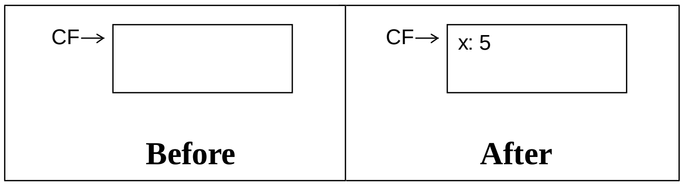
By Joshua Cantrell
jjc@cory.berkeley.edu

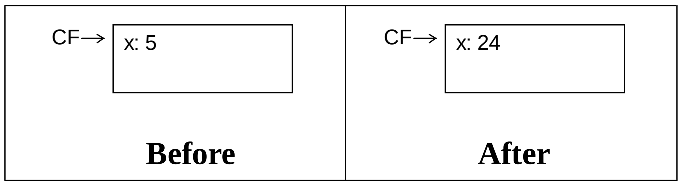
How to Draw Environment Diagrams

Environments are built from frames, which contain variables , which contain pointers to data. These frames are linked together in a way that makes drawing environment diagrams useful for conceptulizing how they work. The steps used to build environments all begin with evaluating expressions in a frame. The current frame being worked on can be isolated from other frames when dealing with evaluation when you don't search for variables to evaluate.

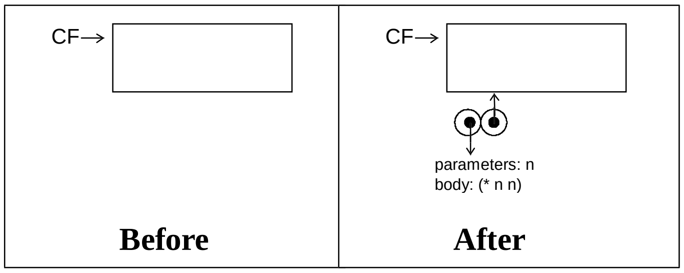
How to bind variables names in a frame:
If *define* is evaluated then the value is said to be bound to the current frame, which I'll signify by CF. For example, if I evaluated the expression "(define x 5)" then this is what would happen.



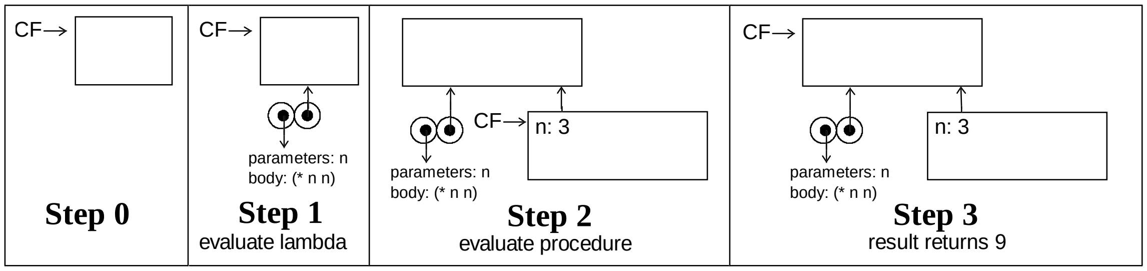
How to change the value of an already bound variable in a frame:
In the case where a variable's name is already defined or bound, then we use the *set!* command instead of *define*. When *set!* is evaluated and cannot find a variable with the given name in the current environment, it returns an error, but when it does find the given variable name, it redirects the name's pointer to the value given. For example, if I evaluted the expression "(set! x 24)" then this is what would happen.



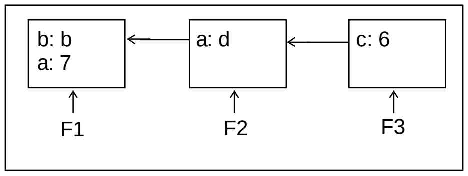
When a lambda is evaluated in an environment:
The evaluation of *lambda* creates a procedure. It is important to note that the body of a *lambda* is not evaluated when the *lambda* is evaluated. The body is only evaluated when the resulting procedure is evaluated. The procedure is drawn as two circles beside each other, the left one pointing at the arguments and body of the resultant procedure and the right one pointing at the current frame the *lambda* is evaluated within. Note that a procedure does not have to be bound to a variable name to exist and be used. To link the procedure to a variable name, the *define* or *set!* procedures can be used. For example, if I evaluated the expression "(lambda (n) (* n n))" then this would happen.



When a procedure is evaluated in an environment:
When a procedure is evaluated, with or without arguments, it creates a new frame, and its evaluation is started by this new frame becoming the current frame. The new frame has a pointer to the frame the procedure points to, and the parameter names are bound in the frame with the values the procedure was called with. When the procedure is done being evaluated, the current frame points to the same frame it pointed to before the procedure was evaluated. Let's consider evaluating the expression "((lambda (n) (* n n)) 3)". Notice how first the *lambda* is evaluated and then the procedure is evaluated afterwards. You can't create a frame before having a procedure to create it.



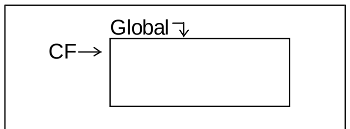
How to find values given variable names:
Variables are found by first searching the current frame for the given variable name. If that fails, then the search continues by going to the frame pointed to by the current frame, and searching there. Still failing, the search will continue following the pointer of frames to other frames until the first occurance of the variable name, and then the variable is found. Given the string of frames below, the variable 'a' is unbound in frame F3, but bound in frames F1 and F2. If the variable 'a' was sought for in F3, it would not find it, and move on to F2 where it would find it. Notice that F2 and F3 cannot reach the variable 'a' in F1 because it is bound in a frame before F1. This series of frames is referred to as an *environment*.



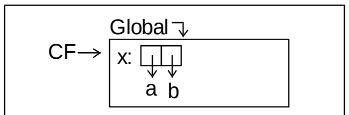
By Joshua Cantrell
jjc@cory.berkeley.edu

Following the execution of Scheme commands using an environment diagram:
An important skill is being able to invision these environment diagrams in your mind when working with Scheme. These can be great aids to you when debugging a complex program or you need to decifier someone else's Scheme code. I'll go over a simple, but complete example of the execution of some Scheme commands. One thing to notice is that frames are never removed in Scheme conceptually, but linger around for eternity. The real interpreter makes a list of all data, frames, and procedures that are currently being pointed to by something else and still has the possibility of being used, then removes everything that isn't needed (this is called garbage collecting). I'll not do any garbage collecting in this example just to show you what the conceptual picture of a growing environment looks like. Some of the steps that just return to the Global environment are left out to save space.

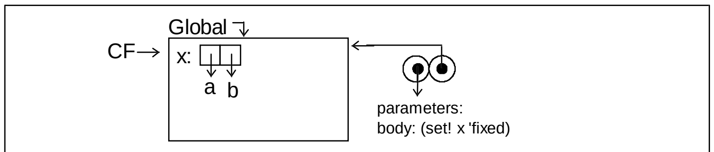
```
> (define x (cons 'a 'b))
> (define (fix-x) (set! x 'fixed))
> (define (make-machine foot pedal)
    (set! x 'break)
    (let ((a 'an-a)
          (x 'an-x))
      (lambda (msg)
        (cond ((eq? msg 'foot) foot)
              ((eq? msg 'pedal) pedal)
              ((eq? msg 'fix)
               (define x 'not-x)
               (fix-x))
              (else (cons a x)) ))))
> (define bike (make-machine 'shoe 'gear))
> (bike 'foot)
> (bike 'fix)
> (bike 'foo)
```



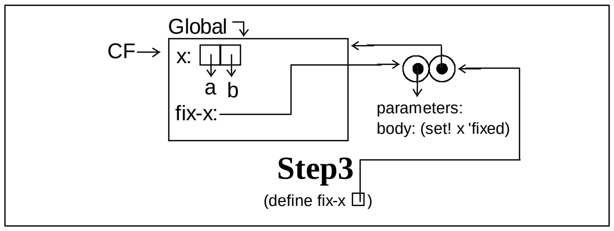
Step0



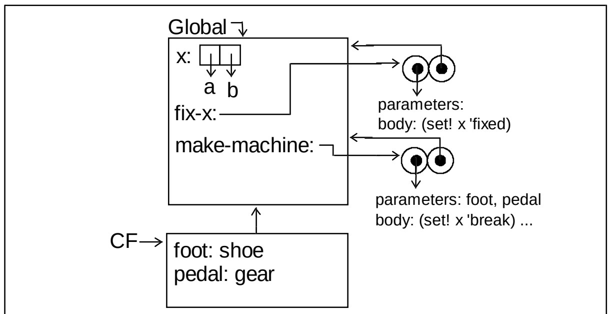
Step1
(define x (cons 'a 'b))



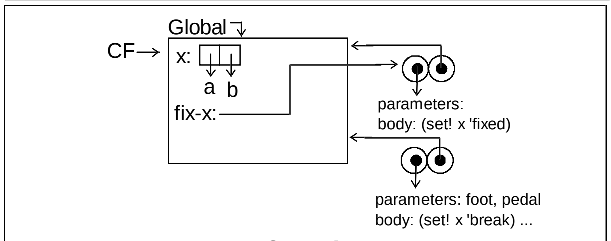
Step2
(define (fix-x) (set! x 'fixed))



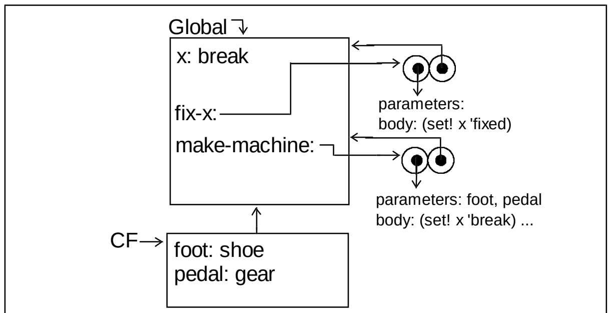
Step3
(define fix-x □)



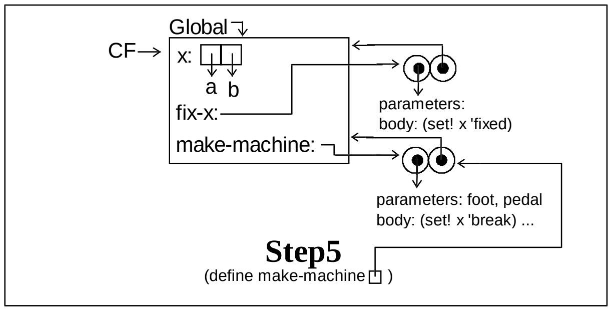
Step6
(define bike (make-machine 'shoe 'gear))



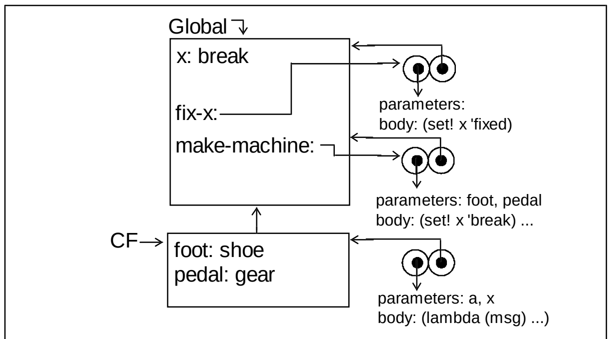
Step4
(define (make-machine ...) ...)



Step7
(set! x 'break)

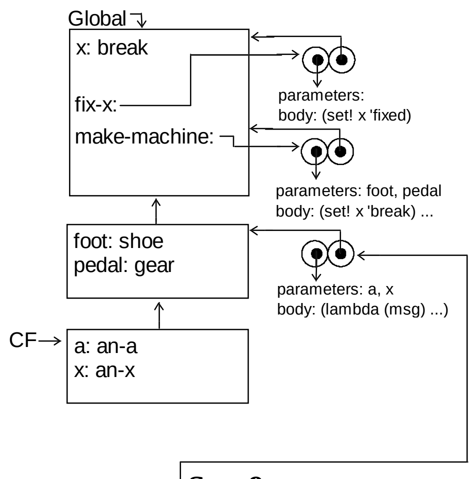


Step5
(define make-machine □)



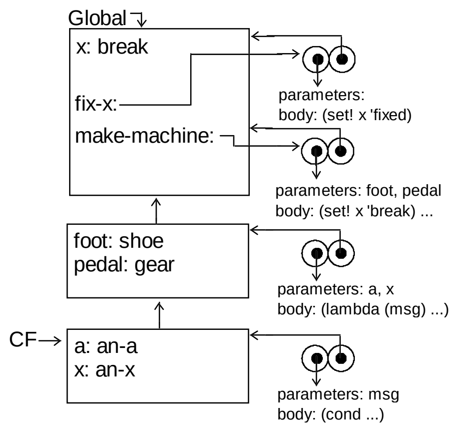
Step8
(let ((a 'an-a) ...) ...)

By Joshua Cantrell
jjc@cory.berkeley.edu



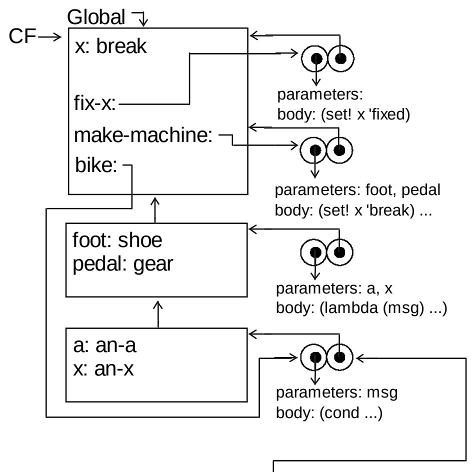
Step9

(☐ 'an-a 'an-x)



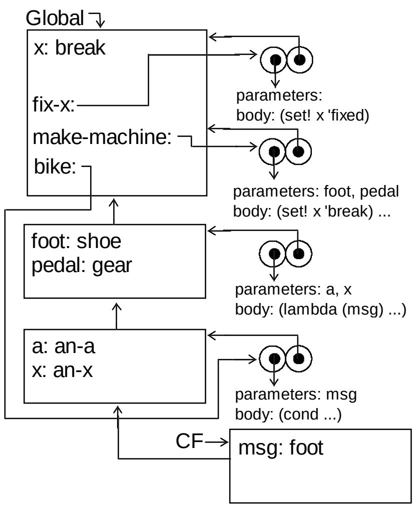
Step10

(lambda (msg) (cond ...))



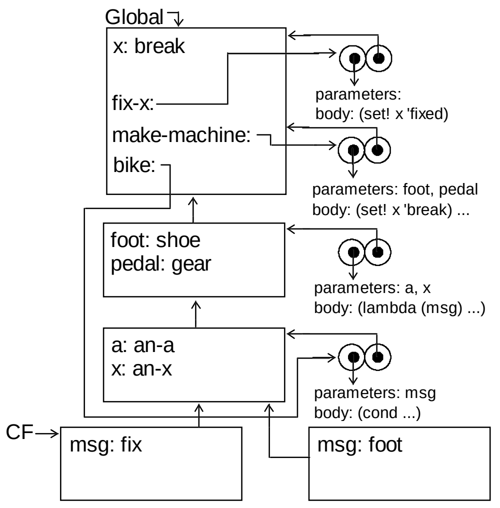
Step11

(define bike ☐)



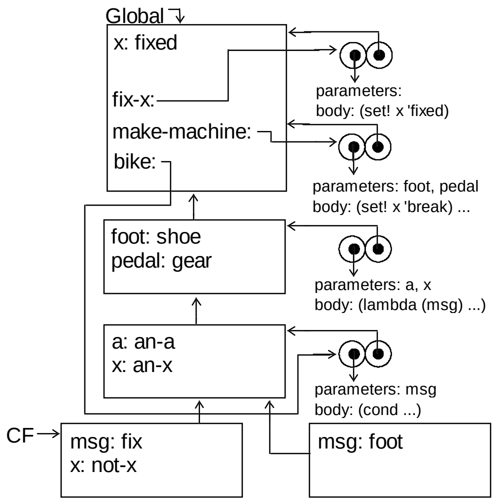
Step12

(bike 'foot) ⟶ shoe



Step13

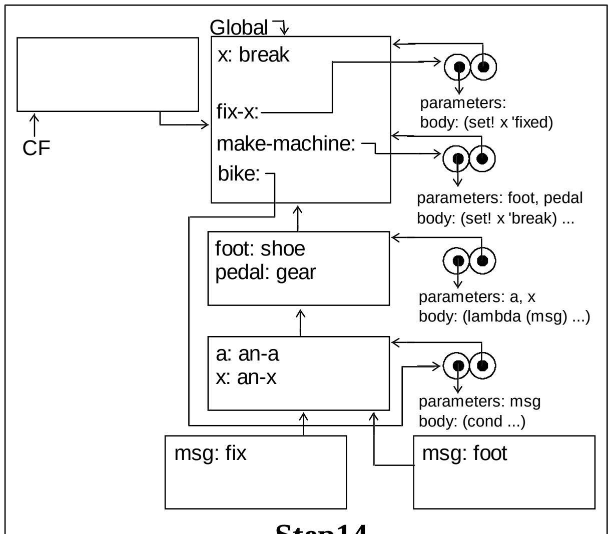
(bike 'fix)



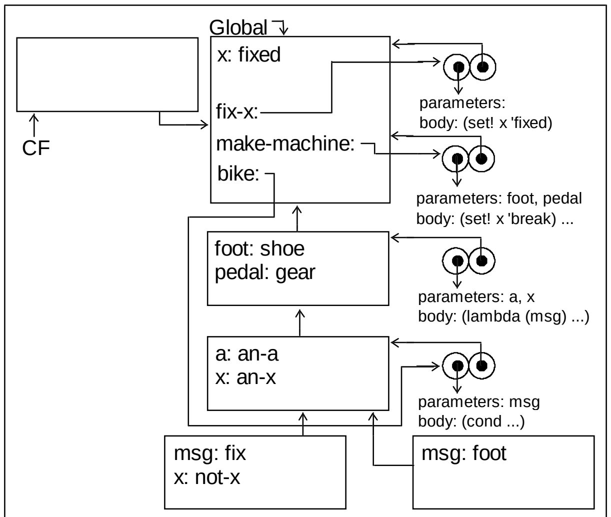
Step15

(define x 'not-x)

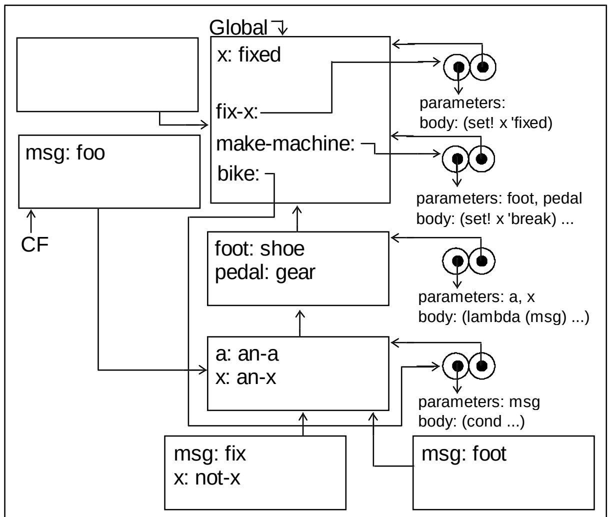
By Joshua Cantrell
jjc@cory.berkeley.edu



Step14
(fix-x)



Step16
(set! x 'fixed)



Step17
(bike 'foo)$\longrightarrow$(an-a . an-x)