

Given that the following were evaluated sequentially, draw the corresponding environment diagram.

```
> (define (quiz-cons a b)
  (let ((cons (cons (a b))))
    (lambda (msg)
      (define wasted 6)
      (cond ((eq? 'bark msg) b)
            ((eq? 'woof msg) a)
            ((eq? 'cons msg) cons)
            (else (error "ERR" msg))))))
> (define apple (quiz-cons 'apple 'core))
> (define no-apple (quiz-cons 'apple "Oh! There's more!"))
> (cdr (no-apple 'cons))
> (define ghastly (cons (apple 'bark) (no-apple 'woof)))
> (set-car! ghastly apple)
```

Evaluating: (define (quiz-cons a b) ...)

This has syntactic sugar, so translate it into real code.

Step 1

Evaluating: (define quiz-cons (lambda (a b) ...))

Define is a special form. We must evaluate the expression to find out what the name quiz-cons will point to.

Step 2

Evaluating: (lambda (a b) (let (...)))

Lambda is a special form. This creates a procedure which is drawn as a bubble pair. One bubble points to the parameters and body of the procedure. The other bubble points to the frame it was evaluated in (the Current Frame).

Step 3

Evaluating: (define quiz-cons m)

The expression in the define special form has been evaluated, now we create the name quiz-cons in the current frame and make it point to the result of the evaluated expression.

Step 4

Evaluating: (define apple (quiz-cons 'apple 'core))

Define is a special form. We must evaluate the expression to find out what the name apple will point to.

Step 5

Evaluating: (quiz-cons 'apple 'core)

This expression is not a special form. Each sub-expression must be evaluated (in any order).

Step 6

Evaluating: quiz-cons Evaluating: 'apple Evaluating: 'core

Evaluating the name quiz-cons, we look in the current frame and find the name exists, so it evaluates to what quiz-cons points to in the current frame. Evaluating 'apple' returns the symbol apple. Likewise, evaluating 'core' returns the symbol core.

Step 7

Evaluating: (cons (cons (cons a b)) (lambda (msg) ...))

This expression is not a special form. Now that each expression has been evaluated, the first expression is a procedure (or an error occurs) and it is evaluated with the given arguments (a wrong number of arguments is also an error). When the procedure is evaluated, a frame is created that points to the same frame the evaluated procedure points to. The parameter names are bound in the frame with the proper arguments. The CF pointer moves to the new frame and starts evaluating the body of the procedure. All previous CFs I'll label as CF(-n) so we know where to return when the body of the procedure has been evaluated.

Step 8

Evaluating: ((lambda (cons) (lambda (msg) ...)) (cons a b))

Now that we have removed the syntactic sugar, we see that this expression is not a special form. Each sub-expression must be evaluated (in any order).

Step 11

Evaluating: (let ((cons (cons (cons a b))) (lambda (msg) ...)))

The let expression is syntactic sugar for an evaluated lambda. We must translate this into real code.

Step 9

Evaluating: (lambda (cons) (lambda (msg) ...)) Evaluating: (cons a b)

We evaluate the lambda special form as we did before, drawing the bubble pair. The other expression is not a special form and each sub-expression must be evaluated (in any order).

Step 12

Evaluating: ((lambda (cons) (lambda (msg) ...)) (cons a b))

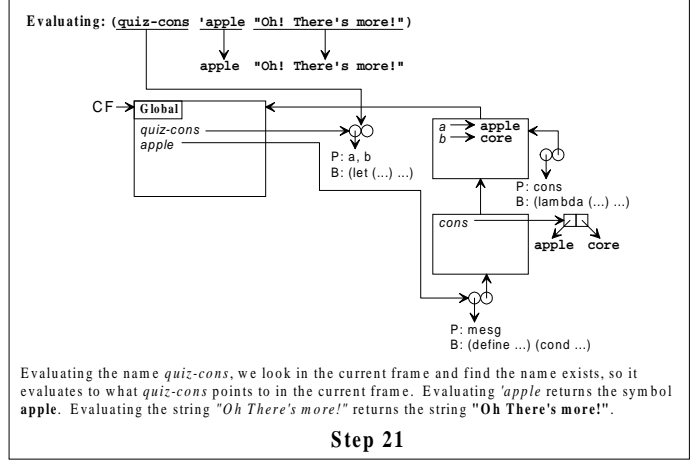
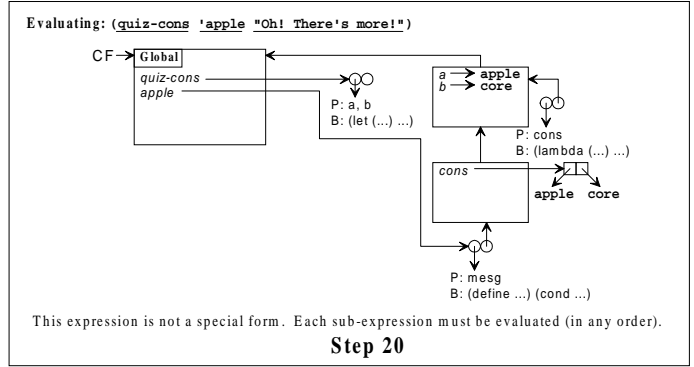
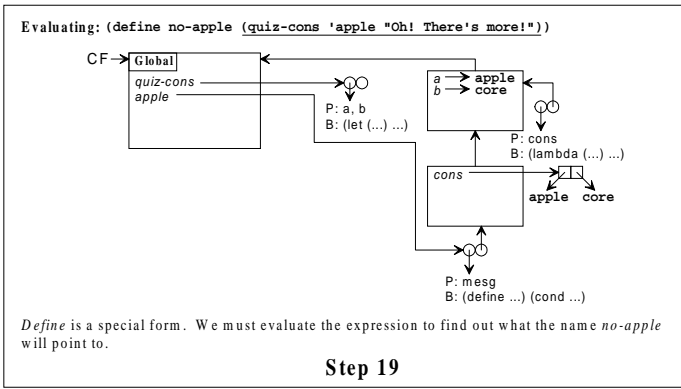
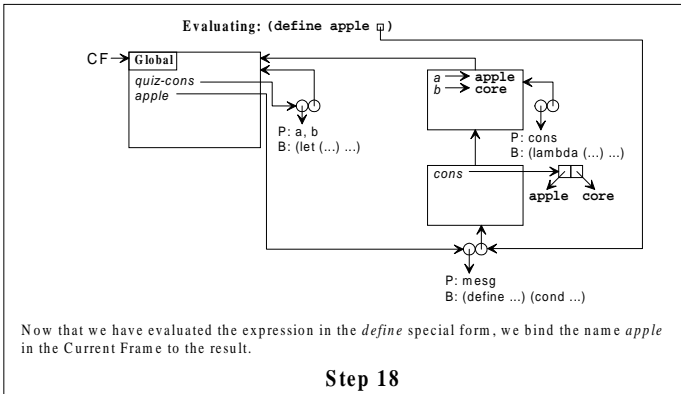
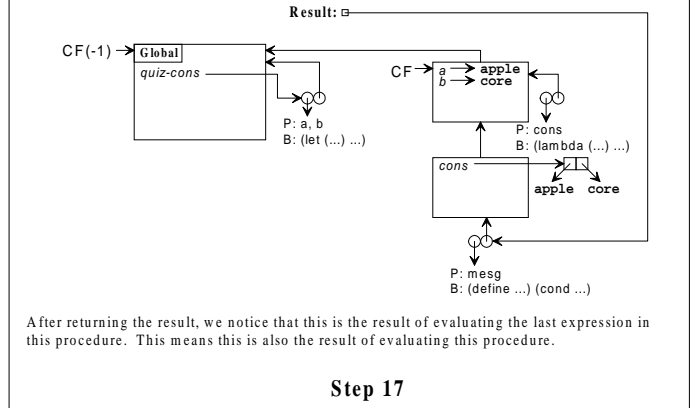
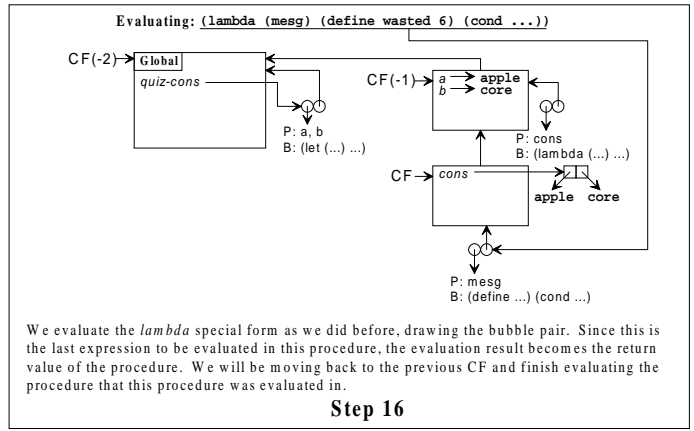
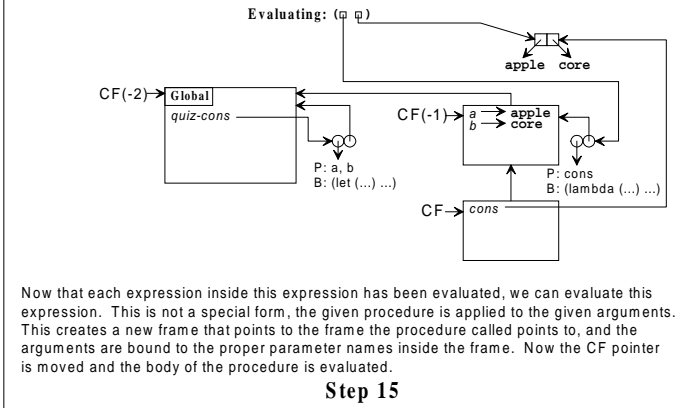
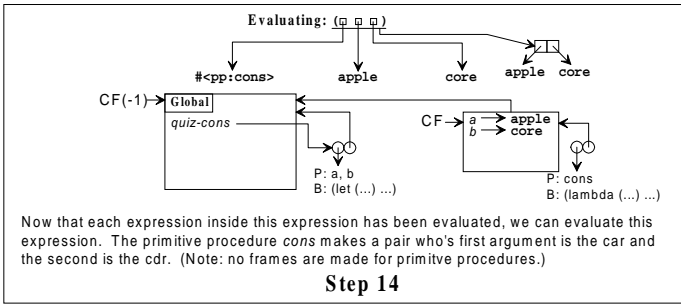
Now that we have removed the syntactic sugar, we see that this expression is not a special form. Each sub-expression must be evaluated (in any order).

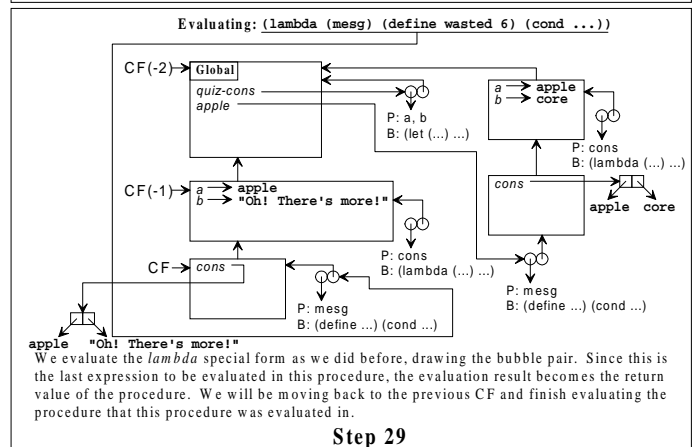
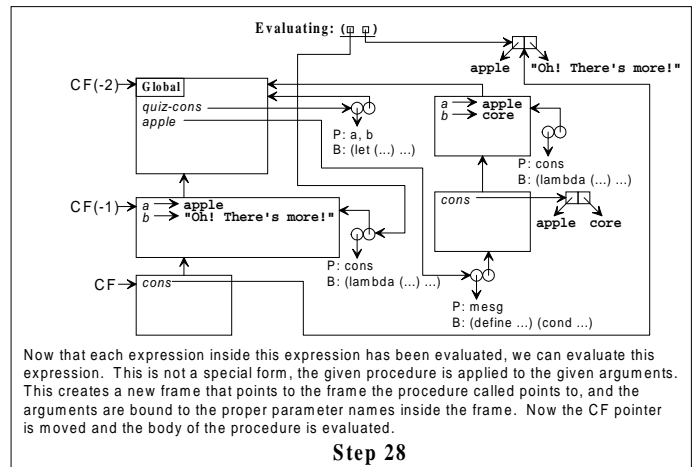
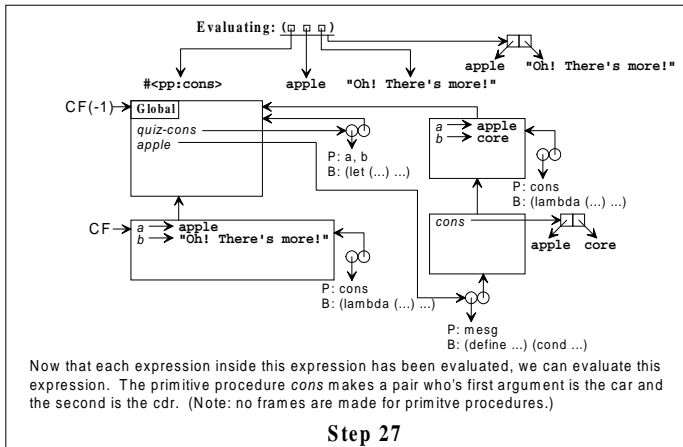
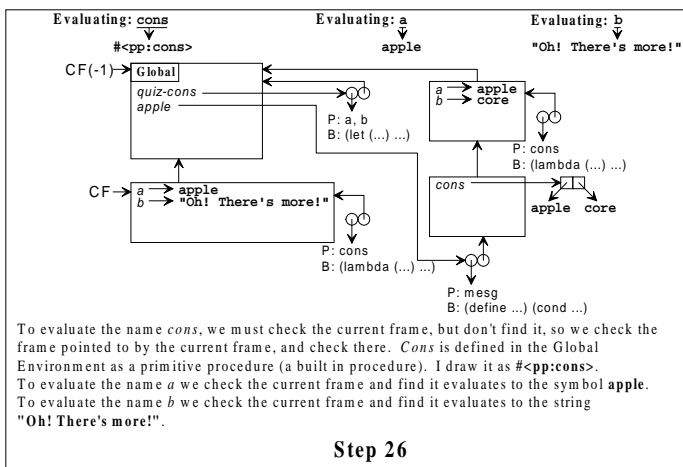
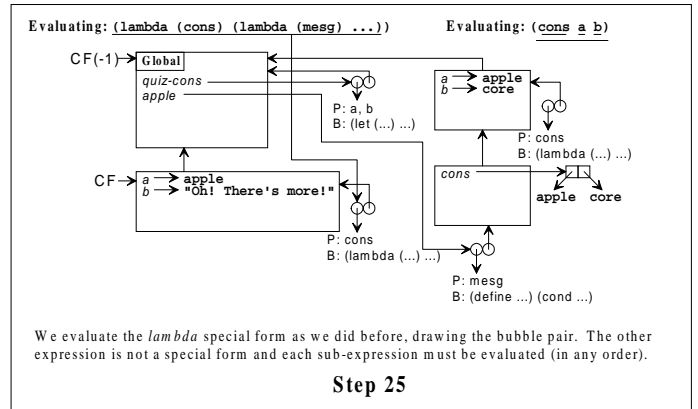
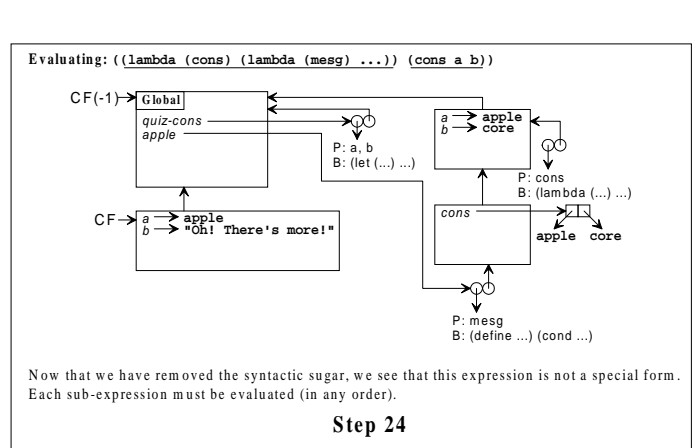
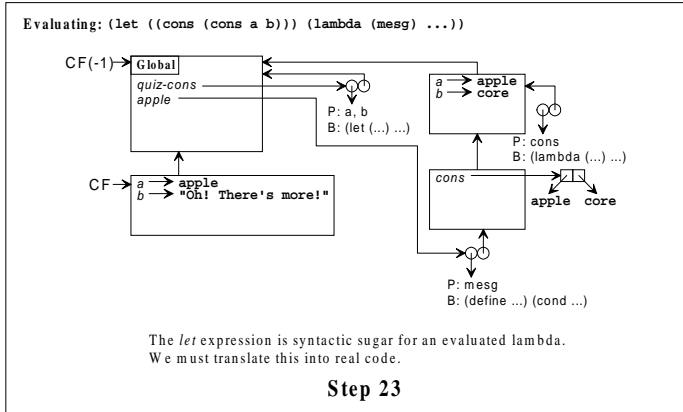
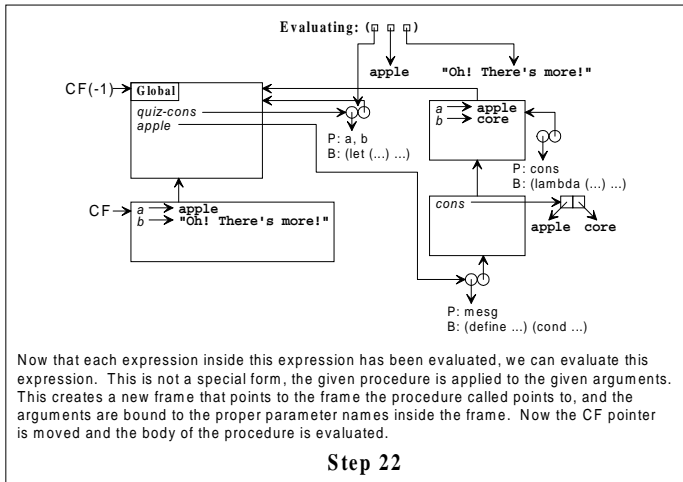
Step 10

Evaluating: cons Evaluating: a Evaluating: b

To evaluate the name cons, we must check the current frame, but don't find it, so we check the frame pointed to by the current frame, and check there. Cons is defined in the Global Environment as a primitive procedure (a built in procedure). I draw it as #<pp:cons>. To evaluate the name a we check the current frame and find it evaluates to the symbol apple. To evaluate the name b we check the current frame and find it evaluates to the symbol core.

Step 13





Evaluating: (cond ... ((eq? 'woof msg) a) ...)

#<pp:eq?> ← woof → cons

Cond is a special form, so first we need to evaluate the first conditional expression. To evaluate it, we need to evaluate all of the parts. The name *eq?* follows the path of pointers to frames from the Current Frame until the Global Environment (since it is undefined in all of the frames inbetween), and is found to be a primitive procedure, #<pp:eq?>. 'woof' evaluates to the symbol *woof*. The name *msg* is found in the Current Frame and evaluates to *cons*.

Step 39

Evaluating: (cond ... ((eq? 'woof msg) a) ...)

#<pp:eq?> ← woof → cons → #f

Evaluating the primitive procedure *eq?* on the symbols *woof* and *cons* results in #f because they are unequal. The result of #f in the *cond* expression causes us to move on to the next conditional predicate and evaluate it.

Step 40

Evaluating: (cond ... ((eq? 'cons msg) cons) ...)

#<pp:eq?> ← cons → cons

Cond is a special form, so first we need to evaluate the first conditional expression. To evaluate it, we need to evaluate all of the parts. The name *eq?* follows the path of pointers to frames from the Current Frame until the Global Environment (since it is undefined in all of the frames inbetween), and is found to be a primitive procedure, #<pp:eq?>. 'cons' evaluates to the symbol *cons*. The name *msg* is found in the Current Frame and evaluates to *cons*.

Step 41

Evaluating: (cond ... ((eq? 'cons msg) cons) ...)

#<pp:eq?> ← cons → cons → #t

Evaluating the primitive procedure *eq?* on the symbols *cons* and *cons* results in #t because they are equal. The result of #t in the *cond* expression causes us to evaluate the clause associated with this predicate.

Step 42

Evaluating: (cond ... (msg cons) ...)

#t → apple → "Oh! There's more!"

To evaluate the name *cons* we look in the Current Frame, don't find it, and then look in the frame pointed to by the Current Frame, and find it is equal to a particular pair. Note that the pair I'm drawing represents exactly the same pair, and is not a duplicate! Since this is the last expression in the predicate, and the *cond* expression was the last expression in the procedure, the result of the procedure is the result of evaluating the name *cons* in the Current Frame, and we move the CF pointer back to the Global Environment and leave the procedure.

Step 43

Evaluating: (msg cons) → "Oh! There's more!"

#<pp:cdr> ←

Now that we have evaluated all of the expressions with this expression, we note that it isn't a special form and evaluate the primitive procedure *cdr* on the given argument. The result, "Oh! There's more!", is then returned to the user at the prompt because that's who evaluated it.

Step 44

Evaluating: (define ghashly (cons (apple 'bark) (no-apple 'woof)))

Define is a special form. We must evaluate the expression to find out what the name *ghashly* will point to. That expression is made up of other expressions, so we need to evaluate them first.

Step 45

Evaluating: cons Evaluating: (apple 'bark) Evaluating: (no-apple 'woof)

Evaluating the name *cons*, we find the primitive procedure *cons*. In order to evaluate the other expressions, their sub expressions must be evaluated. The name *apple* evaluates to the procedure it points to, and *bark* evaluates to the symbol *bark*. Note that we don't evaluate the last expression until we finish evaluating the one we've started on.

Step 46

Evaluating: (msg cons)

Now that each expression inside this expression has been evaluated, we can evaluate this expression. This is not a special form, the given procedure is applied to the given arguments. This creates a new frame that points to the frame the procedure called points to, and the arguments are bound to the proper parameter names inside the frame. Now the CF pointer is moved and the body of the procedure is evaluated.

Step 47

Evaluating: (define wasted 6)

Define is a special form. We must evaluate the expression to find out what the name *quiz-cons* will point to. The *6* evaluates to the value *6*. This result is then bound to the name *wasted* in the Current Frame.

Step 48

Evaluating: (cond ((eq? 'bark msg) b) ...)

#<pp:eq?> ← bark → bark

Cond is a special form, so first we need to evaluate the first conditional expression. To evaluate it, we need to evaluate all of the parts. The name *eq?* follows the path of pointers to frames from the Current Frame until the Global Environment (since it is undefined in all of the frames inbetween), and is found to be a primitive procedure, #<pp:eq?>. 'bark' evaluates to the symbol *bark*. The name *msg* is found in the Current Frame and evaluates to *bark*.

Step 49

Evaluating: (cond ((eq? 'bark msg) b) ...)

#<pp:eq?> ← bark → bark → #t

Evaluating the primitive procedure *eq?* on the symbols *bark* and *bark* results in #t because they are equal. The result of #t in the *cond* expression causes us to evaluate the clause associated with this predicate.

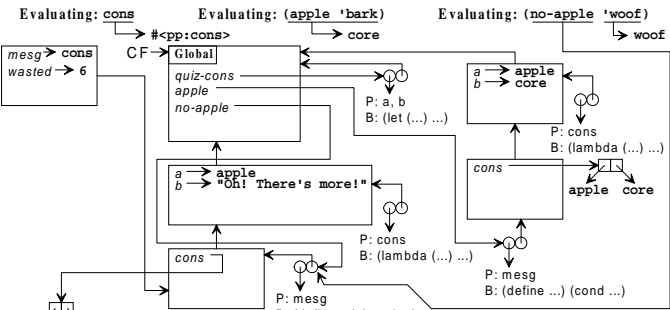
Step 50

For Steps 49 through 51, refer to the Environment Diagram in Step 48. They all look alike.

Evaluating: (cond ... (a b) ...) → core

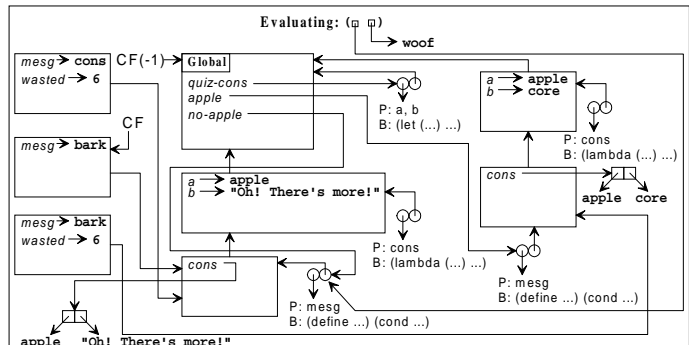
To evaluate the name *b* we look in the Current Frame, don't find it, and then look in the frame pointed to by the Current Frame, and find it is equal to the symbol *core*. Since this is the last expression in the predicate, and the *cond* expression was the last expression in the procedure, the result of the procedure is the result of evaluating the name *b* in the Current Frame, and we move the CF pointer back to the Global Environment and leave the procedure.

Step 51



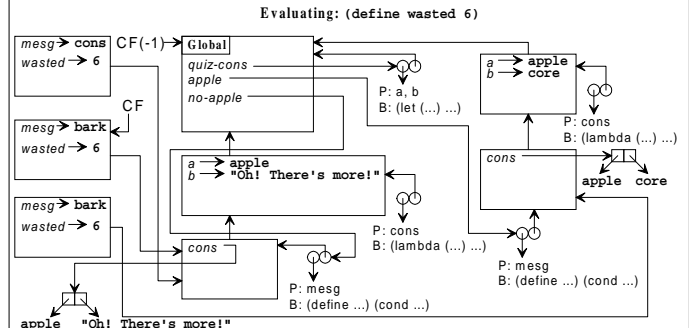
Now we need to evaluate the last expression. The name *no-apple* evaluates to the procedure it points to, and *'woof* evaluates to the symbol *woof*.

Step 52



Now that each expression inside this expression has been evaluated, we can evaluate this expression. This is not a special form, the given procedure is applied to the given arguments. This creates a new frame that points to the frame the procedure called points to, and the arguments are bound to the proper parameter names inside the frame. Now the CF pointer is moved and the body of the procedure is evaluated.

Step 53



Define is a special form. We must evaluate the expression to find out what the name *quiz-cons* will point to. The *6* evaluates to the value *6*. This result is then bound to the name *wasted* in the Current Frame.

Step 54

For Steps 55 through 59, refer to the Environment Diagram in Step 54. They all look alike.

Evaluating: (cond ((eq? 'bark mesg) b) ...)

#<pp:eq?> ← bark → woof

Cond is a special form, so first we need to evaluate the first conditional expression. To evaluate it, we need to evaluate all of the parts. The name *eq?* follows the path of pointers to frames from the Current Frame until the Global Environment (since it is undefined in all of the frames in between), and is found to be a primitive procedure, #<pp:eq?>. *'bark* evaluates to the symbol *bark*. The name *mesg* is found in the Current Frame and evaluates to *woof*.

Step 55

Evaluating: (cond ((a a) b) ...)

#<pp:eq?> ← bark → woof

Evaluating the primitive procedure *eq?* on the symbols *bark* and *woof* results in #f because they are unequal. The result of #f in the *cond* expression causes us to move on to the next conditional predicate and evaluate it.

Step 56

Evaluating: (cond ... ((eq? 'woof mesg) a) ...)

#<pp:eq?> ← woof → woof

Cond is a special form, so first we need to evaluate the first conditional expression. To evaluate it, we need to evaluate all of the parts. The name *eq?* follows the path of pointers to frames from the Current Frame until the Global Environment (since it is undefined in all of the frames in between), and is found to be a primitive procedure, #<pp:eq?>. *'woof* evaluates to the symbol *woof*. The name *mesg* is found in the Current Frame and evaluates to *woof*.

Step 57

Evaluating: (cond ... ((a a) woof) ...)

#<pp:eq?> ← woof → woof → #t

Evaluating the primitive procedure *eq?* on the symbols *woof* and *woof* results in #t because they are equal. The result of #t in the *cond* expression causes us to evaluate the clause associated with this predicate.

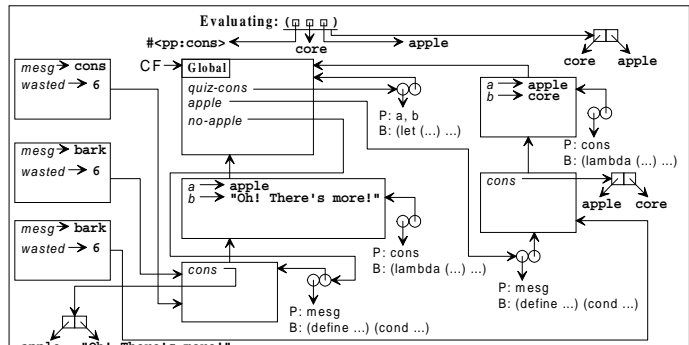
Step 58

Evaluating: (cond ... (a) ...)

#t → apple

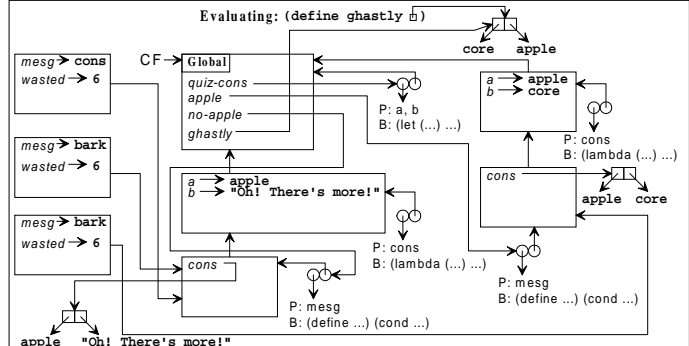
To evaluate the name *a* we look in the Current Frame, don't find it, and then look in the frame pointed to by the Current Frame, and find it is equal to the symbol *apple*. Since this is the last expression in the predicate, and the *cond* expression was the last expression in the procedure, the result of the procedure is the result of evaluating the name *a* in the Current Frame, and we move the CF pointer back to the Global Environment and leave the procedure.

Step 59



Now that each expression inside this expression has been evaluated, we can evaluate this expression. The primitive procedure *cons* makes a pair whose first argument is the car and the second is the cdr. (Note: no frames are made for primitive procedures.)

Step 60



The expression in the *define* special form has been evaluated, now we create the name *ghashtly* in the current frame and make it point to the result of the evaluated expression.

Step 61

