1. A **binding** means an association between a Scheme symbol (called a "**name**") and a Scheme value (called a "**value**"). A name can be the name of a procedure or variable, an argument to a procedure, a variable in a `let` expression, etc. For instance, if in the Scheme interpreter you type:
   ```
   (define n 10)
   ```
   then the fact that the name `n` refers to the number `10` is stored as a binding in an environment.
2. A **frame** is a collection of zero or more bindings as well as a pointer to another frame, which is called its "enclosing environment". There is one special frame which does not have a pointer to another frame, and that is the **top-level environment** (also known as the **global environment**).
3. An **environment** is a sequence of frames, starting from a particular frame

and going back through each frame's enclosing environment until the global environment is reached. Note that every frame defines a particular environment, but the environment is more than just the one frame (except for the global environment, which is just one frame). A given environment diagram will have one environment for every frame in the environment.

In the environment model, an expression is always evaluated in the context of a particular environment. The environment determines what values correspond to the names occurring in the expression. The purpose of an environment is to provide a way to associate a value with a particular name. The way this works is that the first frame in the environment is searched to see if it contains a binding for that name. If so, the associated value is used. If not, the first frame of the enclosing

environment is searched, and so on up to the global environment. If the name is not found there, an error is reported.

Environment model:

1. `define` creates bindings in a frame. Top-level `define`s create bindings in the top-level (global) environment. Internal defines create bindings in the environment in which they are evaluated.
2. `set!` changes bindings. `set!` **never** creates a binding. Top-level `set!`s modify bindings in the global environment. Any other `set!` will modify the first binding it finds, starting from the environment in which it was evaluated and proceeding back through the chain of environments until

it reaches the global environment. If it doesn't find a suitable binding, it generates an error message.

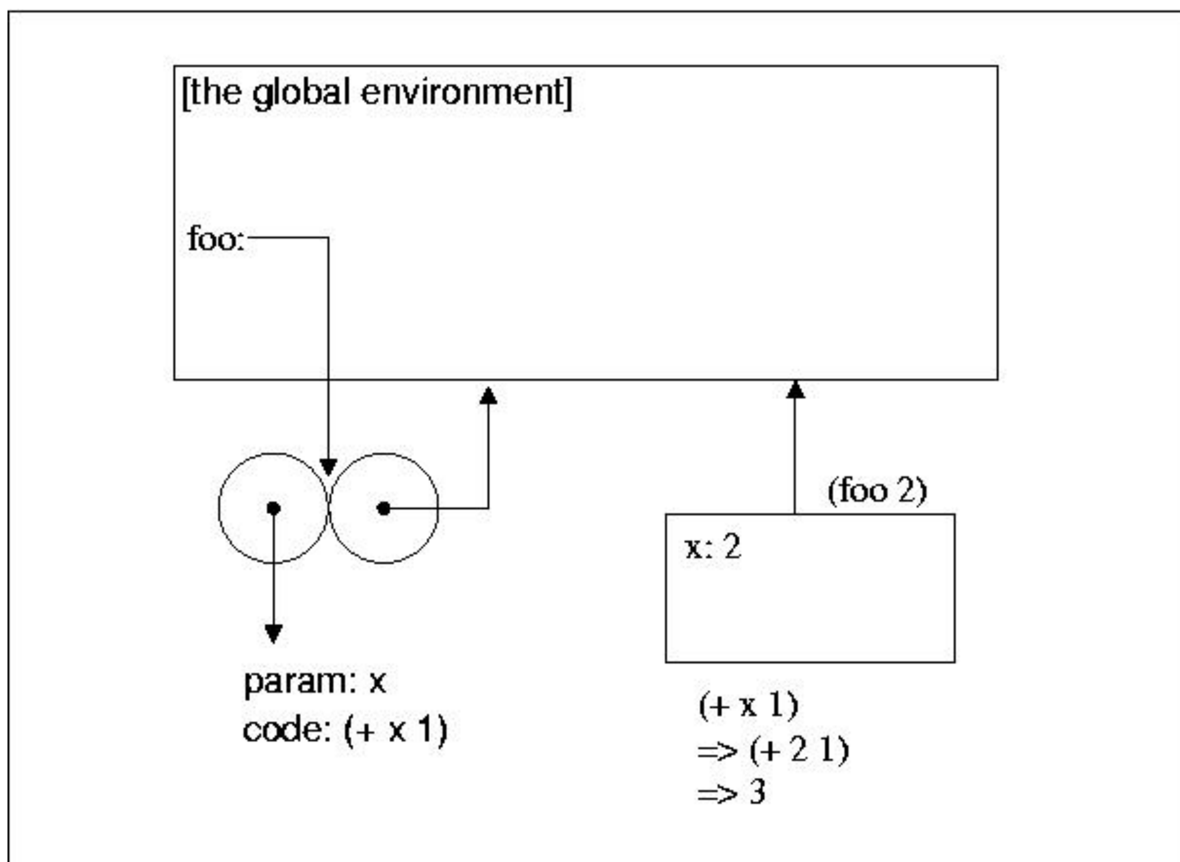
3. **Evaluating** (creating) a **`lambda`** expression (compound procedure) creates a pair (**not** a cons pair; just think of it as an abstract pair). One part of the pair represents the formal parameters of the `lambda` expression and the text of the code in the body of the `lambda`, while the other is a pointer into the environment in which the `lambda` was evaluated (known as the "enclosing environment").
4. **Applying** a **`lambda`** expression (compound procedure) to its arguments creates a new frame. The bindings in the frame are the formal parameters of the `lambda` expression, bound to their values (the arguments of the `lambda` expression). The frame's enclosing environment is the environment that

the `lambda` pair points to. The body of the `lambda` expression is evaluated in the context of the environment defined by the new frame.

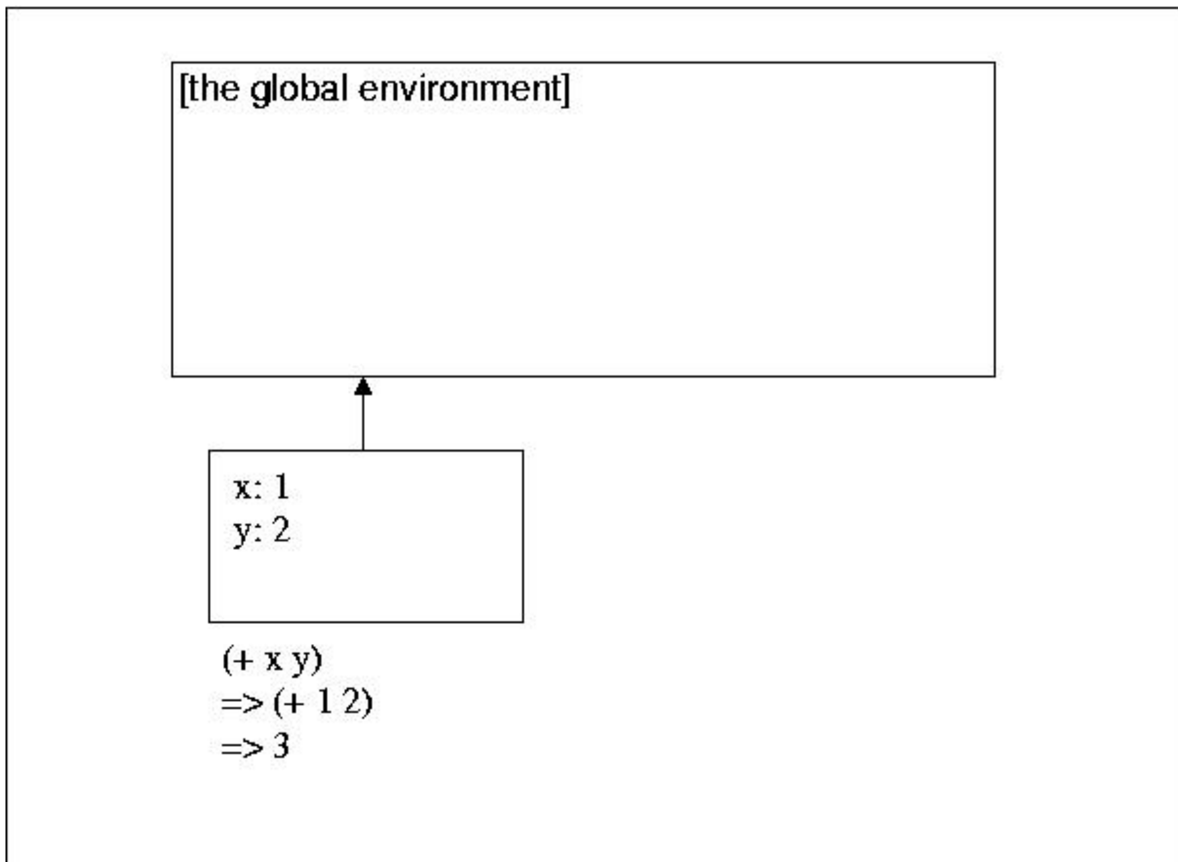
5. Evaluating a `let` generates a frame with the specified bindings and evaluates its expression in the context of that frame. This is just like evaluating a `lambda` expression and immediately applying it to its arguments, because that's actually what a `let` is.
6. Bindings are always between a symbol and a value, **not** between a symbol and another symbol.
7. You should write out procedures in the desugared `lambda` form, which is much clearer when drawing environment diagrams.
8. **[Subtle point]** Any value bound in the bindings part of a `let` expression is evaluated in the environment in which

the `let` expression itself was evaluated in.

```
(define (foo x) (+ x 1))
   (foo 2)
```

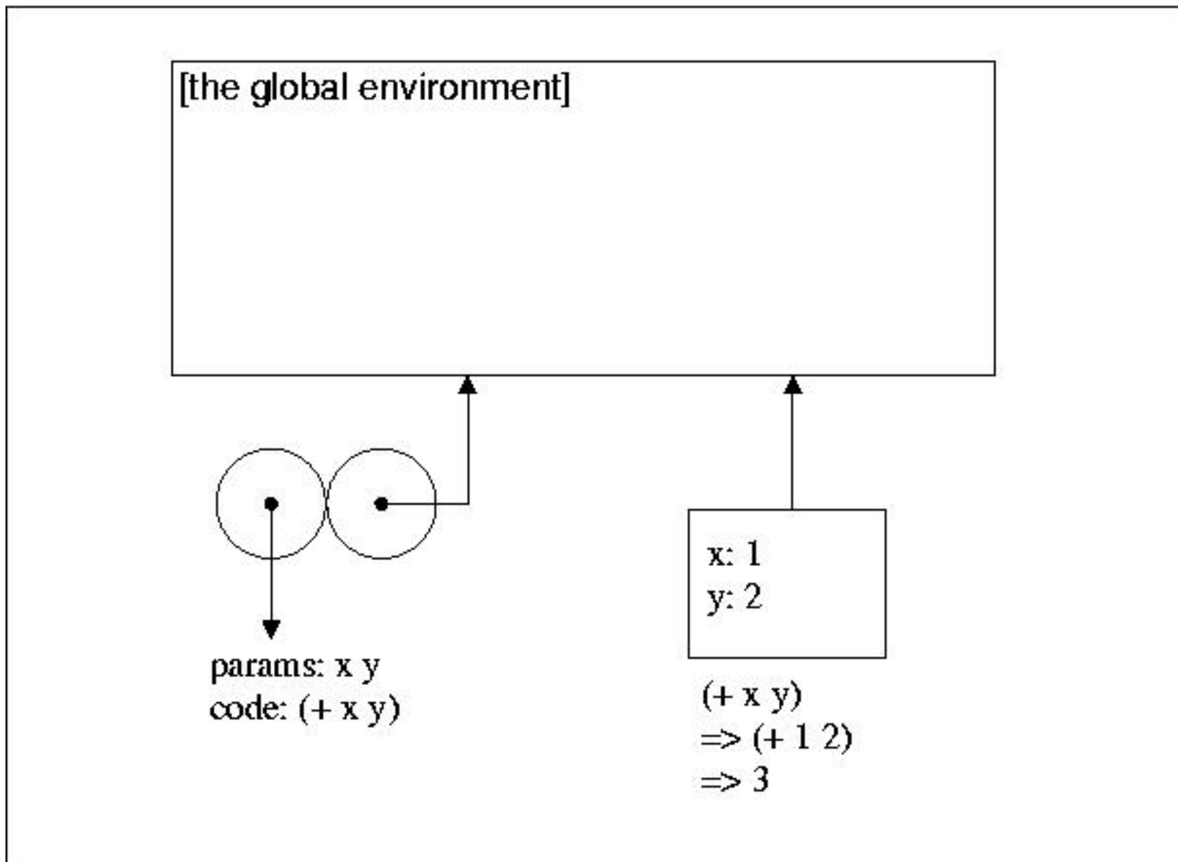



```
(let ((x 1)
      (y 2))
    (+ x y))
```

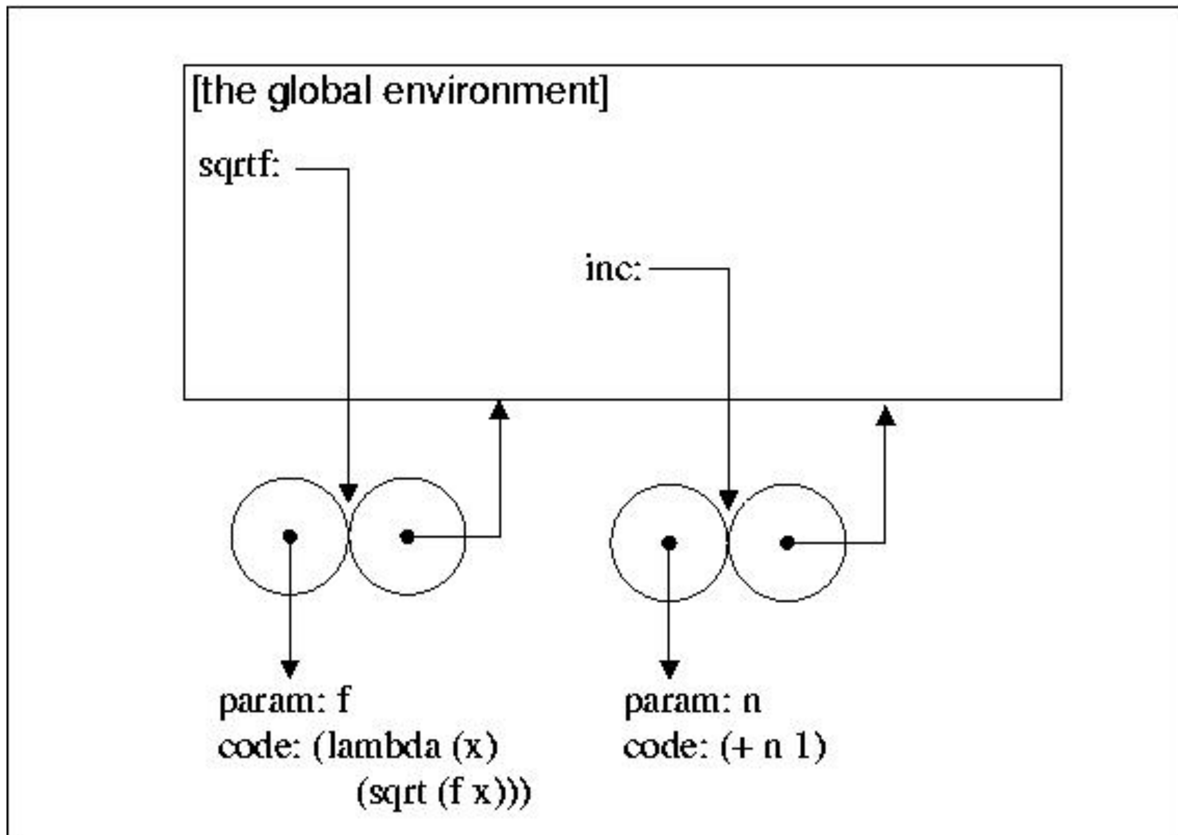


expand let into lambda:

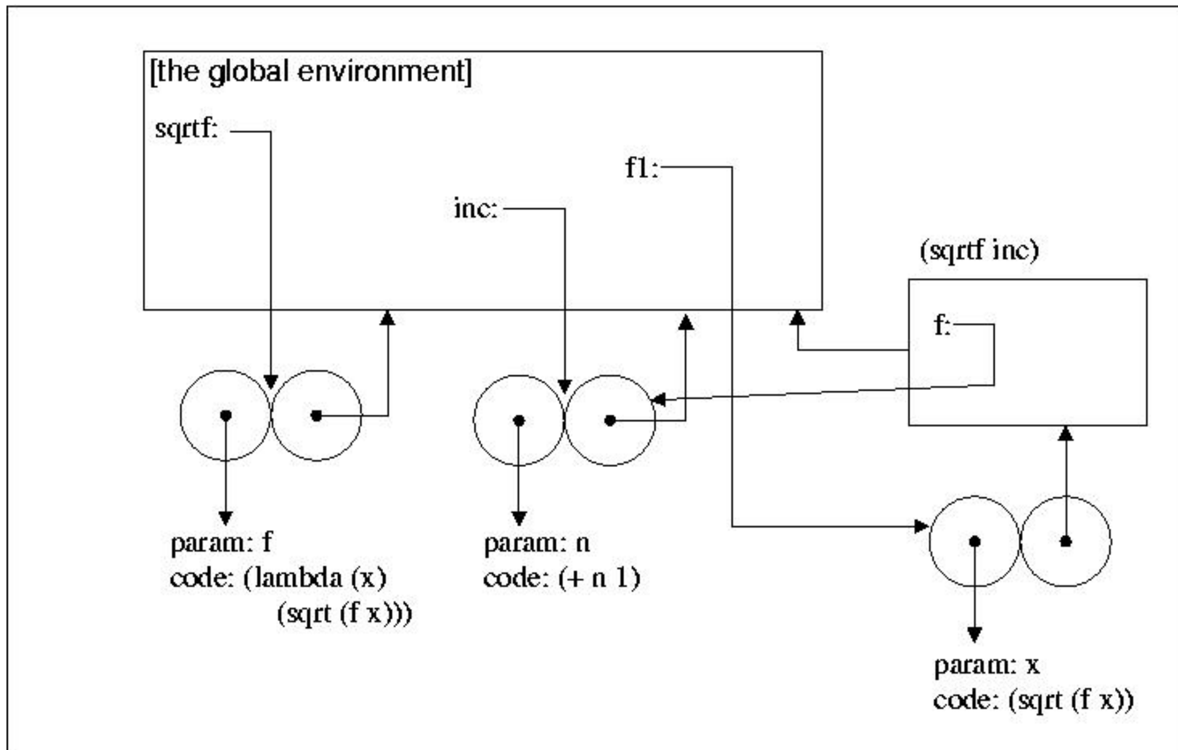
```
((lambda (x y) (+ x y)) 1 2)
```




```
(define (sqrtf f)
      (lambda (x) (sqrt (f x))))
    (define (inc n) (+ n 1))
    (define f1 (sqrtf inc))
    (f1 3)
```



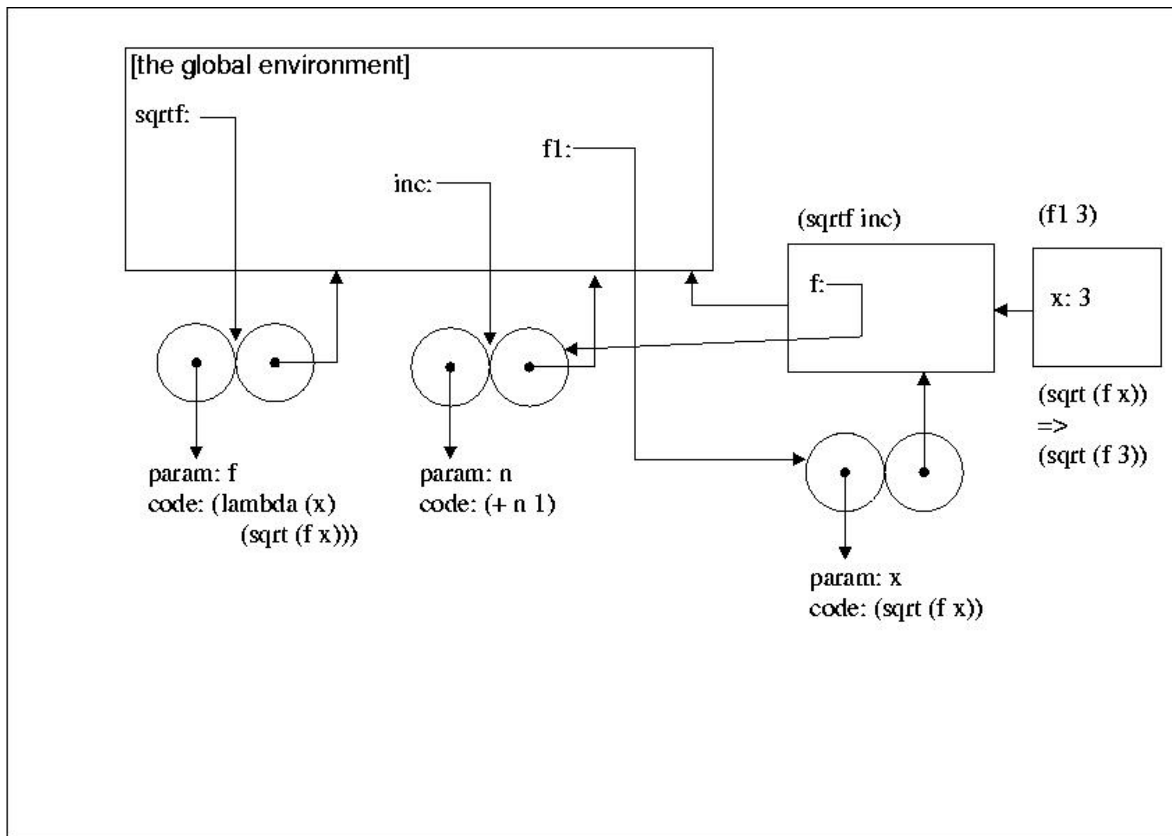
now we have to evaluate (sqrtf inc) and bind it to f1:



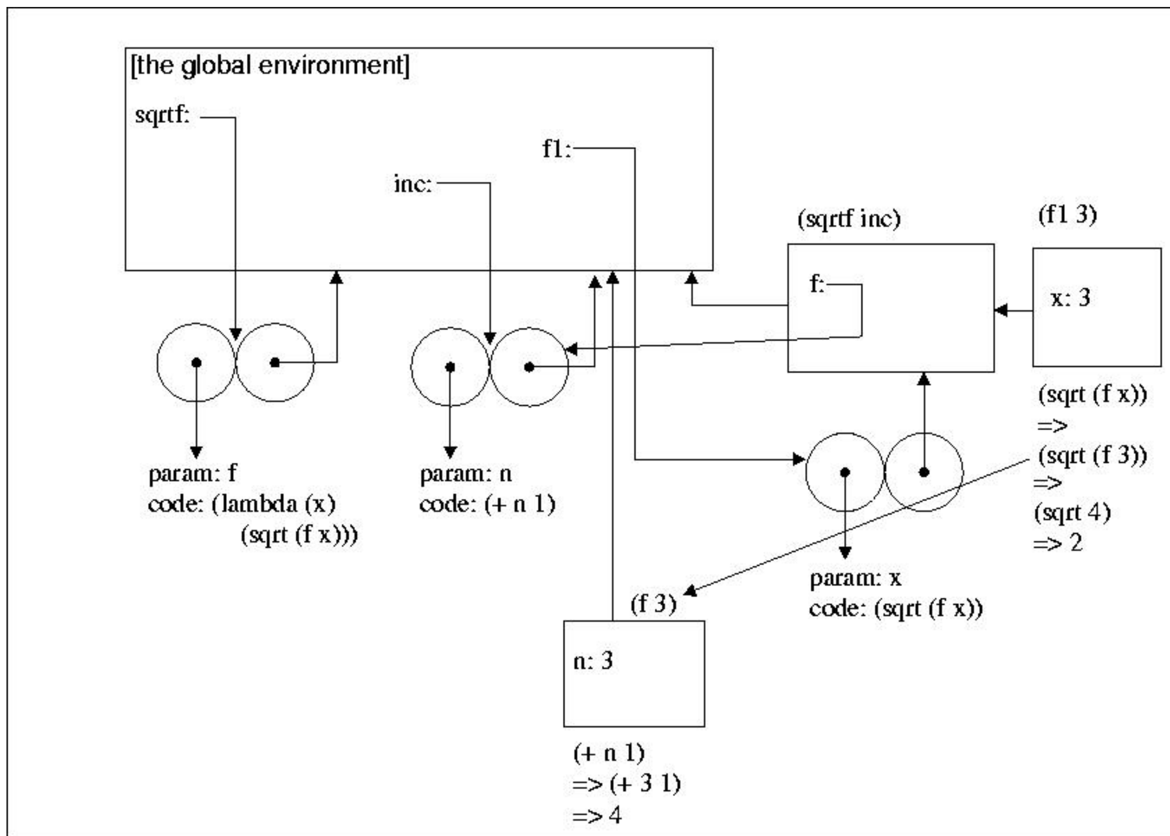
First, the expression `(sqrtf inc)` is an application of a `lambda` expression (the one that `sqrtf` is bound to) to its arguments, so by rule 4 we have to create a new frame with `f` (the parameter of the `lambda` expression) bound to the **value** of `inc` (which is a pair representing another `lambda` expression). Notice that we don't bind `f` to the **name** `inc` but to its **value** (rule 6). Then we have to execute the body of `sqrtf` in the context of the new frame. This body is just `(lambda (x) (sqrt`

`(f x))`, which means that we have to evaluate a `lambda` expression again. This means that we create a pair (rule 3) which points back to the environment in which the `lambda` is evaluated in (in this case, the frame containing `f`). The other part of the pair points to the parameters of the `lambda` (in this case, just `x`) and the code in the body of the `lambda` (in this case, `(sqrt (f x))`). This `lambda` expression is bound to the symbol `f1` in the global environment (because the entire line of code was executed at the top level of the interpreter). This is the first procedure definition we've seen where the environment pointer of the lambda points to an environment which is different from the one which points to the `lambda` expression itself. The `lambda` expression thus has its own private environment for looking up values which is different from the global environment.

Now we have to evaluate `(f1 3)`. Since `f1` is bound to a `lambda` expression, this is an application of a `lambda` expression to its arguments, so we use rule 4, which means that we have to make another frame. In this case, the frame binds `x` to `3`, which should be obvious to you by now. The new frame points back to the frame containing `f`, because that was the frame that the right part of the `lambda` pair `f1` pointed to. This gives rise to this environment diagram:



We have to evaluate `(sqrt (f x))` in the context of this new frame, and since `x` is bound to `3`, this means we have to evaluate `(f 3)` first and then plug the result back into the expression `(sqrt (f 3))`. This leads to this (final) environment diagram:



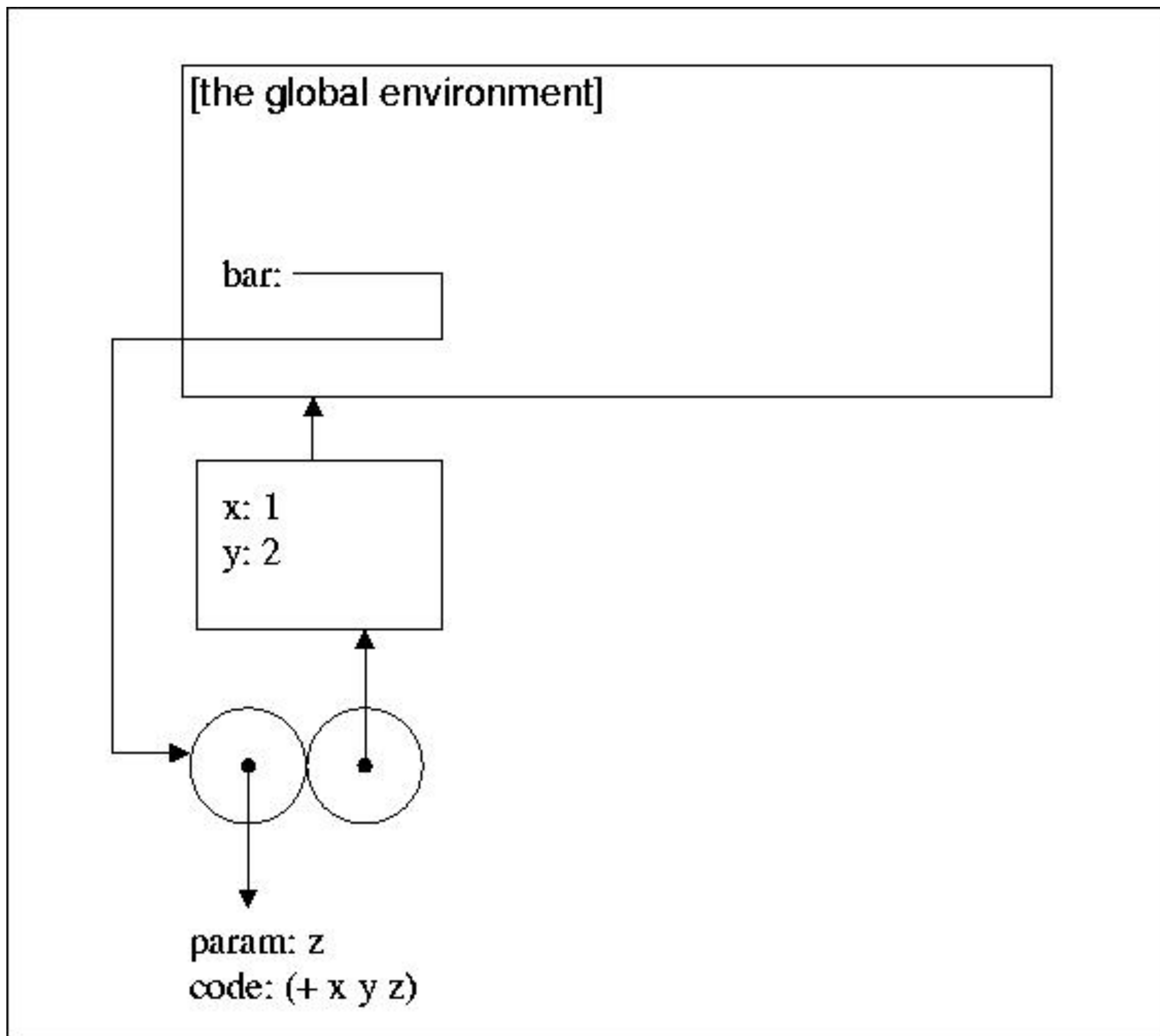
Since `f` is bound to the `lambda` pair corresponding to the `inc` procedure, evaluating `(f 3)` creates a new frame with `n` (from `inc`) bound to `3`. This frame points back to the global environment, because the `lambda` pair corresponding to the `inc` procedure points back to the global environment, and we're applying a `lambda` expression to an argument (rule 4). After a couple of steps, this yields `4`. This value is plugged back into `(sqrt`

`(f 3))` as the value of `(f 3)`, giving `(sqrt 4)` which is just `2`. The result of evaluating this code is therefore `2`.

This example involves both `let` and `lambda`.

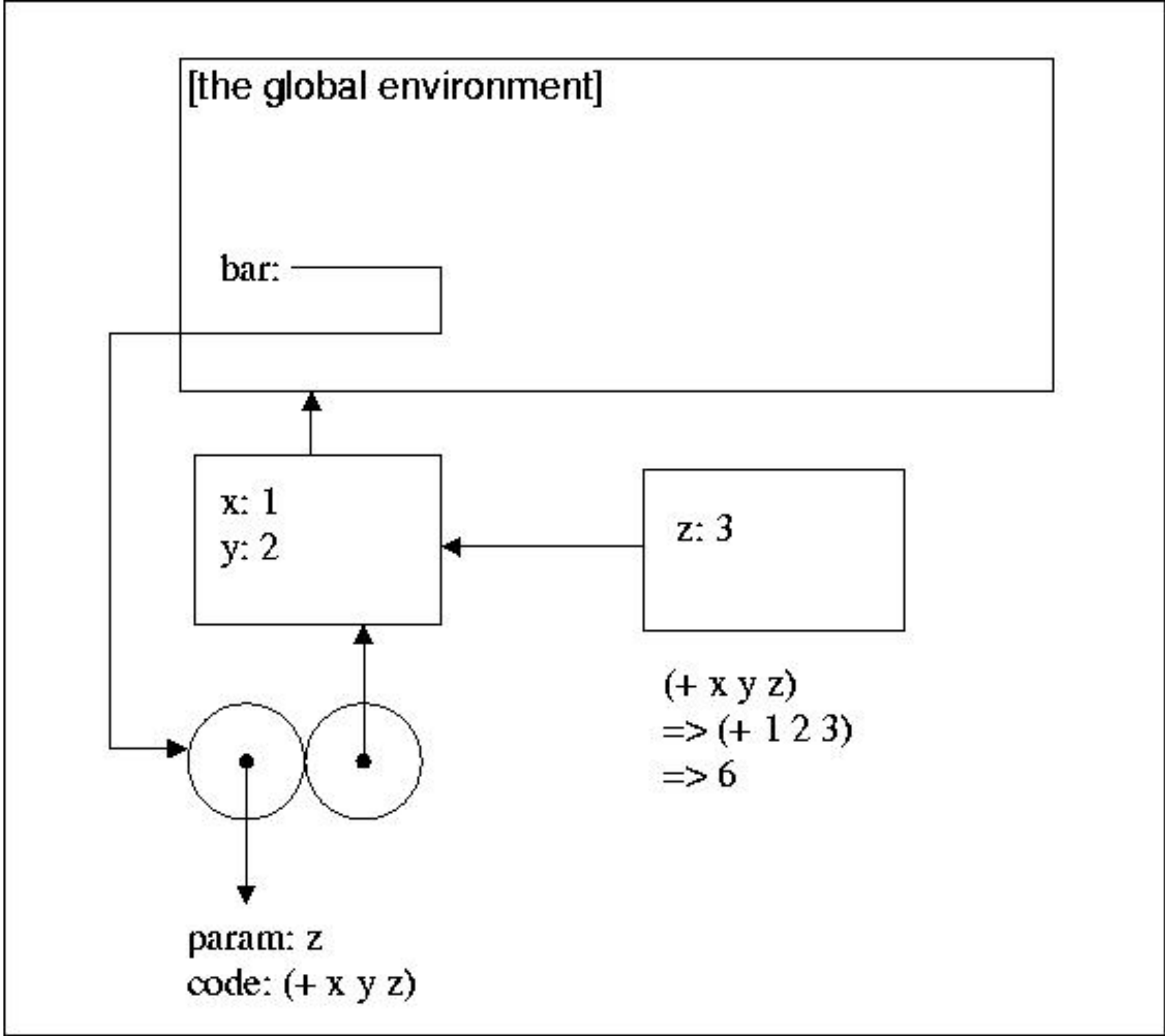
```
(define bar
  (let ((x 1)
        (y 2))
    (lambda (z) (+ x y
z))))

(bar 3)
```

After we evaluate the first line, the environment looks like this:



We see that the `let` expression has created a frame (rule 5) with `x` bound to `1` and `y` bound to `2`. Then we evaluate the `lambda` expression in the context of the environment corresponding to the new frame, which means that we have to make a `lambda` pair (rule 3). The `lambda` pair's enclosing environment is thus the frame created by the `let` expression.

Finally, since we are evaluating a `define`, we have to make a binding from the name `bar` to the `lambda` expression we just created (rule 1). Note that `bar` is a procedure.
After we evaluate `(bar 3)` the environment looks like this:



We see that evaluating `(bar 3)` means

applying a `lambda` expression to its arguments, so (by rule 4) we have to create a frame with the formal parameters bound to their corresponding values. In this case, the name `z` is bound to its value `3`. The enclosing environment of the newly-created frame is the environment pointed to by the `lambda` pair, which is the frame which contains bindings for `x` and `y`. Then we evaluate the code `(+ x y z)` in the context of the new frame, which results in `(+ 1 2 3)`, or `6`.

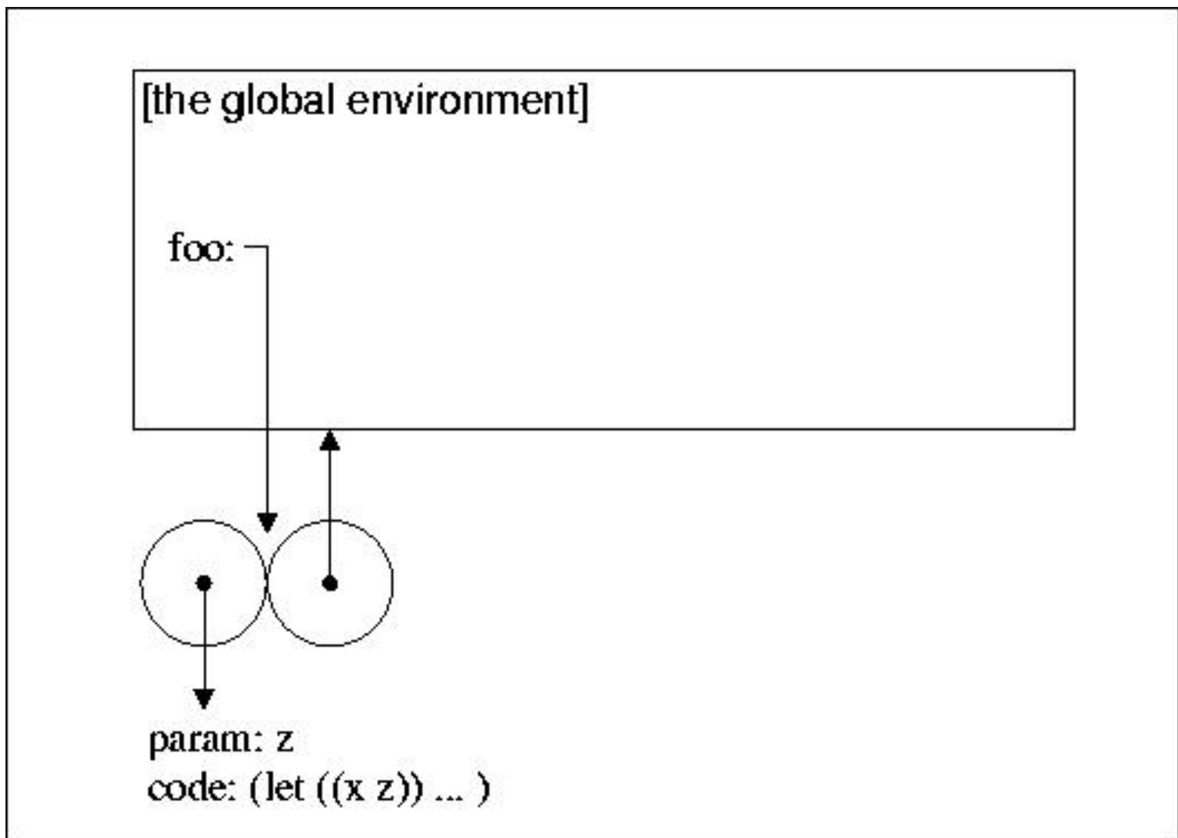
Here is a more complicated example involving message-passing and `set!`.

```
(define (foo z)
  (let ((x z))
    (let ((y (+ x z)))
```

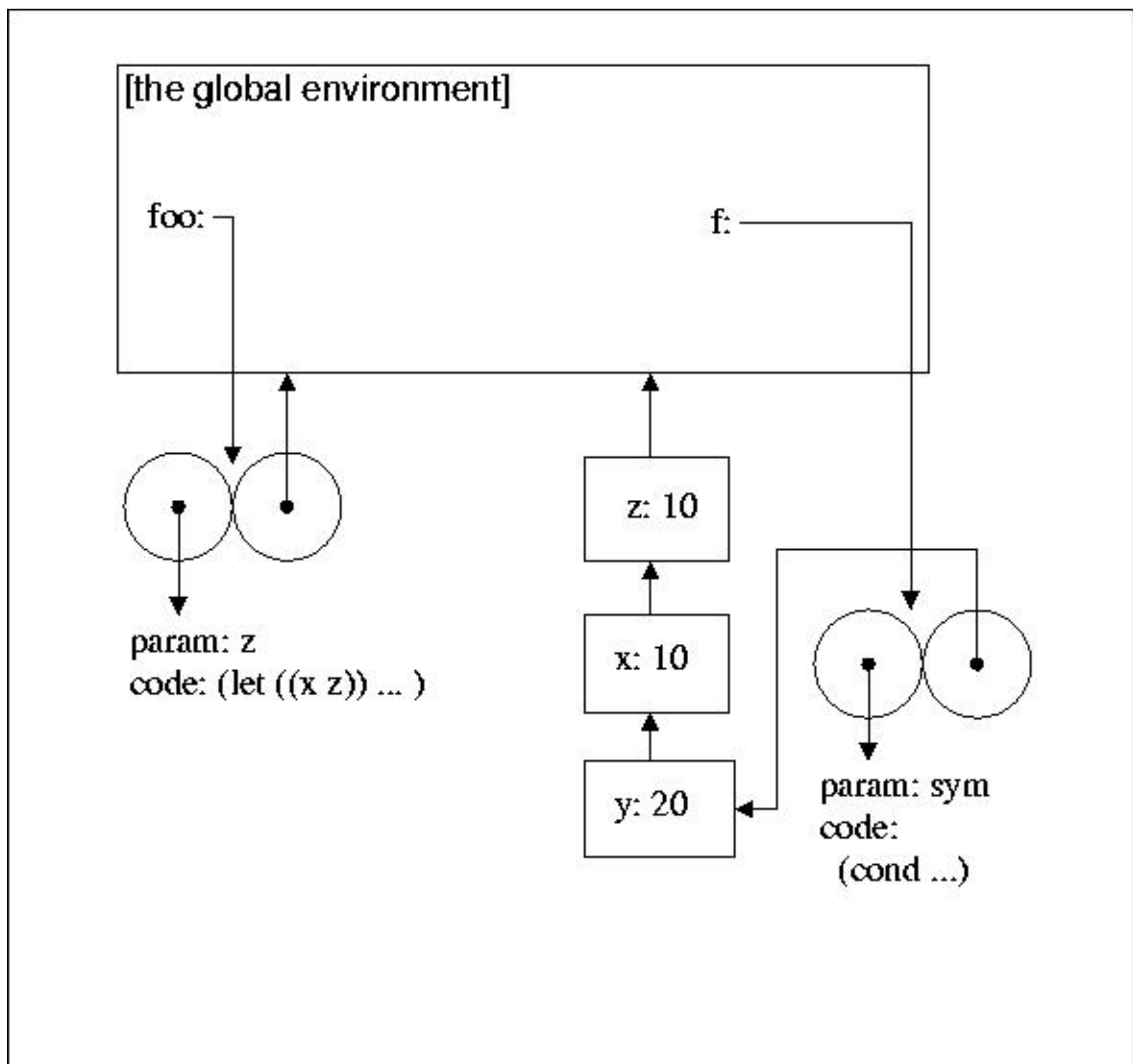
```
        (lambda (sym)
          (cond ((eq? sym
'x) x)
                ((eq? sym
'y) y)
                ((eq? sym
'bump-x)  (set! x (+ x z)))
                ((eq? sym
'bump-y)  (set! y (+ y z)))
                ((eq? sym
'reset-x) (set! x 0))
                ((eq? sym
'reset-y) (set! y 0)))))))

    (define f (foo 10))
    (f 'x)
    (f 'y)
    (f 'bump-x)
    (f 'bump-y)
    (f 'x)
    (f 'y)
    (f 'reset-x)
    (f 'reset-y)
```

```
(f 'x)
(f 'y)
```

After evaluating the `define`, the environment diagram looks like this:



what happens when we evaluate `(foo 10)` and bind it to the name `f` in the global environment. The resulting environment diagram looks like this:

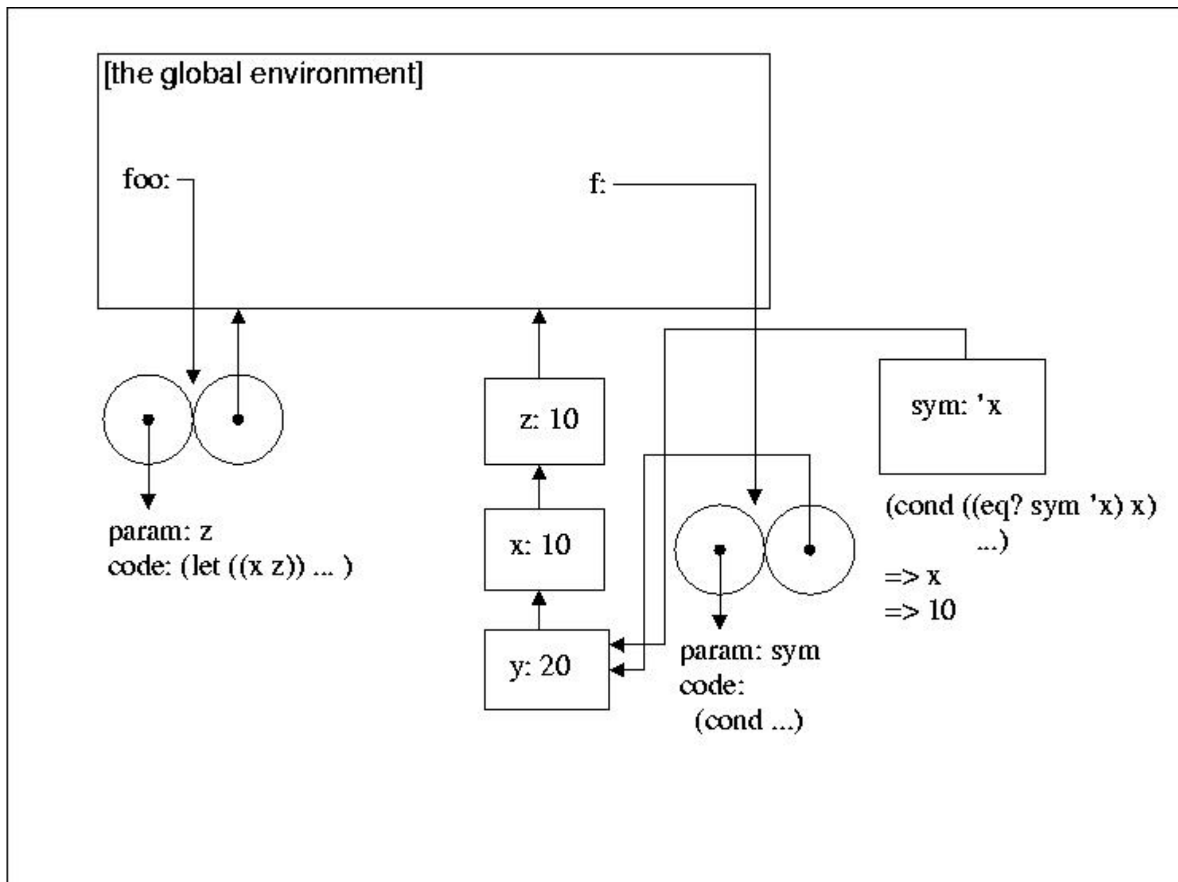


First of all, the expression `(foo 10)` is an application of a `lambda` expression to an argument, so by rule 4, we have to create a new frame. This frame has the name `z` bound to the value `10`. Then we have to evaluate the body of the `lambda` expression in the context of this frame. The first thing we have to evaluate is a

`let` expression, which (by rule 5) creates another frame whose enclosing environment is the current environment (which in this case starts at the frame that binds `z` to `10`). The new frame binds `x` to the **current value** of `z` (**not** the symbol `z` (remember rule 6)). So it binds `x` to `z`'s current value, which is `10`. Then we evaluate the next `let` expression in the context of this frame. This means that (by rule 5 again) we have to create **another** frame; this one binds the name `y` to the **current value** of the expression `(+ x z)`, which is `20`. This frame's enclosing environment starts with the last frame we defined (the one that binds `x` to `10`). Now, in the context of this latest frame we've just defined, we have to evaluate another `lambda` expression. By rule 3, this means we create a pair whose environment pointer points to the last frame we created (because that was the environment we

evaluated the `lambda` expression in). The `lambda` has a single parameter called `sym`, and some code which starts in a `cond` expression; we haven't written the whole thing out for you to keep the diagram uncluttered. Finally, the new `lambda` expression gets bound to the name `f` in the top-level environment, which is the environment in which the `define` expression was evaluated (rule 1).

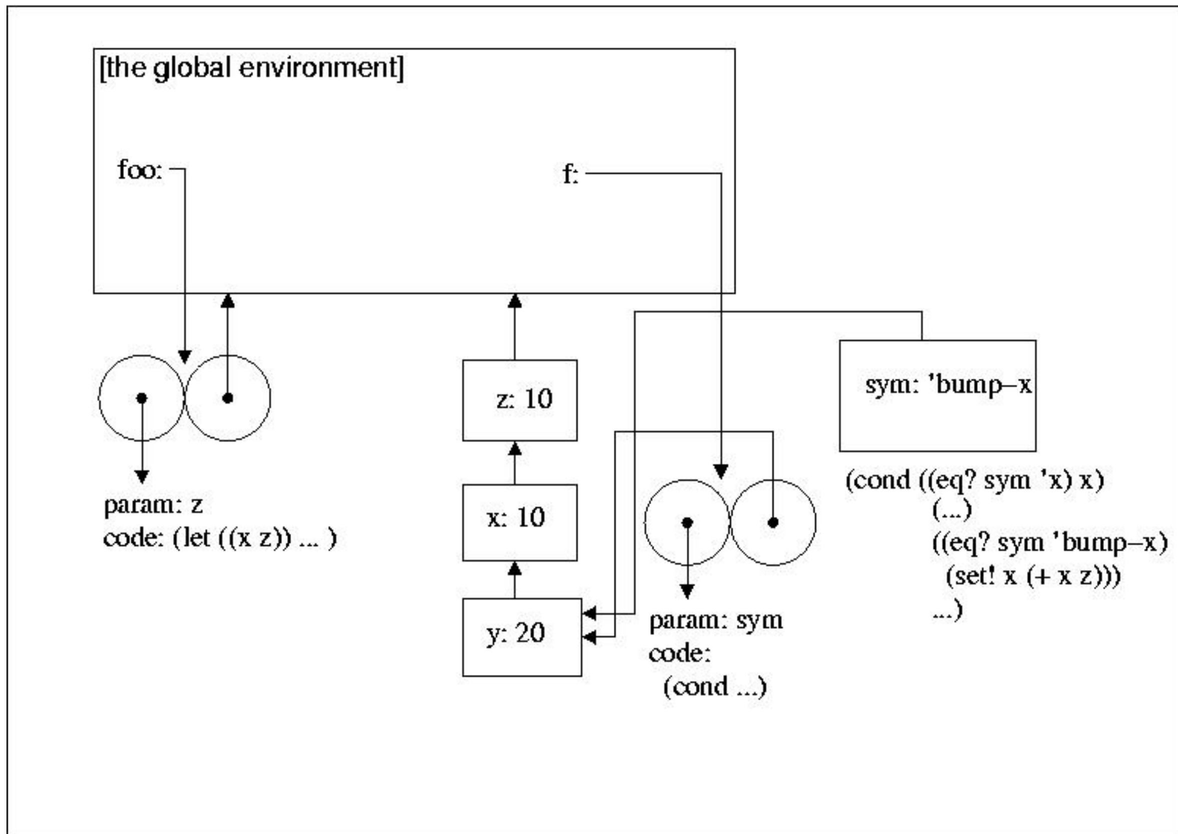
Now evaluate `(f 'x)`.



Since we are applying a `lambda` expression to its argument (which in this case is the symbol `'x`), we create a new frame with the name `sym` bound to `'x`. Then we have to evaluate the big `cond` expression. Fortunately, the first case is the relevant one because `(eq? sym 'x)` evaluates to true. This means that we have to return the value of `x` in the current environment. There is no binding for `x` in the top frame of the environment (the one

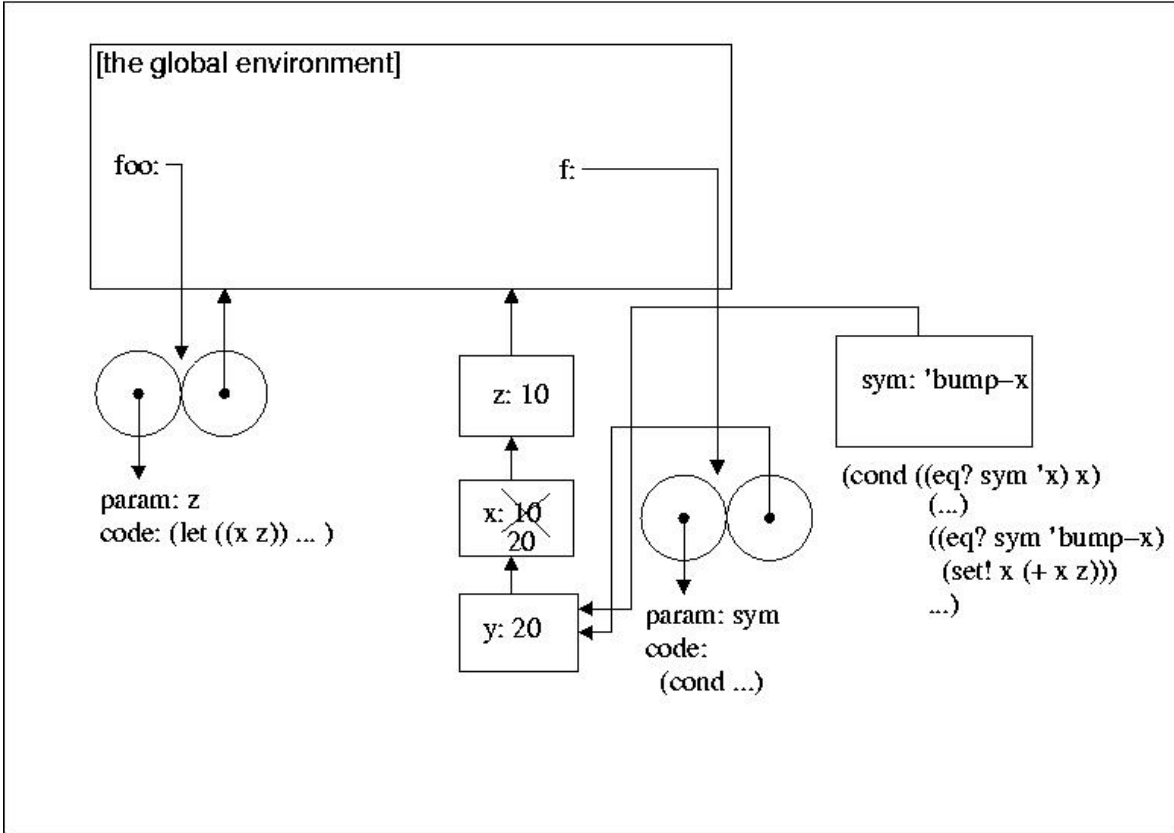
containing the binding for `sym`), so we have to go back to the enclosing frame and look there. There's no binding there either (just a binding for `y`), so we have to go yet another frame back, until finally we find a binding for `x`, and the value is `10`. This is the value of the expression we're evaluating. Similarly, evaluating `(f 'y)` would give `20`.

When we evaluate `(f 'bump-x)`, the resulting environment diagram will look like this:



Again, we apply the `lambda` expression to its argument, which creates a new frame (rule 4) which binds the name `sym` to `'bump-x`. Then we have to evaluate the `cond` expression again. We find the case we're looking for in the line `((eq? sym 'bump-x) (set! x (+ x z)))` which we have to evaluate. First we have to evaluate the current value of the expression `(+ x z)`, which is `20`. Then we have to set the value of `x` in the current

environment to `20`. Again, we have to go down a few frames to find the binding of `x`, but when we do we change the binding (rule 2). The environment diagram now looks like this:

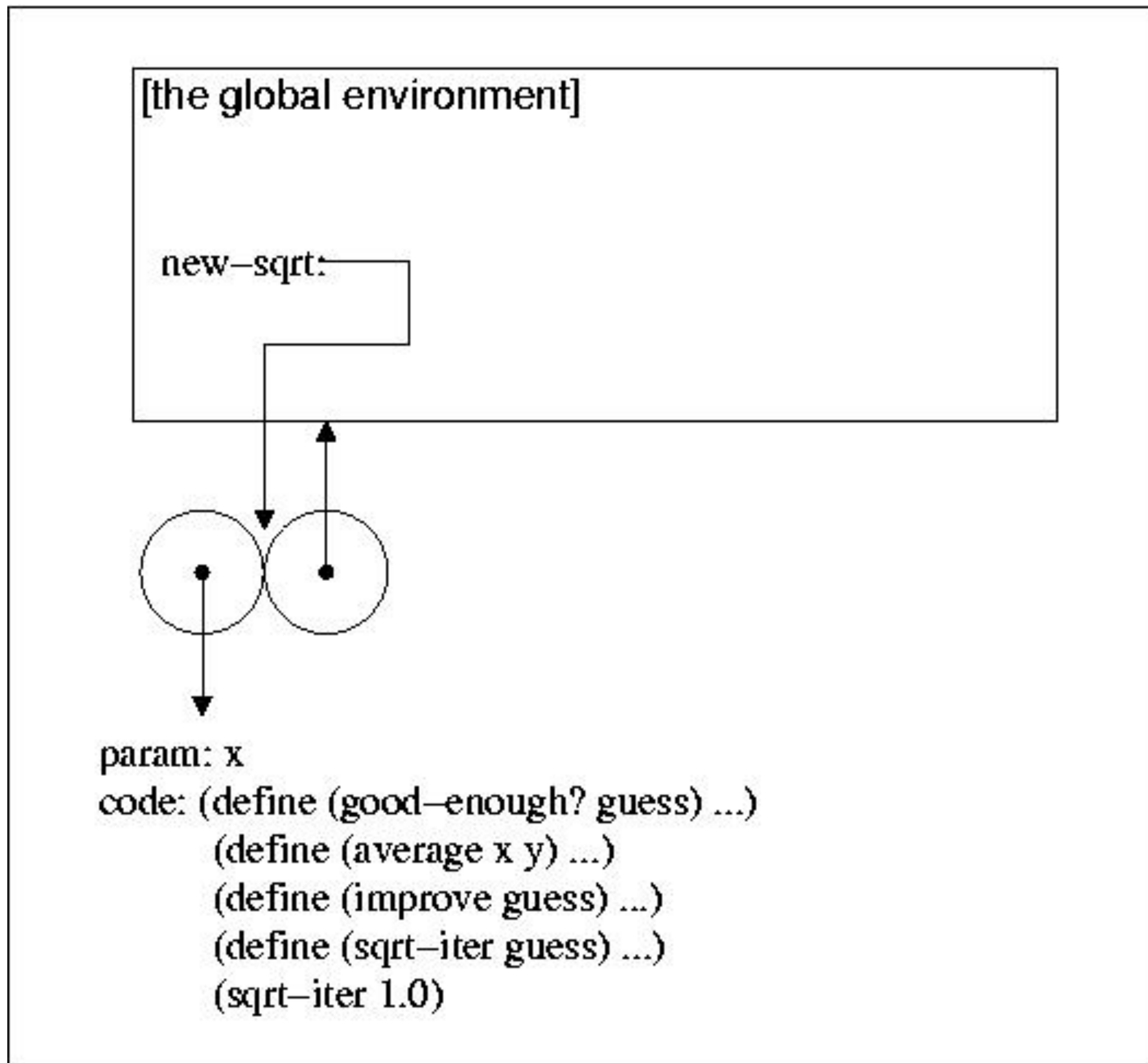


etc etc

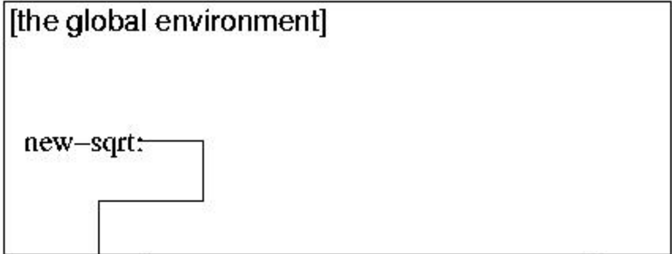
internal `defines`:

```
(define (new-sqrt x)
```

```
      (define (good-enough?
guess)
        (< (abs (- (square
guess) x)) 0.000001))
      (define (average x y)
        (/ (+ x y) 2))
      (define (improve
guess)
        (average guess (/ x
guess)))
      (define (sqrt-iter
guess)
        (if (good-enough?
guess)
            guess
            (sqrt-iter
(improve guess))))
      (sqrt-iter 1.0))

    (new-sqrt 2.0)
```



evaluate `(new-sqrt 2.0)`:

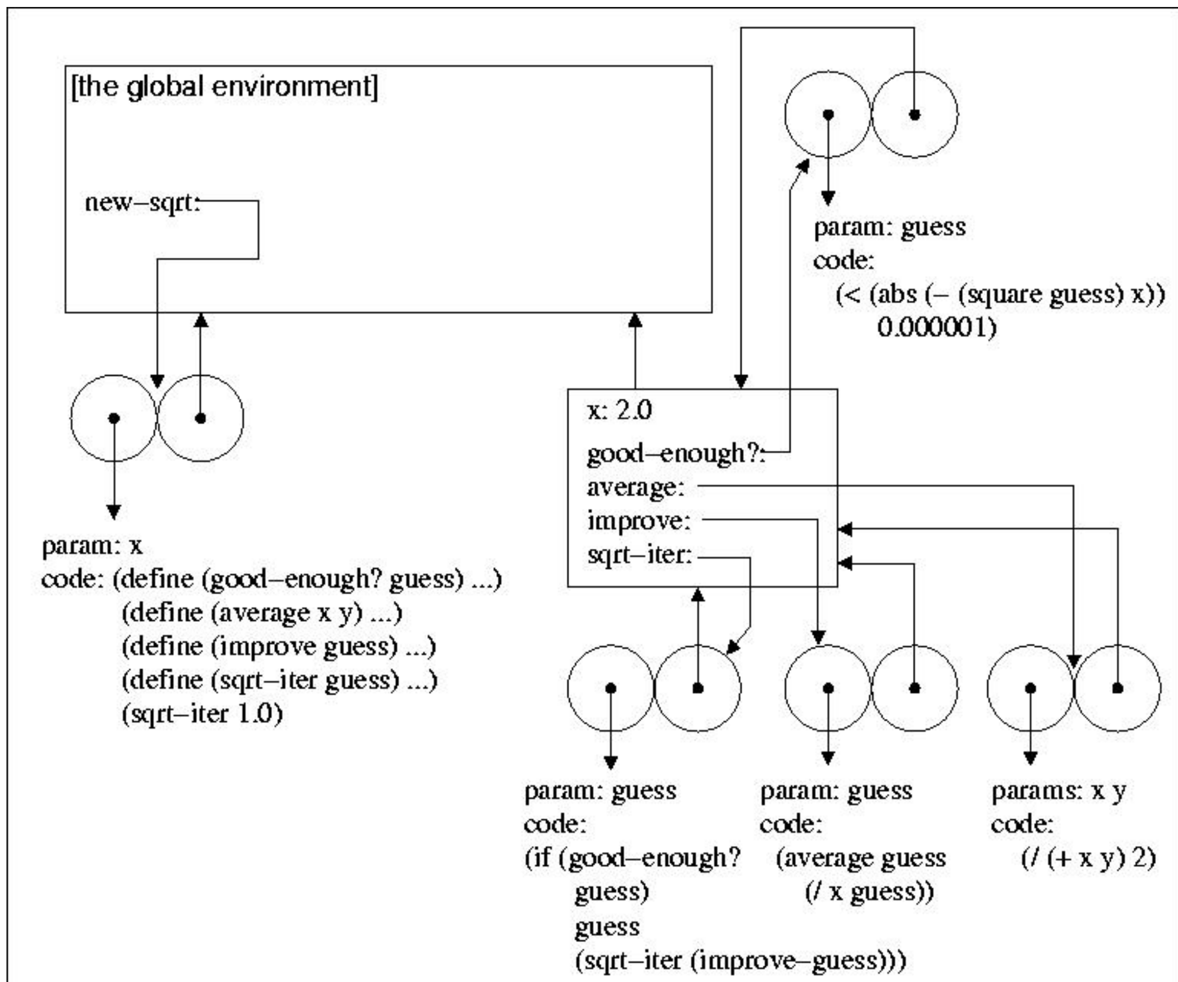


x: 2.0

param: x

code:
```
(define (good–enough? guess) ...)
(define (average x y) ...)
(define (improve guess) ...)
(define (sqrt–iter guess) ...)
(sqrt–iter 1.0)
```

```
(define (good–enough? guess) ...)
(define (average x y) ...)
(define (improve guess) ...)
(define (sqrt–iter guess) ...)
(sqrt–iter 1.0)
```



Note that the internal `define`s created bindings in the new frame, **not** in the top-level frame. Now evaluate (sqrt-iter 1.0) etc..

let::

What rule 8 says is that if the names being bound in the bindings part of the `let` expression are bound to expressions that have to be evaluated before the binding occurs, these expressions are evaluated in the same environment that the `let` expression itself is evaluated in. For instance, if we have:

```
(define x 25)
(let ((x 10)
      (y (+ x 20)))
  (+ x y))
```

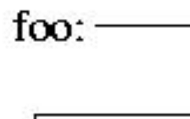
then `y` is bound to the result of evaluating the expression `(+ x 20)` **in the same environment that the `let` was evaluated in**. In this case, the `let` was evaluated in the top-level environment, so `(+ x 20)` evaluated in this environment yields `45`, which is what `y` is bound to inside the `let` expression. The value of the `let` expression is thus `55`. This can be confusing when the examples get more

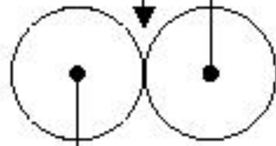
complex, so we'll work through a more complicated example below.
Consider this code:

```
(define (foo x)
  (let ((y (lambda (w) (+ x
w)))
        (z (lambda (w) (* x
w))))
    (y (z x))))
(foo 10)
```

Here, we are evaluating two `lambda` expressions inside a `let` expression. What does the environment diagram look like? Well, first of all it's pretty obvious that evaluating the `define` will create a `lambda` pair and bind it to the name `foo`. This will look like this:

[the global environment]

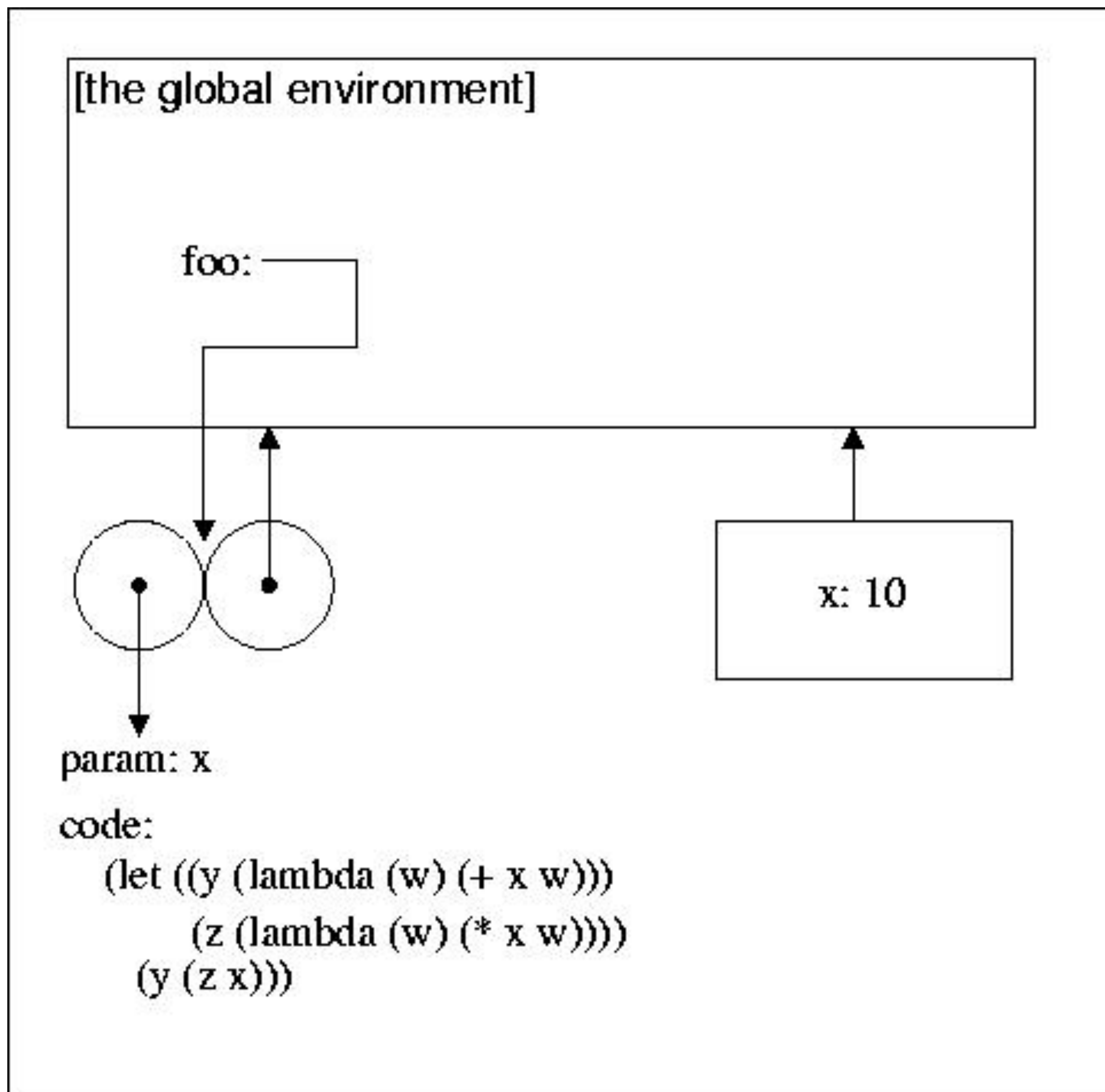




param: x

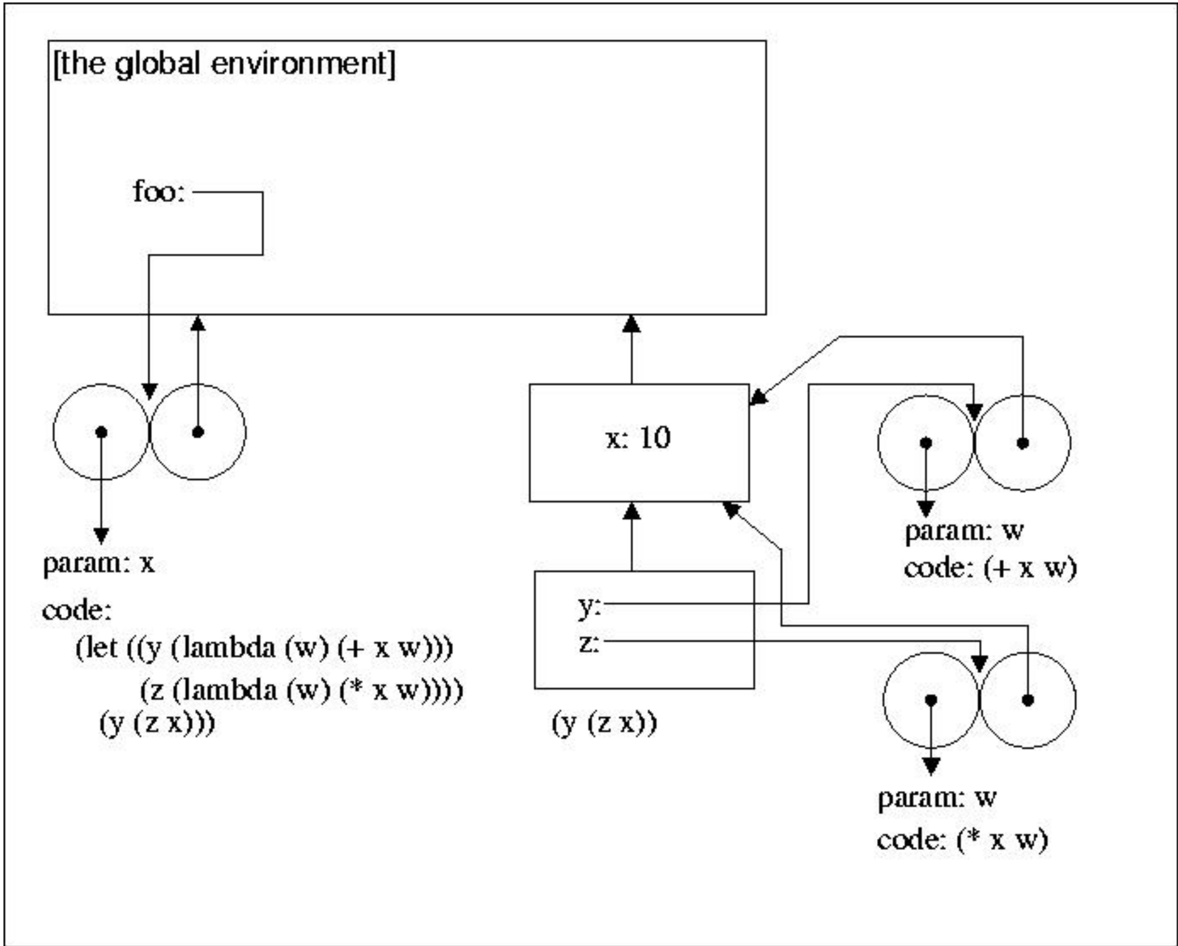
code:

```
(let ((y (lambda (w) (+ x w)))
      (z (lambda (w) (* x w))))
  (y (z x)))
```



Now we have to evaluate the `let` expression in the environment defined by the new frame. To do this, we create another frame which binds the names `y` and `z` to the result of evaluating two different `lambda` expressions. This is where the subtlety occurs. What are the enclosing environments for the `lambda`

expressions? Following rule 8, we realize that it will be the environment in which the `let` expression itself was evaluated in, which starts with the frame that bound `x` to `10`. Thus, the environment diagram will now look like this:



Now we have to evaluate the expression `(y (z x))` in the context of the new environment (the one which binds `y` and

z). x is bound to 10, so (z x) equals (z 10), which equals (* x 10), which equals (* 10 10), which is 100. So the expression reduces to (y 100), which reduces to (+ x 100), which is (+ 10 100), which is 110.