Lesson #7: Recursive List Processing

What's It All About?

Some functions are specified, and presented in pseudocode, which afford opportunities for straightforward recursive list processing implementation. Also, the tracing mechanism in Lisp is introduced.

Rac

Recall `car`? The iconic Lisp function that returns the first element of a nonempty list. We will write the function `rac`, a function featured in "The Little Lisper", that returns the last element of a nonempty list.

Furthermore, we will illustrate the behavior of `rac` by **tracing** its execution on a number of lists. For the record, you can **trace** a function by requiring `racket/trace` and then adding the form `( trace <name-of-function> )` to the file in which the function is defined.

Pseudocode

```
to compute the last element of nonempty list l do
   if l is a singleton list then
      return the first element of l
   else
      return the last element of the "rest" (cdr) of l
   end
end
```

Source

```
#lang racket

( require racket/trace )

( define ( rac l )
  ( cond
    ( ( = ( length l ) 1 ) ( car l ) )
    ( else ( rac ( cdr l ) ) )
  )
)

( trace rac )
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( rac '( one ) )
>(rac '(one))
<'one
'one
> ( rac '( red yellow blue green ) )
>(rac '(red yellow blue green))
>(rac '(yellow blue green))
>(rac '(blue green))
>(rac '(green))
<'green
'green
>
> ( rac '( a b c d e f g ) )
>(rac '(a b c d e f g))
>(rac '(b c d e f g))
>(rac '(c d e f g))
>(rac '(d e f g))
>(rac '(e f g))
>(rac '(f g))
>(rac '(g))
<'g
'g
>
```

Rdc

Recall `cdr`? The iconic Lisp function that returns all of the elements of a nonempty list except for the first element. We will write the function `rac`, a function featured in "The Little Lisper", that returns a list containing all of the elements of a given nonempty list except for the last element.

Furthermore, we will illustrate the behavior of `rdc` by **tracing** its execution on a number of lists.

Pseudocode

```
to compute all but the last element in a nonempty list l do
   if l is a singleton list then
      return the empty list
   else
      return the list containing the first element of l followed by
      all but the last element in the "rest" of l
   end
end
```

Source

```
( define ( rdc l )
  ( cond
    ( ( empty? ( cdr l ) ) '() )
    ( else ( cons ( car l ) ( rdc ( cdr l ) ) ) )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( rdc ' ( one ) )
>(rdc '(one))
<'()
'()
> ( rdc '( red yellow blue green ) )
>(rdc '(red yellow blue green))
> (rdc '(yellow blue green))
> >(rdc '(blue green))
> > (rdc '(green))
< < '()
< <'(blue)
< '(yellow blue)
<'(red yellow blue)
'(red yellow blue)
>
> ( rdc '( a b c d e f g ) )
>(rdc '(a b c d e f g))
> (rdc '(b c d e f g))
> >(rdc '(c d e f g))
> > (rdc '(d e f g))
> > >(rdc '(e f g))
> > > (rdc '(f g))
> > > >(rdc '(g))
< < < <'()
< < < '(f)
< < <'(e f)
< < '(d e f)
< <'(c d e f)
< '(b c d e f)
<'(a b c d e f)
'(a b c d e f)
>
```

Palindrome?

Recall the concept of a **palindrome**? It is a something that reads the same forwards and backwards. Like: (a b b a), like (wonderful), like (), like (1 two 3 four 5 four 3 two 1). We will write a function called `palindrome?` that

returns true if a given list is a palindrome, false if it is not.

Furthermore, we will illustrate the behavior of `palindrome?` by **tracing** its execution on a number of lists.

Pseudocode

```
to determine if list x is palindromic do
   if x is empty then
      return true
   else if x is a singleton then
      return true
   else if the first and last elements of x are equal then
      return the result of asking if the list without
      the first and last elements is a palindrome
   else
      return false
   end
end
```

Source

```
( define ( palindrome? l )
  ( cond
    ( ( empty? l ) #t )
    ( ( empty? ( cdr l ) ) #t )
    ( ( equal? ( car l ) ( rac l ) )
      ( palindrome? ( cdr ( rdc l ) ) )
    )
    ( else
      #f
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( palindrome? '() )
>(palindrome? '())
<#t
#t
> ( palindrome? '( pumpkin ) )
>(palindrome? '(pumpkin))
<#t
#t
> ( palindrome? '( hey hey ) )
>(palindrome? '(hey hey))
<#t
#t
> ( palindrome? '( this that ) )
```

```
>(palindrome? '(this that))
<#f
#f
> ( palindrome? '( 1 two 3 four 5 four 3 two 1 ) )
>(palindrome? '(1 two 3 four 5 four 3 two 1))
>(palindrome? '(two 3 four 5 four 3 two))
>(palindrome? '(3 four 5 four 3))
>(palindrome? '(four 5 four))
>(palindrome? '(5))
<#t
#t
> ( palindrome? '( apple peach banana pumpkin apple ) )
>(palindrome? '(apple peach banana pumpkin apple))
>(palindrome? '(peach banana pumpkin))
<#f
#f
>
```

Snoc

Recall `cons`? The iconic Lisp function that constructs a list by placing a list object at the beginning of a list. We will write the function `snoc`, a function featured in "The Little Lisper", that returns a list constructed by adding a lisp object to the end of a list.

Furthermore, we will illustrate the behavior of `snoc` by **tracing** its execution on a number of lists.

Pseudocode

```
to compute the list with object o at the end of list l do
   if l is empty then
      return the list containing o
   else
      return the list consisting of first element of l added to the beginning of
      the list with object o added to the end of the "rest" of list l
   end
end
```

Source

```
( define ( snoc obj lst )
  ( cond
    ( ( empty? lst )
      ( list obj )
    )
    ( else
      ( cons ( car lst ) ( snoc obj ( cdr lst ) ) )
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( snoc 'x '() )
>(snoc 'x '())
<'(x)
'(x)
> ( snoc 'x '( a b c d e ) )
>(snoc 'x '(a b c d e))
> (snoc 'x '(b c d e))
> >(snoc 'x '(c d e))
> > (snoc 'x '(d e))
> > >(snoc 'x '(e))
> > > (snoc 'x '())
< < < '(x)
< < <'(e x)
< < '(d e x)
< <'(c d e x)
< '(b c d e x)
<'(a b c d e x)
'(a b c d e x)
>
```

Sum

We will write a function called `sum` that returns the sum of the elements in a list of numbers. Furthermore, we will illustrate the behavior of `palindrome?` by **tracing** its execution on a number of lists.

Pseudocode

```
to determine the sum of the elements in a list of numbers do
   if the list is empty then
      return 0
   else
      return the first element of the list plus
      the sum of the elements in the "rest" of the list
   end
end
```

Source

```
( define ( sum nl )
  ( cond
    ( ( empty? nl ) 0 )
    ( else
      ( + ( car nl ) ( sum ( cdr nl ) ) )
    )
```

```
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( sum '() )
>(sum '())
<0
0
> ( sum '( 1 2 3 4 5 6 7 8 9 10 ) )
>(sum '(1 2 3 4 5 6 7 8 9 10))
> (sum '(2 3 4 5 6 7 8 9 10))
> >(sum '(3 4 5 6 7 8 9 10))
> > (sum '(4 5 6 7 8 9 10))
> > >(sum '(5 6 7 8 9 10))
> > > (sum '(6 7 8 9 10))
> > > >(sum '(7 8 9 10))
> > > > (sum '(8 9 10))
> > > > >(sum '(9 10))
> > > > > (sum '(10))
> > > >[10] (sum '())
< < < <[10] 0
< < < < < 10
< < < < <19
< < < < 27
< < < <34
< < < 40
< < <45
< < 49
< <52
< 54
<55
55
>
```

Iota

We will write a function called `iota` that takes a positive integer n and returns a list of the numbers from 1 to n, and then illustrate the behavior of the function by tracing it.

Pseudocode

```
to compute ( iota n ) do
   if ( n = 1 ) then
      return ( 1 )
   else
      return ( iota ( n - 1 ) ) with n tacked onto the end of the list
```

```
   end
end
```

Source

```
( define ( iota n )
  ( cond
    ( ( = n 1 ) '( 1 ) )
    ( else
      ( snoc n ( iota ( - n 1 ) ) )
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( iota 1 )
>(iota 1)
<'(1)
'(1)
> ( iota 2 )
>(iota 2)
> (iota 1)
< '(1)
<'(1 2)
'(1 2)
> ( iota 7 )
>(iota 7)
> (iota 6)
> >(iota 5)
> > (iota 4)
> > >(iota 3)
> > > (iota 2)
> > > >(iota 1)
< < < <'(1)
< < < '(1 2)
< < <'(1 2 3)
< < '(1 2 3 4)
< <'(1 2 3 4 5)
< '(1 2 3 4 5 6)
<'(1 2 3 4 5 6 7)
'(1 2 3 4 5 6 7)
>
```

Take-from

We will write a function called `take-from` that takes two parameters, an object `o` and a list `l`, which returns a list containing the elements of list `l` with all occurrences of the object `o` deleted.

Pseudocode

```
to compute take-from o l do
   if l is empty then
      return the empty list
   else if the first element of l equals the object o then
      return the result of applying take-from to o and the "rest" of l
   else
      return the first element of l together with the result of applying
      take-from to o and the "rest" of l
   end
end
```

Source

```
( define ( take-from o l )
  ( cond
    ( ( empty? l )
      '()
    )
    ( ( equal? ( car l ) o )
      ( take-from o ( cdr l ) )
    )
    ( else
      ( cons ( car l ) ( take-from o ( cdr l ) ) )
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( take-from 'b '( a b c b d b e ) )
>(take-from 'b '(a b c b d b e))
> (take-from 'b '(b c b d b e))
> (take-from 'b '(c b d b e))
> >(take-from 'b '(b d b e))
> >(take-from 'b '(d b e))
> > (take-from 'b '(b e))
> > (take-from 'b '(e))
> > >(take-from 'b '())
< < <'()
< < '(e)
```

```
< <’(d e)
< ’(c d e)
<’(a c d e)
’(a c d e)
> ( take-from ’what ’() )
>(take-from ’what ’())
<’()
’()
> ( take-from ’x ’( x y z ) )
>(take-from ’x ’(x y z))
>(take-from ’x ’(y z))
> (take-from ’x ’(z))
> >(take-from ’x ’())
< <’()
< ’(z)
<’(y z)
’(y z)
>
```

Take-one-from

We will write a function called `take-one-from` that takes two parameters, an object `o` and a list `l`, which returns a list containing the elements of list `l` with just one occurrence of the object `o` deleted, provide it has such an occurrence.

Pseudocode

```
to compute take-one-from o l do
   if l is empty then
      return the empty list
   else if the first element of l equals object o then
      return the "rest" of l
   else
      return the first element of l together with the result of applying
      take-one-from to o and the "rest" of l
   end
end
```

Source

```
( define ( take-one-from o l )
  ( cond
    ( ( empty? l )
      ’()
    )
    ( ( equal? ( car l ) o )
      ( cdr l )
    )
    ( else
```

```
      ( cons ( car l ) ( take-one-from o ( cdr l ) ) )
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( take-one-from 'b '( a b a c a d a e ) )
>(take-one-from 'b '(a b a c a d a e))
> (take-one-from 'b '(b a c a d a e))
< '(a c a d a e)
<'(a a c a d a e)
'(a a c a d a e)
> ( take-one-from 'x '( x y z ) )
>(take-one-from 'x '(x y z))
<'(y z)
'(y z)
> ( take-one-from 'x '( a b c ) )
>(take-one-from 'x '(a b c))
> (take-one-from 'x '(b c))
> >(take-one-from 'x '(c))
> > (take-one-from 'x '())
< < '()
< <'(c)
< '(b c)
<'(a b c)
'(a b c)
> ( take-one-from 'z '() )
>(take-one-from 'z '())
<'()
'()
>
```

Random-permutation

We will write a function called `random-permutation` that takes a list as its sole parameter and returns a random permutation of the list. Also, we will trace it a number of times.

Pseudocode

```
to create a random permutation of list x do
   if x is empty then
      return the empty list
   else
      let element be a randomly selected element of x
      let remainder be the list with one occurrence of the element removed
      return the list consisting of element followed by a random permutation of remainder
```

```
   end
end
```

Source

```
( define ( random-permutation x )
  ( cond
    ( ( empty? x )
      '()
    )
    ( else
      ( define element
        ( list-ref x ( random ( length x ) ) )
      )
      ( define remainder
        ( take-one-from element x )
      )
      ( cons element ( random-permutation remainder ) )
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( random-permutation '( c o m p u t e r ) )
>(random-permutation '(c o m p u t e r))
> (random-permutation '(c o m p u e r))
> >(random-permutation '(o m p u e r))
> > (random-permutation '(o m p u r))
> > >(random-permutation '(o m p u))
> > > (random-permutation '(m p u))
> > > >(random-permutation '(m u))
> > > > (random-permutation '(m))
> > > > >(random-permutation '())
< < < < <'()
< < < < '(m)
< < < <'(u m)
< < < '(p u m)
< < <'(o p u m)
< < '(r o p u m)
< <'(e r o p u m)
< '(c e r o p u m)
<'(t c e r o p u m)
'(t c e r o p u m)
> ( random-permutation '( c o f f e e ) )
>(random-permutation '(c o f f e e))
> (random-permutation '(c o f f e))
> >(random-permutation '(o f f e))
> > (random-permutation '(o f f))
> > >(random-permutation '(o f))
> > > (random-permutation '(f))
```

```
> > > >(random-permutation '())
< < < <'()
< < < '(f)
< < <'(o f)
< < '(f o f)
< <'(e f o f)
< '(c e f o f)
<'(e c e f o f)
'(e c e f o f)
>
```