CS 312
Environment Model Diagrams

Spring 2007

The Substitution Model

- Recall the substitution model:
  - Bind variables at “let” constructs
  - Bind function arguments at function calls
  - Substitute bindings in the let body or function body
- Rules:

```
let val x = v in e end -> e{v/x}
(fn x => e)(v) -> e{v/x}
```

- Example:

```
let val x = 3 in x * x end
-> 3 * 3 -> 9
```

Problems

- Substitution model:
  - Useful for understanding program execution
  - Inefficient as an implementation
- Problem 1: We must traverse the code just to perform substitutions; the code will be traversed again when we execute it
- Problem 2: Substitutions can lead to code blow-up

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

Problems

- Problem 3: SM doesn’t work in a straightforward way with imperative features:

```
let val x = ref 1
    val y = x
in  y := 2; !x end
->  ...
-> (ref 1) := 2; !(ref 1)
-> 1 (* wrong *)
```

- We would need to use a memory location “l” instead of “ref 1”, then substitute l into the code, and keep track of l’s value on the side

The Environment Model

- Solution: the environment model
  - Idea: use an environment to store bindings of variables
  - No substitutions
  - Environment is a map from variables to values
  - Values are looked up lazily, when needed
- Example:

Program:

```
let val x = 2
    val y = “hello”
in
    x + size(y)
end
```

Environment:

The Environment Model

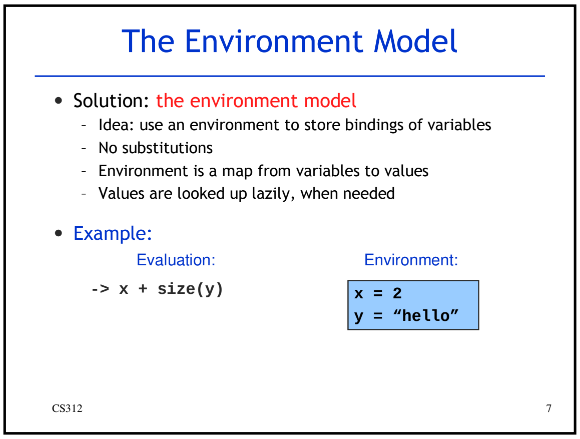
- Solution: the environment model
  - Idea: use an environment to store bindings of variables
  - No substitutions
  - Environment is a map from variables to values
  - Values are looked up lazily, when needed
- Example:

Evaluation:

```
-> let val y = “hello”
   in
        x + size(y)
   end
```

Environment:

```
x = 2
```



The Environment Model

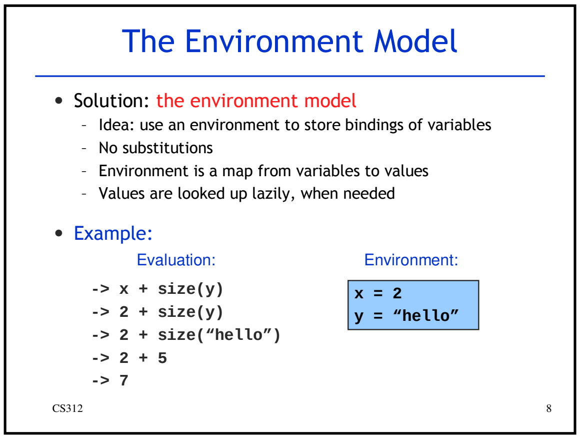
- Solution: the environment model
  - Idea: use an environment to store bindings of variables
  - No substitutions
  - Environment is a map from variables to values
  - Values are looked up lazily, when needed
- Example:

Evaluation:

```
-> x + size(y)
```

Environment:

```
x = 2
y = “hello”
```



The Environment Model

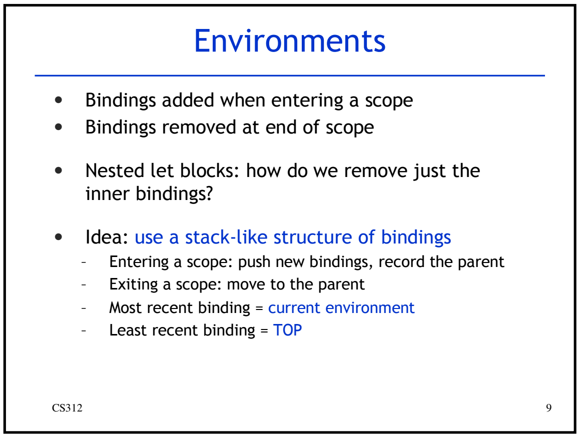
- Solution: the environment model
  - Idea: use an environment to store bindings of variables
  - No substitutions
  - Environment is a map from variables to values
  - Values are looked up lazily, when needed
- Example:

Evaluation:

```
-> x + size(y)
-> 2 + size(y)
-> 2 + size(“hello”)
-> 2 + 5
-> 7
```

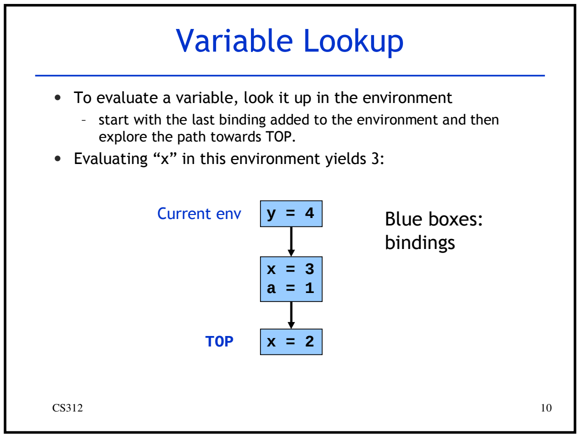
Environment:

```
x = 2
y = “hello”
```



Environments

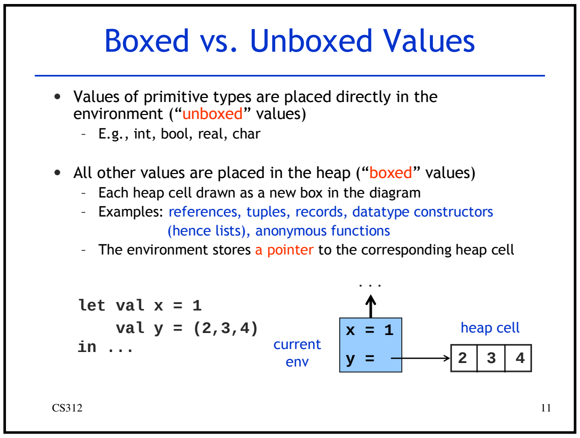
- Bindings added when entering a scope
- Bindings removed at end of scope
- Nested let blocks: how do we remove just the inner bindings?
- Idea: use a stack-like structure of bindings
  - Entering a scope: push new bindings, record the parent
  - Exiting a scope: move to the parent
  - Most recent binding = current environment
  - Least recent binding = TOP



Variable Lookup

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

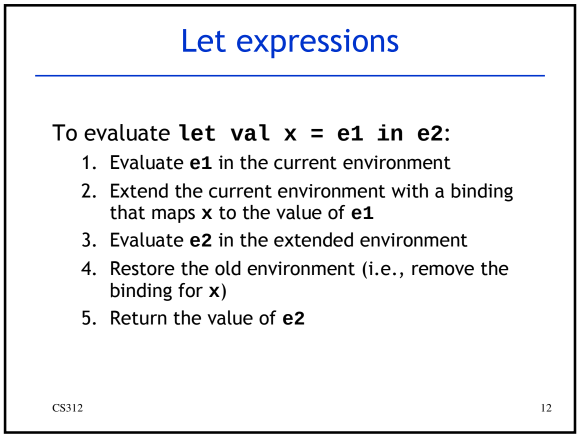
Blue boxes: bindings



Boxed vs. Unboxed Values

- Values of primitive types are placed directly in the environment (“unboxed” values)
  - E.g., int, bool, real, char
- All other values are placed in the heap (“boxed” values)
  - Each heap cell drawn as a new box in the diagram
  - Examples: references, tuples, records, datatype constructors (hence lists), anonymous functions
  - The environment stores a pointer to the corresponding heap cell

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



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

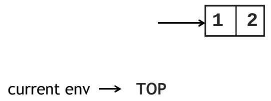
```
let val x = (1,2) in (x,3) end
```

current env → TOP

Let Example

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

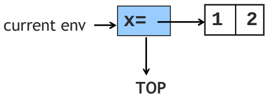
1. Evaluating (1,2) yields a pointer to a heap cell.



Let Example

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

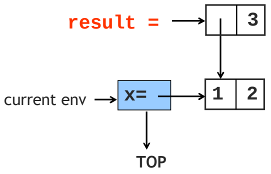
1. Evaluating (1,2) yields a pointer to a heap cell.
2. Extend the environment with a binding for x.



Let Example

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

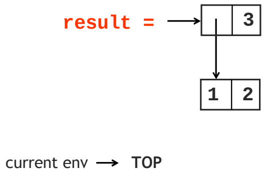
1. Evaluating (1,2) yields a pointer to a heap cell.
2. Extend the environment with a binding for x.
3. Evaluate the body of the let in the current env.



Let Example

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

1. Evaluating (1,2) yields a pointer to a heap cell.
2. Extend the environment with a binding for x.
3. Evaluate the body of the let in the current env.
4. Restore the environment and return the result.



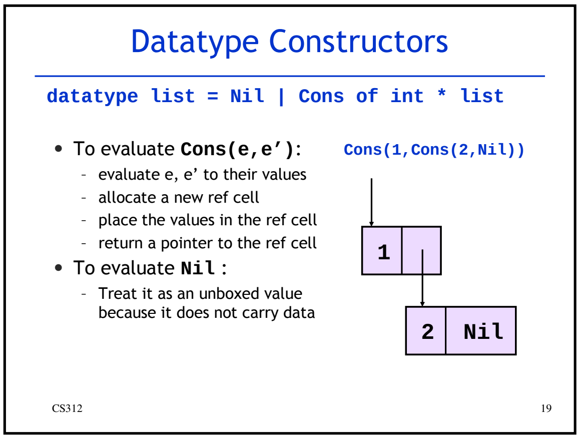
Multiple Declarations

- To evaluate:

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

- Do the same the same thing as you would for:

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



Datatype Constructors

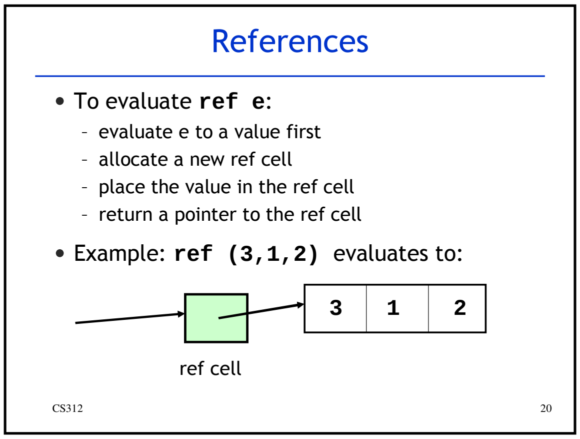
```
datatype list = Nil | Cons of int * list
```

- To evaluate **`Cons(e,e')`**:
  - evaluate e, e' to their values
  - allocate a new ref cell
  - place the values in the ref cell
  - return a pointer to the ref cell
- To evaluate **`Nil`** :
  - Treat it as an unboxed value because it does not carry data

`Cons(1,Cons(2,Nil))`

1

2 Nil

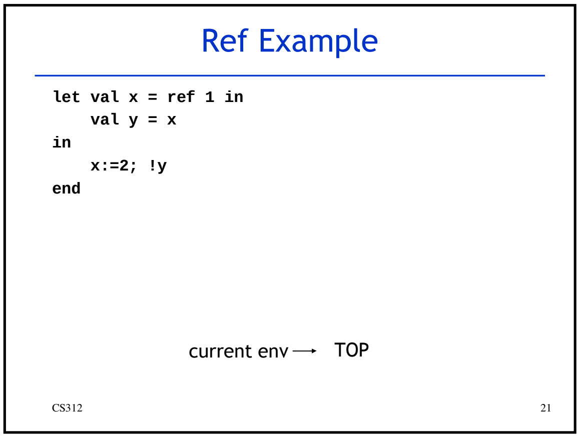


References

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

3 1 2

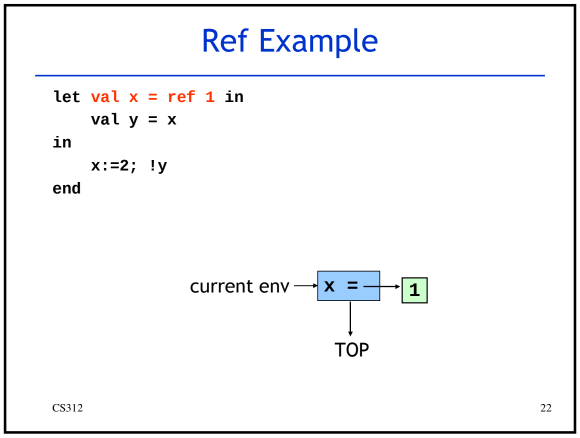
ref cell



Ref Example

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

current env → TOP

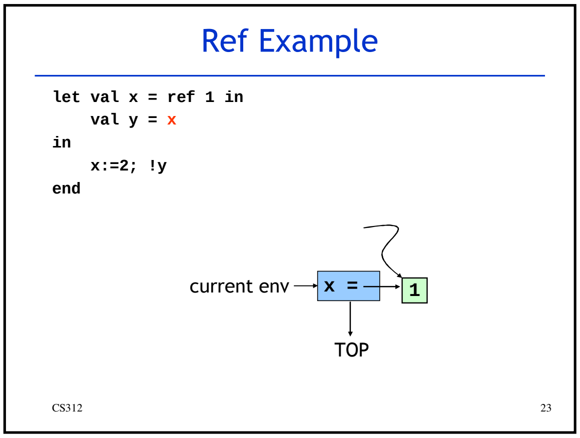


Ref Example

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

current env → x = → 1

TOP

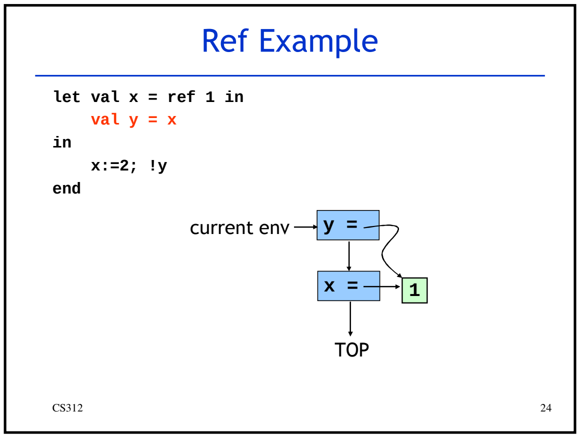


Ref Example

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

current env → x = → 1

TOP



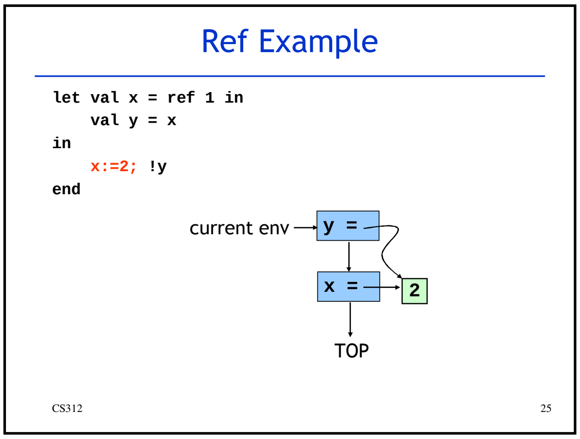
Ref Example

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

current env → y =

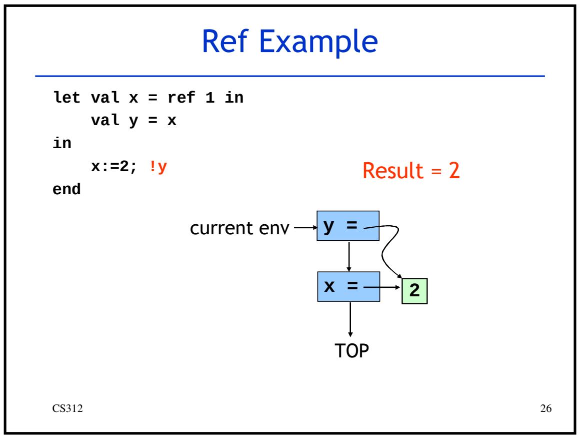
x = → 1

TOP



Ref Example

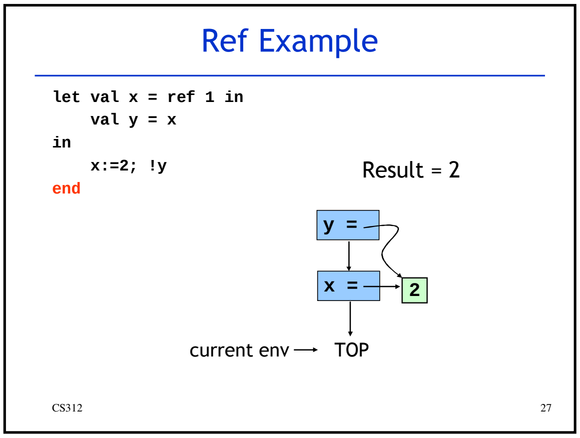
```
let val x = ref 1 in
    val y = x
in
    x:=2; !y
end
```



Ref Example

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

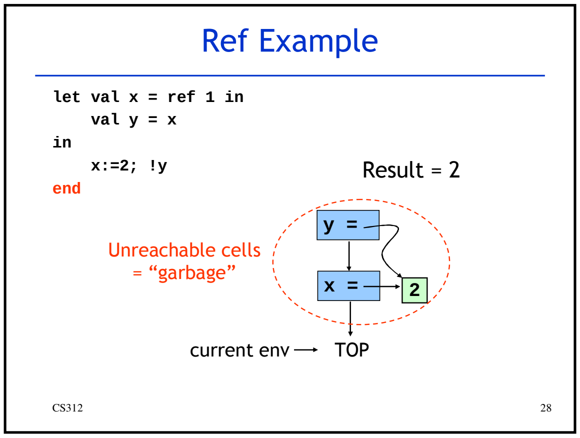
Result = 2



Ref Example

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

Result = 2

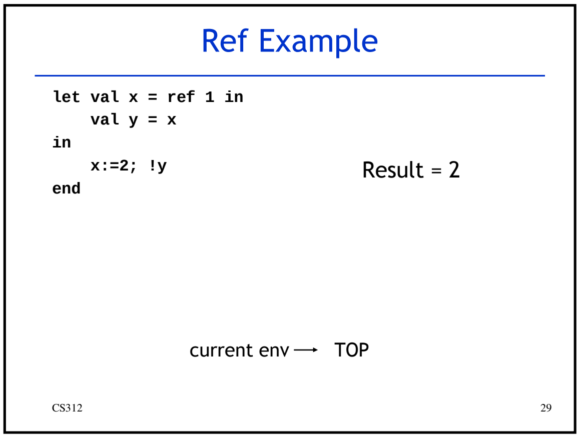


Ref Example

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

Result = 2

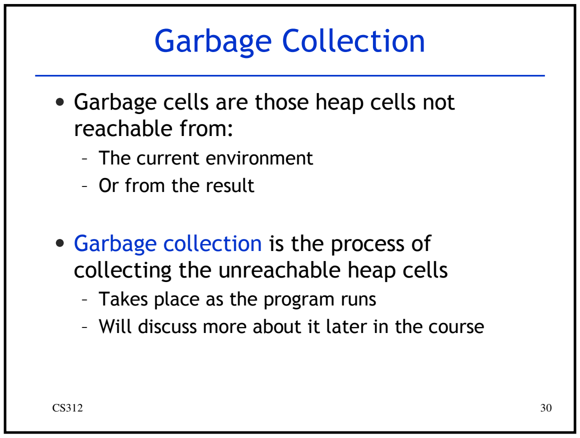
Unreachable cells = “garbage”



Ref Example

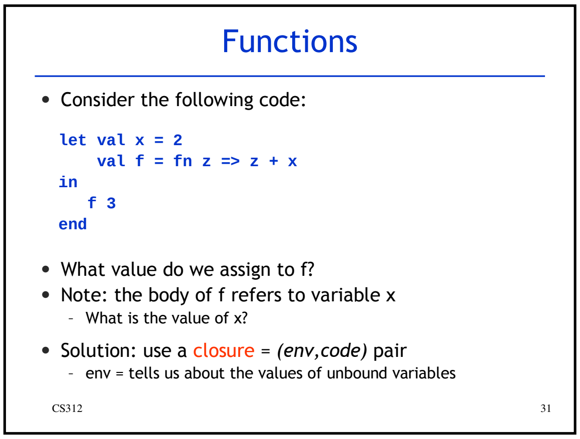
```
let val x = ref 1 in
    val y = x
in
    x:=2; !y
end
```

Result = 2



Garbage Collection

- Garbage cells are those heap cells not reachable from:
  - The current environment
  - Or from the result
- Garbage collection is the process of collecting the unreachable heap cells
  - Takes place as the program runs
  - Will discuss more about it later in the course

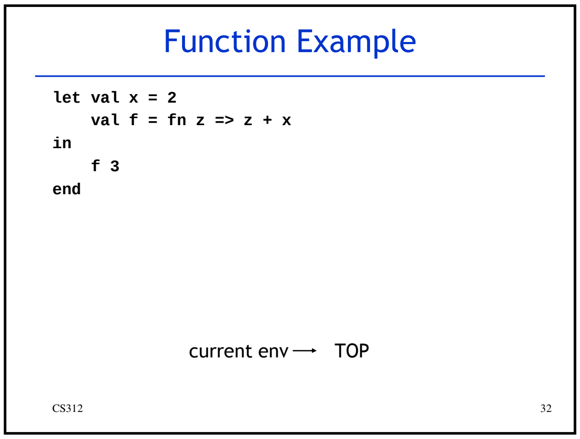


Functions

- Consider the following code:

```
let val x = 2
    val f = fn z => z + x
in
   f 3
end
```

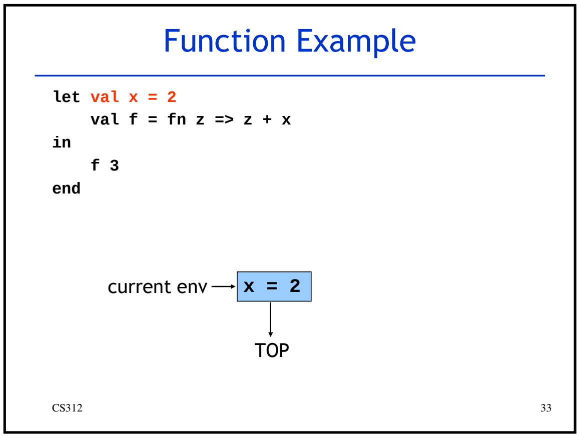
- What value do we assign to f?
- Note: the body of f refers to variable x
  - What is the value of x?
- Solution: use a closure = *(env,code)* pair
  - env = tells us about the values of unbound variables



Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

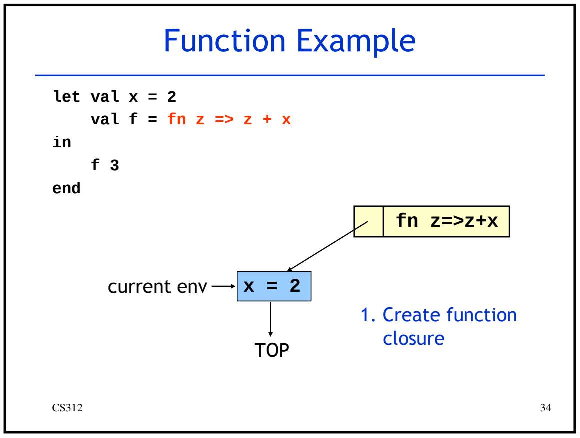
current env ⟶ TOP



Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

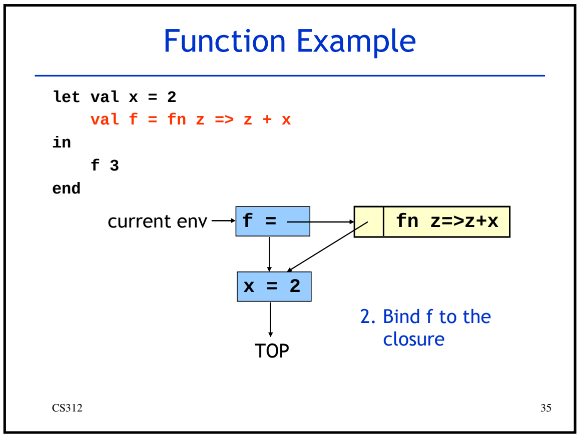
current env ⟶ x = 2 ⟶ TOP



Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

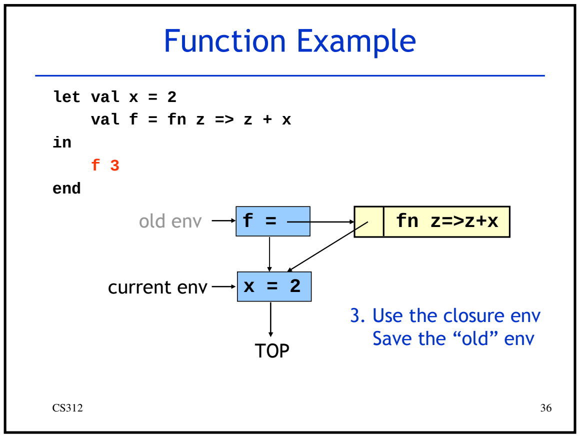
1. Create function closure



Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

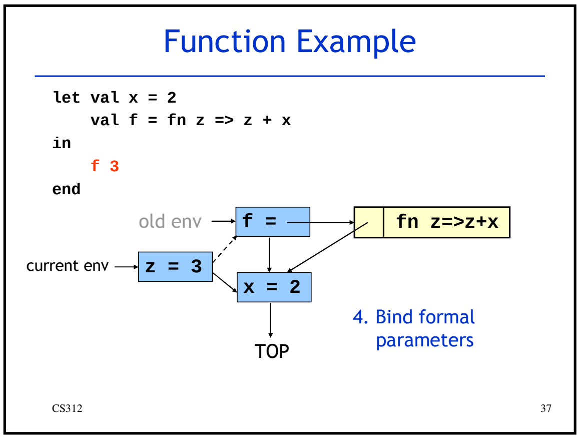
2. Bind f to the closure



Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

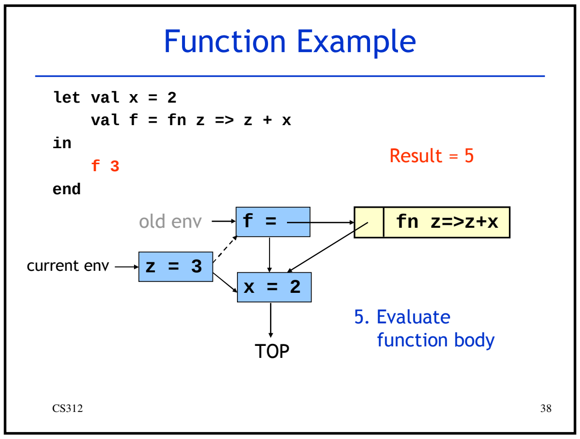
3. Use the closure env Save the “old” env



Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

4. Bind formal parameters

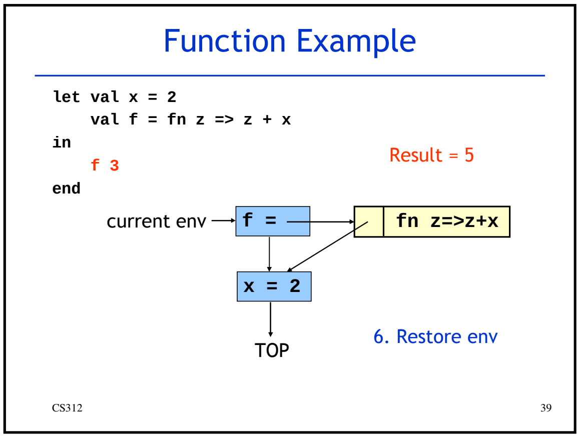


Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

Result = 5

5. Evaluate function body

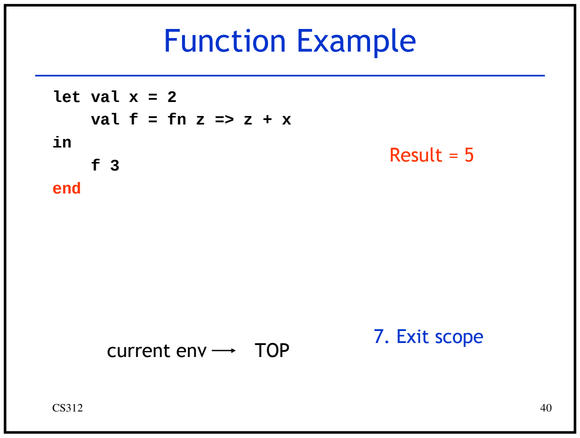


Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

Result = 5

6. Restore env



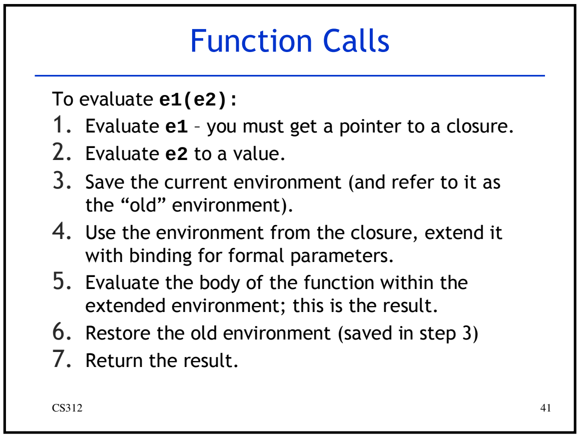
Function Example

```
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

Result = 5

current env → TOP

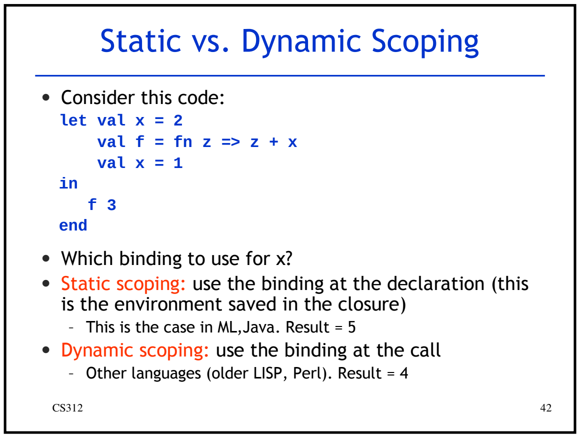
7. Exit scope



Function Calls

To evaluate **e1(e2)**:

1. Evaluate **e1** - you must get a pointer to a closure.
2. Evaluate **e2** to a value.
3. Save the current environment (and refer to it as the “old” environment).
4. Use the environment from the closure, extend it with binding for formal parameters.
5. Evaluate the body of the function within the extended environment; this is the result.
6. Restore the old environment (saved in step 3)
7. Return the result.



Static vs. Dynamic Scoping

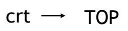
- Consider this code:

```
let val x = 2
    val f = fn z => z + x
    val x = 1
in
   f 3
end
```

- Which binding to use for x?
- Static scoping: use the binding at the declaration (this is the environment saved in the closure)
  - This is the case in ML,Java. Result = 5
- Dynamic scoping: use the binding at the call
  - Other languages (older LISP, Perl). Result = 4

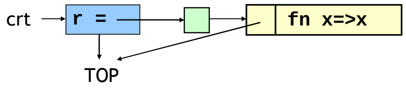
Simulating Recursion

```
let val r = ref (fn x=>x)
    val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end
```



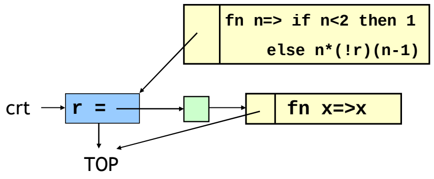
Simulating Recursion

```
let val r = ref (fn x=>x)
    val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end
```



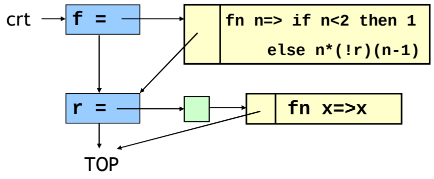
Simulating Recursion

```
let val r = ref (fn x=>x)
    val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end
```



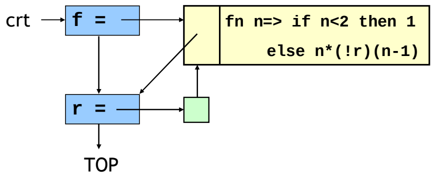
Simulating Recursion

```
let val r = ref (fn x=>x)
    val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end
```



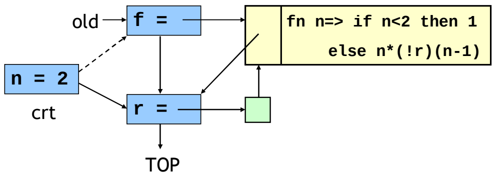
Simulating Recursion

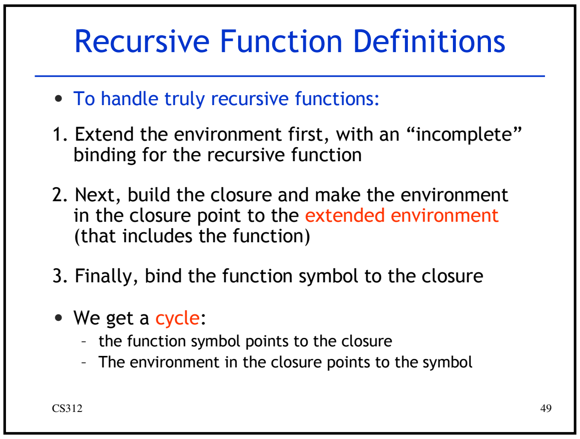
```
let val r = ref (fn x=>x)
    val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end
```



Simulating Recursion

```
let val r = ref (fn x=>x)
    val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end
```



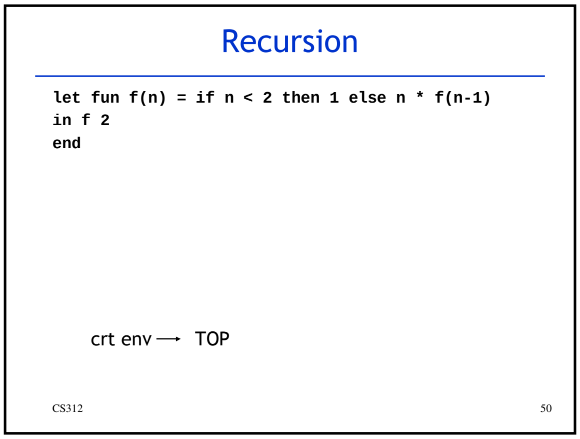


Recursive Function Definitions

- To handle truly recursive functions:

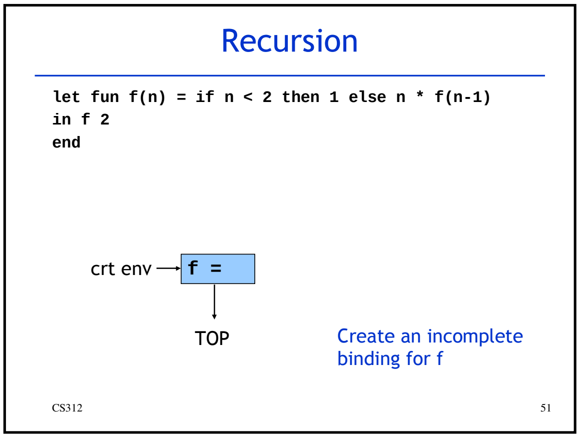
1. Extend the environment first, with an “incomplete” binding for the recursive function
2. Next, build the closure and make the environment in the closure point to the extended environment (that includes the function)
3. Finally, bind the function symbol to the closure

- We get a cycle:
  - the function symbol points to the closure
  - The environment in the closure points to the symbol



Recursion

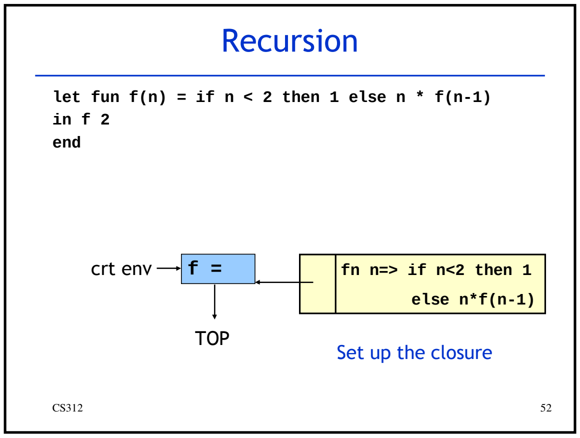
```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```



Recursion

```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

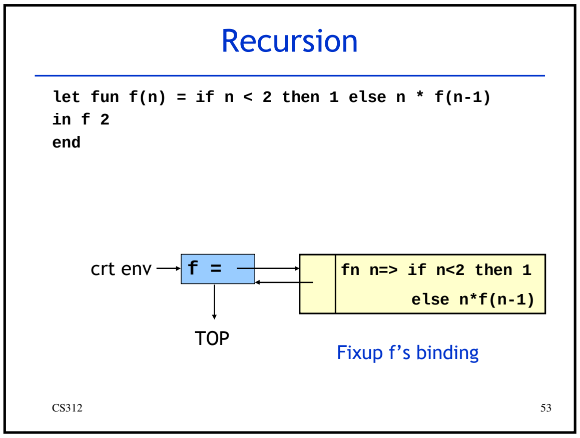
Create an incomplete binding for f



Recursion

```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

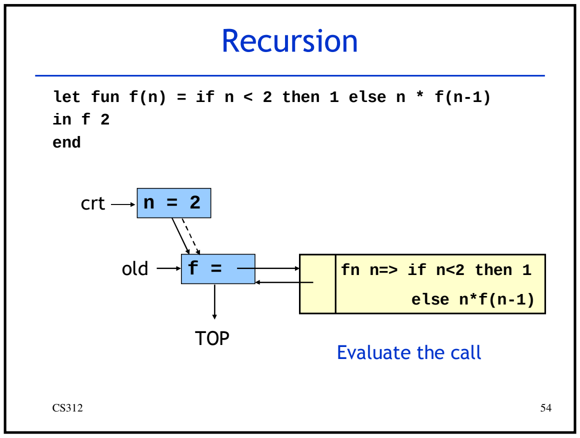
Set up the closure



Recursion

```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

Fixup f’s binding



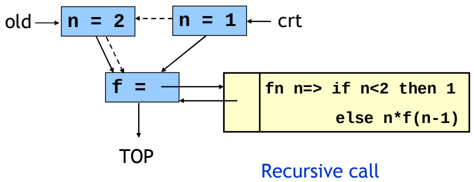
Recursion

```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

Evaluate the call

Recursion

```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

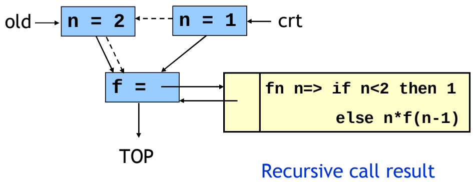


Recursive call

Recursion

```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

Result = 1

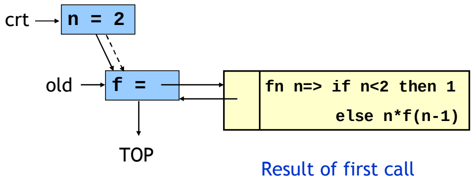


Recursive call result

Recursion

```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

Result = 2

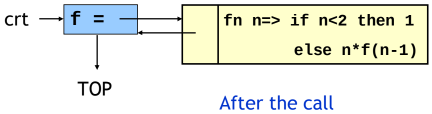


Result of first call

Recursion

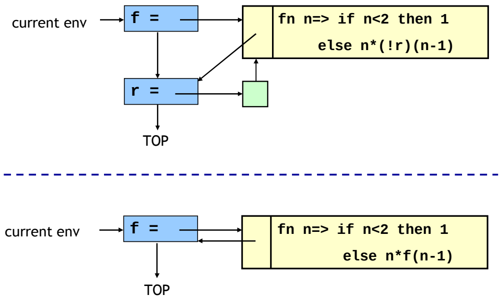
```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

Result = 2



After the call

Comparison


current env
f =
fn n=> if n<2 then 1
else n*(!r)(n-1)
r =
TOP
current env
f =
fn n=> if n<2 then 1
else n*f(n-1)
TOP