```
let val x = (3,4)
    val y = (x,x)
in
     #1(#2 y)
end
```

```
let val x = (3,4)
in
    let val y = (x,x)
    in
      #1(#2 y)
    end
end
```



Refs

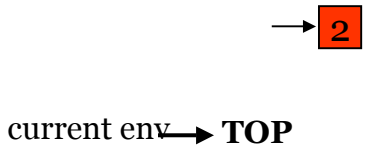
- To evaluate **`ref e`**, evaluate e to a value first, and then allocate a new ref cell, place the value in the ref cell, and return a pointer to the ref cell. For instance, **`ref (1,2,3)`** evaluates to:



ref cells = red boxes.

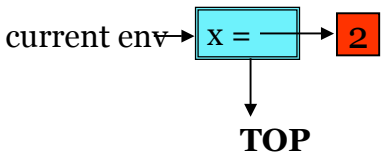
Ref Example

```
let val x = ref 2 in
    val y = x
in
    x:=1; !y
end
```



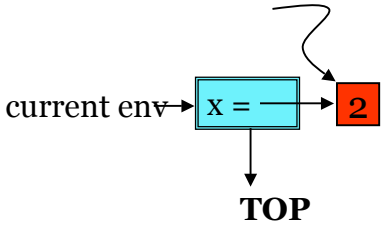
Ref Example

```
let val x = ref 2 in
    val y = x
in
    x:=1; !y
end
```



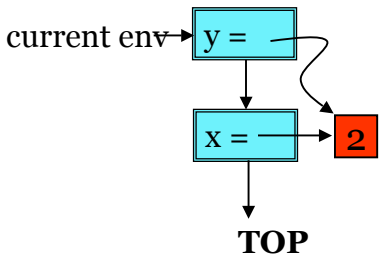
Ref Example

```
let val x = ref 2 in
    val y = x
in
    x:=1; !y
end
```



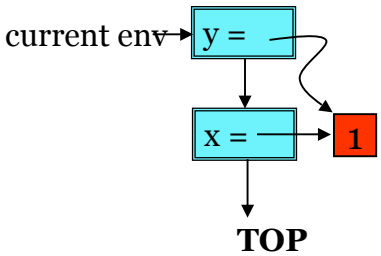
Ref Example

```
let val x = ref 2 in
    val y = x
in
    x:=1; !y
end
```



Ref Example

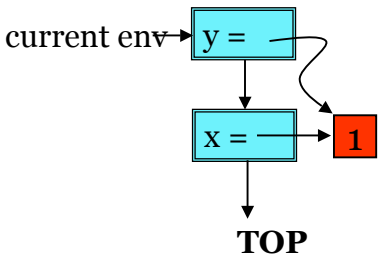
```
let val x = ref 2 in
    val y = x
in
    x:=1; !y
end
```



Ref Example

```
let val x = ref 2 in
    val y = x
in
    x:=1; !y
end
```

Result= 1



Functions

```
let val x = 2
    val f = fn z:int => x
in
let val x = "bye"
  in
    f(size(x))
  end
```

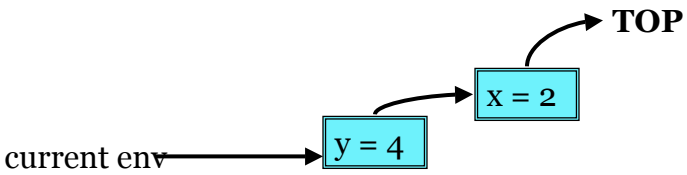
Static scope:
ML, Java, Scheme, …

Dynamic scope:
Perl, Python, BASIC

- How do we make sure the environment has the (correct) binding for *x*?
  - We must keep track of the environment at the point where the function was evaluated.
  - Function evaluation: `fn z:int => x`, not `f(size(x))`
- We create a *closure*
  - A pair of a function and its environment

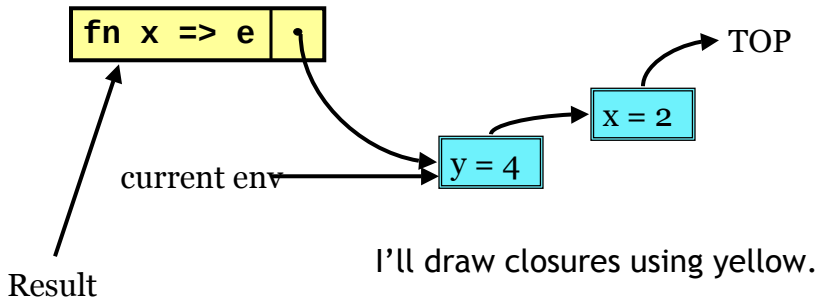
Functions

- To evaluate a function `(fn x => e)` create a *closure* out of the function and the current environment and return a pointer to the closure.



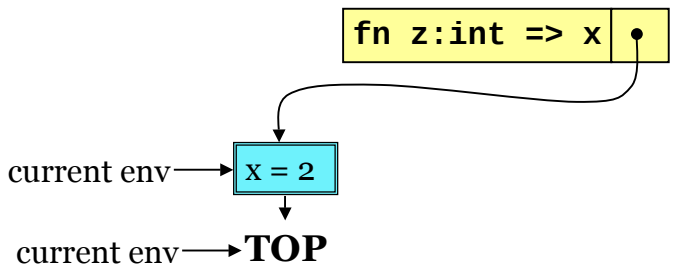
Creating closures

- To evaluate a function `(fn x => e)` create a *closure* out of the function and the current environment and return a pointer to the closure.



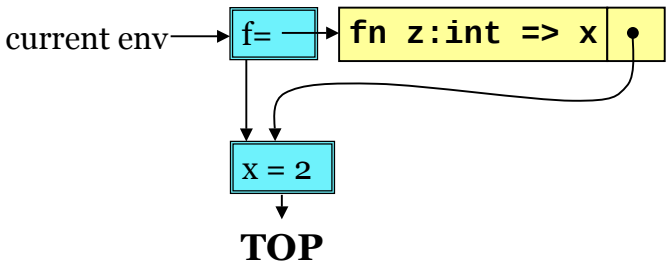
Function Example

```
let val x = 2
    val f = fn z:int => x
  in
  let val x = "bye"
    in
      f(size(x))
    end
```



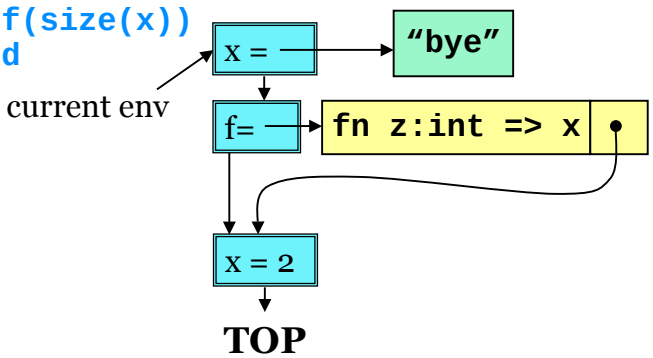
Function Example

```
let val x = 2
    val f = fn z:int => x
  in
  let val x = "bye"
    in
      f(size(x))
    end
```



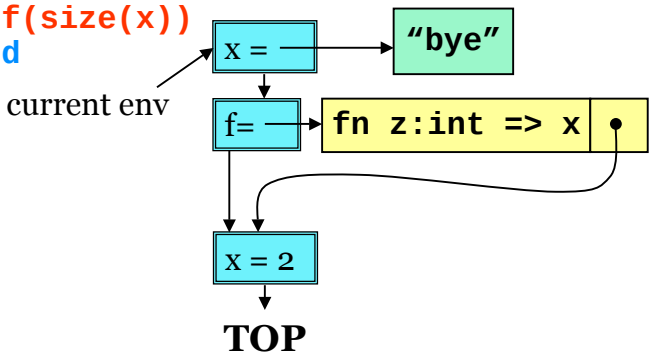
Function Example

```
let val x = 2
    val f = fn z:int => x
  in
  let val x = "bye"
    in
      f(size(x))
    end
```



Function Example

```
let val x = 2
    val f = fn z:int => x
  in
  let val x = "bye"
    in
      f(size(x))
    end
```



Function Calls

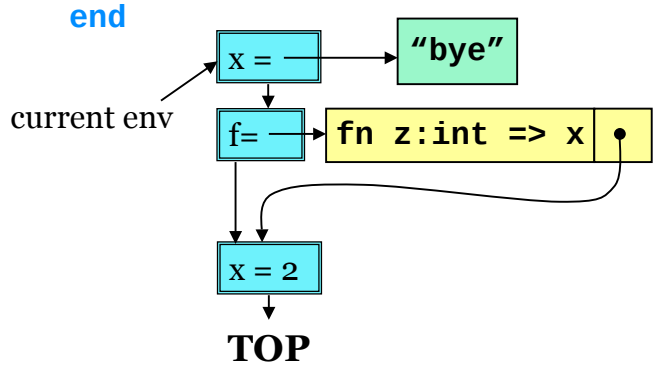
To evaluate **`e1(e2)`**:

1. evaluate **`e1`** -- you should get a pointer to a closure.
2. evaluate **`e2`** to a value.
3. save the current environment -- we'll come back to it after the function call.
4. extend the environment of the closure, mapping the formal argument to the actual argument.
5. evaluate the body of the function within the extended environment -- this gives us our result value.
6. restore the old environment (saved in step 3)
7. return the result.

Function Call Example

```
let val x = 2
    val f = fn z:int => x
  in
  let val x = "bye"
    in
      f(size(x))
    end
```

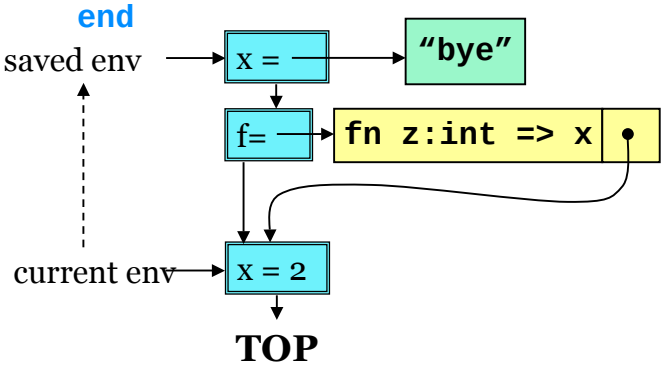
1. Evaluate e1, e2



Function Call Example

```
let val x = 2
    val f = fn z:int => x
  in
  let val x = "bye"
    in
      f(size(x))
    end
```

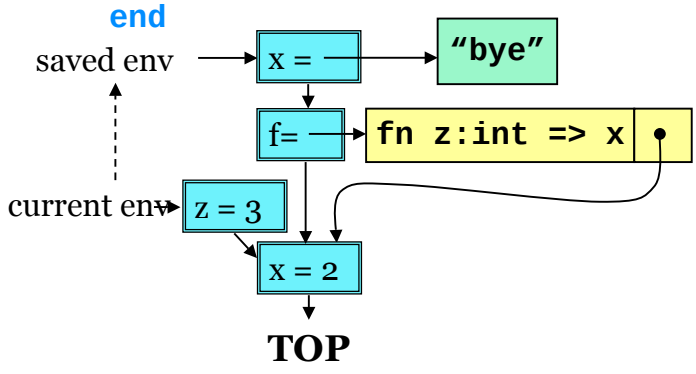
1. Evaluate e1, e2
2. Save environ.



Function Call Example

```
let val x = 2
    val f = fn z:int => x
  in
  let val x = "bye"
    in
      f(size(x))
    end
```

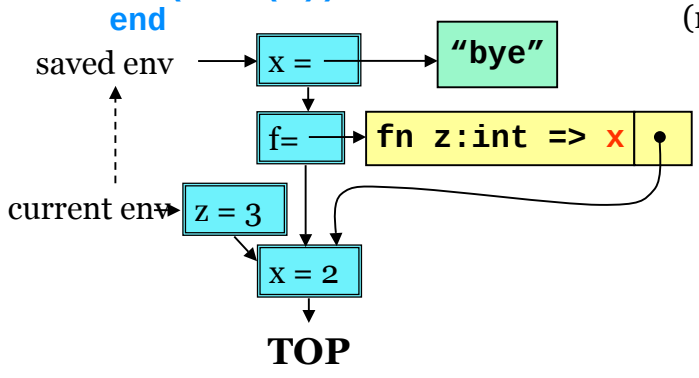
1. Evaluate e1, e2
2. Save environ.
3. Extend env with actual



Function Call Example

```
let val x = 2
    val f = fn z:int => x
  in
  let val x = "bye"
    in
      f(size(x))
    end
```

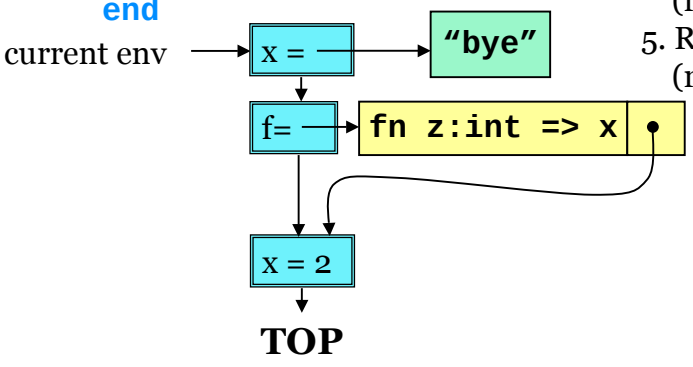
1. Evaluate e1, e2
2. Save environ.
3. Extend env with actual
4. Evaluate body (result= 2)



Function Call Example

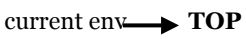
```
let val x = 2
    val f = fn z:int => x
 in
 let val x = "bye"
   in
     f(size(x))
   end
```

1. Evaluate e1, e2
2. Save environ.
3. Extend env with actual
4. Evaluate body (result= 2)
5. Restore env. (result= 2)



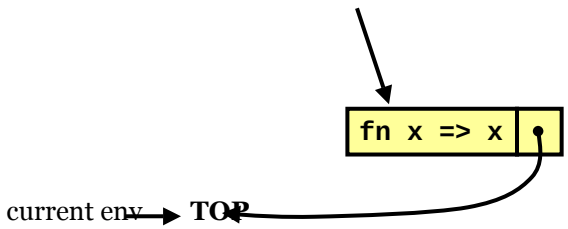
Creating a cycle

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



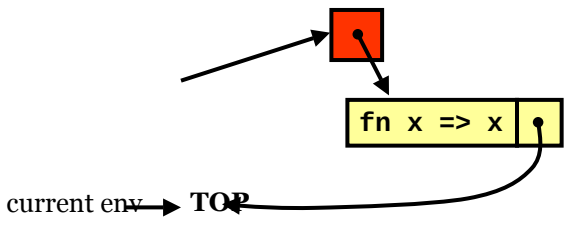
Creating a cycle

```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



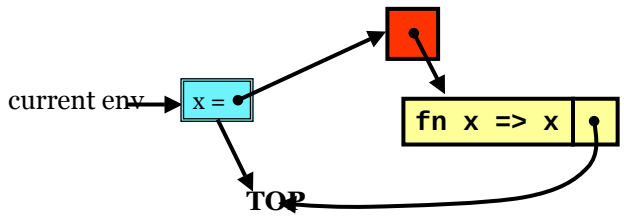
Creating a cycle

```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



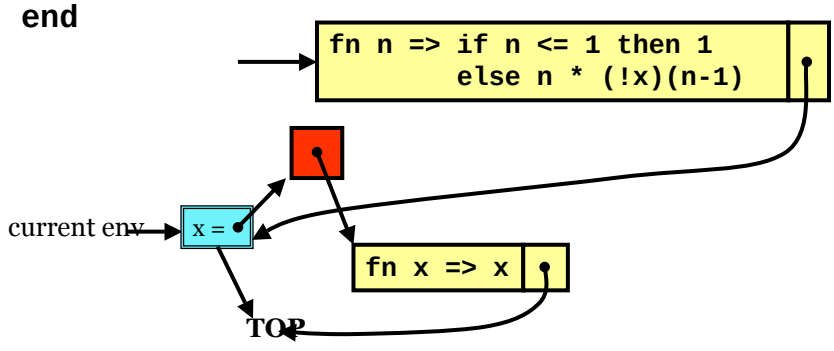
Creating a cycle

```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



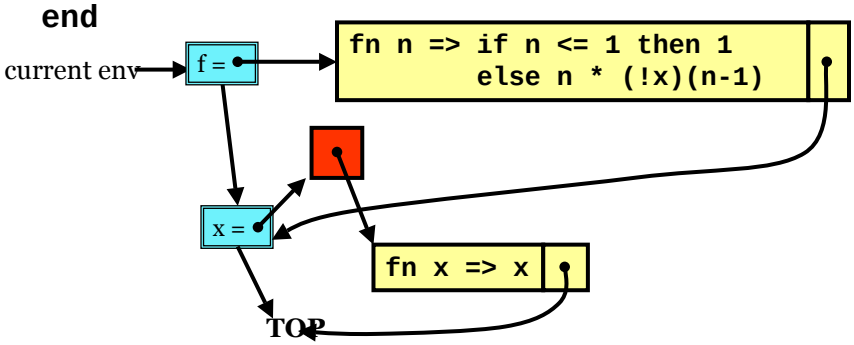
Creating a cycle

```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



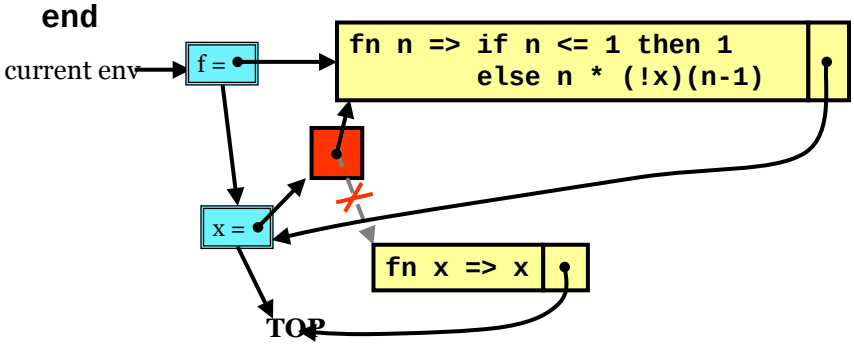
Creating a cycle

```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



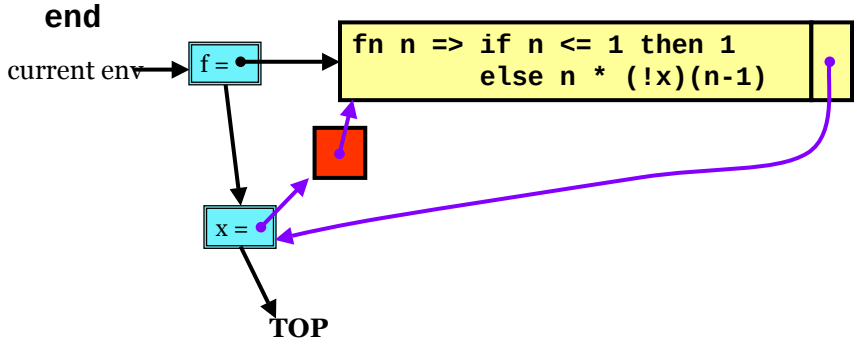
Creating a cycle

```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



Creating a cycle

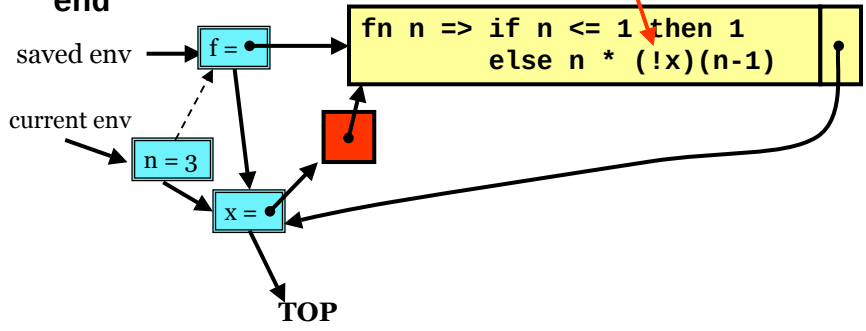
```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



Creating a cycle

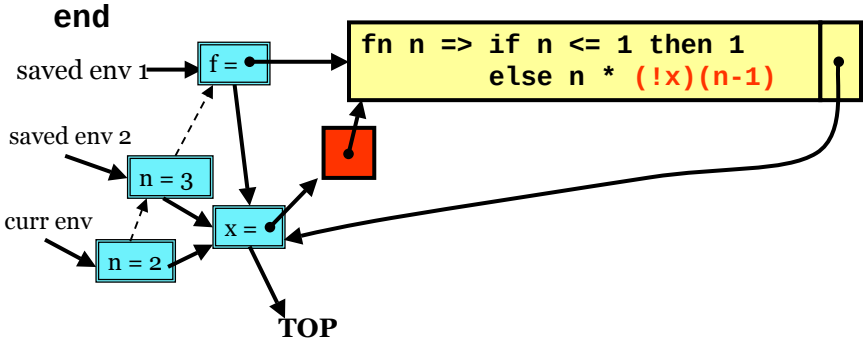
```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```

Note: !x is the same as f



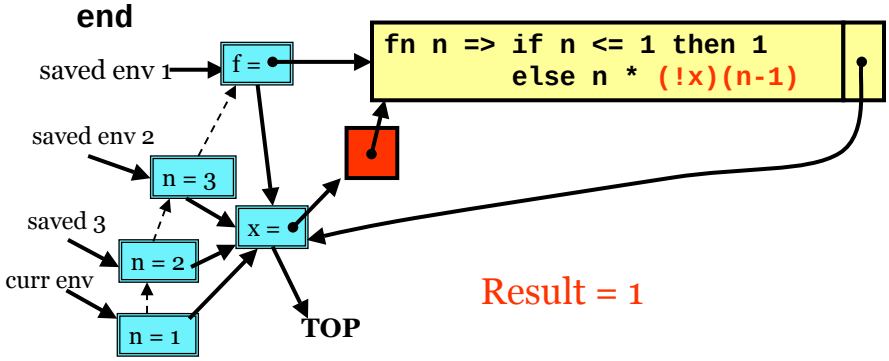
Creating a cycle

```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



Creating a cycle

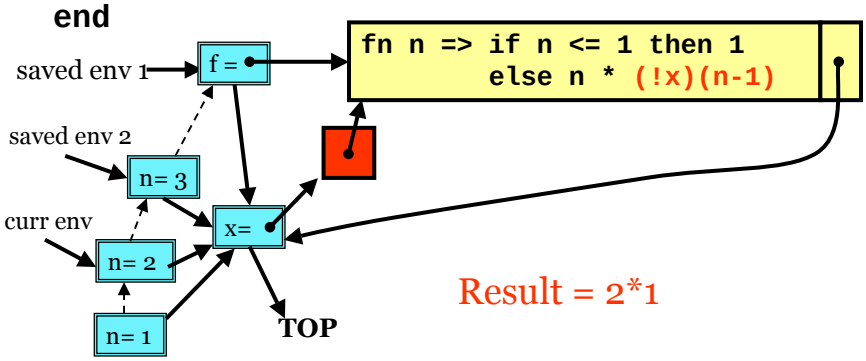
```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



Result = 1

Creating a cycle

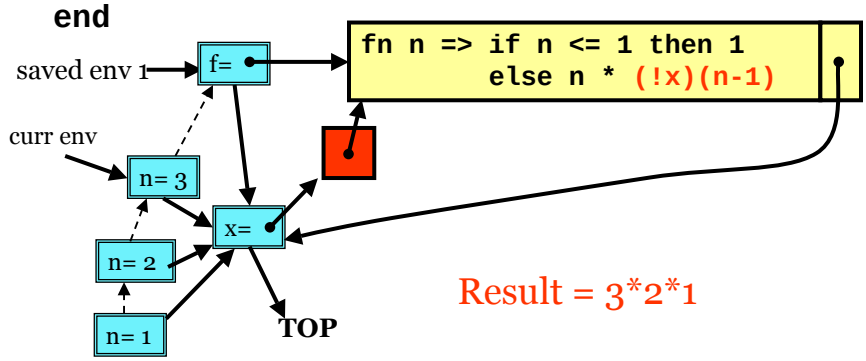
```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



Result = 2*1

Creating a cycle

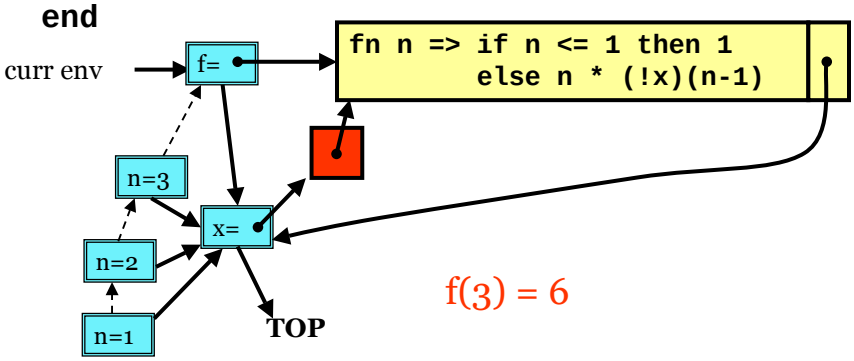
```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



Result = 3*2*1

Creating a cycle

```
let val x = ref (fn x => x)
    val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
   f(3)
end
```



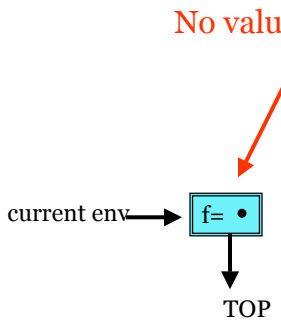
Recursion

```
let fun f(n) =
   if n <= 1 then 1 else n * f(n-1)
in
   f(3)
end
```

1. create a new binding for f **before creating the closure** and extend the current environment with it (but don't put in the value yet.)
2. **now** create a closure for f that uses the extended environment.
3. fix the binding to use the closure!

Recursion

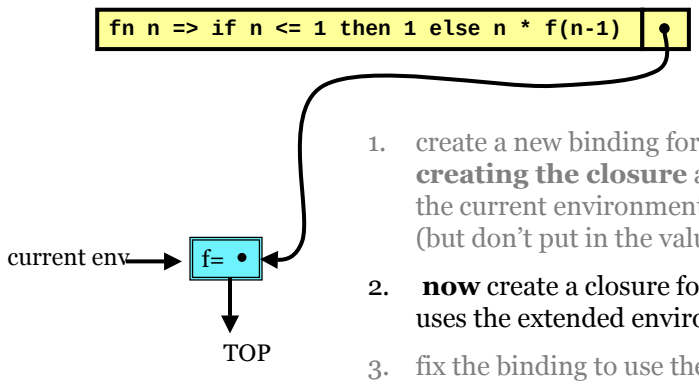
```
let fun f(n) => if n <= 1 then 1 else n * f(n-1)
in
   f(3)
end
```



1. create a new binding for f **before creating the closure** and extend the current environment with it (but don't put in the value yet.)
2. **now** create a closure for f that uses the extended environment.
3. fix the binding to use the closure!

Recursion

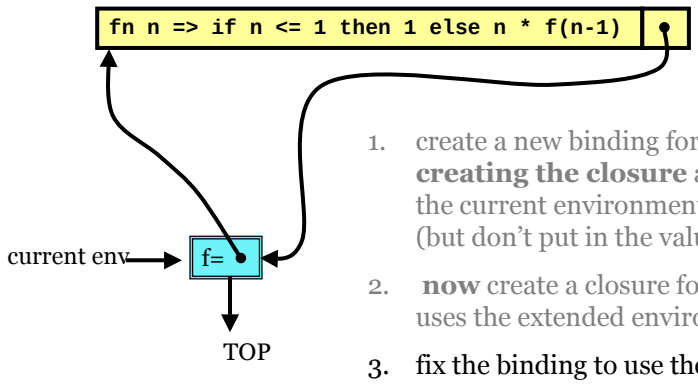
```
let fun f(n) => if n <= 1 then 1 else n * f(n-1)
in
   f(3)
end
```



1. create a new binding for f **before creating the closure** and extend the current environment with it (but don't put in the value yet.)
2. **now** create a closure for f that uses the extended environment.
3. fix the binding to use the closure!

Recursion

```
let fun f(n) => if n <= 1 then 1 else n * f(n-1)
in
   f(3)
end
```

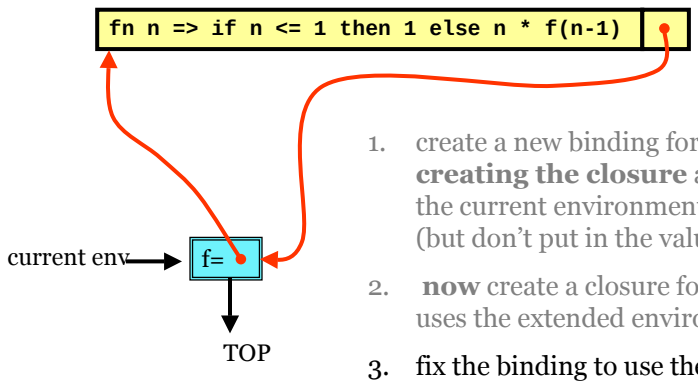


1. create a new binding for f **before creating the closure** and extend the current environment with it (but don't put in the value yet.)
2. **now** create a closure for f that uses the extended environment.
3. fix the binding to use the closure!

Cycle

```
let fun f(n) => if n <= 1 then 1 else n * f(n-1)
in
   f(3)
end
```

- Closure points to environment
- Environment points to closure

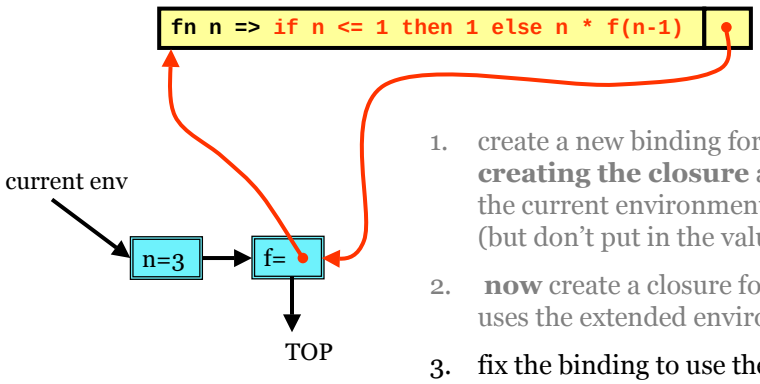


1. create a new binding for f **before creating the closure** and extend the current environment with it (but don't put in the value yet.)
2. **now** create a closure for f that uses the extended environment.
3. fix the binding to use the closure!

Cycle

```
let fun f(n) => if n <= 1 then 1 else n * f(n-1)
in
   f(3)
end
```

- Closure points to environment
- Environment points to closure



1. create a new binding for f **before creating the closure** and extend the current environment with it (but don't put in the value yet.)
2. **now** create a closure for f that uses the extended environment.
3. fix the binding to use the closure!

Comparison