Damian Randall

Racket Programming Assignment #5: RLP and HoFs

Abstract:
In this assignment we are asked to use higher order functions along with recursive functions

---

Task 1 - Simple List Generators

Task 1a - iota

Code:

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

  ( cond
    ( ( = integer 1 ) '( 1 ) )
    ( else
      ( snoc integer ( iota ( - integer 1 ) ) )
      )
    )
  )
```

Demo:

Damian Randall

```
> (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 n lst)
  (cond
    ((= n 0)
     empty
     )
    ((> n 0)
     (append (list lst) (same (- n 1) lst))
     )
    )
  )
```

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 haskel rust))
'((racket prolog haskel rust) (racket prolog haskel rust))
>
```

Task 1c - Alternator

Code:

```
( 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

Code:

```
( 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)
```

Damian Randall

Task 2 - Counting

Task 2a - Accumulation Counting

Code:

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

Demo:

Damian Randall

```
> ( 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:

```
> ( 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 3 Association Lists

Task 3a - Zip

Code:

```
( define ( zip lstA lstB )
  ( cond
    ( ( empty? lstA ) empty)
    ( else
      ( cons ( list* ( car lstA ) ( car lstB ) ) ( zip ( cdr lstA ) ( cdr lstB ) ) )
      )
    )
  )
```

Demo:

Damian Randall

```
> ( 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 lstObj lst )
  ( cond
    ( ( empty? lst ) empty )
    ( ( eq? lstObj ( caar lst ) ) ( car lst ) )
    ( else ( assoc lstObj ( 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 n lst )
  ( cond
    ( ( eq? n ( caar lst ) ) ( cdar lst ) )
```

```
    ( else ( nr->note n ( 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 lstA lstB )
( map ( lambda (x) ( nr->note x lstB ) ) lstA )
)
```

Demo:

Damian Randall

```
> ( 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
#<procedure:nrs->notes>
'(4 4 4 4 3 3 3 2 2 1)
'((1 . "C") (2 . "D") (3 . "E") (4 . "F") (5 . "G") (6 . "A") (7 . "B"))
>  ( 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 lstA lstB )
( string-join ( nrs->notes lstA lstB ) )
)
```

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"
```

Damian Randall