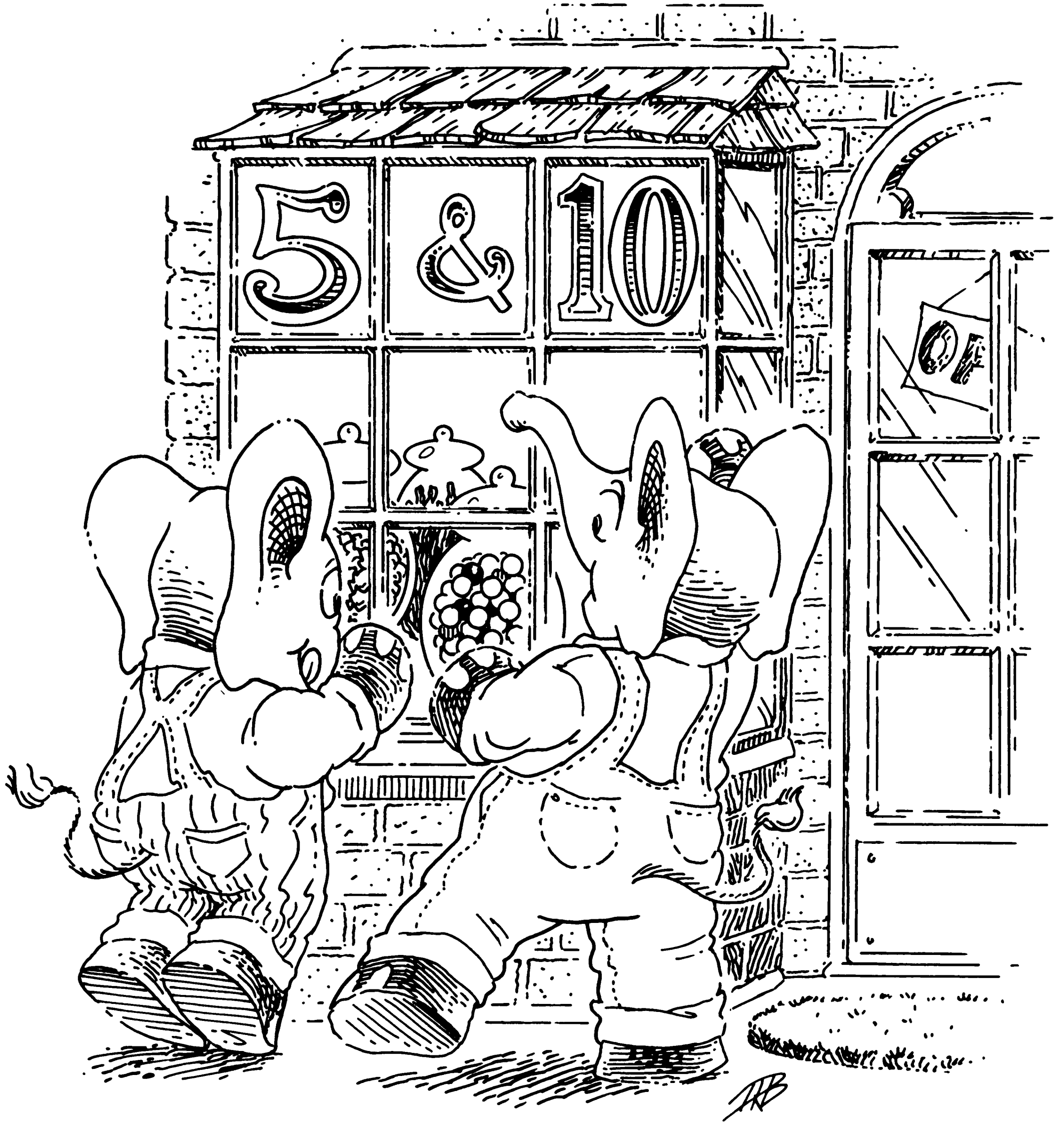
20. What's in Store?



Do you remember tables from chapter 10?

A table is something that pairs names with values.

How did we represent tables?

We used lists and entries.

Could a table be anything else?

Yes, a function. A table acts like a function, because it pairs names with values, in the same way that functions pair arguments with results.

So let's use functions to make tables. Here is a way to make an empty table:

```
(define the-empty-table
  (lambda (name)
    ...))
```

Don't fill in the dots!

In *The Little Schemer* we used (*car* (**quote** ())).

What does that do?

It breaks The Law of Car.

If a table is a function, how can we extract whatever is associated with a name?

We apply the table to the name.

Write the function *lookup* that does that.

```
(define lookup
  (lambda (table name)
    (table name)))
```

Can you explain how *extend* works?

```
(define extend
  (lambda (name1 value table)
    (lambda (name2)
      (cond
        ((eq? name2 name1) value)
        (else (table name2))))))
```

Here are our words:
"It takes a name and a value together with a table and returns a table. The new table first compares its argument with the name. If they are identical, the value is returned. Otherwise, the new table returns whatever the old table returns."

What is the value of

```
(define x 3)
```

No answer.

What is (*value* *e*)
where
e is (define x 3)

No answer.

What is *value*

The name is familiar from chapter 10. But, the function *value* there does not handle (**define** ...).

So the new *value* might be defined like this.

```
(define value
  (lambda (e)
    ...
    (cond
      ((define? e) (*define e))
      (else (the-meaning e))) ... ))
```

Yes, this might do for a while. And don't bother filling in the dots, now. We will do that later.

Should we continue with (**letcc** ...) now?

Oh no!

Okay, we'll wait until later.

Whew!

Do we need *define?*

We don't need to define it now, because it is easy, but here it is anyway.

```
(define define?
  (lambda (e)
    (cond
      ((atom? e) #f)
      ((atom? (car e))
       (eq? (car e) (quote define)))
      (else #f))))
```

Do we need **define*

Yes, we need it. With (**define** ...), we can add new definitions.

Here is **define*

```
(define global-table
  ... the-empty-table ...)
```

```
(define *define
  (lambda (e)
    (set! global-table
      (extend
        (name-of e)
        (box
          (the-meaning
            (right-side-of e)))
        global-table))))
```

This function looks like one of those functions that remembers its arguments.

Yes, **define* uses *global-table* to remember those values that were **define**d.

The table appears to be empty at first.

Is it empty?

We shall soon find out.

When **define* extends a table with a name and a value, will the name always stand for the *same* value?

No, with (**set!** ...) we can change what a name stands for, as we have often seen.

Is this the reason why **define* puts the value in a *box* before it extends the table?

If we knew what a *box* was, the answer might be yes.

Here is the function that makes *box*es:

```
(define box
  (lambda (it)
    (lambda (sel)
      (sel it (lambda (new)
                (set! it new))))))
```

Does this remind you of something we have discussed before?

It should: *bons* from chapter 18 is a similar function.

Here is a function that changes the contents of a *box*

```
(define setbox
  (lambda (box new)
    (box (lambda (it set) (set new)))))
```

Write the function *unbox* which extracts a value from a *box*

That's easy:

```
(define unbox
  (lambda (box)
    (box (lambda (it set) it))))
```

So, is it true that if a name is paired with a *box*ed value that we can change what the name stands for without changing the table?

Yes, it is. Using *setbox* changes the contents of the box but the table stays the *same*.

What is the value of *x*

3.

What is (*value e*)
where
e is x

3.

Here is *the-meaning*

```
(define the-meaning
  (lambda (e)
    (meaning e lookup-in-global-table)))
```

What do you think *lookup-in-global-table* does?

The function *lookup-in-global-table* is a function that takes a name and looks up its value in *global-table*. It is easy to define:

```
(define lookup-in-global-table
  (lambda (name)
    (lookup global-table name)))
```

Is it true that *lookup-in-global-table* is just like a table?

Yes, it is a function that takes a name and returns the value that is paired with the name in *global-table*.

Does this mean *lookup-in-global-table* is like *global-table*

Yes and no. Since **define* changes *global-table*, *lookup-in-global-table* is always just like the most recent *global-table*, not like the one we have now.

Have we seen this before?

Remember $Y_!$ from chapter 16?

Is it important that we always have the most recent value of *global-table*

Yes, we will soon see why that is.

Here is *meaning*

```
(define meaning
  (lambda (e table)
    ((expression-to-action e)
     e table)))
```

What do you think the function *expression-to-action* does?

It translates *e* to a function that knows what to do with the expression and the table.

Do we need to define *expression-to-action*

No, we have seen it in chapter 10; it is easy; and it can wait until later.

Fine, we will consider it later.

Okay.

Here is the most trivial action.

```
(define *quote
  (lambda (e table)
    (text-of e)))
```

Can you define **identifier*

The function **identifier* is similar to **quote*, but it uses *table* to look up what a given name is paired with.

And what is a name paired with?

A name is paired with a box that contains its current value. So **identifier* must *unbox* the result of looking up the value.

And how does **identifier* look up the value?

It's best to have **identifier* use *lookup*, which finds the box that is paired with the name in the table.

```
(define *identifier
  (lambda (e table)
    (unbox (lookup table e))))
```

What is the value of

(**set!** *x* 5)

No answer.

What is the value of *x*

5.

What is (*value* *e*)
where
e is (set! x 5)

No answer.

What is (*value* *e*)
where
e is x

5.

How does **set* differ from **identifier*

It too looks up the box that is paired with the name in a (set! ...) expression, but it changes the contents of the box instead of extracting it.

Where does the new value for the box come from?

It is the value of the right-hand side in a (set! ...) expression.

Can you write **set* now?

Yes, it just means translating the words into a definition:

```
(define *set
  (lambda (e table)
    (setbox
      (lookup table (name-of e))
      (meaning (right-side-of e) table))))
```

Can you describe what **set* does?

Yes.
"The function *lookup* returns the box that is paired with the name whose value is to be changed. The box is then changed so that it contains the value of the right-hand side of the (set! ...) expression."

What is the value of (**lambda** (x) x)	It is a function.
What is (*value* *e*) where *e* is (lambda (x) x)	It could also be a function.
What is the value of ((**lambda** (y) (**set!** x 7) y) 0)	0.
What is the value of x	7.
What is (*value* *e*) where *e* is ((lambda (y) (set! x 7) y) 0)	0.
What is (*value* *e*) where *e* is x	7.
Here is **lambda*	That's interesting, but what are *beglis* and *box-all*?

```
(define *lambda
  (lambda (e table)
    (lambda (args)
      (beglis (body-of e)
        (multi-extend
          (formals-of e)
          (box-all args)
          table)))    ))
```

Okay one more:

```
(define beglis
  (lambda (es table)
    (cond
      ((null? (cdr es))
       (meaning (car es) table))
      (else ((lambda (val)
               (beglis (cdr es) table))
             (meaning (car es) table))))))
```

Can you define *box-all*

Trivial, with that kind of name:

```
(define box-all
  (lambda (vals)
    (cond
      ((null? vals) (quote ()))
      (else (cons (box (car vals))
                  (box-all (cdr vals)))))))
```

Take a look at *beglis*
What is
((**lambda** (*val*) ...)
(*meaning* (*car es*) *table*))

It is the same as
(**let** ((*val* (*meaning* (*car es*) *table*)))
...)
which first determines the value of (*meaning* (*car es*) *table*) and then the value of the value part.

Why didn't we use (**let** ...)

Our functions will work for all the definitions that we need for them. And they do not need to deal with expressions of the shape (let ...) because we know how to do without them.

How do you do without (**let** ...) in
(**let** ((x 1)) ($+$ x 10))

Like this: it's the same as
((**lambda** (x) ($+$ x 10)) 1).

Do you remember how to do without (**let** ...) in
(**let** ((x 1) (y 10)) ($+$ x y))

Yes, it's the same as
((**lambda** (x y) ($+$ x y)) 1 10).

So what does
(**let** ((*val* (*meaning* (*car es*) *table*)))
(*beglis* (*cdr es*) *table*))
do for *beglis*

First, it determines the value of (*meaning* (*car es*) *table*) and names it *val*. And then, it determines the value of (*beglis* (*cdr es*) *table*).

What happens to the value named *val*

Nothing. It is ignored.

Why did we determine a value that is ignored in the end?	Because the values of all but the last expression in the value part of a (**lambda** ...) are ignored.
Can you summarize now what the function *beglis* does for **lambda*	We summarize: "The function *beglis* determines the values of a list of expressions, one at a time, and returns the value of the last one."
How does **lambda* work?	When given (lambda (x y ...) ...), it returns the function that is in the inner box of **lambda*.
What does that function do?	It takes the values of the arguments and apparently extends *table*, pairing each formal name, x, y, ..., with the corresponding argument value.

Write the function *multi-extend*, which takes a list of names, a list of values, and a table and constructs a new table with *extend*

No problem.

```
(define multi-extend
  (lambda (names values table)
    (cond
      ((null? names) table)
      (else
        (extend (car names) (car values)
          (multi-extend
            (cdr names)
            (cdr values)
            table)))))))
```

Okay, so now that we know how *table* is extended, what happens after the new table is constructed?	The function that represents a (lambda ...) expression uses the resulting table to determine the value of the body of the (lambda ...) expression, which was the first argument to **lambda*.

Which parts of the table can change even though the table stays the same?

Each box that the table remembers for any given name may change its value.

That's how (**set!** ...) works, right?

True.

Write *odd?* and *even?* as recursive functions.

Do you mean this pair of functions?

```
(define odd?
  (lambda (n)
    (cond
      ((zero? n) #f)
      (else (even? (sub1 n))))))
```

```
(define even?
  (lambda (n)
    (cond
      ((zero? n) #t)
      (else (odd? (sub1 n))))))
```

Yes, what is (*value e*)
where
e is
```
(define odd?
  (lambda (n)
    (cond
      ((zero? n) #f)
      (else (even? (sub1 n))))))
```

No answer.

What is (*value* (**quote** odd?))

A function.

Which table does the function use when we ask (*value e*)
where
e is (odd? 0)

The function extends *lookup-in-global-table* by pairing n with (a box containing) 0.

And then?

Eventually we get the result: #f.

Does this table know about odd?	It sure does.
Does this table know about even?	Not yet.
Does this mean that (*value* *e*) where *e* is (odd? 1) does not have an answer?	Not yet.
(*value* *e*) where *e* is (define even? (lambda (n) (cond ((zero? n) #t) (else (odd? (sub1 n))))))	No answer.
What is (*value* *e*) where *e* is (odd? 1)	#t . Time for tea and cookies.
Can you explain why?	Here is how we can explain it: "The table that is embedded in the representation of odd? is *lookup-in-global-table*. It is like a table, but when it is given a name, it looks in the most current value of *global-table* for the value that goes with the name. Since *global-table* may grow, *lookup* is guaranteed to look through all definitions ever made.
Have we seen this method of changing a function before?	Yes, when we derived $Y_!$ in chapter 16, and when we discussed *lookup-in-global-table*.
If **lambda* represents (lambda ...) with a function, how does **application* work?	That is easy. It just applies the value of the first expression in an application to the values of the rest of the application's expressions.

Here is the function **application*

```
(define *application
  (lambda (e table)
    ((meaning (function-of e) table)
     (evlis (arguments-of e) table))))
```

The functions *function-of* and *arguments-of* are easy ones, and we can write them later. But what does the function *evlis* do?

The function *evlis* determines the values of a list of expressions, one at a time, and returns the list of values. It is quite similar to *beglis*.

```
(define evlis
  (lambda (args table)
    (cond
      ((null? args) (quote ()))
      (else
        ((lambda (val)
           (cons val
             (evlis (cdr args) table)))
         (meaning (car args) table))))))
```

Why do we use ((**lambda** (*val*) ...) ...) in *evlis*

We still don't have (**let** ...).

Do we need ((**lambda** (*val*) ...) ...) here too?

Yes,[1] here and in *beglis*.
Thank you, John Reynolds.

[1] S: So that our definitions always work in Scheme.

What happens when we determine the value of (*value* *e*)
where
e is (car (cons 0 (quote ())))

The function *value* uses the function *the-meaning*, which in turn uses *meaning* to determine a value.

And then?

Then *expression-to-action* determines that (car (cons 0 (quote ()))) is an application, so that **application* takes over.

Does this mean the value of
(*meaning* (**quote** car) *table*)
must be a function?

Yes,
because **application* expects (*function-of* *e*) to be represented as a (**lambda** ...), no matter what *e* is.

What kind of function does **application* expect from (*meaning e table*) where
e is car

It will need to be a function that takes all of its arguments in a list and then does the right thing.

How many values should the list contain that (*meaning* (**quote** car) *table*) receives?

Exactly one.

And what kind of value should this be?

The value must be a list. And then we take its *car*.

Define the function that we can use to represent *car*

Let's call it *:car*.

```
(define :car
  (lambda (args-in-a-list)
    (car (car args-in-a-list))))
```

Are there other primitives for which we should have a representation?

Yes, *cdr* is one, and *add1* is another.

We should have a function that makes representations for such functions.

Here is one:

```
(define a-prim
  (lambda (p)
    (lambda (args-in-a-list)
      (p (car args-in-a-list)))))
```

We also need one for functions like *cons* that take two arguments.

No problem: now the argument list must contain exactly two elements, and we just do what is necessary:

```
(define b-prim
  (lambda (p)
    (lambda (args-in-a-list)
      (p (car args-in-a-list)
         (car (cdr args-in-a-list))))))
```

And now we can define **const*

```
(define *const
  (lambda (e table)
    (cond
      ((number? e) e)
      ((eq? e #t) #t)
      ((eq? e #f) #f)
      ((eq? e (quote cons))
       (b-prim cons))
      ((eq? e (quote car))
       (a-prim car))
      ((eq? e (quote cdr))
       (a-prim cdr))
      ((eq? e (quote eq?))
       (b-prim eq?))
      ((eq? e (quote atom?))
       (a-prim atom?))
      ((eq? e (quote null?))
       (a-prim null?))
      ((eq? e (quote zero?))
       (a-prim zero?))
      ((eq? e (quote add1))
       (a-prim add1))
      ((eq? e (quote sub1))
       (a-prim sub1))
      ((eq? e (quote number?))
       (a-prim number?)))))
```

Can you rewrite **const* using (**let** ...)

Where? Why? There are no repeated expressions.

What is (*value e*)
where
e is
```
(define ls
  (cons
    (cons
      (cons 1 (quote ()))
      (quote ()))
    (quote ())))
```

We add ls to *global-table* and rember what it stands for.

What is (*value e*)
where
e is (car (car (car ls)))

1.

How do we determine this value?	It is an application, so we need to find out what car is and the value of the argument.
How do we determine the value of car	We use the function **const*: (**const* (**quote** car)) tells us.
And that is?	It is the same as (*a-prim car*), which is like *:car*.
How do we determine the value of the argument?	It is an application, so we need to find out what car is and the value of the argument.
(*value* (**quote** car))	We use the function **const*: (**const* (**quote** car)) tells us.
And?	It is the same as (*a-prim car*), which is like *:car*.
How do we determine the value of the argument?	It is an application, so we need to find out what car is and the value of the argument.
(*value* (**quote** car))	We use the function **const*: (**const* (**quote** car)) tells us.
How often did we have to figure out the value of (*a-prim car*)	Three times.
Is it the same value every time?	It sure is.
Is this wasteful?	Yes: let's name the value!
Can we really use (**let** ...)	We can: we just saw how to replace it.

Where do we put the (**let** ...)

Around (**cond** ...)?

When would we determine the values in this (**let** ...)

Each time **const* determines the value of car.

So this wouldn't help.

Let's put the (**let** ...) outside of (**lambda** ...).

Here is **const* with (**let** ...)

```
(define *const
  (let ((:cons (b-prim cons))
        (:car (a-prim car))
        (:cdr (a-prim cdr))
        (:null? (a-prim null?))
        (:eq? (b-prim eq?))
        (:atom? (a-prim atom?))
        (:number? (a-prim number?))
        (:zero? (a-prim zero?))
        (:add1 (a-prim add1))
        (:sub1 (a-prim sub1))
        (:number? (a-prim number?)))
    (lambda (e table)
      (cond
        ((number? e) e)
        ((eq? e #t) #t)
        ((eq? e #f) #f)
        ((eq? e (quote cons)) :cons)
        ((eq? e (quote car)) :car)
        ((eq? e (quote cdr)) :cdr)
        ((eq? e (quote null?)) :null?)
        ((eq? e (quote eq?)) :eq?)
        ((eq? e (quote atom?)) :atom?)
        ((eq? e (quote zero?)) :zero?)
        ((eq? e (quote add1)) :add1)
        ((eq? e (quote sub1)) :sub1)
        ((eq? e (quote number?))
         :number?)))))
```

Can you rewrite **const* without (**let** ...)

```
(define *const
  ((lambda (:cons :car :cdr :null?
            :eq? :atom?
            :zero? :add1 :sub1 :number?)
     (lambda (e table)
       (cond
         ((number? e) e)
         ((eq? e #t) #t)
         ((eq? e #f) #f)
         ((eq? e (quote cons)) :cons)
         ((eq? e (quote car)) :car)
         ((eq? e (quote cdr)) :cdr)
         ((eq? e (quote null?)) :null?)
         ((eq? e (quote eq?)) :eq?)
         ((eq? e (quote atom?)) :atom?)
         ((eq? e (quote zero?)) :zero?)
         ((eq? e (quote add1)) :add1)
         ((eq? e (quote sub1)) :sub1)
         ((eq? e (quote number?))
          :number?))))
   (b-prim cons)
   (a-prim car)
   (a-prim cdr)
   (a-prim null?)
   (b-prim eq?)
   (a-prim atom?)
   (a-prim zero?)
   (a-prim add1)
   (a-prim sub1)
   (a-prim number?)))
```

The Fifteenth Commandment

(final version)

Use (let ...) to name the values of repeated expressions in a function definition if they may be evaluated twice for one and the same use of the function. And use (let ...) to name the values of expressions (without set!) that are re-evaluated every time a function is used.

Are we now ready to work with *value*

Almost.

What is missing?

The one kind of expression that we still need to treat is the set of (cond ...) expressions.

Is **cond* simple?

Yes, there is nothing to it. We must determine the first line in the (cond ...) expression for which the question is true.

And when we find one?

Then we determine the value of the answer in that line.

Here is the function **cond* which uses *evcon* to do its job:

```
(define *cond
  (lambda (e table)
    (evcon (cond-lines-of e) table)))
```

Can you define the function *evcon*

By now, this is easy:

```
(define evcon
  (lambda (lines table)
    (cond
      ((else? (question-of (car lines)))
       (meaning (answer-of (car lines))
         table))
      ((meaning (question-of (car lines))
         table)
       (meaning (answer-of (car lines))
         table))
      (else (evcon (cdr lines) table)))))
```

What is (*value* *e*) where *e* is (cond (else 0))	0.
What is (*value* *e*) where *e* is (cond ((null? (cons 0 (quote ()))) 0) (else 1))	1.
What is (*value* *e*) where *e* is (cond)	No answer.
Time to continue with (letcc ...)	Is it time to go to the North Pole?
Yes, (**letcc** *skip* ...) remembers the North Pole so that *skip* can find its way back. How does it do this?	We are about to find out.
What does *skip* stand for in (**letcc** *skip* ...)	We said it was like a function.
Why is it like a function?	We use (*skip* 0) when we want to go to the North Pole named *skip*.
How is it different from a function?	When we use *skip*, it forgets everything that is about to happen.
How can ***letcc* name a North Pole that remembers what is left to do?	With (**letcc** *skip* ...).
And now that *skip* is a North Pole, how can we turn it into a function that ***application* can use?	The North Pole *skip* stands for a function of one argument. So the function that represents it for ***application* must take a list that contains the representation of this argument.

Can we use something that we have seen before to make this function?

Yes, we can use (*a-prim skip*). This is exactly the kind of function we need.

What is the name for the function just created?

If (letcc skip ...) is the expression that **letcc* receives, then skip is the name.

And how do we associate this name with the function we created?

We can use *extend* to put the new pair into the table that **letcc* receives.

Here is the function **letcc*

```
(define *letcc
  (lambda (e table)
    (letcc skip
      (beglis (ccbody-of e)
        (extend
          (name-of e)
          (box (a-prim skip))
          table)))))
```

Can you describe what it does?

It sets up the North Pole *skip*, turns it into a function that **application* can use, associates the name in *e* with this function, and evaluates the value part of the expression.

That's exactly what happens.

Whew.

But what would happen if we tried to determine the value of (*value e*)
where
  e is z

The name z hasn't been used with define yet.

So what would happen?

We still would like to have a good answer to this question. We have not yet finished the function *the-empty-table*.

Have you forgotten about forgetting? We just showed you how it works.

It is wrong to ask for the value of a name that is not in the table.

What should happen when something wrong happens?

We could forget all pending computations.[1]

[1] We could also use (**letcc** ...) to remember how the computation would have proceeded, if nothing wrong had happened.

True enough. And how can we forget such pending computations?

We use (**letcc** ...).

Where should the North Pole be while we determine (*value e*)

Right at the beginning of *value*:

```
(define value
  (lambda (e)
    (letcc the-end
      ...
      (cond
        ((define? e) (*define e))
        (else (the-meaning e))))))
```

But what can we put in the place of the dots?

Well, we probably should remember *the-end* until we are done.

Perhaps we should use (**set!** ...) to remember it.

Yes, we have always used (**set!** ...) to remember things.

Here is the final definition of *value*

```
(define value
  (lambda (e)
    (letcc the-end
      (set! abort the-end)
      (cond
        ((define? e) (*define e))
        (else (the-meaning e))))))
```

Can you finish this?

We need to define *abort*:

```
(define abort)
```

And how does *abort* help us?

We should probably use it with *the-empty-table*, which is why we redefined *value* in the first place.

Can we now use *abort* inside of *the-empty-table* so that it no longer breaks The Law of Car?

Definitely. Here is how we can fill in the dots in a better way:

```
(define the-empty-table
  (lambda (name)
    (abort
      (cons (quote no-answer)
        (cons name (quote ()))))))
```

We didn't talk about *expression-to-action* and *atom-to-action*

```
(define expression-to-action
  (lambda (e)
    (cond
      ((atom? e) (atom-to-action e))
      (else (list-to-action e)))))
(define atom-to-action
  (lambda (e)
    (cond
      ((number? e) *const)
      ((eq? e #t) *const)
      ((eq? e #f) *const)
      ((eq? e (quote cons)) *const)
      ((eq? e (quote car)) *const)
      ((eq? e (quote cdr)) *const)
      ((eq? e (quote null?)) *const)
      ((eq? e (quote eq?)) *const)
      ((eq? e (quote atom?)) *const)
      ((eq? e (quote zero?)) *const)
      ((eq? e (quote add1)) *const)
      ((eq? e (quote sub1)) *const)
      ((eq? e (quote number?)) *const)
      (else *identifier))))
```

Is there anything left to do?

Yes, a few simple things:

```
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
       (cond
         ((eq? (car e) (quote quote))
          *quote)
         ((eq? (car e) (quote lambda))
          *lambda)
         ((eq? (car e) (quote letcc))
          *letcc)
         ((eq? (car e) (quote set!))
          *set)
         ((eq? (car e) (quote cond))
          *cond)
         (else *application)))
      (else *application))))
```

Here are a few more:

```
(define text-of
  (lambda (x)
    (car (cdr x))))
(define formals-of
  (lambda (x)
    (car (cdr x))))
(define body-of
  (lambda (x)
    (cdr (cdr x))))
(define ccbody-of
  (lambda (x)
    (cdr (cdr x))))
(define name-of
  (lambda (x)
    (car (cdr x))))
(define right-side-of
  (lambda (x)
    (cond
      ((null? (cdr (cdr x))) 0)
      (else (car (cdr (cdr x)))))))
(define cond-lines-of
  (lambda (x)
    (cdr x)))
(define else?
  (lambda (x)
    (cond
      ((atom? x) (eq? x (quote else)))
      (else #f))))
(define question-of
  (lambda (x)
    (car x)))
(define answer-of
  (lambda (x)
    (car (cdr x))))
(define function-of
  (lambda (x)
    (car x)))
(define arguments-of
  (lambda (x)
    (cdr x)))
```

It returns 0 if there is no right-hand side. This handles definitions like
(**define** *global-table*)
where there is no right-hand side.

What is unusual about *right-side-of*

How does it take care of such definitions?	It makes up a value for the name until it is changed to what it is supposed to be.
So what's the value of all of this?	It makes people hungry.
What is (*value* *e*) where *e* is (value 1)	(no-answer value).
How can we teach *value* what value means?	We need to determine the value of (*value* *e*) where *e* is (define value (lambda (e) (letcc the-end (set! abort the-end) (cond ((define? e) (*define e)) (else (the-meaning e)))))).
And then?	Then the answer to our original question is (no-answer define?).
So we also need to add define? to *global-table*	Yes, we do. And while we are at it, we might as well add *define, the-meaning, lookup, lookup-in-global-table, and a few others.
Are you sure we didn't forget anything?	We can try it out.
How can we find out what other functions we need?	The same way that we found out that we needed define?.
What is (*value* *e*) where *e* is (value 1)	First we decide that *e* is not a definition, so we determine the value of (*the-meaning* *e*).

And then?	Then we determine the value of (*meaning e lookup-in-global-table*).
Is this all?	No. After we find out that *e* is an application, we need to determine (*meaning f table*) and (*meaning a table*) where *f* is value *a* is 1 and *table* is *lookup-in-global-table*.
Is it easy from here on?	The value of value is a function and the value of 1 is 1. The function that represents *value* extends *table* by pairing *e* with 1. And now the function works basically like *value*.
Does that mean that we get the result 1	Yes, because we added all the things we needed to *global-table*.
If *e* is some expression so that (*value e*) makes sense and if *f* represents *e*, then we can always determine the same value by calculating (*value value-on-f*) where *value-on-f* is the result of (*cons v* (*cons f* (**quote** ()))) where *v* is value	That is complicated and true.
Isn't it heavy duty work?	It sure burns a lot of calories, but of course that only means that we will soon be ready for a lot more food.

Enjoy yourself with a great dinner:

((escargots garlic)
(chicken Provençal)
((red wine) and Brie))†

† No, you don't have to eat the parentheses.