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.

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))))
```

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.

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?)))
```