Racket Assignment #5: RLP and HoFs

Written by **David Hennigan**

Abstract

This assignment includes 7 tasks, all utilizing either recursive list processing, high order functions, or some combination of both. Task 4 was my personal favorite, involving the generation of music via recursive list generation and the high order function of mapping generated numbers to keys.

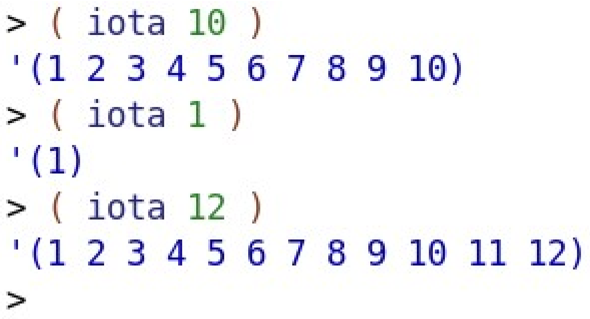
Task 1 – Simple List Generators

Task 1a – iota

Function Definition

```
( define ( iota n )
  ( cond
    ( ( = n 0 ) '() )
    ( else
      ( snoc n ( iota ( - n 1 ) ) )
    )
  )
)


( define ( snoc n l )
  ( cond
    ( ( empty? l )
      ( list n )
    )
    ( else
      ( cons ( car l ) ( snoc n ( cdr l ) ) )
    )
  )
)
```

Demo



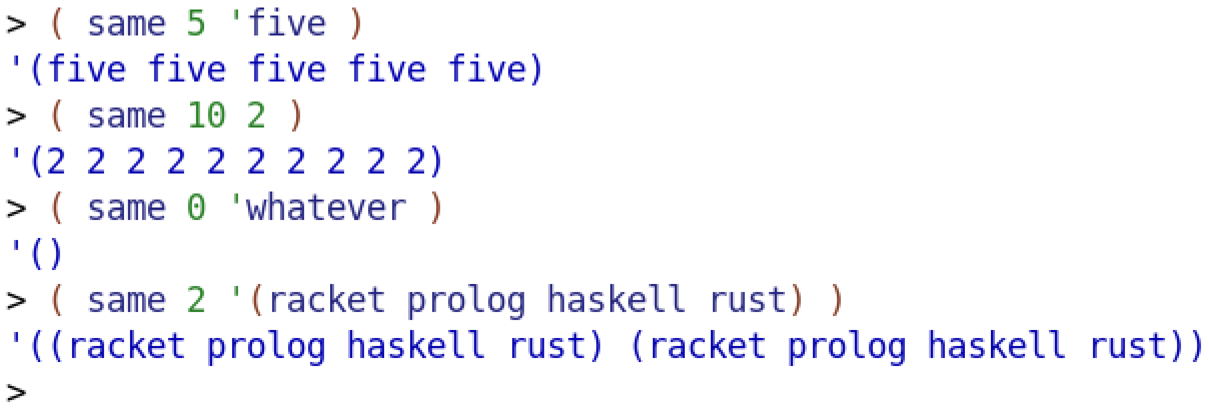
```
> ( iota 10 )
'(1 2 3 4 5 6 7 8 9 10)
> ( iota 1 )
'(1)
> ( iota 12 )
'(1 2 3 4 5 6 7 8 9 10 11 12)
>
```

Task 1b – Same

Function Definition

```
( define ( same n obj )
  ( cond
    ( ( = n 0 ) '() )
    ( else
      ( cons obj ( same ( - n 1 ) obj) )
    )
  )
)
```

Demo



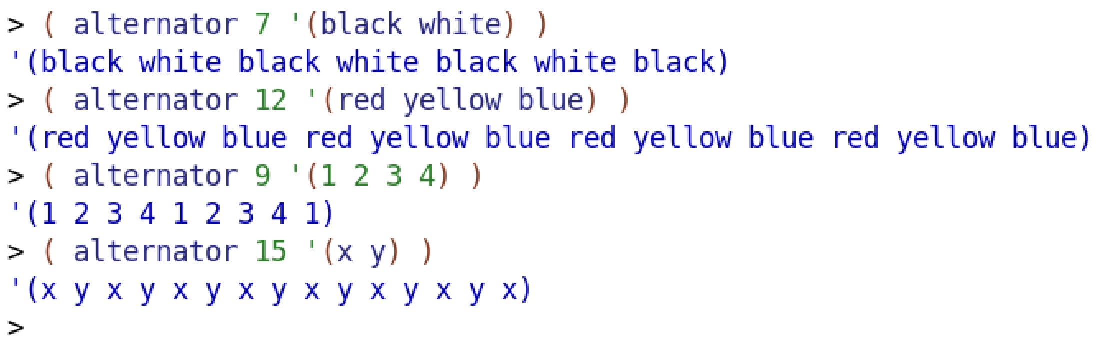
```
> ( same 5 'five )
'(five five five five five)
> ( same 10 2 )
'(2 2 2 2 2 2 2 2 2 2)
> ( same 0 'whatever )
'()
> ( same 2 '(racket prolog haskell rust) )
'((racket prolog haskell rust) (racket prolog haskell rust))
>
```

Task 1c – Alternator

Function Definition

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


( define ( snoc n l )
  ( cond
    ( ( empty? l )
      ( list n )
    )
    ( else
      ( cons ( car l ) ( snoc n ( cdr l ) ) )
    )
  )
)
```

Demo



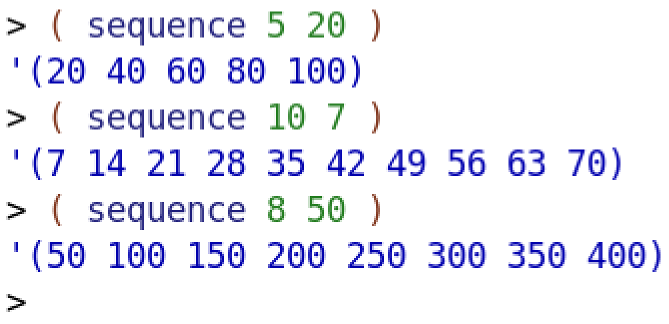
```
> ( alternator 7 '(black white) )
'(black white black white black white black)
> ( alternator 12 '(red yellow blue) )
'(red yellow blue red yellow blue red yellow blue red yellow blue)
> ( alternator 9 '(1 2 3 4) )
'(1 2 3 4 1 2 3 4 1)
> ( alternator 15 '(x y) )
'(x y x y x y x y x y x y x y x)
>
```

Task 1d – Sequence

Function Definition

```
( define ( sequence n num )
  ( map ( lambda (x) ( * x num ) ) ( iota n ) )
 )
```

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)
>
```

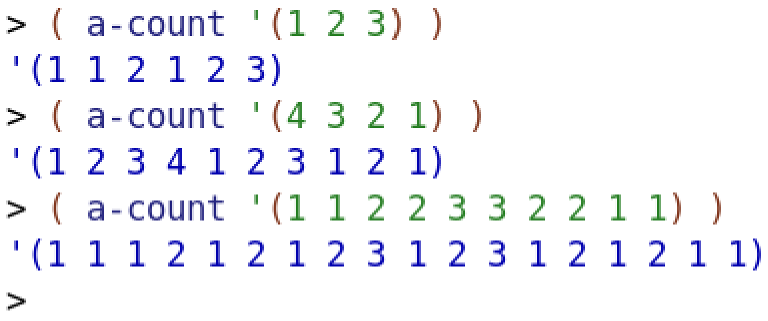
Task 2 – Counting

Task 2a – Accumulation Counting

Function Definition

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

Demo



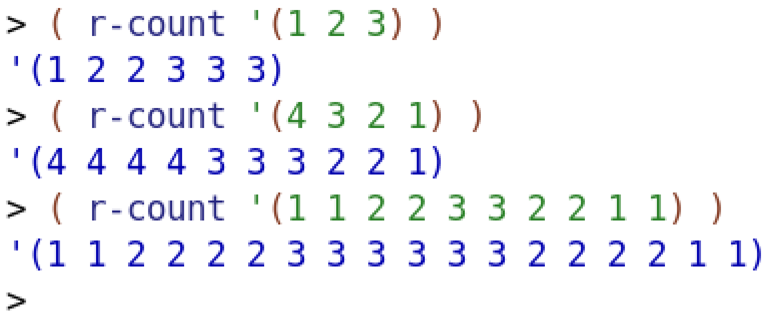
```
> ( a-count '(1 2 3) )
'(1 1 2 1 2 3)
> ( a-count '(4 3 2 1) )
'(1 2 3 4 1 2 3 1 2 1)
> ( a-count '(1 1 2 2 3 3 2 2 1 1) )
'(1 1 1 2 1 2 1 2 3 1 2 3 1 2 1 2 1 1)
>
```

Task 2b – Repetition Counting

Function Definition

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

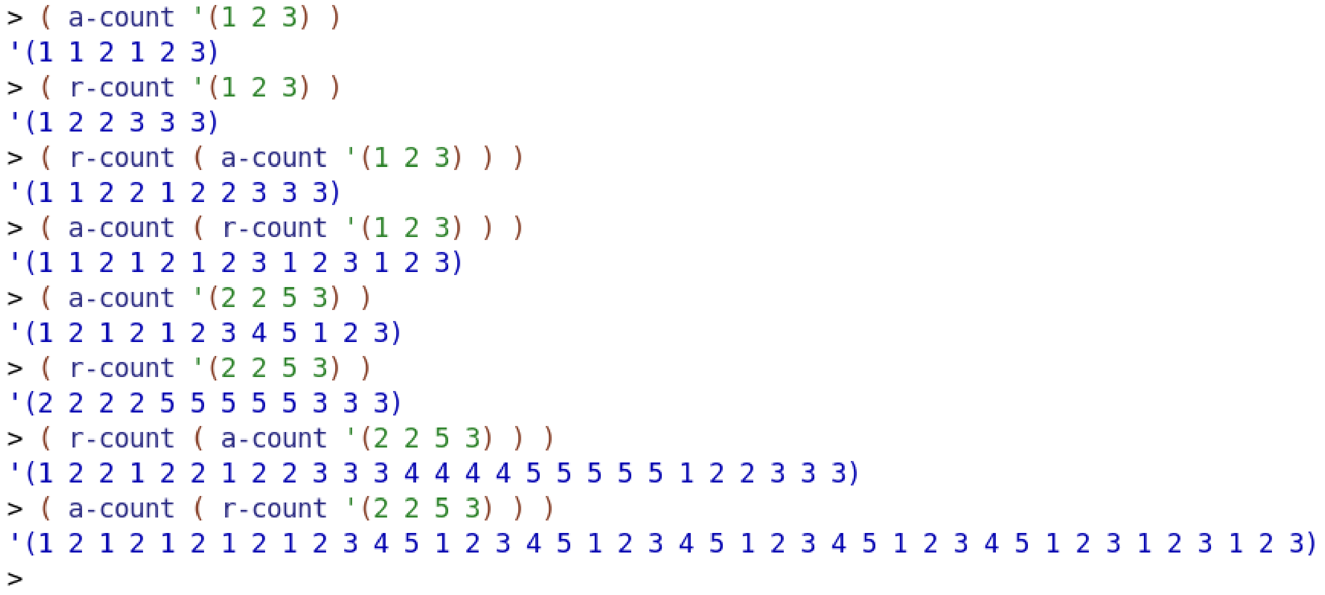
Demo



```
> ( 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 2c – 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)
>
```

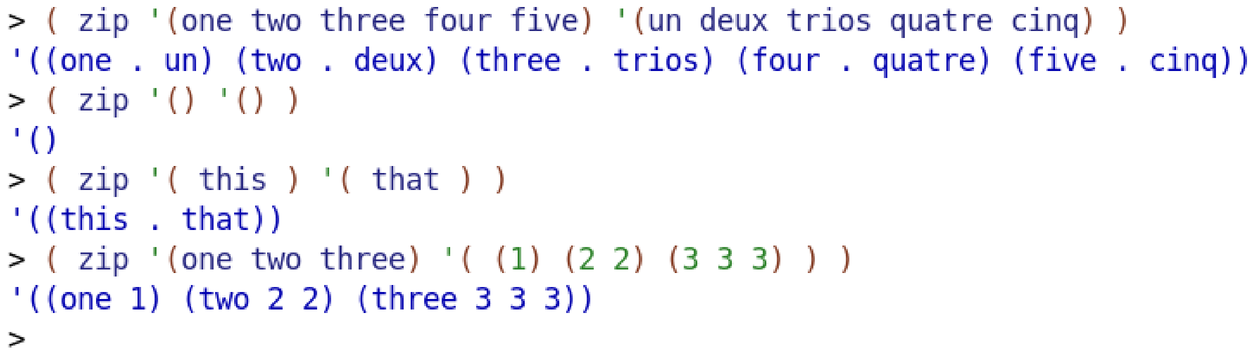
Task 3 – Association Lists

Task 3a – Zip

Function Definition

```
( define ( zip l1 l2 )
  ( cond
    ( ( empty? l1 ) '() )
    ( else
      ( cons ( list* ( car l1 ) ( car l2 ) ) ( zip ( cdr l1 ) ( cdr l2 ) ) )
    )
  )
)
```

Demo



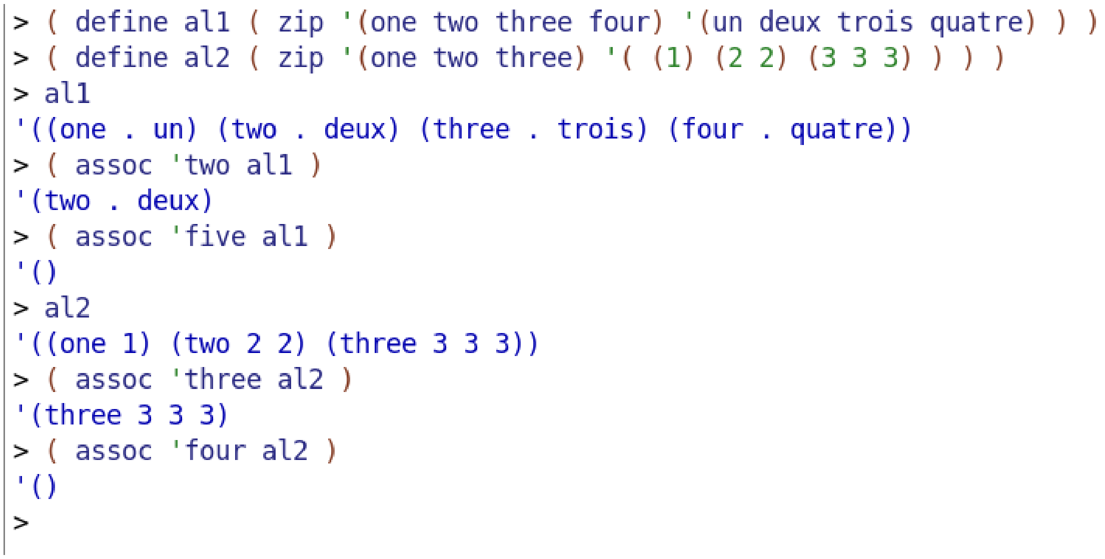
```
> ( zip '(one two three four five) '(un deux trios quatre cinq) )
'((one . un) (two . deux) (three . trios) (four . quatre) (five . cinq))
> ( zip '() '() )
'()
> ( zip '( this ) '( that ) )
'((this . that))
> ( zip '(one two three) '( (1) (2 2) (3 3 3) ) )
'((one 1) (two 2 2) (three 3 3 3))
>
```

Task 3b – Association

Function Definition

```
( define ( assoc obj l )
  ( cond
    ( ( empty? l ) '() )
    ( ( eq? obj ( caar l ) ) ( car l ) )
    ( else ( assoc obj ( cdr l ) ) )
  )
)
```

Demo



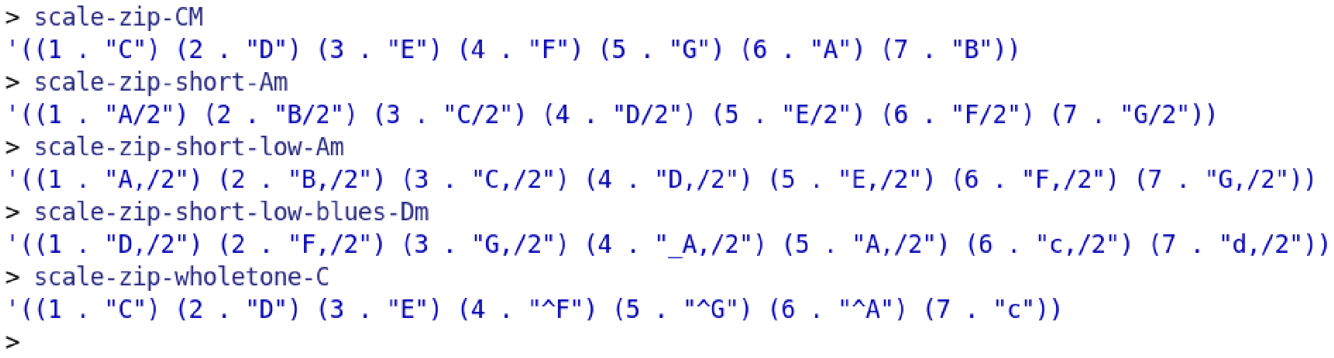
```
> ( define al1 ( zip '(one two three four) '(un deux trois quatre) ) )
> ( define al2 ( zip '(one two three) '( (1) (2 2) (3 3 3) ) ) )
> al1
'((one . un) (two . deux) (three . trois) (four . quatre))
> ( assoc 'two al1 )
'(two . deux)
> ( assoc 'five al1 )
'()
> al2
'((one 1) (two 2 2) (three 3 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"))
>
```

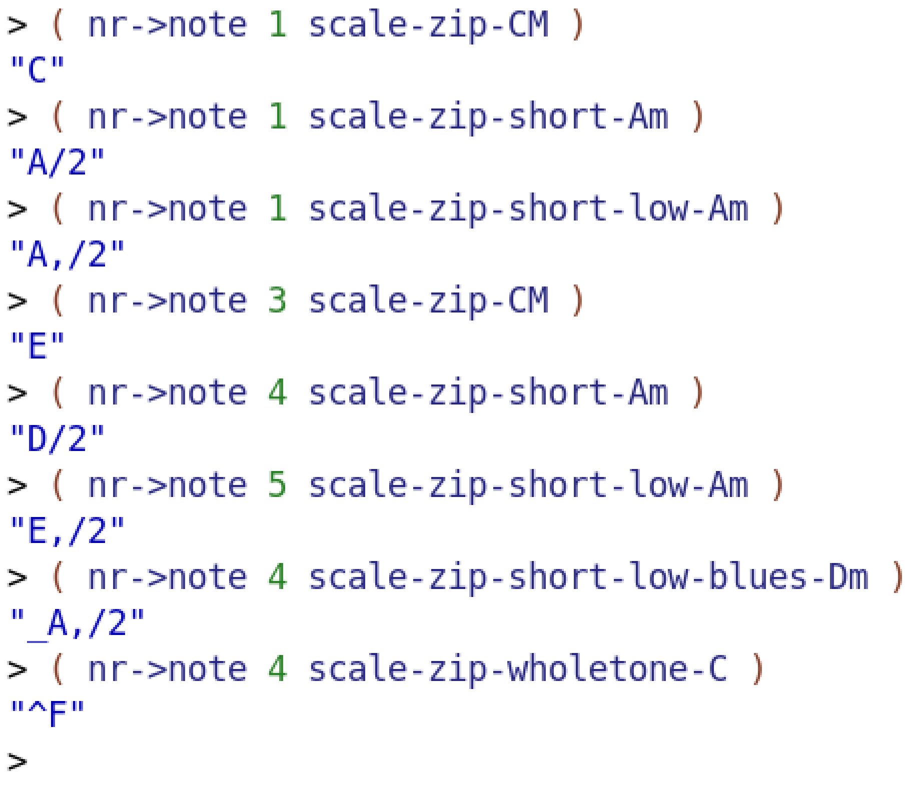
Task 4 – Numbers to Notes to ABC

Task 4a – nr → note

Function Definition

```
( define ( nr->note n al )
  ( cond
    ( ( eq? n ( caar al ) ) ( cdar al ) )
    ( else ( nr->note n ( cdr al ) ) )
  )
)
```

Demo



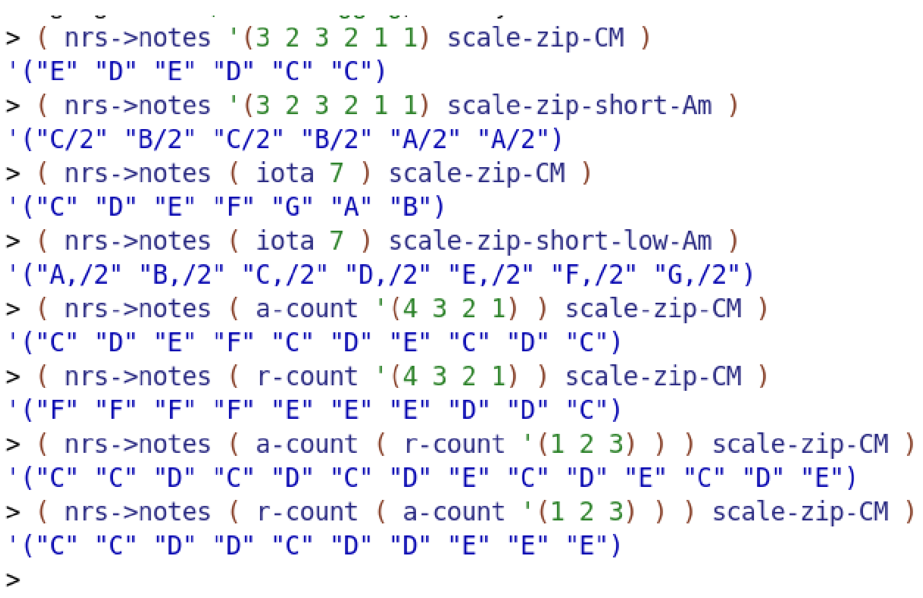
```
> ( nr->note 1 scale-zip-CM )
"C"
> ( nr->note 1 scale-zip-short-Am )
"A/2"
> ( nr->note 1 scale-zip-short-low-Am )
"A,/2"
> ( nr->note 3 scale-zip-CM )
"E"
> ( nr->note 4 scale-zip-short-Am )
"D/2"
> ( nr->note 5 scale-zip-short-low-Am )
"E,/2"
> ( nr->note 4 scale-zip-short-low-blues-Dm )
"_A,/2"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

Task 4b – nrs → notes

Function Definition

```
( define ( nrs->notes nl al )
  ( map ( lambda (x) ( nr->note x al ) ) nl )
)
```

Demo



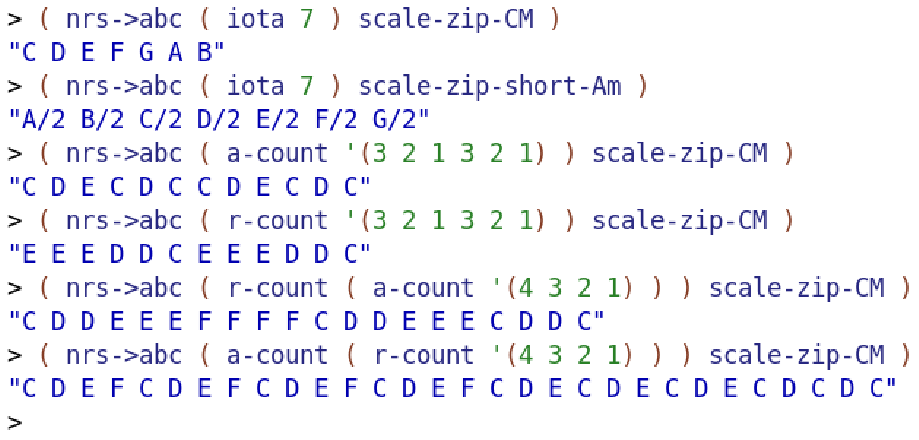
```
> ( nrs->notes '(3 2 3 2 1 1) scale-zip-CM )
'("E" "D" "E" "D" "C" "C")
> ( nrs->notes '(3 2 3 2 1 1) scale-zip-short-Am )
'("C/2" "B/2" "C/2" "B/2" "A/2" "A/2")
> ( nrs->notes ( iota 7 ) scale-zip-CM )
'("C" "D" "E" "F" "G" "A" "B")
> ( nrs->notes ( iota 7 ) scale-zip-short-low-Am )
'("A,/2" "B,/2"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

Function Definition

```
( define ( nrs->abc nl al )
  ( string-join ( nrs->notes nl al ) )
)
```

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"
>
```

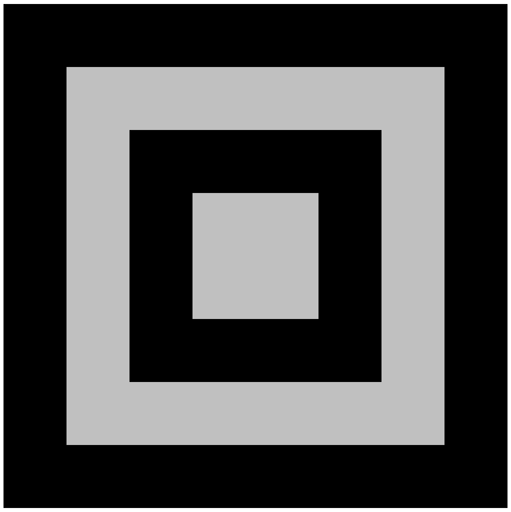
Task 5 Stella

Function Definition

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

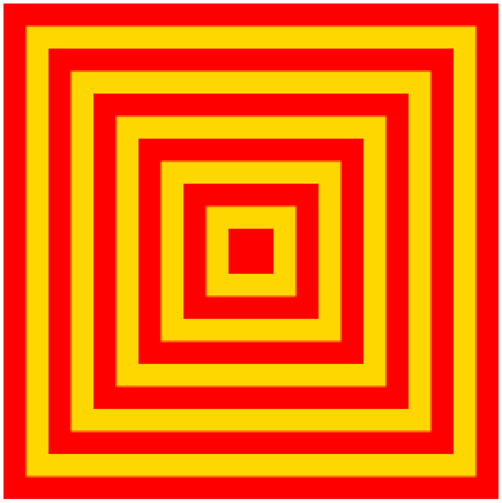
The Three Demos

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

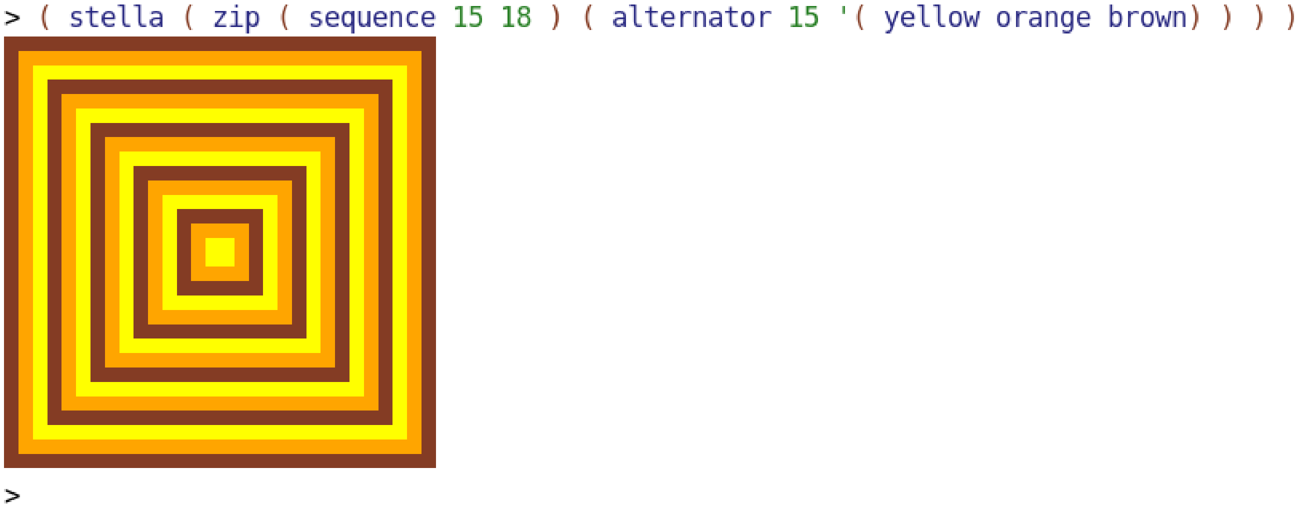


```
>
```

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



```
>
```



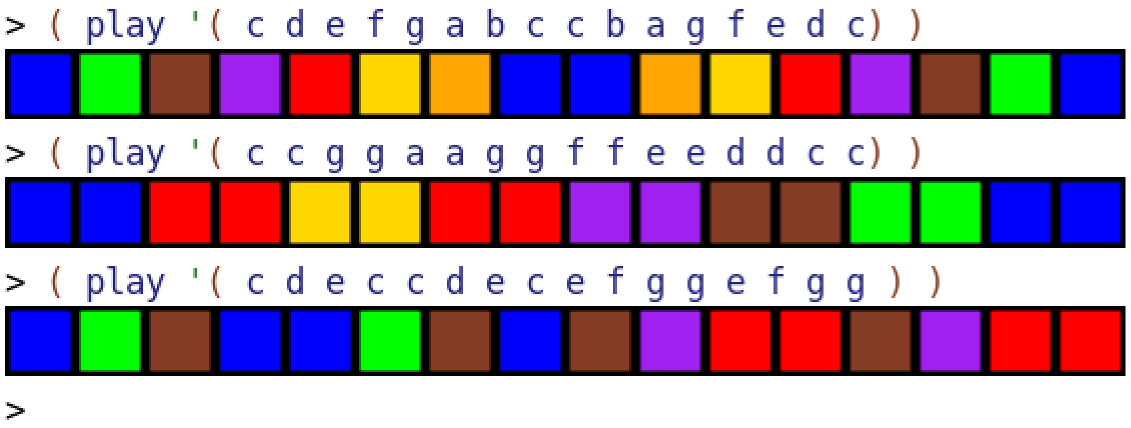
```
> ( stella ( zip ( sequence 15 18 ) ( alternator 15 '( yellow orange brown) ) ) )
>
```

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 ) )
( define cb-a-list ( zip color-names boxes ) )

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

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

( define ( play pitches )
  ( foldr beside empty-image ( map ( lambda (c) ( color->box c ) ) ( map ( lambda (pitch) ( pc->color pitch ) ) pitches ) ) )
 )
```

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 ) )

>
```

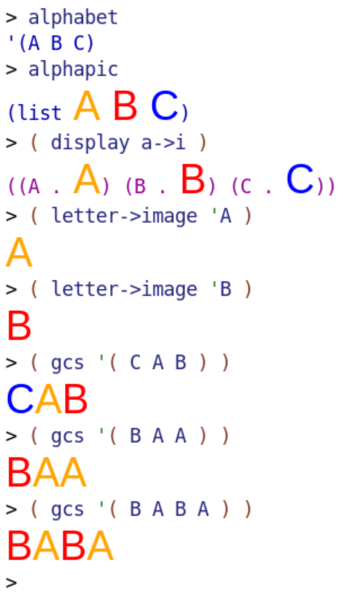
Task 7 – Grapheme to Color Synesthesia

Code

```
( define AI ( text "A" 36 "orange" ) )
( define BI ( text "B" 36 "red" ) )
( define CI ( text "C" 36 "blue" ) )
( define DI ( text "D" 36 "green" ) )
( define EI ( text "E" 36 "Cornflower Blue" ) )
( define FI ( text "F" 36 "Firebrick" ) )
( define GI ( text "G" 36 "Pink" ) )
( define HI ( text "H" 36 "Yellow" ) )
( define II ( text "I" 36 "Maroon" ) )
( define JI ( text "J" 36 "Cornsilk" ) )
( define KI ( text "K" 36 "Medium Sea Green" ) )
( define LI ( text "L" 36 "Dodger Blue" ) )
( define MI ( text "M" 36 "Light Cyan" ) )
( define NI ( text "N" 36 "Tomato" ) )
( define OI ( text "O" 36 "Black" ) )
( define PI ( text "P" 36 "Hot Pink" ) )
( define QI ( text "Q" 36 "Gold" ) )
( define RI ( text "R" 36 "Dark Khaki" ) )
( define SI ( text "S" 36 "Turquoise" ) )
( define TI ( text "T" 36 "Dark Magenta" ) )
( define UI ( text "U" 36 "Light Grey" ) )
( define VI ( text "V" 36 "Rosy Brown" ) )
( define WI ( text "W" 36 "Dark Slate Blue" ) )
( define XI ( text "X" 36 "Green Yellow" ) )
( define YI ( text "Y" 36 "Dark Orange" ) )
( define ZI ( text "Z" 36 "Navajo White" ) )
( 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 WI XI YI
ZI ) )



( define ( zip l1 l2 )
  ( cond
    ( ( empty? l1 ) '() )
    ( else
      ( cons ( list* ( car l1 ) ( car l2 ) ) ( zip ( cdr l1 ) ( cdr l2 ) ) )
    )
  )
)

( define ( assoc obj l )
  ( cond
    ( ( empty? l ) '() )
    ( ( eq? obj ( caar l ) ) ( car l ) )
    ( else ( assoc obj ( cdr l ) ) )
  )
)

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

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

( define ( gcs letters )
  ( foldr beside empty-image ( map ( lambda (letter) ( letter->image letter ) ) letters ) )
```

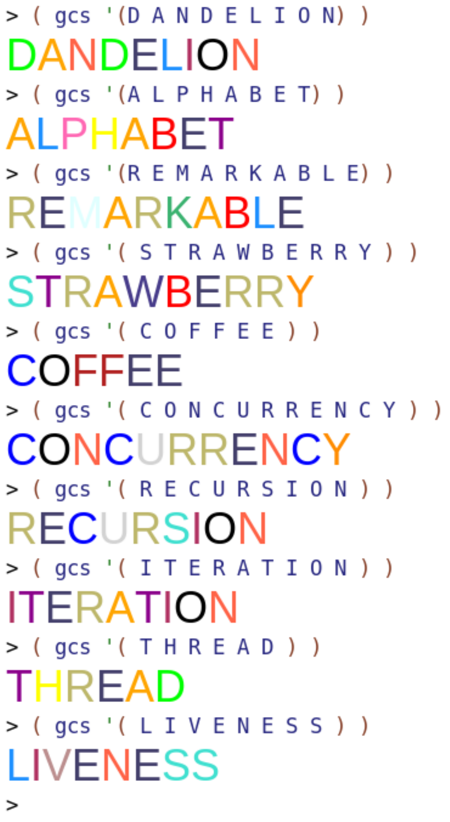
)

Demo 1



```
> 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
>
```

Demo 2



```
> ( gcs '(D A N D E L I O N) )
DANDELION
> ( gcs '(A L P H A B E T) )
ALPHABET
> ( gcs '(R E M A R K A B L E) )
REMARKABLE
> ( gcs '( S T R A W B E R R Y ) )
STRAWBERRY
> ( gcs '( C O F F E E ) )
COFFEE
> ( gcs '( C O N C U R R E N C Y ) )
CONCURRENCY
> ( gcs '( R E C U R S I O N ) )
RECURSION
> ( gcs '( I T E R A T I O N ) )
ITERATION
> ( gcs '( T H R E A D ) )
THREAD
> ( gcs '( L I V E N E S S ) )
LIVENESS
>
```