Lesson #6: Basic Lisp Programming

What's It All About?

This is the first of three lessons that feature list processing in Lisp. This lesson focusses mainly on referencers and constructors. The second list processing lesson will focus on recursive list processing. The third list processing lesson will focus on higher order functions.

The Icons of List Processing

The functions `car`, `cdr` and `cons` are defined, illustrated, and classified. In the process, your are invited to do a little mental exercise.

Definitions

Three functions are more closely, intimately, definitionally associated with Lisp than are any other functions. These are the list processing functions `CAR`, `CDR` and `CONS`.

- The `CAR` of a nonempty list is a reference to the first element of the list.
- The `CDR` of a nonempty list is a reference to the list consisting of everything but the first element of the list.
- The `CONS` of an object O and a list L is the list whose car is O and whose cdr is L.

Abstract Session featuring `CAR`, `CDR` and `CONS`

1. What is the `CAR` of `(RED GREEN BLUE)`?

2. What is the `CDR` of `(RED GREEN BLUE)`?

3. What is the `CAR` of `((1 3 5) SEVEN NINE)`?

4. What is the `CDR` of `((1 3 5) SEVEN NINE)`?

5. What is the `CAR` of `("Desde El Alma")`?

6. What is the `CDR` of `("Desde El Alma")`?

7. What is the `CONS` of `ESPRESSO` and `(LATTE CAPPUCCINO)`?

8. What is the `CONS` of `(A B C)` and `(1 2 3)`?

9. What is the `CONS` of `SYMBOL` and `()`?

The Icons of List Processing

Racket Session featuring `CAR`, `CDR` and `CONS`

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( car '( red green blue ) )
'red
> ( cdr '( red green blue ) )
'(green blue)
> ( car '( ( 1 3 5 ) seven nine ) )
'(1 3 5)
> ( cdr '( ( 1 3 5 ) seven nine ) )
'(seven nine)
> ( car '( "Desde El Alma" ) )
"Desde El Alma"
> ( cdr '( "Desde El Alma" ) )
'()
> ( cons 'ESPRESSO '( LATTE CAPPUCCINO ) )
'(ESPRESSO LATTE CAPPUCCINO)
> ( cons '( a b c ) '( 1 2 3) )
'((a b c) 1 2 3)
> ( cons 'SYMBOL '() )
'(SYMBOL)
>
```

Referencers and Constructors

A **referencer** is a form which returns a reference to a part of a given structure. A **constructor** is a form which returns a reference to a structure constructed from some number of lisp objects. Can you classify each of the three icons of list processing according to whether it is a referencer or a constructor?

Motivation for Additional Lisp Processing Functions

Athough the icons of list processing provide all the list referencing/constructing functionality that you need in a theoretical sense, it would be a nightmare to actually program in Lisp using just the three icons. A few exercises should give you a feel for just why this is so!

Referencing a list element

Suppose that the symbol `L` is bound to a list of length 5. Using only the icons of list processing (not necessarily all of them), how would you write a Lisp form to compute (reference) the fourth element of the list. Thus, for example:

If `L` $\longrightarrow$ `( ANT BAT CAT DOG EEL )`, then the value of the form will be `DOG`.

Invitation to think a thought: Referencing a list element "from scratch"

```
> ( define animals '(ant bat cat dog eel) )
> ( define questions '(who what when where why) )
> animals
'(ant bat cat dog eel)
> questions
'(who what when where why)
> <<how do you reference the fourth element of animals?>>
'dog
> <<how do you reference the fourth element of questions?>>
'where
```

Lisp Session: Referencing a list element "from scratch"

```
> ( define animals '(ant bat cat dog eel) )
> ( define questions '(who what when where why) )
> animals
'(ant bat cat dog eel)
> questions
'(who what when where why)
> ( car ( cdr ( cdr ( cdr animals ) ) ) )
'dog
> ( car ( cdr ( cdr ( cdr questions ) ) ) )
'where
```

Lisp Session: `list-ref` (`nth` in Common Lisp)

```
> ( define animals '(ant bat cat dog eel) )
> ( define questions '(who what when where why) )
```

```
> animals
'(ant bat cat dog eel)
> questions
'(who what when where why)
> ( list-ref animals 3 )
'dog
> ( list-ref questions 3 )
'where
>
```

Creating a list

Suppose that the symbol `a` is bound to a list object. Same for the symbol `b`. Same for `c`. Using only the icons of list processing (not necessarily all of them), write a Lisp form to compute the list of length three whose first element is the value of `a`, whose second element is the value of `b`, and whose third element is the value of `c`. Thus, for example:

If `a` $\longrightarrow$ `apple` and `b` $\longrightarrow$ `peach` and `c` $\longrightarrow$ `cherry`, then the value of the form will be `(apple peach cherry)`.

Invitation to think a thought: Creating a list "from scratch"

```
> ( define a ( random 10 ) )
> ( define b ( random 10 ) )
> ( define c ( random 10 ) )
> <<how do you create the list of elements to which a, b, and c are bound?>>
'(1 9 4)
```

Lisp Session: Creating a list "from scratch"

```
> ( define a ( random 10 ) )
> ( define b ( random 10 ) )
> ( define c ( random 10 ) )
> ( cons a ( cons b ( cons c '() ) ) )
'(8 0 6)
```

Lisp Session: Creating a list using `list`

```
> ( define a ( random 10 ) )
> ( define b ( random 10 ) )
> ( define c ( random 10 ) )
> ( list a b c )
'(4 7 7)
```

Appending one list to another list

Suppose that the symbol `x` is bound to a list of length two. Same for the symbol `y`. Using only the icons of list processing (not necessarily all of them), write a Lisp form to compute the list of length four consisting of the concatenation of `x` with `y`. Thus, for example:

If `x` $\longrightarrow$ `(one fish)` and `y` $\longrightarrow$ `(two fish)` then the value of the form will be `(one fish two fish)`.

Invitation to think a thought: Appending two lists "from scratch"

```
> ( define x '(one fish) )
> ( define y '(two fish) )
> x
'(one fish)
> y
'(two fish)
> <<how do you append the two lists, x and y?>>
'(one fish two fish)
```

Lisp Session: Appending two lists "from scratch"

```
> ( define x '(one fish) )
> ( define y '(two fish) )
> x
'(one fish)
> y
'(two fish)
> ( cons ( car x ) ( cons ( car ( cdr x ) ) y ) )
'(one fish two fish)
```

Lisp Session: Appending two lists using `append`

```
> ( define x '(one fish) )
> ( define y '(two fish) )
> x
'(one fish)
> y
'(two fish)
> ( append x y )
'(one fish two fish)
```

Note on `list-ref` and `list` and `append`

You should like `list-ref` and `list` and `append`! They are very easy to use! They are wonderfully general!

Redacted Racket Session Featuring Referencers and Constructors

Playing Lisp can help you to think in Lisp. Perhaps you would like to write down, on paper or in the wetware of you mind, the responses that Racket would make to the given forms in the following redacted Racket session.

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define languages '(racket prolog haskell rust) )
> languages


> 'languages


> ( quote languages )


> ( car languages )


> ( cdr languages )


> ( car ( cdr languages ) )


> ( cdr ( cdr languages ) )


> ( cadr languages )


> ( cddr languages )


> ( first languages )


> ( second languages )
```

```
> ( third languages )


> ( list-ref languages 2 )


> ( define numbers '(1 2 3) )
> ( define letters '(a b c) )
> ( cons numbers letters )


> ( list numbers letters )


> ( append numbers letters )


> ( define animals '(ant bat cat dot eel) )
> ( car ( cdr ( cdr ( cdr animals ) ) ) )


> ( cadddr animals )


> ( list-ref animals 3 )


> ( define a 'apple )
> ( define b 'peach )
> ( define c 'cherry )
> ( cons a ( cons b ( cons c '() ) ) )


> ( list a b c )


> ( define x '(one fish) )
> ( define y '(two fish) )
> ( cons ( car x ) ( cons ( car ( cdr x ) ) y ) )


> ( append x y )


>
```

Example Program: Sampler

The simple little program presented selects an element at random from a given list. The list is provided by means of the `read` function, which will read any S-expression, including a list.

Source

```
( define ( sampler )
  ( display "(?):  " )
  ( define the-list ( read ) )
  ( define the-element
    ( list-ref the-list ( random ( length the-list ) ) )
  )
  ( display the-element ) ( display "\n" )
  ( sampler )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( sampler )
(?):  ( red orange yellow green blue indigo violet )
indigo
(?):  ( red orange yellow green blue indigo violet )
blue
(?):  ( red orange yellow green blue indigo violet )
orange
(?):  ( red orange yellow green blue indigo violet )
green
(?):  ( red orange yellow green blue indigo violet )
yellow
(?):  ( red orange yellow green blue indigo violet )
orange
(?):  ( aet ate eat eta tae tea )
aet
(?):  ( aet ate eat eta tae tea )
aet
(?):  ( aet ate eat eta tae tea )
tae
(?):  ( aet ate eat eta tae tea )
eta
(?):  ( aet ate eat eta tae tea )
tae
(?):  ( aet ate eat eta tae tea )
aet
(?):  ( 0 1 2 3 4 5 6 7 8 9 )
7
(?):  ( 0 1 2 3 4 5 6 7 8 9 )
3
(?):  ( 0 1 2 3 4 5 6 7 8 9 )
4
(?):  ( 0 1 2 3 4 5 6 7 8 9 )
4
```

```
(?):  ( 0 1 2 3 4 5 6 7 8 9 )
9
(?):  ( 0 1 2 3 4 5 6 7 8 9 )
6
(?):  . . user break
>
```

Basic List Processing Example - Playing Cards

Suppose that playing cards are represented as lists of length two, the first element of which represents a rank, the second component of which represents a suit. Furthermore, suppose that:

- Ranks are represented by the atoms `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `X`, `J`, `Q`, `K`, and `A`.
- Suits are represented by the atoms `C`, `D`, `H`, and `S`.

Thus, for example, the two of spades would be represented by `(2 S)`, and the jack of spades would be represented by `(J S)`.

Note that, unlike most Lisps, Racket symbols are case sensitive. Thus, for example, `a` is a different symbol from `A`.

Note also basic Boolean operators and relational equality operators are presented in passing, since you already have knowledge these operators from your previous programming experience, syntactic differences notwithstanding.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define c1 '( 7 C ) )
> ( define c2 '( Q H ) )
> c1
'(7 C)
> c2
'(Q H)
> ( rank c1 )
7
> ( suit c1 )
'C
> ( rank c2 )
'Q
> ( suit c2 )
'H
> ( red? c1 )
#f
> ( red? c2 )
#t
> ( black? c1 )
#t
> ( black? c2 )
```

```
#f
> ( aces? '( A C ) '( A S ) )
#t
> ( aces? '( K S ) '( A C ) )
#f
> ( ranks 4 )
'((4 C) (4 D) (4 H) (4 S))
> ( ranks 'K )
'((K C) (K D) (K H) (K S))
> ( length ( deck ) )
52
> ( display ( deck ) )
((2 C) (2 D) (2 H) (2 S) (3 C) (3 D) (3 H) (3 S) (4 C) (4 D) (4 H) (4 S) (5 C) (5 D) (5 H)
(5 S) (6 C) (6 D) (6 H) (6 S) (7 C) (7 D) (7 H) (7 S) (8 C) (8 D) (8 H) (8 S) (9 C) (9 D)
(9 H) (9 S) (X C) (X D) (X H) (X S) (J C) (J D) (J H) (J S) (Q C) (Q D) (Q H) (Q S) (K C)
(K D) (K H) (K S) (A C) (A D) (A H) (A S))
> ( pick-a-card )
'(Q C)
> ( pick-a-card )
'(2 D)
> ( pick-a-card )
'(9 C)
> ( pick-a-card )
'(X D)
> ( pick-a-card )
'(3 H)
> ( pick-a-card )
'(2 H)
>
```

Source

```
#lang racket

( define ( ranks rank )
  ( list
    ( list rank 'C )
    ( list rank 'D )
    ( list rank 'H )
    ( list rank 'S )
  )
)

( define ( deck )
  ( append
    ( ranks 2 )
    ( ranks 3 )
    ( ranks 4 )
    ( ranks 5 )
    ( ranks 6 )
    ( ranks 7 )
```

```
    ( ranks 8 )
    ( ranks 9 )
    ( ranks 'X )
    ( ranks 'J )
    ( ranks 'Q )
    ( ranks 'K )
    ( ranks 'A )
  )
)

( define ( pick-a-card )
  ( define cards ( deck ) )
  ( list-ref cards ( random ( length cards ) ) )
)

( define ( show card )
  ( display ( rank card ) )
  ( display ( suit card ) )
)

( define ( rank card )
  ( car card )
)

( define ( suit card )
  ( cadr card )
)

( define ( red? card )
  ( or
    ( equal? ( suit card ) 'D )
    ( equal? ( suit card ) 'H )
  )
)

( define ( black? card )
  ( not ( red? card ) )
)

( define ( aces? card1 card2 )
  ( and
    ( equal? ( rank card1 ) 'A )
    ( equal? ( rank card2 ) 'A )
  )
)
```