Lesson #5: Historical Lisp in Racket

What's It All About?

The plan is to **loosely** present the ten functions of historical Lisp by embedding some Scheme code within some English prose. The Scheme code will be lifted from a Racket session. The English prose will be a more or less extemporaneous description of the thoughts that come to mind as I enter the code into the machine. My hope is that you will find the informality to be appealing in one way or another.

The "looseness" of the presentation will permit us to continue our study of Racket as we focus our attention on historical Lisp.

The Ten Functions of Historical Lisp

The agenda, more precisely, is simply to discuss the most salient functions of historical Lisp: `quote`, `eval`, `car`, `cdr`, `cons`, `eq`, `atom`, `lambda`, `define`, and `cond` in a manner that is more or less consistent with the late 1950s take on these functions, but that features Scheme code rather than Lisp 1.5 code.

Recall REPL? Recall that it repeatedly reads an S-expression, evaluates it, and prints the result. We are about to make ample use of REPL!

Quote and Eval

An expression is considered to be a **constant** if it evaluates to itself. That is the standard definition of a constant. Consider ...

```
> 9
9
> "red"
"red"
> 'red
'red
>
```

Evidently, `9` and `"red"` and `'red` are all constants, since they evaluate to themselves. It turns out that `'whatever` can be expressed as `( quote whatever )`. The former mode of expression is how people tend to like to read and write a quoted S-expression. The latter mode of expression consistent with the actual Lisp code.

```
> ( quote red )
'red
> 'red
'red
>
```

It is best to think of `quote` as an evaluation suppression form. It simply returns its unevaluated argument. Selective suppression of evaluation via quote is extremely important in an evaluation hungry language like Lisp!

```
> red
# red: undefined; cannot reference an identifier before its definition
> ( quote red )
'red
```

I took a liberty in representing the error message just now, by pruning it just a bit, but I preserved its essence. Given a symbolic atom (like `red`), `eval` will try to evaluate it by finding a value to which it has been bound. If it has not been bound to a value, Lisp will express that fact in no uncertain terms.

The evaluation of expressions involving the application of an operator to operands is straight forward. This is accomplished by means of a simple process called "standard evaluation". **Standard evaluation** is the following two step process:

1. Evaluate all of the operands
2. Apply the operator to the values of the operands

Some examples of standard form evaluation involving numeric processing:

```
> ( + 3 4 )
7
> ( + ( / 3 1 ) ( * 2 2 ) )
7
> ( + 1 2 3 4 5 6 7 8 9 10 )
55
> ( / ( * 10 ( + 10 1 ) ) 2 )
55
```

Given a list, eval will, generally speaking, interpret its first element as a function (operation), the remaining elements as operands, and perform standard evaluation.

But what of the case when Lisp encounters a list in which the first element is not a function?

```
> ( red yellow blue )
#  red: undefined; cannot reference an identifier before its definition
```

Trouble! That is, except in the case of "special forms", like quote! Again, I took a liberty in representing the error message, by pruning it just a bit, but, again, I preserved its essence.

Recap: What have we done so far? We have explicitly discussed `quote` while implicitly considering `eval` (which is embedded in the REPL). If you would actually like to read `eval`, find your way to McCarthy's presentation of the function on page 13 of "Lisp 1.5 Programmer's Manual" - one of the most important pages ever written in the history of the field of computer science!

Car, Cdr and Cons

Time to consider the "big three" when it comes to list processing operators: car, cdr and cons. The keep things clean, we will, for now, consider only S-expressions that can be represented as lists.

Recall that the car of a nonempty list is simply the first element of the list.

1. What is the `car` of the list `( apple peach cherry )`?

2. What is the `car` of the list `( ( lisp 1959 ) ( prolog 1971 ) ( haskell 1990 ) )`?

The same questions, but posed in the context of the read-eval-print loop:

```
> ( car '( apple peach cherry ) )
'apple
> ( car '( ( lisp 1959 ) ( prolog 1971 ) ( haskell 1990 ) ) )
'(lisp 1959)
>
```

Recall that the cdr of a nonempty list is the list with the first element removed.

1. What is the `cdr` of the list `( apple peach cherry )`?
2. What is the `cdr` of the list `( ( lisp 1959 ) ( prolog 1971 ) ( haskell 1990 ) )`?

The same questions, but posed in the context of the read-eval-print loop:

```
> ( cdr '( apple peach cherry ) )
'(peach cherry)
> ( cdr '( ( lisp 1959 ) ( prolog 1971 ) ( haskell 1990 ) ) )
'((prolog 1971) (haskell 1990))
>
```

The cons of an S-expression and a list is the list whose car is the given S-expression and cdr is the given list.

1. What is the `cons` of the symbol `apple` and the list `( peach cherry )`?
2. What is the `cons` of the list `( lisp 1959 )` and the list `( ( prolog 1971 ) ( haskell 1990 ) )`?

The same questions, but posed in the context of the read-eval-print loop:

```
> ( cons 'apple '( peach cherry ) )
'(apple peach cherry)
> ( cons '( lisp 1959 ) '( ( prolog 1971 ) ( haskell 1990 ) ) )
'((lisp 1959) (prolog 1971) (haskell 1990))
>
```

Recap: What have we done so far? After saying something about `quote` and `eval`, we defined and illustrated the application of the three most iconic functions in Lisp.

Eq and Atom

The `eq` function is used to test for the equality of symbolic atoms. Symbolic atoms are a bit like immutable character strings, except that they have some "special" powers: (1) they can be compared for equality with great speed (a result of how they are stored in a hash table), and (2) they can be used quite flexibly in the service of knowledge representation.

In Racket, predicate names tend to end with a question mark, by convention. Thus, in Racket, `eq` is "spelled" `eq?`.

```
> ( eq? 'a 'b )
#f
> ( eq? 'a 'a )
#t
```

In Racket, the Boolean values are represented by `#f` for "false" and `#t`, most notably, for "true". The particular representation of these values in Lisp depends on dialect. In Common Lisp, for example, "false" is represented as `nil`, and "true" is represented, most notably, as `t`.

There is no `atom` (or `atom?`) predicate in Racket. This may be due to the fact that the concept of "atom" tends to be dialect dependent. For completeness in this quasi-historical introduction to Lisp, we can cook up a Racket predicate, using a Boolean operators and another built in predicate, that will pass as an atom checker, provided we don't look at it too closely.

```
> ( define (atom? x) ( not ( pair? x ) ) )
> ( atom? 'a )
#t
> ( atom? '(a b c ) )
#f
> ( atom? 4 )
#t
> ( atom? '( a . b ) )
#f
```

Recap: What have we done so far? After saying something about `quote` and `eval`, and the "big three" list processing functions, we talked about `eq` and `atom`. In passing, it may be worth mentioning that both `eq` and `atom` are featured in McCarthy's definition of `eval`.

Lambda

Lisp is an implementation of the Lambda calculus, as is reflected in: (1) anonymous functions that channel Church's lambda expressions, and (2) applications of those functions. In Lisp, anonymous functions take the form:

```
<function> ::= ( lambda <lambda-list> <body> )
<lambda-list> ::= ( <dbd-id-list> )
<body> ::= <dbd-form-list>
<dbd-id-list> ::= <empty> | <id> <dbd-id-list>
<dbd-form-list> ::= <empty> | <form> <dbd-form-list>
```

Thus, for example, each of the following would be a "lambda function":

1. `( lambda (x) ( * x x ) )`
2. `( lambda ( x y ) ( cons x ( cons x ( cons y ( cons y '() ) ) ) ) )`
3. `( lambda ( a b c ) ( define s ( / ( + a b c ) 2.0 ) ) ( * s ( - s a ) ( - s b ) ( - s c ) ) )`

Lambda functions can be applied in the same way that named functions can be applied. In the following interaction, a function with one parameter is applied twice, a function with two parameters is applied twice, and a function with three parameters is applied one time.

```
> ( ( lambda (x) ( * x x ) ) 5 )
25
> ( ( lambda (x) ( * x x ) ) 9 )
81
> ( ( lambda ( x y ) ( cons x ( cons x ( cons y ( cons y '() ) ) ) ) ) 1 2 )
'(1 1 2 2)
> ( ( lambda ( x y ) ( cons x ( cons x ( cons y ( cons y '() ) ) ) ) ) 'hey 'now )
'(hey hey now now)
> ( ( lambda ( a b c )
      ( define s ( / ( + a b c ) 2.0 ) )
      ( * s ( - s a ) ( - s b ) ( - s c ) )
    )
    3 4 5
  )
36.0
```

Recap: We have now considered eight of the historic Lisp functions, including the function which ties Lisp directly to the Lambda calculus. In due time we will talk about how lambda of functions tend to be used in Lisp. Moreover, if time permits, we will talk just a little bit about the Lambda calculus.

Define

John McCarthy, in the "Lisp 1.5 Programmer's Manual", refers to `define` as a pseudofunction, because it is used primarily to alter the state of the system, and only secondarily for its value. The `define` function associates a name with a value, possibly a function.

The following definitions were placed in the Definitions area of DrRacket and then loaded into the Interactions area by clicking on the `Run` icon.

```
( define lisp-born 1959 )

( define favorite-pies '( cherry peach apple ) )

( define square ( lambda ( x ) ( * x x ) ) )

( define seeing-double
  ( lambda ( x y ) ( cons x ( cons x ( cons y ( cons y '() ) ) ) ) )
)
```

Here you see some interactions, in which forms that reference the defined entities are presented to Racket in the Interactions area.

```
> lisp-born
1959
> favorite-pies
'(cherry peach apple)
> ( square 5 )
25
> ( square 11 )
121
> ( seeing-double 'meow 'woof )
```

```
'(meow meow woof woof)
> ( seeing-double 'oh 'no )
'(oh oh no no)
>
```

Although functions can be defined in the manner illustrated above, they are usually defined with a syntax that avoids explicit mention of the lambda function.

```
( define ( square x ) ( * x x ) )

( define ( seeing-double x y )
  ( cons x ( cons x ( cons y ( cons y '() ) ) ) )
)
```

This variant of definition is just "sugar", as they say, for the more basic form.

As has been mentioned in passing, a **form** is an s-expression that can be evaluated by `eval`. A point worth emphasizeing is this: the **value of a function** is the value of the last form in the list of forms. For example, the value of the following function is the value of `the-area` (the last form in the body of the function definition):

```
( define ( area-of-circle diameter )
  ( define radius ( / diameter 2 ) )
  ( define radius-squared ( square radius ) )
  ( define the-area ( * pi radius-squared ) )
  the-area
)
```

And the application:

```
> ( area-of-circle 20 )
314.1592653589793
>
```

Quick check: Can you rewrite the following function definition in a manner that explicitly uses a lambda function?

```
( define ( average a b ) ( / ( + a b ) 2.0 ) )
```

If you should need a hint, take a look at the two definitions of the `square` function that were just presented. How might you verify that you got the translation correct?

Recap: We have now considered nine of the historic Lisp functions, including the important "binding" function which can be used to establish variable bindings or function bindings.

Cond

Lisp introduced the conditional expression. Soon after inventing Lisp, John McCarthy successfully lobbied for the inclusion of a conditional construct in ALGOL at a meeting in Paris of collaborators on the design of the language.

A fairly readable, simplified BNF grammar for the `cond` form was presented in Lesson 3. Even without reference to that, the basic form of the conditional construct can probably be inferred from the three example definitions provided in the following Definitions area text. I took the third one from the Racket guide (with just a bit of syntactic adjustment), simply because I thought it might add a little bit of fun to this presentation.

```
( define ( rgb color-name )
  ( cond
    ( ( eq? color-name 'red )
      '( 255 0 0 )
    )
    ( ( eq? color-name 'green )
      '( 0 255 0 )
    )
    ( ( eq? color-name 'blue )
      '( 0 0 255 )
    )
    ( ( eq? color-name 'purple )
      '( 106 13 173 )
    )
    ( ( eq? color-name 'yellow )
      '( 255 255 0 )
    )
    ( else
      'unknown-color-name
    )
  )
)

( define ( determine operator operand )
  ( cond
    ( ( eq? operator 'difference )
      ( define maximum ( max ( car operand ) ( cadr operand ) ( caddr operand ) ) )
      ( define minimum ( min ( car operand ) ( cadr operand ) ( caddr operand ) ) )
      ( - maximum minimum )
    )
    ( ( eq? operator 'average )
      ( define sum ( + ( car operand ) ( cadr operand ) ( caddr operand ) ) )
      ( / sum ( length operand ) )
    )
  )
)

( define ( got-milk? list )
  ( cond
    ( ( null? list ) #f )
    ( ( eq? 'milk ( car list ) ) #t )
    ( else ( got-milk? ( cdr list ) ) )
  )
)
```

Some interactions illustrating application of the functions:

```
> ( rgb 'blue )
'(0 0 255)
> ( rgb 'yellow )
'(255 255 0)
> ( rgb 'purple )
'(106 13 173)
> ( rgb 'orange )
'unknown-color-name
```

```
> ( determine 'difference '( 11 100 55 ) )
89
> ( determine 'difference '( 5 20 -1 ) )
21
> ( determine 'average '( 1 2 9 ) )
4
> ( determine 'average '( 9 5 22 ) )
12
> ( got-milk? '( coffee ) )
#f
> ( got-milk? '( coffee with cream ) )
#f
> ( got-milk? '( coffee with milk ) )
#t
>
```

The `cond` construct is a function in Lisp, and so it always returns a value. If none of the conditions "fire", then a specially designated dialect dependent value will be returned (e.g. `nil` or <`void`>). Otherwise, the value returned will be determined by the first case whose opening form evaluates to "true", by which we mean anything other than the value which represents "false". Once a case is selected, the forms of the case which follow the firing form will be evaluated in turn. The value of the `cond` will be the value of the last form to be evaluated within the selected case.

In Racket, if the conditional form of a case is `else`, as it was in the `rgb` function, it must be the last case of the `cond`.

Recap: That is it! All ten functions of historical Lisp have been reviewed. Daniel P Friedman, in the first edition of "The Little Lisper" (something of a classic), refers to Lisp's essential conditional form as "the mighty cond". I recall that, in part, because it gives me an opportunity to mention that the lead Racketeer, Matthias Felleisen, collaborated with Friedman on successive editions of "The Little Lisper" and variants of that book.