Lesson #8: Higher Order Functions

What's It All About?

The concept of a higher order function is introduced. The focus at this time will be on functions which take functions as arguments, rather than on functions which return functions as values. (Later in the course, especially when discussing "currying", we will focus on functions which return functions as values.) In this lesson, a small number of higher order functions are defined, and several standard higher order functions provided by Racket are introduced by way of word and example.

What is a higher order function?

A **higher order function** is a function that takes a function as an argument (perhaps more than one) or returns a function as a result.

Using higher-order functions increases the expressive power of programming languages by allowing common programming patterns of function application to be encapsulated as functions within the language itself.

Defining Filter-in

Source

```
( define ( filter-in p l )
  ( cond
    ( ( empty? l )
      '()
    )
    ( ( p ( car l ) )
      ( cons ( car l ) ( filter-in p ( cdr l ) ) )
    )
    ( else
      ( filter-in p ( cdr l ) )
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define numbers '(1 2 3 4 5 6 7 8 9 10) )
```

```
> ( define colors '("red" "yellow" "blue" "green" "purple") )
> ( filter-in even? numbers )
'(2 4 6 8 10)
> ( filter-in ( lambda (sw) ( < ( string-length sw ) 4 ) ) colors )
'("red")
> ( filter-in ( lambda (sw) ( < ( string-length sw ) 5 ) ) colors )
'("red" "blue")
> ( filter-in ( lambda (sw) ( < ( string-length sw ) 6 ) ) colors )
'("red" "blue" "green")
>
```

Defining Filter-out

Source

```
( define ( filter-out p l )
  ( cond
    ( ( empty? l )
      '()
    )
    ( ( p ( car l ) )
      ( filter-out p ( cdr l ) )
    )
    ( else
      ( cons ( car l ) ( filter-in p ( cdr l ) ) )
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define numbers '(1 2 3 4 5 6 7 8 9 10) )
> ( define colors '("red" "yellow" "blue" "green" "purple") )
> ( filter-out even? numbers )
'(1 3 5 7 9)
> ( filter-out ( lambda (sw) ( < ( string-length sw ) 4 ) ) colors )
'("yellow" "blue" "green" "purple")
> ( filter-out ( lambda (sw) ( < ( string-length sw ) 5 ) ) colors )
'("yellow" "green" "purple")
> ( filter-out ( lambda (sw) ( < ( string-length sw ) 6 ) ) colors )
'("yellow" "purple")
>
```

Filter

There is a function in Racket called `filter` that acts like the `filter-in` function.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define numbers '(1 2 3 4 5 6 7 8 9 10) )
> ( define colors '("red" "yellow" "blue" "green" "purple") )
> ( filter even? numbers )
'(2 4 6 8 10)
> ( filter ( lambda (sw) ( < ( string-length sw ) 4 ) ) colors )
'("red")
> ( filter ( lambda (sw) ( < ( string-length sw ) 5 ) ) colors )
'("red" "blue")
> ( filter ( lambda (sw) ( < ( string-length sw ) 6 ) ) colors )
'("red" "blue" "green")
>
```

> Defining Apply-to-all

Source

```
( define ( apply-to-all f l )
  ( cond
    ( ( empty? l )
      '()
    )
    ( else
      ( cons ( f ( car l ) ) ( apply-to-all f ( cdr l ) ) )
    )
  )
)
```

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( apply-to-all car '( ( a b ) ( c d e ) ( f g h i ) ) )
'(a c f)
> ( apply-to-all cdr '( ( a b ) ( c d e ) ( f g h i ) ) )
'((b) (d e) (g h i))
> ( apply-to-all cadr '( ( a b ) ( c d e ) ( f g h i ) ) )
'(b d g)
> ( apply-to-all ( lambda (n) ( + n 1 ) ) '( 1 2 3 4 5 ) )
'(2 3 4 5 6)
> ( apply-to-all ( lambda (n) ( - n 1 ) ) '( 1 2 3 4 5 ) )
'(0 1 2 3 4)
> ( apply-to-all empty? '( a () b () ( c ) ( d e ) ) )
'(#f #t #f #t #f #f)
> ( apply-to-all length '( () ( a ) ( b c d e ) ) )
```

```
'(0 1 4)
>
```

Map

There is a funtion in Racket called `map` that acts like the `apply-to-all` function.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( map car '( ( a b ) ( c d e ) ( f g h i ) ) )
'(a c f)
> ( map cdr '( ( a b ) ( c d e ) ( f g h i ) ) )
'((b) (d e) (g h i))
> ( map cadr '( ( a b ) ( c d e ) ( f g h i ) ) )
'(b d g)
> ( map ( lambda (n) ( + n 1 ) ) '( 1 2 3 4 5 ) )
'(2 3 4 5 6)
> ( map ( lambda (n) ( - n 1 ) ) '( 1 2 3 4 5 ) )
'(0 1 2 3 4)
> ( map empty? '( a () b () ( c ) ( d e ) ) )
'(#f #t #f #t #f #f)
> ( map length '( () ( a ) ( b c d e ) ) )
'(0 1 4)
>
```

Example Program Featuring Map and Filter: Stem-Leaf Plot

This example program makes a rather big production out of computing a stem-leaf plot for a set of exam grades. The reason for this approach is precisely to illustrate some extremely basic uses of the `filter` function and the `map` function.

The presentation will consist of demo/source pairs, each of which is preceded by just a few words of orientation.

> Preliminary function: Random grade generator

It will be useful to have a random grade generator when it comes time to demonstrate the behavior of this program. The generator presented will generate uniform random grades over the interval 0..100, inclusive. (It would be sad if the grades for an exam actually exhibited this sort of behavior!)

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( generate-grades 10 )
'(14 93 67 95 81 57 35 100 69 82)
> ( generate-grades 15 )
'(52 88 13 76 6 18 93 69 95 13 69 60 65 91 44)
> ( generate-grades 20 )
'(100 10 66 93 54 44 40 83 98 34 39 76 61 18 97 91 73 62 91 24)
>
```

Source

```
( define ( generate-grades n )
  ( map random ( make-list n 101 ) )
)
```

> The Stem and Leaf functions

For a particular grade, we will want a function to compute just the stem, and another function to compute just the leaf. Each of these will be used repeatedly to get the stem and the leaf for each score in the set of exam grades

---

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( stem 75 )
7
```

```
> ( leaf 75 )
5
> ( stem 9 )
0
> ( leaf 9 )
9
> ( stem 100 )
10
> ( leaf 100 )
0
> ( stem 92 )
9
> ( leaf 92 )
2
>
```

Source

```
( define ( stem grade )
  ( quotient grade 10 )
)

( define ( leaf grade )
  ( remainder grade 10 )
)
```

> Long form Stem-Leaf Tree generator

This function transforms a set of exam grades into a stem-leaf tree. In long form, it computes partial results and displays them in order to make the approach to stem-leaf tree creation abundantly clear. Noteworthy bits of code:

1. The primitive `sort` function takes a relational function as its second argument, and thus is a higher order function.
2. The `map` function is used to compute and collect stems.
3. The `map` function is used to compute and collect leaves.
4. The `map` function is used to construct an association-list of stem/leaf pairs.
5. The `branch` function takes a stem as its sole parameter and returns an association list representing all of the leaves with the given stem, which is being termed a "branch". This function uses the `filter` function with a **lambda function** to select the leaves.
6. The `map` function then makes good use of the `branch` function to compute all ten branches of the stem/leaf tree.
7. The `live branches` of the tree (i.e., those with at least one leaf) are computed by the `filter` function which uses a **lambda function** to do the selecting from the list of branches.
8. The branches on the list of live branches are then restructured into a list of "trimmed live branches", for lack of a better term. The most salient aspect of this computation is the use of **three** instances of the `map` function!

Demo

```
Welcome to DrRacket, version 8.1 [cs].
```

```
Language: racket, with debugging; memory limit: 128 MB.
> ( stem-leaf-tree-lf ( generate-grades 20 ) )
grades --> (64 55 18 32 39 19 19 87 81 66 66 44 58 88 36 82 50 94 32 77)

sorted-grades --> (94 88 87 82 81 77 66 66 64 58 55 50 44 39 36 32 32 19 19 18)

stems --> (9 8 8 8 8 7 6 6 6 5 5 5 4 3 3 3 3 1 1 1)

leaves --> (4 8 7 2 1 7 6 6 4 8 5 0 4 9 6 2 2 9 9 8)

a-list --> ((9 . 4) (8 . 8) (8 . 7) (8 . 2) (8 . 1) (7 . 7) (6 . 6) (6 . 6) (6 . 4) (5 . 8)
(5 . 5) (5 . 0) (4 . 4) (3 . 9) (3 . 6) (3 . 2) (3 . 2) (1 . 9) (1 . 9) (1 . 8))

branches --> (() ((9 . 4)) ((8 . 8) (8 . 7) (8 . 2) (8 . 1)) ((7 . 7)) ((6 . 6) (6 . 6)
(6 . 4)) ((5 . 8) (5 . 5) (5 . 0)) ((4 . 4)) ((3 . 9) (3 . 6) (3 . 2) (3 . 2)) () ((1 . 9)
(1 . 9) (1 . 8)) ())

live-branches --> (((9 . 4)) ((8 . 8) (8 . 7) (8 . 2) (8 . 1)) ((7 . 7)) ((6 . 6) (6 . 6)
(6 . 4)) ((5 . 8) (5 . 5) (5 . 0)) ((4 . 4)) ((3 . 9) (3 . 6) (3 . 2) (3 . 2)) ((1 . 9)
(1 . 9) (1 . 8)))

trimmed-live-branches --> ((9 (4)) (8 (8 7 2 1)) (7 (7)) (6 (6 6 4)) (5 (8 5 0)) (4 (4))
(3 (9 6 2 2)) (1 (9 9 8)))

'((9 (4)) (8 (8 7 2 1)) (7 (7)) (6 (6 6 4)) (5 (8 5 0)) (4 (4)) (3 (9 6 2 2)) (1 (9 9 8)))
>
```

Source

```
( define ( stem-leaf-tree-lf grades )
  ( echo 'grades grades )
  ( define sorted-grades ( sort grades > ) )
  ( echo 'sorted-grades sorted-grades )
  ( define stems ( map stem sorted-grades ) )
  ( echo 'stems stems )
  ( define leaves ( map leaf sorted-grades ) )
  ( echo 'leaves leaves )
  ( define a-list ( map cons stems leaves ) )
  ( echo 'a-list a-list )
  ( define ( branch n ) ( filter ( lambda (x) ( = ( car x ) n ) ) a-list ) )
  ( define branches ( map branch ( list 10 9 8 7 6 5 4 3 2 1 0 ) ) )
  ( echo 'branches branches )
  ( define live-branches
    ( filter ( lambda (l) ( not ( empty? l ) ) ) branches )
  )
  ( echo 'live-branches live-branches )
  ( define ( trim-branch b ) ( map cdr b ) )
  ( define trimmed-live-branches
    ( map list
      ( map caar live-branches )
      ( map trim-branch live-branches )
    )
  )
  ( echo 'trimmed-live-branches trimmed-live-branches )
```

```
  trimmed-live-branches
)

( define ( echo symbol value )
   ( display symbol )
   ( display " --> " )
   ( display value )
   ( display "\n\n" )
)
```

> Stem-Leaf Tree generator

This function is the same as the long form function, but with all of the calls to `echo` removed.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( stem-leaf-tree ( generate-grades 20 ) )
'((10 (0)) (9 (5)) (8 (5 0)) (7 (7 4 4 3 1)) (6 (9 6)) (5 (8 8 6)) (4 (5)) (3 (9)) (1 (5 5)) (0 (9 3)))
> ( stem-leaf-tree ( generate-grades 20 ) )
'((9 (8 7 5 5 0)) (8 (0)) (7 (9 7)) (4 (3)) (3 (8 2)) (2 (4 2)) (1 (9 3 3)) (0 (7 5 3 3)))
> ( stem-leaf-tree ( generate-grades 20 ) )
'((8 (7)) (6 (6 4 3 0)) (5 (6 4 3 0)) (4 (9 7)) (3 (1)) (2 (0)) (1 (9 8 8 1)) (0 (7 1 1)))
>
```

Source

```
( define ( stem-leaf-tree grades )
  ( define sorted-grades ( sort grades > ) )
  ( define stems ( map stem sorted-grades ) )
  ( define leaves ( map leaf sorted-grades ) )
  ( define a-list ( map cons stems leaves ) )
  ( define ( branch n ) ( filter ( lambda (x) ( = ( car x ) n ) ) a-list ) )
  ( define branches ( map branch ( list 10 9 8 7 6 5 4 3 2 1 0 ) ) )
  ( define live-branches
    ( filter ( lambda (l) ( not ( empty? l ) ) ) branches )
  )
  ( define ( trim-branch b ) ( map cdr b ) )
  ( define trimmed-live-branches
     ( map list
       ( map caar live-branches )
       ( map trim-branch live-branches )
     )
   )
  trimmed-live-branches
)
```

> Branch-String conversion function

In anticipation of defining the function to display the stem-leaf plot, a function to convert a "trimmed live branch" to a character string is defined. The "leaf string" computation is the most interesting line in the function. The others are quite straightforward.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( branch->string '( 9 ( 7 5 4 4 3 ) ) )
"9:    7 5 4 4 3\n"
> ( branch->string '( 2 ( 5 ) ) )
"2:    5\n"
> ( branch->string '( 7 ( 9 7 6 5 4 3 2 2 1 ) ) )
"7:    9 7 6 5 4 3 2 2 1\n"
>
```

Source

```
( define ( branch->string trimmed-live-branch )
  ( define branch trimmed-live-branch )
  ( define the-stem ( car branch ) )
  ( define the-leaves ( cadr branch ) )
  ( define stem-string ( number->string the-stem ) )
  ( define leaf-string ( string-join ( map number->string the-leaves ) " " ) )
  ( define branch-string
    ( string-append
      stem-string
      ":"
      ( blank-string ( - 5 ( string-length stem-string ) ) )
      leaf-string
      "\n"
     )
  )
  branch-string
)

( define ( blank-string n )
  ( cond
    ( ( = n 0 ) "" )
    ( else
      ( string-append " " ( blank-string ( - n 1 ) ) )
    )
  )
)
```

> Stem-Leaf plot

Note, within the `display-stem-leaf-tree` function, the use of `map` to display the list of branch strings after the use of `map` to generate and collect the branch strings into a list.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( stem-leaf-plot ( generate-grades 50 ) )
10:   0
9:    5 4 3 3 3 0
8:    8 5
7:    9 8 7 5 4
6:    8 7 6 4 4 2
5:    9 8 3
4:    6 5 4 3 3 3 3 1
3:    9 8 8 7 5 0
2:    9 9 8 3 1 0
1:    5 2
0:    8 3 3 1 0
> ( stem-leaf-plot ( generate-grades 50 ) )
9:    7 7 6 4 2 0
8:    9 8 7 7 6
7:    7 3 2 2
6:    8 6 5 5 5 1 1 1
5:    4 3 3
4:    8 7 6 5 2
3:    9 2
2:    8 7 4 0
1:    9 4 0
0:    9 7 7 6 5 3 3 2 1 0
>
```

Source

```
( define ( stem-leaf-plot grades )
  ( define tree ( stem-leaf-tree grades ) )
  ( display-stem-leaf-tree tree )
  ( display "")
)

( define ( display-stem-leaf-tree tree )
  ( map display ( map branch->string tree ) )
)
```

Patterns of Computation

Higher order functions can be seen as generalized functions, based on the recognition of certain patterns of computation. For example:

1. Some functions collect elements of a list that satisfy some property. The `filter` function is based on this pattern of computation.
2. Some functions transform every element of a list in some way. The `map` function is based on this pattern of computation.
3. Some functions combine elements of a list using some operator. (Think `sum` and `product` from Lesson 9 on recursive list processing.) The `foldr` function is based on this pattern of computation.

Defining Sum

Recall `sum` from Lesson 9 on recursive list processing?

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( sum '( 1 3 5 7 9 ) )
25
> ( sum '( 2 2 2 2 2 2 2 ) )
14
>
```

Source

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

Defining Product

The recursive `product` function definition is quite like the recursive `sum` function defintion.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( product '( 1 3 5 7 9 ) )
945
> ( product '( 2 2 2 2 2 2 2 ) )
128
>
```

Source

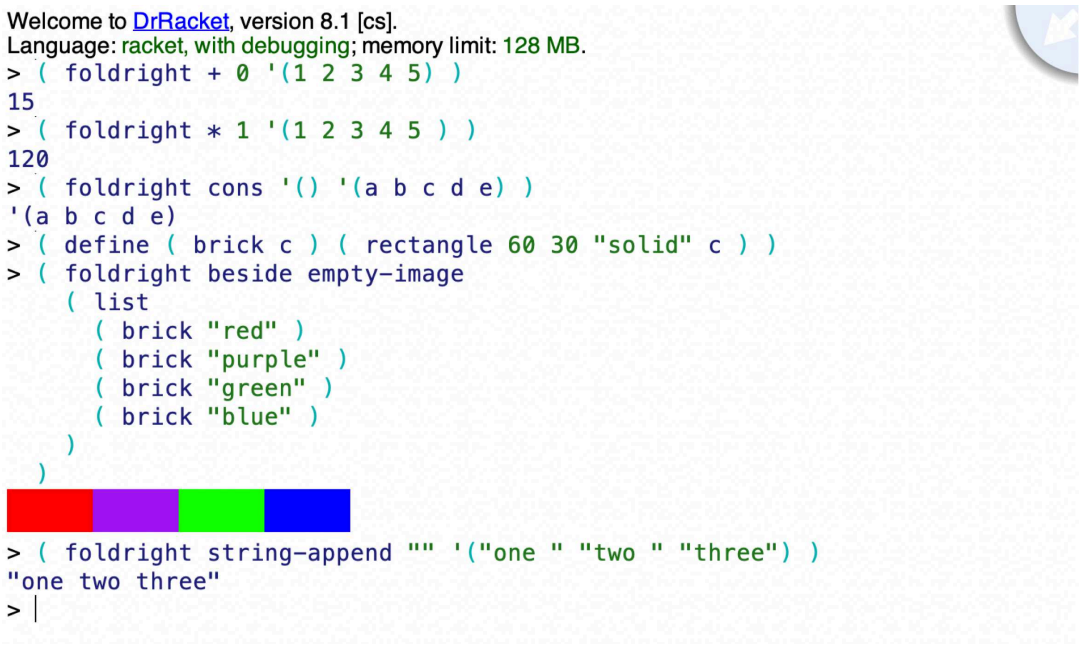
```
( define ( product nl )
  ( cond
    ( ( empty? nl ) 1 )
    ( else
      ( * ( car nl ) ( product ( cdr nl ) ) )
    )
  )
)
```

Defining Foldright

The "fold right" function combines elements of a list **from left to right** according to the given operator, with the given value serving to complete the computation.Thus, for example: the following two lines are functionally equivalent:

- `( foldright cons '() '( a b c d ) )`
- `( cons 'a ( cons 'b ( cons 'c ( cons 'd () ) ) ) )`

Demo



Source

```
( define ( foldright operator value list )
  ( cond
    ( ( empty? list ) value )
    ( else
      ( operator ( car list ) ( foldright operator value ( cdr list ) ) )
    )
  )
)
```

Foldr

There is a function in Racket called `foldr` that acts like the `foldright` function.

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( foldr + 0 '( 1 2 3 4 5 ) )
15
> ( foldr * 1 '( 1 2 3 4 5 ) )
120
> ( foldr cons '() '( a b c d e ) )
'(a b c d e)
> ( define ( brick c ) ( rectangle 60 30 "solid" c ) )
> ( foldr beside empty-image
    ( list
      ( brick "red" )
      ( brick "purple" )
      ( brick "green" )
      ( brick "blue" )
    )
  )
```

```
> ( foldr string-append "" '("one " "two " "three" ) )
"one two three"
>
```

Defining Foldleft

The "fold left" function combines elements of a list **from right to left** according to the given operator, with the given value serving to complete the computation.Thus, for example: the following two lines are functionally equivalent:

- `( foldleft cons '() '( a b c d ) )`
- `( cons 'd ( cons 'c ( cons 'b ( cons 'a () ) ) ) )`

Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( foldleft + 0 '(1 2 3 4 5) )
15
> ( foldleft * 1 '(1 2 3 4 5) )
120
> ( foldleft cons '() '(a b c d e) )
'(e d c b a)
> ( define ( brick c ) ( rectangle 60 30 "solid" c ) )
> ( foldleft beside empty-image
    ( list
      ( brick "red" )
      ( brick "purple" )
      ( brick "green" )
      ( brick "blue" )
    )
  )
```

```
> ( foldleft string-append "" '(" one" " two" "three") )
"three two one"
>
```

Source

```
( define ( foldleft operator value list )
  ( cond
    ( ( empty? list ) value )
    ( else
      ( operator ( rac list ) ( foldleft operator value ( rdc list ) ) )
    )
  )
)
```

Note that the functions `rac` and `rdc` were defined in Lesson 9 on recursive list processing.

Foldl

There is a function in Racket called `foldl` that acts like the `foldleft` function.

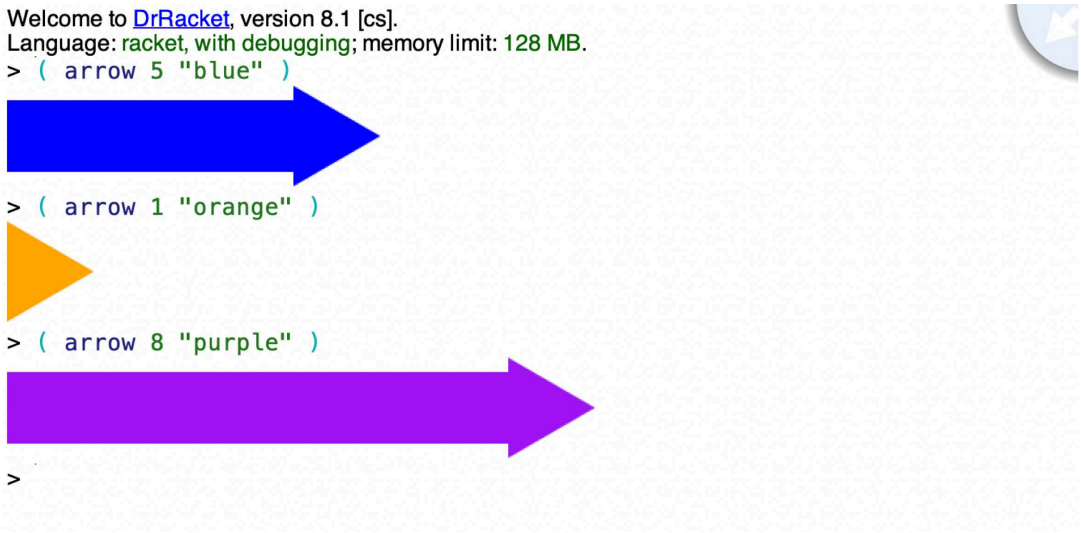
Demo

```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( foldl + 0 '( 1 2 3 4 5 ) )
15
> ( foldl * 1 '( 1 2 3 4 5 ) )
120
> ( foldl cons '() '( a b c d e ) )
'(e d c b a)
> ( define ( brick c ) ( rectangle 60 30 "solid" c ) )
> ( foldl beside empty-image
    ( list
      ( brick "red" )
      ( brick "purple" )
      ( brick "green" )
      ( brick "blue" )
    )
  )
```

```
> ( foldl string-append "" '(" one" " two" "three" ) )
"three two one"
>
```

Example: Graphical Arrow, revisited

Demo



Source

```
( define ( arrow n c )
  ( define part ( square 50 "solid" c ) )
  ( define head ( rotate 30 ( triangle 70 "solid" c ) ) )
  ( define stem ( foldr beside empty-image ( make-list ( - n 1 ) part ) ) )
  ( beside stem head )
)
```