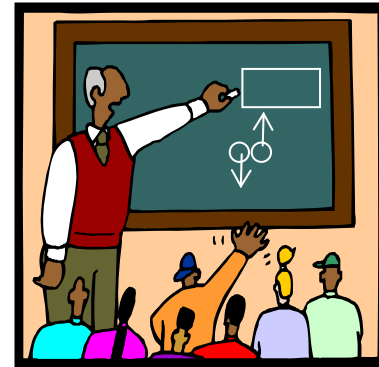
6.001 recitation 14 4/6/06

- environment diagrams



Dr. Kimberle Koile

environment model

How does the environment model differ from the substitution model?
What's an environment? How do we represent it? What's an "enclosing environment"?
What does the overall shape of environment diagrams connote about a language?

Substitution model: variable = name for value
procedure = functional description

environment model: variable = place into which store value
procedure = object with inherited context
expressions now only have meaning with respect to a context

environment = binding context; represented as a sequence of frames

enclosing environment = sequence of frames "above" a frame

environment diagram review

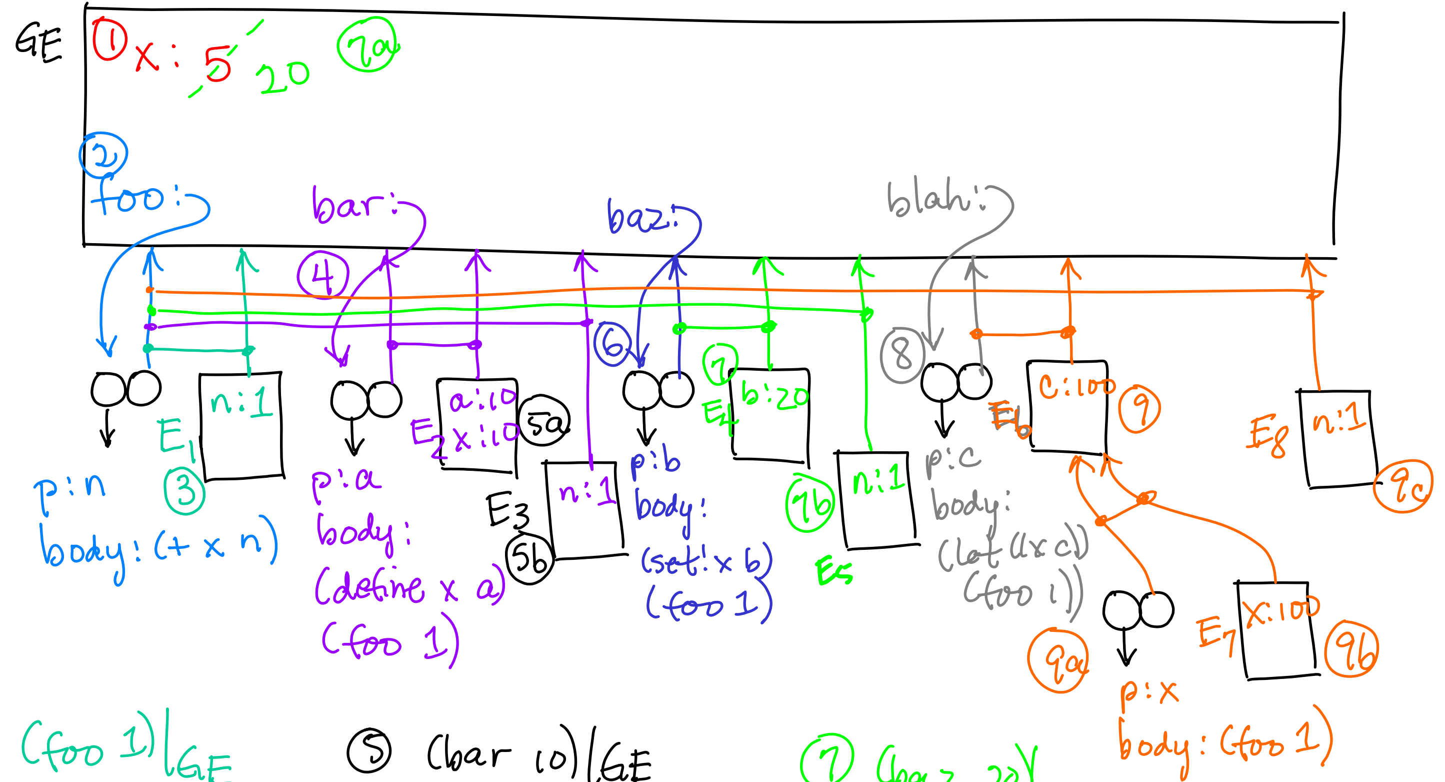
Eval

- **name:** look up name in the (lowest frame of the) current environment. If you find it, return the value, otherwise do the lookup in the parent frames of the current environment.
- **(lambda (params) body):** create a “double bubble” with code pointer to params and body, and env pointer to current environment.
- **(define name value):** evaluate value and create or replace the binding for name in the (lowest frame of the) current environment.
- **(set! name value):** evaluate value and replace the closest binding for name in the environment frame chain, starting with the lowest frame of the current environment
- (proc args …): evaluate proc and args, then do the apply steps
- Otherwise, follow the correct rule (if, cond, etc.).

Apply

1. Drop a new frame.
2. Link frame ptr of new frame to (lowest frame of the) environment referenced by env pointer of double bubble.
3. Bind params of double bubble in the new frame.
4. Eval the body in the new frame.

examples (solution)



① 1. (define x 5)

② 2. (define (foo n)
(+ x n))

③ 3. (foo 1) => 6

④ 4. (define (bar a)
(define x a)
(foo 1))

⑤ 5. (bar 10) => 6

⑥ 6. (define (baz b)
(set! x b)
(foo 1))

⑦ 7. (baz 20) => 21

⑧ 8. (define (blah c)
(let ((x c))
(foo 1)))

⑨ 9. (blah 100) ⇒ 21

③ (foo 1)|GE
(+ x n)|E1
(+ 5 1)|E1 ⇒ 6

⑤ (bar 10)|GE
⑤a (define x a)|E2
(foo 1)|E2
⑤b (+ x n)|E3
(+ 5 1)|E3 ⇒ 6

Note: For dynamic scoping, E3 would point to E2, the calling frame.

⑦ (baz 20)|GE
⑦a (set! x b)|E4
⑦b (foo 1)|E4
(+ x n)|E5
(+ 20 1)|E5 ⇒ 21

⑨ (blah 100)|GE
(let ((x c)) (foo 1))|E6 ⇒ ((λ(x) (foo 1)) ~~c~~ 100)
⑨a ⑨b

⑨c (foo 1)|E7
(+ x n)|E8
(+ 20 1)|E8 ⇒ 21

let, lambda, set!, define solution

Let turns into a lambda that makes a local frame: desugar the let into the lambda, then immediately apply the lambda:

e.g (let ((a 0)) (foo a)) => ((lambda (a) (foo a)) 0)| GE

Lambda becomes a procedure object in the **current** environment (i.e., the lambda is captured by the current environment). So even if applied elsewhere, free variables are looked up in the **environment of definition**, not the environment of call.

Set! works on the nearest frame in the current environment chain that contains a binding for x; this may not be the current frame. Causes an error if a bining for x does not exist.

Define always works on the current frame only. Replaces a binding for x if it existed previously or creates a new binding for x if it did not exist previously.

Example of set! vs define:

```
① (define x 0)
② (define f
     (lambda (y)  (define x (+ y  10))  x))
③ (define g
     (lambda (y)  (set! x (+ y  10))  x))
④ (f 5)  => 15

     x => 0

⑤ (g 5)  => 15

     x => 15
```

4a (define x (+ y 10))|E1

5a (set! x (+ y 10))|E2