# Racket Programming Assignment #5: RLP and HoFs

## What's It All About?

Working within the DrRacket PDE, please do each of the following tasks. Some of the tasks involve straightforward recursive list processing. Some involve the use of higher-order functions. Some involve a mix of both.

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.

## Overall Charge

Generate a solution document template that is consistent with the accompanying solution template. Then, working within the DrRacket PDE, please do each of the tasks, adding source code and demos to your template in the appropriate manner.

## Task 1 - Simple List Generators

For this task you are asked to write and demo three simple list generating functions using recursion, and one more using a classic composition of higher order functions.

## Task 1a - iota

## Function Definition

Define a **recursive** function called `iota` according to the following specification:

1. The one and only parameter is presumed to be a natural number (positive integer).
2. The value of the function will be a list consisting of the natural numbers from 1 through the value of the parameter.

## Demo

Mimic the following 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)
>
```

## Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 1b - Same

## Function Definition

Define a **recursive** function called `same` according to the following specification:

1. The first parameter is presumed to be a nonnegative integer.
2. The second parameter is presumed to be a Lisp object.
3. The value of the function is a list of length equal to the value of the first parameter containing just that many instances of the value of the second parameter.

## Demo

```
> ( 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))
>
```

## Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 1c - Alternator

### Function Definition

Define a **recursive** function called `alternator` according to the following specification:

1. The first parameter is presumed to be a nonnegative integer.
2. The second parameter is presumed to be a list of Lisp objects.
3. The value of the function is a list of length equal to the value of the first parameter, where the elements are drawn from the value of the second parameter according to a linear ordering given by the cyclic interpretation of the of elements in the value of the second parameter. Please see the demo for clarification.

### Demo

```
> ( 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)
>
```

### Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 1d - Sequence

### Function Definition

Define a function called `sequence` according to the following specification:

1. The first parameter is presumed to be a nonnegative integer.

2. The second parameter is presumed to be a number.
3. The value of the function is a list of length equal to the value of the first parameter containing the integers from 1 to the value of the first parameter, each multiplied by the second parameter.

**Constraint: Rather that using recursion for this function, make good use of the `map` function, writing a `lambda` function for its first argument, and applying `iota` to obtain the second argument.**

## 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)
>
```

## Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 2 - Counting

For this task you are asked to write and demo two counting programs, and then recreate one additional, more elaborate, demo.

## Task 2a - Accumulation Counting

### Function Definition

Define a **recursive** function called `a-count` according to the following specification:

1. The one and only parameter is presumed to be a list of natural numbers.
2. The value of the function will be a list in which each element k of the input list is replace by the numbers from 1 through k. Please see the demo for clarification.

### Demo

Mimic the following 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)
>
```

### Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 2b - Repetition Counting

### Function Definition

Define a **recursive** function called `r-count` according to the following specification:

1. The one and only parameter is presumed to be a list of natural numbers.
2. The value of the function will be a list in which each element k of the input list is replace by k occurrences of k. Please see the demo for clarification.

## Demo

Mimic the following 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)
>
```

## Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 2c - Mixed Counting Demo

This demo will involve mixing applications of accumulation counting and repetition counting in various ways.

## 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)
>
```

## Add to the Solution Document

Simply add the mixed demo to your solution document.

## Task 3 - Association Lists

This task involves defining and demoing two recursive functions, one to create an “association list”, and one to look up a key value in an association list.

## Task 3a - Zip

### Function Definition

Define a **recursive** function called `zip` according to the following specification:

1. The first parameter is presumed to be a list of objects.
2. The second parameter is presumed to be a list of objects of the same length as the value of the first parameter.
3. The value of the function will be a list of pairs obtained by “consing” successive elements of the two lists.

### Demo

```
> ( 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))
>
```

### Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 3b - Assoc

## Function Definition

Define a **recursive** function called `assoc` according to the following specification:

1. The first parameter is presumed to be a lisp object.
2. The second parameter is presumed to be an association list.
3. The value of the function will be the first pair in the given association list for which the car of the pair equals the value of the first parameter, or '() if there is no such element.

## 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 )
'()
>
```

## Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 3c - Establishing some Association Lists

## Code

Please add the following code to your definitions buffer:

```
( 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

This demo simply consists of assuring that the three association lists are functioning properly.

```
> 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"))
>
```

## Add to the Solution Document

Add the association list definitions and your re-creation of the demo to your solution document.

## Task 4 - Numbers to Notes to ABC

This task involves defining three conversion functions. The first features an association list. The second features the definition of a `lambda` function in the context of the `map` function. The third uses the `string-join` primitive.

## Task 4a - nr->note

### Function Definition

Define a function called `nr->note` according to the following specification:

1. The first parameter is presumed to be an integer between 1 and 7, inclusive.
2. The second parameter is presumed to be an association list which maps the numbers 1 through 7 to a musical note in ABC notation.
3. The value of the function will be the musical note in ABC notation that corresponds to the input number, as specified by the association list.

### 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"
>
```

### Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 4b - nrs->notes

## Function Definition

Define a function called `nrs->notes` according to the following specification:

1. The first parameter is presumed to be a list of integers between 1 and 7, inclusive.
2. The second parameter is presumed to be an association list which maps the numbers 1 through 7 to a musical note in ABC notation.
3. The value of the function will be a list of the musical notes in ABC notation that corresponds to the input list of integers, as specified by the association list.

**Constraint: Use the `map` function, and write your own `lambda` function, which converts an integer to the appropriate note in ABC notation, to embed within the map function.**

## 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")
>
```

## Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## Task 4c - nrs->abc

## Function Definition

Define a function called `nrs->abc` according to the following specification:

1. The first parameter is presumed to be a list of integers between 1 and 7, inclusive.
2. The second parameter is presumed to be an association list which maps the numbers 1 through 7 to a musical note in ABC notation.
3. The value of the function will be a string representation of the ABC representation of the musical notes that correspond to the input list of integers, as specified by the association list.

**Constraint: Look up the specification of the `string_join` primitive function, and make good use of it!**

## 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 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"
>
```

## Add to the Solution Document

Add your function definition and your re-creation of the demo to your solution document.

## An Extra Little Something (Optional)

Before moving on to the next task, perhaps you would like to listen to a few sound files relating to the "composition by counting" method? If so, you will find some here:

`https://www.cs.oswego.edu/~blue/arts_forever/music/MxM/CompositionByCounting/index.html`

## Task 5 - Stella

This task involves writing a program to create nested square images in the style of Frank Stella using higher order functions. The function that you are asked to define will be specified in words, and the constraint that you use higher order functions in your work will be made explicit. Then, a recursive program to render the images will be presented, by way of edification. After that, three demos will be presented. Finally, something will be said which pertains to building your solution document.

## Function Definition

Define a function called `stella` according to the following specification:

1. The one and only parameter is presumed to be an **association list** containing pairs which specify the squares that will make up the Stella image. For each pair in the association list, the **car** represents a square side length and the **cdr** represents a square color. Furthermore, the pairs are ordered from small to large according to side length.
2. The value of the function will be a nested square image that is obtained by overlaying one square upon another, as indicated by the given association list. Please see the demos for clarification.

**Constraint: You must use `foldr` in writing this function. Furthermore, within the call to `foldr` you must make use of the `map` function, which should in turn make use of a function that generates a square given the size and color of the square in terms of a pair (i.e., a cons cell).**

## Recursive Implementation of the Definition

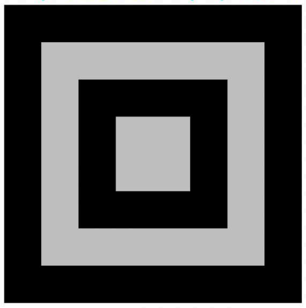
Please note that this function is meant for you to read. I was thinking that you might like to constrast this recursive version with the higher order function version that you are being asked to write.

```
( define ( stella spec )
  ( cond
    ( ( empty? spec ) empty-image )
    ( else
      ( overlay
        ( make-square ( car spec ) )
        ( stella ( cdr spec ) )
      )
    )
  )
)

( define ( make-square pair )
  ( square ( car pair ) 'solid ( cdr pair ) )
)
```

## Demo

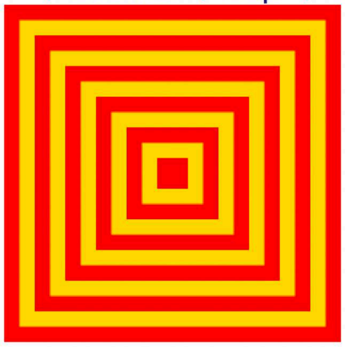
For your solution document, please recreate the three demos that I am providing, and two more, each of which uses the `sequence` program and the `alternator` program to help generate a value to pass along to the `stella` function. Furthermore, please do your best to render nice images in the style of Stella which look somewhat different from mine in terms of square sizes and colors.

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



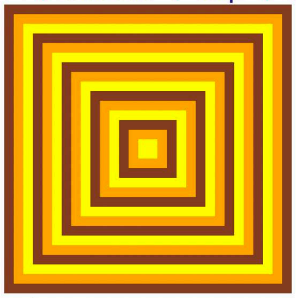
```
>
```

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



```
>
```

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



```
>
```

## Add to the Solution Document

Add your code for the function definition and the five required demos to your solution document.

## Task 6 - Chromesthetic Renderings

This problem is based loosely on the phenomenon of chromesthesia, the mapping of musical pitch classes to colors.

## Function Definition

Define a function called `play` according to the following specification:

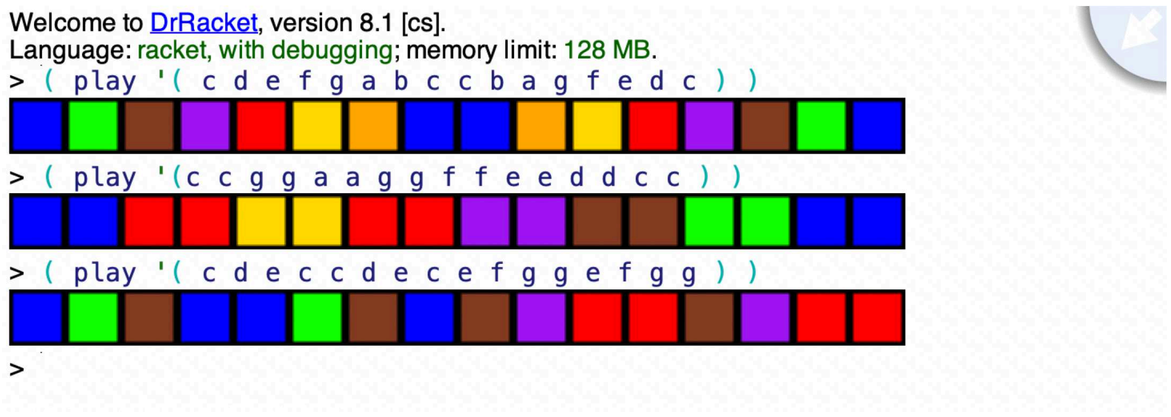
1. The sole parameter is a list of pitch names drawn from the set {c, d, e, f, g, a, b}.
2. The result is an image consisting of a sequence of colored squares with black frames, with the colors determined by the following mapping: c→blue; d→green; e→brown; f→purple; g→red; a→gold; b→orange.

**Constraint: Your function definition must use `map` twice and `foldr` one time.**

## Some auxilliary for you to use

```
( 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 ) )
)
```

## Demo

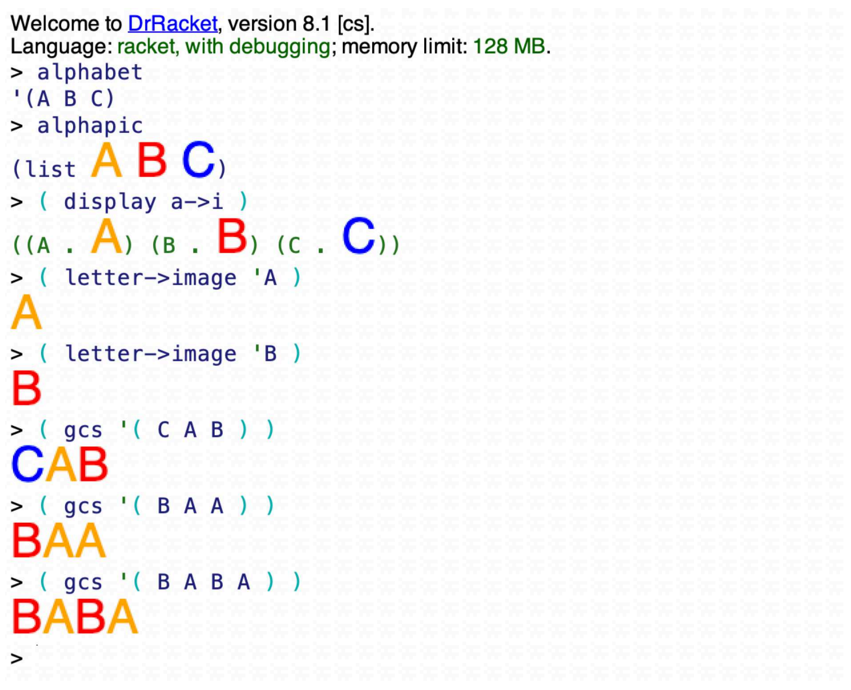


```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( play '( c d e f g a b c c b a g f e d c ) )
> ( play '(c c g g a a g g f f e e d d c c ) )
> ( play '( c d e c c d e c e f g g e f g g ) )
>
```

## Add to the Solution Document

Add your code for the function definition and the demo to your solution document.

## Task 7 - Grapheme to Color Synesthesia



```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> alphabet
'(A B C)
> alphapic
(list A B C)
> ( display a->i )
((A . A) (B . B) (C . C))
> ( letter->image 'A )
A
> ( letter->image 'B )
B
> ( gcs '( C A B ) )
CAB
> ( gcs '( B A A ) )
BAA
> ( gcs '( B A B A ) )
BABA
>
```

## Code Development

1. Add the following code to your source file, after the code which is associated with the previous tasks:

```
( define AI (text "A" 36 "orange") )
( define BI (text "B" 36 "red") )
( define CI (text "C" 36 "blue") )

( define alphabet '(A B C) )
( define alphapic ( list AI BI CI ) )

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

2. Making good use of the association list to which the variable `a->i` is bound, and making good use of the `assoc` function, write the `letter->image` function, and test it.
3. Making good use of both the `map` function and the `foldr` function, write the `gcs` function, and test it.
4. Do the "Demo 1" activity (see below).
5. Copy/paste the demo into your solution document.
6. Extend the program so that it will map the full alphabet, not just the first three letters, to colored images.
7. Add your code for the program (all of it) and the demo to your solution document.
8. Do the "Demo 2" activity (see below).

9. Copy/paste the demo into your solution document.

## Demo 1

Recreate the demo that opens this grapheme to color synesthesia task.

## Demo 2

Create a demo that illustrates grapheme to color synesthesia on at least 10 words, including alphabet and dandelion.

## Due Date

Tuesday, March 28, 2023