## Recursion Using a Cyclic Structure

- ⟨⟨PLAI 9.1⟩⟩
- This doesn't work because `fact` is not bound (also, we need to impliment `if`)

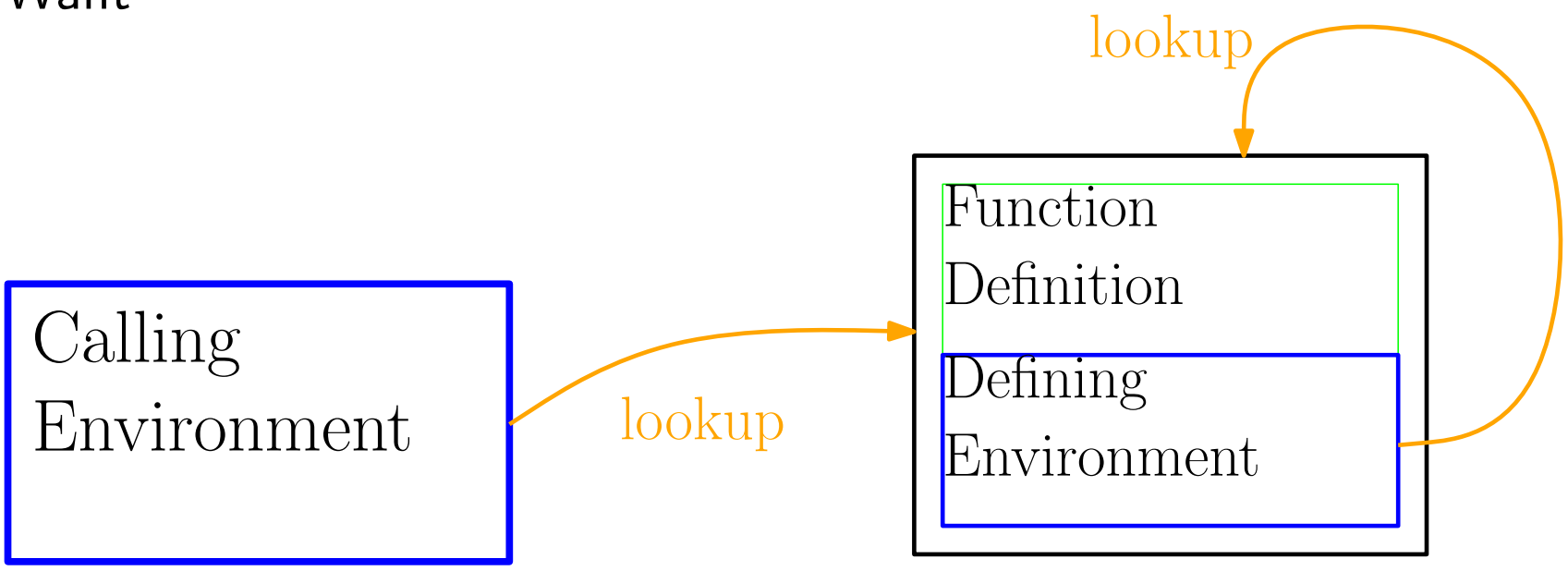
```
{with {fact
       {fun {n}
            {if {zero? n} 1
                {* n {call fact {- n 1}}}}}}
      {call fact 5}}
```

- We need some primitive other than `with`

2

```
{rec {fact {fun {n}
                {if {zero? n} 1
                    {* n {call fact {- n 1}}}}}}}
  {call fact 5}}
```

- Want



## Boxes and Mutation

- To implement a circular structure, we will use **side-effects**.
- Specifically, a new kind of Racket value which supports mutation: a box.

3

```
(define glarf (box 7))

(* 6 (unbox glarf))

(set-box! glarf 17)
(* 6 (unbox glarf))
```

## Racket facilities for working with side-effects

- To evaluate a sequence of Racket expressions, you wrap them in a ‘begin’ expression.

4

```
(begin
  (display "one plus one is")
  2)
```

Racket facilities for working with side-effects

- To evaluate a sequence of Racket expressions, you wrap them in a 'begin' expression.

```
(begin
  (display "one plus one is")
  2)
```

1. An important thing to note is that 'set-box!' is much like 'display' etc, it returns a value that is not printed in the Racket REPL, because there is no point in using the result of a 'set-box!', it is called for the side-effect it generates.
2. Languages like C blur this distinction between returning a value and a side-effect with its assignment statement.
3. Actually we now have side effects of two kinds: mutation of state, and I/O (at least the O part). (Actually, there is also infinite looping that can be viewed as another form of a side effect.)

- Some places with an "implicit 'begin'": the body of a function (or any lambda expression), the body of a 'let' (and 'let'-relatives), the consequence positions in 'cond', 'match', and 'cases' clauses.

5
```
(cond
  [#t (display "one plus one is") 11])
```

- 'cond' without an 'else' can make sense, if all you're using it it for is side-effects.
- 'when' & 'unless' are one-sided ifs

6
```
(when #t
  (display "two plus two is ")
  (display "four"))
```

- When any one of these things is used, you can tell that side-effects are probably involved, because there is no point in any of them otherwise.
- Any name that ends with a '!' ("bang") is used to mark a function that changes state

## Creating a cycle using using boxes

- So how do we create a cycle?
- Boxes have any value, and they can be put in other values like lists, so we can get a circular value like



```
(define foo (list 1 (box 3)))
(set-box! (second foo) foo)
(display foo)
```

- Note that the racket printer detects the cycle.
- This exact trick is hard (impossible?) to do with the type checker enabled, but it will turn out we can still use boxes because we already have a "union-type" defined, namely VAL.

## Implementing a Circular Environment

- We want to add Rec variant, with a recursive env.

8

```
...
[With (id : symbol) (named-expr : FLANG)
    (bound-body : FLANG)]
[Rec (id : symbol) (named-expr : FLANG)
    (bound-body : FLANG)]
```

- change environments so they hold (boxof VAL) instead of VAL,

9

```
(define-type ENV
  [EmptyEnv]
  [Extend (name : symbol) (val : (boxof VAL))
      (env : ENV)])
```

- `lookup` returns a box, which behaves like an **lvalue** in C (this isn't needed for Rec, but makes implementing `set!` in FLANG easy)

10

```
(define (lookup name env)
  (type-case ENV env
    [(EmptyEnv) (error 'lookup "free
       variable")]
    [(Extend id boxed-val rest-env)
     (if (eq? id name) boxed-val
                 (lookup name rest-env))]))

(define foo  (Extend 'x (box (NumV 42))
                     (EmptyEnv)))
(set-box! (lookup 'x foo) (NumV 1))
(test (val->number (unbox (lookup 'x foo))) 1)
```

- To extend an environment with Rec, we use



```
(define (extend-rec id expr rest-env)
  (let ([new-cell (box (NumV 42))])
    (let ([new-env (Extend id new-cell
                     rest-env)])
      (let ([value (eval expr new-env)])
        (begin
          (set-box! new-cell value)
          new-env)))))

(define foo
  (extend-rec 'f (Fun 'x (Call (Id 'f)
                               (Id 'x)))
              (EmptyEnv)))
(lookup 'f foo)
```

- We can make this shorter with `let*`

12
```
(define (extend-rec id expr rest-env)
  (let* ([new-cell (box (NumV 42))]
         [new-env  (Extend id new-cell
                     rest-env)]
         [value    (eval expr new-env)])
    (begin (set-box! new-cell value) new-env)))
```

- 'let*' be read almost as a C/Java-ish kind of code:

13
```
fun extend_rec(id, expr, rest_env) {
  new_cell  = new NumV(42);
  new_env   = Extend(id, new_cell, rest_env);
  value     = eval(expr, new_env);
  *new_cell = value;
  return new_env;
}
```

- Given `extend-rec`, the change to eval is trivial



```
(define (eval expr env)
  (type-case FLANG expr
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (box (eval
              named-expr env)) env))]
    [(Rec bound-id named-expr bound-body)
     (eval bound-body
           (extend-rec bound-id named-expr
              env))]
```

```
(trace lookup)

(test (run `{with {x 3}
              {with {f {fun {y} {+ x y}}}
                {with {x 5}
                  {call f 4}}}})
      7)
(test (run `{call {with {x 3}
                    {fun {y} {+ x y}}}
                  4})
      7)
```

- Hurray, we made an infinite loop!



```
(trace eval)
(run `{rec {f {fun {y} {call f 0}}}
          {call f 0}})
```

- Time to add an if

17
```
(define (eval expr env)
  (type-case FLANG expr
      ⋮
    [(If test then-part else-part)
        (if (eq? 0 (val->number (eval test
                                env)))
            (eval else-part env)
            (eval then-part env))]
```

18
```
(test (run `{if 0 1 0})  0)
(test/exn (run `{if {fun {y} y} 1 0})
          "not a number")
```

- Our old friend



```
(run
 `{rec
   {fact
    {fun {n}
         {if n
             {* n {call fact {- n 1}}}
             1}}}
   {call fact 5}})
```