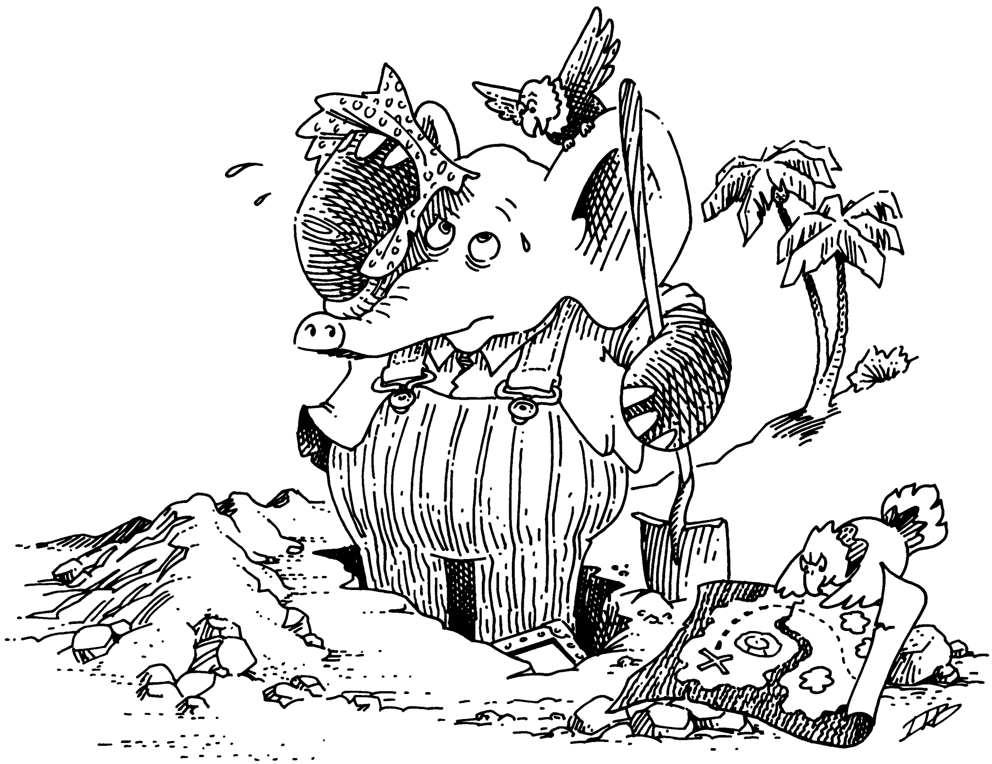
10. What Is the Value of All of This?



An entry is a pair of lists whose first list is a set. Also, the two lists must be of equal length. Make up some examples for entries.

Here are our examples:

```
((appetizer entrée beverage)
 (paté boeuf vin))
```

and

```
((appetizer entrée beverage)
 (beer beer beer))
```

and

```
((beverage dessert)
 ((food is) (number one with us))).
```

How can we build an entry from a set of names and a list of values?

(**define** *new-entry build*)

Try to build our examples with this function.

What is (*lookup-in-entry name entry*) where *name* is entrée and *entry* is ((appetizer entrée beverage) (food tastes good))

tastes.

What if *name* is dessert

In this case we would like to leave the decision about what to do with the user of *lookup-in-entry*.

How can we accomplish this?

lookup-in-entry takes an additional argument that is invoked when *name* is not found in the first list of an entry.

How many arguments do you think this extra function should take?

We think it should take one, *name*. Why?

Here is our definition of *lookup-in-entry*

```
(define lookup-in-entry
  (lambda (name entry entry-f)
    (lookup-in-entry-help name
      (first entry)
      (second entry)
      entry-f)))
```

Finish the function *lookup-in-entry-help*

```
(define lookup-in-entry-help
  (lambda (name names values entry-f)
    (cond
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ ))))
```

```
(define lookup-in-entry-help
  (lambda (name names values entry-f)
    (cond
      ((null? names) (entry-f name))
      ((eq? (car names) name)
       (car values))
      (else (lookup-in-entry-help name
              (cdr names)
              (cdr values)
              entry-f))))))
```

A table (also called an environment) is a list of entries. Here is one example: the empty table, represented by ()
Make up some others.

Here is another one:
(((appetizer entrée beverage)
  (paté boeuf vin))
 ((beverage dessert)
  ((food is) (number one with us)))).

Define the function *extend-table* which takes an entry and a table (possibly the empty one) and creates a new table by putting the new entry in front of the old table.

```
(define extend-table cons)
```

What is
  (*lookup-in-table name table table-f*)
where
  name is entrée
  table is (((entrée dessert)
              (spaghetti spumoni))
             ((appetizer entrée beverage)
              (food tastes good)))
and
  table-f is (lambda (*name*) ...)

It could be either spaghetti or tastes, but *lookup-in-table* searches the list of entries in order. So it is spaghetti.

Write *lookup-in-table*
Hint: Don't forget to get some help.

```
(define lookup-in-table
  (lambda (name table table-f)
    (cond
      ((null? table) (table-f name))
      (else (lookup-in-entry name
              (car table)
              (lambda (name)
                (lookup-in-table name
                  (cdr table)
                  table-f)))))))
```

Can you describe what the following function represents:

```
(lambda (name)
  (lookup-in-table name
    (cdr table)
    table-f))
```

This function is the action to take when the name is not found in the first entry.

In the preface we mentioned that sans serif typeface would be used to represent atoms. To this point it has not mattered. Henceforth, you must notice whether or not an atom is in sans serif.

Remember to be very conscious as to whether or not an atom is in sans serif.

Did you notice that "sans serif" was not in sans serif?

We hope so. This is "sans serif" in sans serif.

Have we chosen a good representation for expressions?

Yes. They are all S-expressions so they can be data for functions.

What kind of functions?

For example, *value*.

Do you remember *value* from chapter 6?

Recall that *value* is the function that returns the natural value of expressions.

What is the value of
(*car* (**quote** (a b c)))

We don't even know what (**quote** (a b c)) is.

What is the value of (*cons rep-a* (*cons rep-b* (*cons rep-c* (**quote** ()))))) where *rep-a* is a *rep-b* is b and *rep-c* is c	It is the same as (a b c).
Great. And what is the value of (*cons rep-car* (*cons* (*cons rep-quote* (*cons* (*cons rep-a* (*cons rep-b* (*cons rep-c* (**quote** ()))))) (**quote** ())))) (**quote** ())))) where *rep-car* is car *rep-quote* is quote *rep-a* is a *rep-b* is b and *rep-c* is c	It is a representation of the expression: (car (quote (a b c))).
What is the value of (*car* (**quote** (a b c)))	a.
What is (*value e*) where *e* is (car (quote (a b c)))	a.
What is (*value e*) where *e* is (quote (car (quote (a b c))))	(car (quote (a b c))).

What is (*value* *e*) where *e* is (add1 6)	7.
What is (*value* *e*) where *e* is 6	6, because numbers are constants.
What is (*value* *e*) where *e* is (quote nothing)	nothing.
What is (*value* *e*) where *e* is nothing	nothing has no value.

What is (*value* *e*)
where
e is ((lambda (nothing)
    (cons nothing (quote ())))
  (quote
    (from nothing comes something)))
	((from nothing comes something)).

What is (*value* *e*)
where
e is ((lambda (nothing)
    (cond
      (nothing (quote something))
      (**else** (quote nothing))))
  #t)
	something.

What is the type of *e* where *e* is 6	**const*.
What is the type of *e* where *e* is #f	**const*.

What is (*value* *e*) where *e* is #f	#f.
What is the type of *e* where *e* is cons	**const*.
What is (*value* *e*) where *e* is car	(primitive car).
What is the type of *e* where *e* is (quote nothing)	**quote*.
What is the type of *e* where *e* is nothing	**identifier*.
What is the type of *e* where *e* is (lambda (x y) (cons x y))	**lambda*.

What is the type of *e*
where
   e is ((lambda (nothing)
            (cond
              (nothing (quote something))
              (else (quote nothing))))
          #t)
	**application*.

What is the type of *e*
where
   e is (cond
          (nothing (quote something))
          (else (quote nothing)))
	**cond*.

How many types do you think there are?

We found six:
**const*
**quote*
**identifier*
**lambda*
**cond*
and
**application*.

How do you think we should represent types?

We choose functions. We call these functions "actions."

If actions are functions that do "the right thing" when applied to the appropriate type of expression, what should *value* do?

You guessed it. It would have to find out the type of expression it was passed and then use the associated action.

Do you remember *atom-to-function* from chapter 8?

We found *atom-to-function* useful when we rewrote *value* for numbered expresssions.

Below is a function that produces the correct action (or function) for each possible S-expression:

```
(define expression-to-action
  (lambda (e)
    (cond
      ((atom? e) (atom-to-action e))
      (else (list-to-action e)))))
```

Define the function *atom-to-action*[1]

```
(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))))
```

[1] Ill-formed S-expressions such as (quote a b), (), (lambda (#t) #t), (lambda (5) 5), (lambda (car) car), (lambda a), (cond (3 c) (else b) (6 a)), and (1 2) are not considered here. They can be detected by an appropriate function to which S-expressions are submitted before they are passed on to *value*.

Now define the help function *list-to-action*

```
(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 cond))
          *cond)
         (else *application)))
      (else *application))))
```

Assuming that *expression-to-action* works, we can use it to define *value* and *meaning*

```
(define value
  (lambda (e)
    (meaning e (quote ()))))
```

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

What is (**quote** ()) in the definition of *value*

It is the empty table. The function *value*,[1] together with all the functions it uses, is called an interpreter.

[1] The function *value* approximates the function **eval** available in Scheme (and Lisp).

Actions do speak louder than words.

How many arguments should actions take according to the above?

Two, the expression *e* and a table.

Here is the action for constants.

```
(define *const
  (lambda (e table)
    (cond
      ((number? e) e)
      ((eq? e #t) #t)
      ((eq? e #f) #f)
      (else (build (quote primitive) e)))))
```

Is it correct?

Yes, for numbers, it just returns the expression, and this is all we have to do for 0, 1, 2, ...
For #t, it returns true.
For #f, it returns false.
And all other atoms of constant type represent primitives.

Here is the action for *quote

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

Define the help function *text-of*

```
(define text-of second)
```

Have we used the table yet?

No, but we will in a moment.

Why do we need the table?

To remember the values of identifiers.

Given that the table contains the values of identifiers, write the action **identifier*

```
(define *identifier
  (lambda (e table)
    (lookup-in-table e table initial-table)))
```

Here is *initial-table*

```
(define initial-table
  (lambda (name)
    (car (quote ()))))
```

When is it used?

Let's hope never. Why?

What is the value of (lambda (x) x)

We don't know yet, but we know that it must be the representation of a non-primitive function.

How are non-primitive functions different from primitives?

We know what primitives do; non-primitives are defined by their arguments and their function bodies.

So when we want to use a non-primitive we need to remember its formal arguments and its function body.

At least. Fortunately this is just the *cdr* of a lambda expression.

And what else do we need to remember?

We will also put the table in, just in case we might need it later.

And how do we represent this?

In a list, of course.

Here is the action **lambda*

```
(define *lambda
  (lambda (e table)
    (build (quote non-primitive)
      (cons table (cdr e)))))
```

What is (*meaning e table*)
where
e is (lambda (x) (cons x y))
and
table is (((y z) ((8) 9)))

$$(\text{non-primitive}\ (\underbrace{(((\text{y z})\ ((8)\ 9)))}_{\text{table}}\ \ \underbrace{(\text{x})}_{\text{formals}}\ \ \underbrace{(\text{cons x y})}_{\text{body}}\ ))$$

It is probably a good idea to define some help functions for getting back the parts in this three element list (i.e., the table, the formal arguments, and the body). Write *table-of formals-of* and *body-of*

```
(define table-of first)
```

```
(define formals-of second)
```

```
(define body-of third)
```

Describe (**cond** ...) in your own words.

It is a special form that takes any number of **cond**-lines. It considers each line in turn. If the question part on the left is false, it looks at the rest of the lines. Otherwise it proceeds to answer the right part. If it sees an **else**-line, it treats that **cond**-line as if its question part were true.

Here is the function *evcon* that does what we just said in words:

```
(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)))))
```

Write *else?* and the help functions *question-of* and *answer-of*

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

```
(define question-of first)
```

```
(define answer-of second)
```

Didn't we violate The First Commandment?

Yes, we don't ask (*null? lines*), so one of the questions in every **cond** better be true.

Now use the function *evcon* to write the **cond* action.

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

```
(define cond-lines-of cdr)
```

Aren't these help functions useful?

Yes, they make things quite a bit more readable. But you already knew that.

Do you understand **cond* now?

Perhaps not.

How can you become familiar with it?

The best way is to try an example. A good one is:
  (**cond e table*)
where
  e is (cond (coffee klatsch) (else party))
and
  table is (((coffee) (#t))
          ((klatsch party) (5 (6)))).

Have we seen how the table gets used?

Yes, **lambda* and **identifier* use it.

But how do the identifiers get into the table?

In the only action we have not defined: **application*.

How is an application represented?

An application is a list of expressions whose *car* position contains an expression whose value is a function.

How does an application differ from a special form, like (**and** ...) (**or** ...) or (**cond** ...)

An application must always determine the meaning of all its arguments.

Before we can apply a function, do we have to get the meaning of all of its arguments?

Yes.

Write a function *evlis* that takes a list of (representations of) arguments and a table, and returns a list composed of the meaning of each argument.

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

What else do we need before we can determine the meaning of an application?

We need to find out what its *function-of* means.

And what then?

Then we apply the meaning of the function to the meaning of the arguments.

Here is **application*

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

Is it correct?

Of course. We just have to define *apply*, *function-of*, and *arguments-of* correctly.

Write *function-of* and *arguments-of*

```
(define function-of car)
```

```
(define arguments-of cdr)
```

How many different kinds of functions are there?

Two: primitives and non-primitives.

What are the two representations of functions?

(primitive *primitive-name*) and
(non-primitive (*table formals body*))
The list (*table formals body*) is called a closure record.

Write *primitive?* and *non-primitive?*

```
(define primitive?
  (lambda (l)
    (eq? (first l) (quote primitive))))
```

```
(define non-primitive?
  (lambda (l)
    (eq? (first l) (quote non-primitive))))
```

Now we can write the function *apply*

Here it is:

```
(define apply¹
  (lambda (fun vals)
    (cond
      ((primitive? fun)
       (apply-primitive
         (second fun) vals))
      ((non-primitive? fun)
       (apply-closure
         (second fun) vals)))))
```

[1] If *fun* does not evaluate to either a primitive or a non-primitive as in the expression ((lambda (x) (x 5)) 3), there is no answer. The function *apply* approximates the function **apply** available in Scheme (and Lisp).

This is the definition of *apply-primitive*

```
(define apply-primitive
  (lambda (name vals)
    (cond
      ((eq? name ___1___)
       (cons (first vals) (second vals)))
      ((eq? name (quote car))
       (car (first vals)))
      ((eq? name (quote cdr))
       (___2___ (first vals)))
      ((eq? name (quote null?))
       (null? (first vals)))
      ((eq? name (quote eq?))
       (___3___ (first vals) ___4___))
      ((eq? name (quote atom?))
       (___5___ (first vals)))
      ((eq? name (quote zero?))
       (zero? (first vals)))
      ((eq? name (quote add1))
       (add1 (first vals)))
      ((eq? name (quote sub1))
       (sub1 (first vals)))
      ((eq? name (quote number?))
       (number? (first vals))))))
```

Fill in the blanks.

1. (**quote** cons)
2. *cdr*[1]
3. *eq?*
4. (*second vals*)
5. *:atom?*

```
(define :atom?
  (lambda (x)
    (cond
      ((atom? x) #t)
      ((null? x) #f)
      ((eq? (car x) (quote primitive))
       #t)
      ((eq? (car x) (quote non-primitive))
       #t)
      (else #f))))
```

[1] The function *apply-primitive* could check for applications of *cdr* to the empty list or *sub1* to 0, etc.

Is *apply-closure* the only function left?

Yes, and *apply-closure* must extend the table.

How could we find the result of (f a b) where
  f is (lambda (x y) (cons x y))
  a is 1
and
  b is (2)

That's tricky. But we know what to do to find the meaning of
  (cons x y)
where
  table is (((x y)
             (1 (2)))).

Why can we do this?

Here, we don't need *apply-closure*.

Can you generalize the last two steps?

Applying a non-primitive function—a closure—to a list of values is the same as finding the meaning of the closure's body with its table extended by an entry of the form

(*formals values*)

In this entry, *formals* is the *formals* of the closure and *values* is the result of *evlis*.

Have you followed all this?

If not, here is the definition of *apply-closure*.

```
(define apply-closure
  (lambda (closure vals)
    (meaning (body-of closure)
      (extend-table
        (new-entry
          (formals-of closure)
          vals)
        (table-of closure)))))
```

This is a complicated function and it deserves an example.

In the following,

```
closure is ((((u v w)
              (1 2 3))
             ((x y z)
              (4 5 6)))
            (x y)
            (cons z x))
```

and

vals is ((a b c) (d e f)).

What will be the new arguments of *meaning*

The new *e* for *meaning* will be (cons z x) and the new *table* for *meaning* will be

```
(((x y)
  ((a b c) (d e f)))
 ((u v w)
  (1 2 3))
 ((x y z)
  (4 5 6))).
```

What is the meaning of (*cons z x*)
where *z* is 6
and
  x is (a b c)
	The same as
	  (*meaning e table*)
	where
	  e is (cons z x)
	and
	  table is (((x y)
	                  ((a b c) (d e f)))
	                 ((u v w)
	                  (1 2 3))
	                 ((x y z)
	                  (4 5 6))).

Let's find the meaning of all the arguments.
What is
  (*evlis args table*)
where
  args is (z x)
and
  table is (((x y)
                  ((a b c) (d e f)))
                 ((u v w)
                  (1 2 3))
                 ((x y z)
                  (4 5 6)))
	In order to do this, we must find both
	  (*meaning e table*)
	where
	  e is z
	and
	  (*meaning e table*)
	where
	  e is x.

What is the (*meaning e table*) where *e* is z	6, by using **identifier*.
What is (*meaning e table*) where *e* is x	(a b c), by using **identifier*.
So, what is the result of *evlis*	(6 (a b c)), because *evlis* returns a list of the meanings.
What is (*meaning e table*) where *e* is cons	(primitive cons), by using **const*.

We are now ready to (*apply fun vals*) where *fun* is (primitive cons) and *vals* is (6 (a b c)) Which path should we take?	The *apply-primitive* path.
Which **cond**-line is chosen for (*apply-primitive name vals*) where *name* is cons and *vals* is (6 (a b c))	The third: ((*eq? name* (**quote** cons)) (*cons* (*first vals*) (*second vals*))).
Are we finished now?	Yes, we are exhausted.
But what about (**define** ...)	It isn't needed because recursion can be obtained from the Y combinator.
Is (**define** ...) really not needed?	Yes, but see *The Seasoned Schemer*.
Does that mean we can run the interpreter on the interpreter if we do the transformation with the Y combinator?	Yes, but don't bother.
What makes *value* unusual?	It sees representations of expressions.
Should *will-stop?* see representations of expressions?	That may help a lot.
Does it help?	No, don't bother—we can play the same game again. We would be able to define a function like *last-try?* that will show that we cannot **define** the new and improved *will-stop?*.
else	Yes, it's time for a banquet.