Racket Programming Assignment #5: RLP and HoFs:

Cameron Francois
March 28th, 2023
CSC 344

Abstract:
Task 1 is dedicated to defining four simple list generators. Three of these require the use of recursion. One requires a classic application of higher order functions. Task 2 features programs generate number sequences by performing some interesting sorts of "counting." These programs serve to channel one of Tom Johnson's many "automantic composition" techniques. Task 3 affords you an opportunity to get acquainted with "association lists," which are a classic data structure introduced in McCarthy's original Lisp. This task also serves as a segue into Task 4, which pertains to the transformation of number sequences to musical notes represented in ABC notation. Task 5 channels Frank Stella, famous for (among other things) his nested squares. Task 6 simulates a cognitive phenomenon known as chromesthesia, the mapping of musical pitchs to colors. Task 7 simulations grapheme to color synesthesia, in which letters are mapped to colors.

Task 1 - Simple List Generators:

Task 1a - iota:

Code:

```
#lang racket

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

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

Demo:

```
> ( iota 10 )
'(1 2 3 4 5 6 7 8 9 10)
> ( iota 1 )
'(1)
> ( iota 12 )
'(1 2 3 4 5 6 7 8 9 10 11 12)
>
```

Task 1b - Same:

Code:

```
( define ( same c x )
   ( cond
      ( ( = c 0 ) '() )
      ( else
        ( cons x ( same ( - c 1 ) x ) )
       )
      )
     )
```

Demo:

```
Language: racket, with debugging, memory limit: 128 MB.
> ( same 5 'five )
'(five five five five five)
> ( same 10 2 )
'(2 2 2 2 2 2 2 2 2 2)
> ( same 0 'whatever )
'()
> ( same 2 '( racket prolog haskell rust ) )
'((racket prolog haskell rust)
  (racket prolog haskell rust))
>
```

Task 1c - Alternator:

Code:

```
( define ( alternator c lst )
   ( cond
      ( ( = c 0 ) '() )
      ( else
        ( cons ( car lst ) ( alternator ( - c 1 ) ( snoc ( car lst ) ( cdr lst ) ) ) )
      )
     )
    )
```

Demo:

```
Language: racket, with debugging, memory limit: 128 MB.
> ( alternator 7 '( black white ) )
'(black white black white black white black)
> ( alternator 12 '( red yellow blue ) )
'(red yellow blue red yellow blue red yellow blue red yellow blue)
> ( alternator 9 '( 1 2 3 4 ) )
'(1 2 3 4 1 2 3 4 1)
> ( alternator 15 '( x y ) )
'(x y x y x y x y x y x y x y x)
```

Task 1d - Sequence:

Code:

```
( define ( sequence c num )
   ( cond
      ( ( = c 0 ) '() )
      ( else
        ( map ( lambda ( x ) ( * x num ) ) ( iota c ) )
      )
     )
    )
```

Demo:

```
> ( sequence 5 20 )
'(20 40 60 80 100)
> ( sequence 10 7 )
'(7 14 21 28 35 42 49 56 63 70)
> ( sequence 8 50 )
'(50 100 150 200 250 300 350 400)
>
```

Task 2 - Counting:

Task 2a - Accumulation Counting:

Code:

```
( define ( a-count lst )
   ( cond
      ( ( empty? lst ) '() )
      ( else
        ( append ( iota ( car lst ) ) ( a-count ( cdr lst ) ) )
      )
     )
    )
```

Demo:

```
> ( a-count '( 1 2 3 ) )
'(1 1 2 1 2 3)
> ( a-count '( 4 3 2 1 ) )
'(1 2 3 4 1 2 3 1 2 1)
> ( a-count '( 1 1 2 2 3 3 2 2 1 1 ) )
'(1 1 1 2 1 2 1 2 3 1 2 3 1 2 1 2 1 1)
>
```

Task 2b - Repetition Counting:

Code:

```
( define ( r-count lst )
   ( cond
      ( ( empty? lst ) '() )
        ( else
          ( append ( same ( car lst ) ( car lst ) ) ( r-count ( cdr lst ) ) )
        )
       )
      )
```

Demo:

```
> ( r-count '( 1 2 3 ) )
'(1 2 2 3 3 3)
> ( r-count '( 4 3 2 1 ) )
'(4 4 4 4 3 3 3 2 2 1)
> ( r-count '( 1 1 2 2 3 3 2 2 1 1 ) )
'(1 1 2 2 2 2 3 3 3 3 3 3 2 2 2 2 1 1)
>
```

Task 2a - Mixed Counting Demo:

Demo:

```
> ( a-count '( 1 2 3 ) )
'(1 1 2 1 2 3)
> ( r-count '( 1 2 3 ) )
'(1 2 2 3 3 3)
> ( r-count ( a-count '( 1 2 3 ) ) )
'(1 1 2 2 1 2 2 3 3 3)
> ( a-count ( r-count '( 1 2 3 ) ) )
'(1 1 2 1 2 1 2 3 1 2 3 1 2 3)
> ( a-count '( 2 2 5 3 ) )
'(1 2 1 2 1 2 3 4 5 1 2 3)
> ( r-count '( 2 2 5 3 ) )
'(2 2 2 2 5 5 5 5 5 3 3 3)
> ( r-count ( a-count '( 2 2 5 3 ) ) )
'(1 2 2 1 2 2 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 1 2 2 3 3 3)
> ( a-count ( r-count '( 2 2 5 3 ) ) )
'(1 2 1 2 1 2 1 2 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 1 2 3 1 2 3)
>
```

Task 3 - Association Lists:

Task 3a - Zip:

Code:

```
( define ( zip list1 list2 )
   ( cond
      ( ( empty? list1 ) '() )
      ( else
        ( cons ( list* ( car list1 ) ( car list2 ) ) ( zip ( cdr list1 ) ( cdr list2 ) ) )
      )
     )
    )
```

Demo:

```
Language: racket, with debugging, memory limit: 128 MB.
> ( zip '( one two three four five ) '( un deux trois quatre cinq ) )
'((one . un) (two . deux) (three . trois) (four . quatre) (five . cinq))
> ( zip '() '() )
'()
> ( zip '( this ) '( that ) )
'((this . that))
> ( zip '(one two three ) '( ( 1 ) ( 2 2 ) ( 3 3 3 ) ) )
'((one 1) (two 2 2) (three 3 3 3))
>
```

Task 3b - Assoc:

Code:

```
( define ( assoc x lst )
   ( cond
      ( ( empty? lst ) '() )
      ( ( eq? x ( caar lst ) ) ( car lst ) )
      ( else
        ( assoc x ( cdr lst ) ) )
    )
   )
```

Demo:

```
> ( define al1( zip '(one two three four ) '(un deux trois quatre ) ) ; # a-list -> zip
   )
> ( define al2 ( zip '(one two three) '( (1) (2 2) (3 3 3) ) ) ; # a-list -> zip
   )
> al1
'((one . un) (two . deux) (three . trois) (four . quatre))
> ( assoc 'two al1 )
'(two . deux)
> ( assoc 'five al1 )
'()
> al2
'((one 1) (two 2 2) (three 3 3 3))
> ( assoc 'three al2 )
'(three 3 3 3)
> ( assoc 'four al2 )
'()
>
```

Task 3c - Establishing some Association Lists:

Code:

```
( define scale-zip-CM ( zip ( iota 7 ) '( "C" "D" "E" "F" "G" "A" "B" ) ) )
( define scale-zip-short-Am ( zip ( iota 7 ) '( "A/2" "B/2" "C/2" "D/2" "E/2" "F/2" "G/2" ) ) )
( define scale-zip-short-low-Am ( zip ( iota 7 ) '( "A,/2" "B,/2" "C,/2" "D,/2" "E,/2" "F,/2" "G,/2" ) ) )
( define scale-zip-short-low-blues-Dm ( zip ( iota 7 ) '( "D,/2" "F,/2" "G,/2" "_A,/2" "A,/2" "c,/2" "d,/2" ) ) )
( define scale-zip-wholetone-C ( zip ( iota 7 ) '( "C" "D" "E" "^F" "^G" "^A" "c" ) ) )
```

Demo:

```
> scale-zip-CM
'((1 . "C") (2 . "D") (3 . "E") (4 . "F") (5 . "G") (6 . "A") (7 . "B"))
> scale-zip-short-Am
'((1 . "A/2") (2 . "B/2") (3 . "C/2") (4 . "D/2") (5 . "E/2") (6 . "F/2") (7 . "G/2"))
> scale-zip-short-low-Am
'((1 . "A,/2") (2 . "B,/2") (3 . "C,/2") (4 . "D,/2") (5 . "E,/2") (6 . "F,/2") (7 . "G,/2"))
> scale-zip-short-low-blues-Dm
'((1 . "D,/2") (2 . "F,/2") (3 . "G,/2") (4 . "_A,/2") (5 . "A,/2") (6 . "c,/2") (7 . "d,/2"))
> scale-zip-wholetone-C
'((1 . "C") (2 . "D") (3 . "E") (4 . "^F") (5 . "^G") (6 . "^A") (7 . "c"))
>
```

Task 4 - Numbers to Notes to ABC:

Task 4a - nr->note:

Code:

```
( define ( nr->note c lst )
   ( cond
      ( ( eq? c ( caar lst ) ) ( cdar lst ) )
      ( else
        ( nr->note c ( cdr lst ) ) )
    )
   )
```

Demo:

```
> ( nr->note 1 scale-zip-CM )
"C"
> ( nr->note 1 scale-zip-short-Am )
"A/2"
> ( nr->note 1 scale-zip-short-low-Am )
"A,/2"
> ( nr->note 3 scale-zip-CM )
"E"
> ( nr->note 4 scale-zip-short-Am )
"D/2"
> ( nr->note 5 scale-zip-short-low-Am )
"E,/2"
> ( nr->note 4 scale-zip-short-low-blues-Dm )
"_A,/2"
> ( nr->note 4 scale-zip-wholetone-C )
"^F"
>
```

Task 4b - nrs->notes:

Code:

```
( define ( nrs->notes list1 list2 )
   ( map ( lambda ( x ) ( nr->note x list2 ) ) list1 )
  )
```

Demo:

```
> ( nrs->notes '(3 2 3 2 1 1) scale-zip-CM )
'("E" "D" "E" "D" "C" "C")
> ( nrs->notes '(3 2 3 2 1 1) scale-zip-short-Am )
'("C/2" "B/2" "C/2" "B/2" "A/2" "A/2")
> ( nrs->notes ( iota 7 ) scale-zip-CM )
'("C" "D" "E" "F" "G" "A" "B")
> ( nrs->notes ( iota 7 ) scale-zip-short-low-Am )
'("A,/2" "B,/2" "C,/2" "D,/2" "E,/2" "F,/2" "G,/2")
> ( nrs->notes ( a-count '(4 3 2 1) ) scale-zip-CM )
'("C" "D" "E" "F" "C" "D" "E" "C" "D" "C")
> ( nrs->notes ( r-count '(4 3 2 1) ) scale-zip-CM )
'("F" "F" "F" "F" "E" "E" "E" "D" "D" "C")
> ( nrs->notes ( a-count ( r-count '(1 2 3) ) ) scale-zip-CM )
'("C" "C" "D" "C" "D" "C" "D" "E" "C" "D" "E" "C" "D" "E")
> ( nrs->notes ( r-count ( a-count '(1 2 3) ) ) scale-zip-CM )
'("C" "C" "D" "D" "C" "D" "D" "E" "E" "E")
>
```

Task 4c - nrs->abc:

Code:

```
( define ( nrs->abc list1 list2 )
   ( string-join ( nrs->notes list1 list2 ) )
  )
```

Demo:

```
> ( nrs->abc ( iota 7 ) scale-zip-CM )
"C D E F G A B"
> ( nrs->abc ( iota 7 ) scale-zip-short-Am )
"A/2 B/2 C/2 D/2 E/2 F/2 G/2"
> ( nrs->abc ( a-count '( 3 2 1 3 2 1 ) ) scale-zip-CM )
"C D E C D C C D E C D C"
> ( nrs->abc ( r-count '( 3 2 1 3 2 1 ) ) scale-zip-CM )
"E E E D D C E E E D D C"
> ( nrs->abc ( r-count ( a-count '(4 3 2 1) ) ) scale-zip-CM )
"C D D E E E F F F C D D E E E C D D C"
> ( nrs->abc ( a-count ( r-count '(4 3 2 1) ) ) scale-zip-CM )
"C D E F C D E F C D E F C D E F C D E C D E C D E C D C D C"
>
```

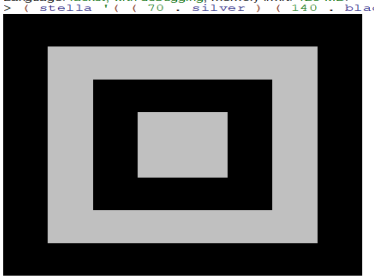
Task 5 - Stella:

Code:

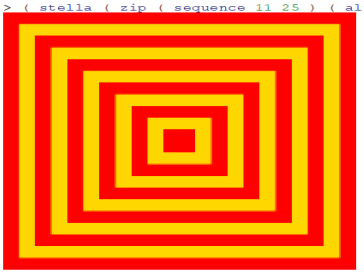
```
(require 2htdp/image )

( define ( stella lst )
   ( foldr overlay empty-image
     ( map ( lambda ( x ) ( square ( car x ) "solid" ( cdr x ) ) ) lst ) )
  )
```

Demo:

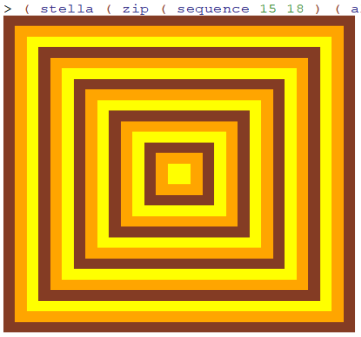
```
> ( stella '( ( 70 . silver ) ( 140 . black ) ( 210 . silver ) ( 280 . black ) ) )
```



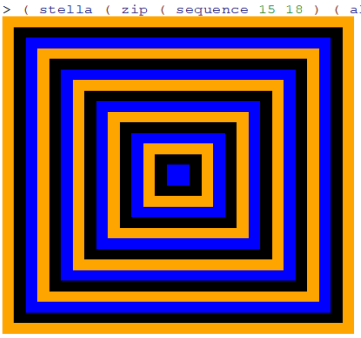
```
> ( stella ( zip ( sequence 11 25 ) ( alternator 11 '( red gold ) ) ) )
```



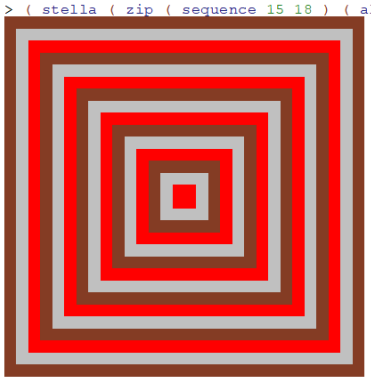
```
> ( stella ( zip ( sequence 15 18 ) ( alternator 15 '( yellow orange brown ) ) ) )
```



```
> ( stella ( zip ( sequence 15 18 ) ( alternator 15 '( blue black orange ) ) ) )
```



```
> ( stella ( zip ( sequence 15 18 ) ( alternator 15 '( red silver brown ) ) ) )
```



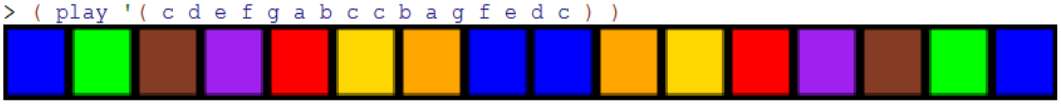
Task 6 - Chromesthetic Renderings:

Code:

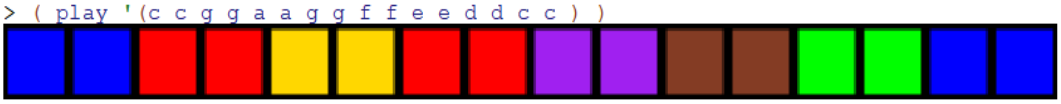
```
( define pitch-classes '( c d e f g a b ) )
( define color-names '( blue green brown purple red yellow orange ) )
( define ( box color )
   ( overlay
     ( square 30 "solid" color )
     ( square 35 "solid" "black" )
   )
  )

( define boxes
   ( list
     ( box "blue" )
     ( box "green" )
     ( box "brown" )
     ( box "purple" )
     ( box "red" )
     ( box "gold" )
     ( box "orange" )
   )
  )
( define pc-a-list ( zip pitch-classes color-names ) ) ; a-list -> zip
( define cb-a-list ( zip color-names boxes ) ) ; a-list -> zip

( define ( pc->color pc )
   ( cdr ( assoc pc pc-a-list ) )
 )

( define ( color->box color )
   ( cdr ( assoc color cb-a-list ) )
 )

( define ( play list )
   ( define lst ( zip pitch-classes boxes ) )
   ( foldr beside empty-image
    ( map ( lambda ( x ) ( cdr (assoc x lst ) ) ) list )
  )
 )
```

Demo:

```
> ( play '( c d e f g a b c c b a g f e d c ) )
```



```
> ( play '(c c g g a a g g f f e e d d c c ) )
```



```
> ( play '( c d e c c d e c e f g g e f g g ) )
```

```
>
```

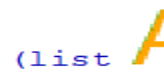
Task 7 - Grapheme to Color Synesthesia:

Code:

```
( define AI (text "A" 36 "orange") )
( define BI (text "B" 36 "red") )
( define CI (text "C" 36 "blue") )
( define DI (text "D" 36 "pink") )
( define EI (text "E" 36 "yellow") )
( define FI (text "F" 36 "purple") )
( define GI (text "G" 36 "brown") )
( define HI (text "H" 36 "green") )
( define II (text "I" 36 "black") )
( define JI (text "J" 36 "maroon") )
( define KI (text "K" 36 "gold") )
( define LI (text "L" 36 "turquoise") )
( define MI (text "M" 36 "firebrick") )
( define NI (text "N" 36 "silver") )
( define OI (text "O" 36 "tomato") )
( define PI (text "P" 36 "coral") )
( define QI (text "Q" 36 "olive") )
( define RI (text "R" 36 "sea shell") )
( define SI (text "S" 36 "lime") )
( define TI (text "T" 36 "sea green") )
( define UI (text "U" 36 "blue") )
( define VI (text "V" 36 "royal blue") )
( define WI (text "W" 36 "teal") )
( define XI (text "X" 36 "navy") )
( define YI (text "Y" 36 "plum") )
( define ZI (text "Z" 36 "dim gray") )

( define alphabet '(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z) )
( define alphapic ( list AI BI CI DI EI FI GI HI II JI KI LI MI NI OI PI QI RI SI TI UI VI VI WI XI YI ZI) )

( define a->i ( zip alphabet alphapic ) )

( define ( letter->image letter )
   ( cdr ( assoc letter a->i ) )
 )

( define ( gcs lst )
   ( foldr beside empty-image
     ( map ( lambda ( x ) ( cdr ( assoc x a->i ) ) ) lst )
   )
 )
```

Demo 1:

```
> alphabet
'(A B C)
> alphapic
```




```
> ( display a->i )
```


```
> ( letter->image 'A )
```


```
> ( letter->image 'B )
```

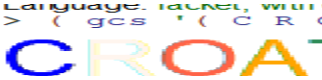

```
> ( gcs '( C A B ) )
```


```
> ( gcs '( B A A ) )
```


```
> ( gcs '( B A B A ) )
```

Demo 2:

```
Language: racket, with debugging; memory limit: 1
> ( gcs '( C R O A T I A ) )
```



```
> ( gcs '( P E N G U I N S ) )
```

```
> ( gcs '( B L U E ) )
```

```
> ( gcs '( R A C K E T ) )
```

```
> ( gcs '( B A S K E T B A L L ) )
```

```
> ( gcs '( C O O K I E S ) )
```

```
> ( gcs '( P H O N E ) )
```

```
> ( gcs '( C A M E R O N ) )
```

```
> ( gcs '( D E M O ) )
```



```
> ( gcs '( C O D E ) )
```