1. The single code prints the exact thing typed after it to the console. It’s similar to the way the double quotes around any characters behave.
2. I expected cons to build a list out of the elements, but it doesn’t work. It’s supposed to build a pair. The second set of code, creates a pair of the current content in the list, adds a new item to the pair, and then sets the value to the list. The code set! Puts the pair into the list.
3. Car returns the first value of the list, and the cdr returns the rest of the list except the first value. Both commands do not manipulate the list.
4. Cadr creates the result of car of cdr of the list. Cdar creates the result of cdr of car of the list. Caadar creates the car of cdr of car of car of list. Also, every a in the word is short for car and every d is short for cdr. You must read the d’s and a’s backwords in order to get the order correct. For example, cadaar means car(car(cdr(car))).

(define capitals '((maryland (annapolis)) (pennsylvania (harrisburg))(delaware (dover))

(virginia (richmond)))) creates a list of those elements listed.

(car (cdr capitals)) returns '(pennsylvania (harrisburg))

(cadr capitals) returns '(pennsylvania (harrisburg))

(cdar capitals) returns '((annapolis))

(cadar capitals) returns '(annapolis)

(caadar capitals) returns 'annapolis

(cdadar capitals) returns '()

(car (cdadar capitals)) returns error

1. Code:

#lang racket

(define third (lambda (x) (caddr x)))

(define fourth (lambda (x) (cadddr x)))

(define fifth (lambda (x) (car(reverse x))))

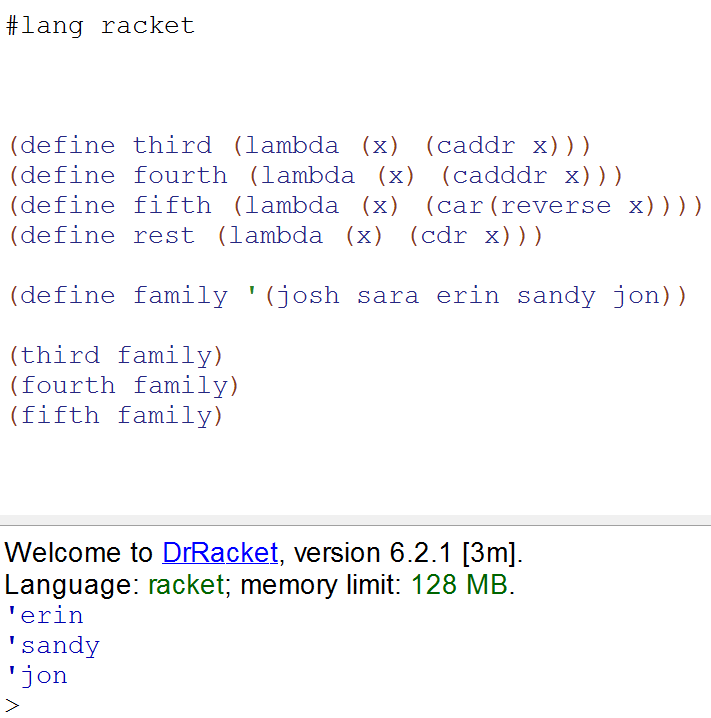
(define rest (lambda (x) (cdr x)))

(define family '(josh sara erin sandy jon))

(third family)

(fourth family)

(fifth family)



1. Code:

#lang racket

(define list '(#t #t #f #t #f))

(define (convert alist)

(map (lambda (x) (if (eq? x #T) 1 0)) alist))

(convert list)

(define (addit lis)

(define sum 0)

(map (lambda (x) (set! sum (+ sum x))) lis)

sum

)

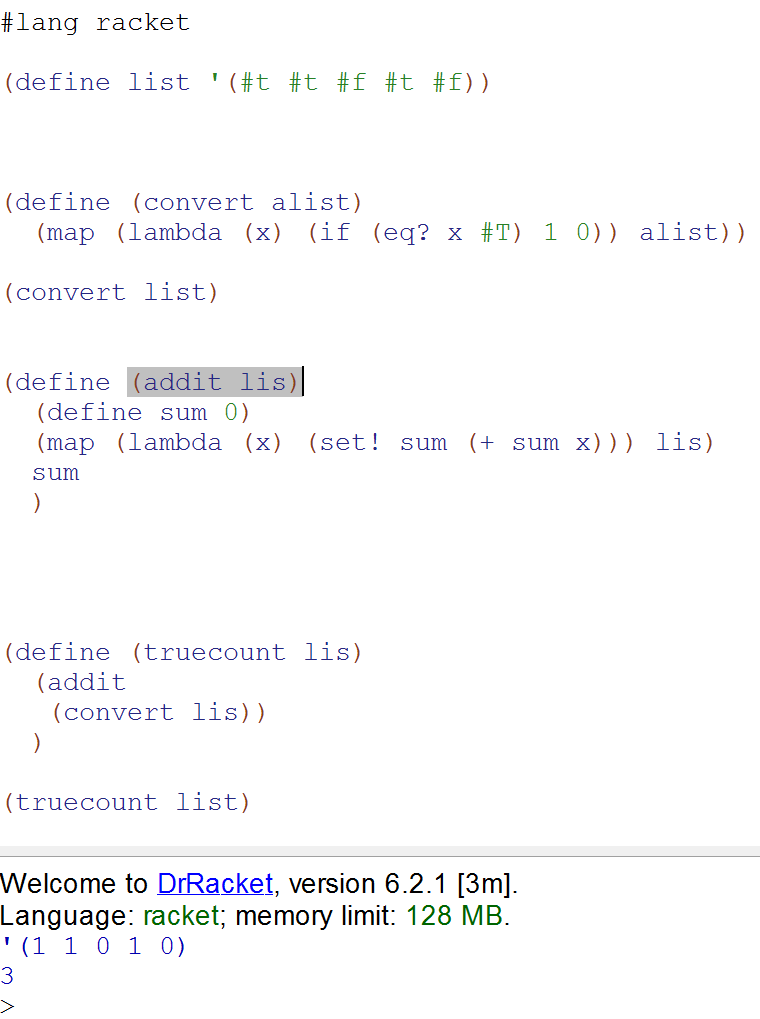
(define (truecount lis)

(addit

(convert lis))

)

(truecount list)



1. Code:

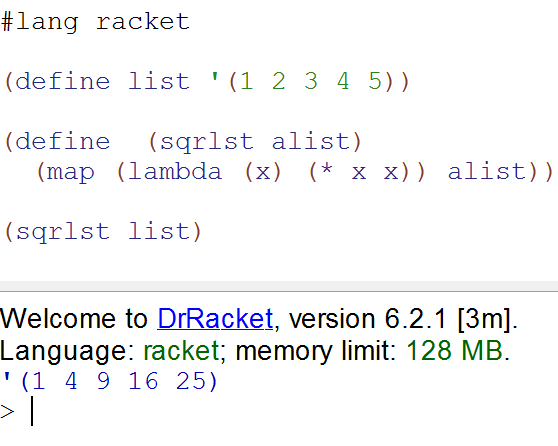
#lang racket

(define list '(1 2 3 4 5))

(define (sqrlst alist)

(map (lambda (x) (\* x x)) alist))

(sqrlst list)



1. Code:

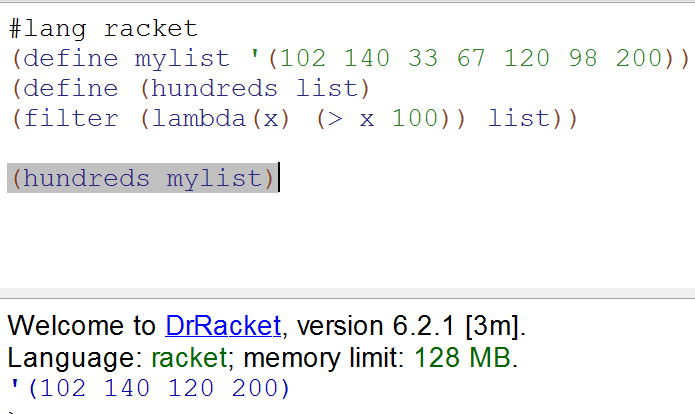
#lang racket

(define mylist '(102 140 33 67 120 98 200))

(define (hundreds list)

(filter (lambda(x) (> x 100)) list))

(hundreds mylist)



1. Code:

#lang racket

(define (collatz n)

(cond

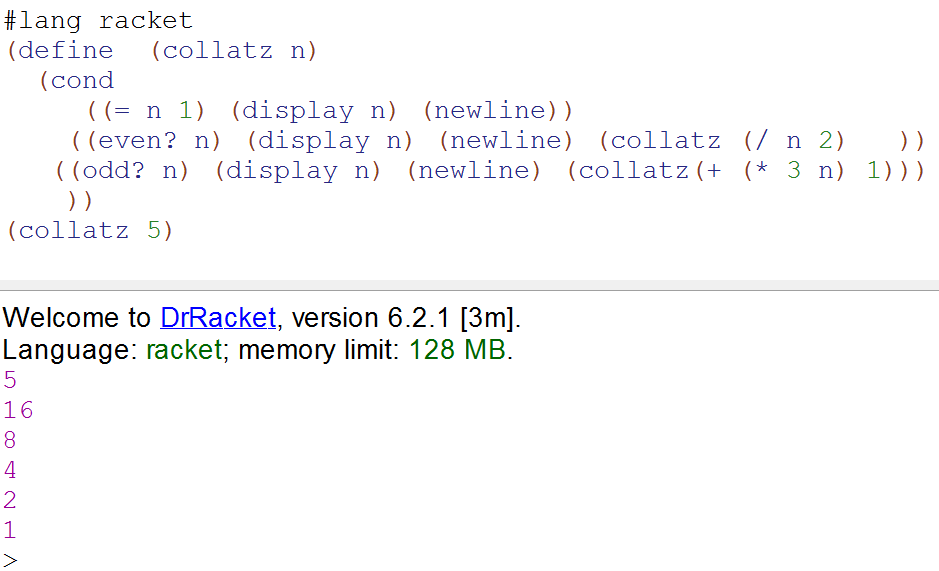
((= n 1) (display n) (newline))

((even? n) (display n) (newline) (collatz (/ n 2) ))

((odd? n) (display n) (newline) (collatz(+ (\* 3 n) 1)))

))

(collatz 5)



1. Code:

ang racket

(define (fizzbuzz n)

(continue n 1))

(define (continue n m)

(cond

((= n m) (newline) )

((and (eq? (modulo m 3) 0) (eq? (modulo m 5) 0)) (display "FizzBuzz") (newline)(continue n (+ m 1)) )

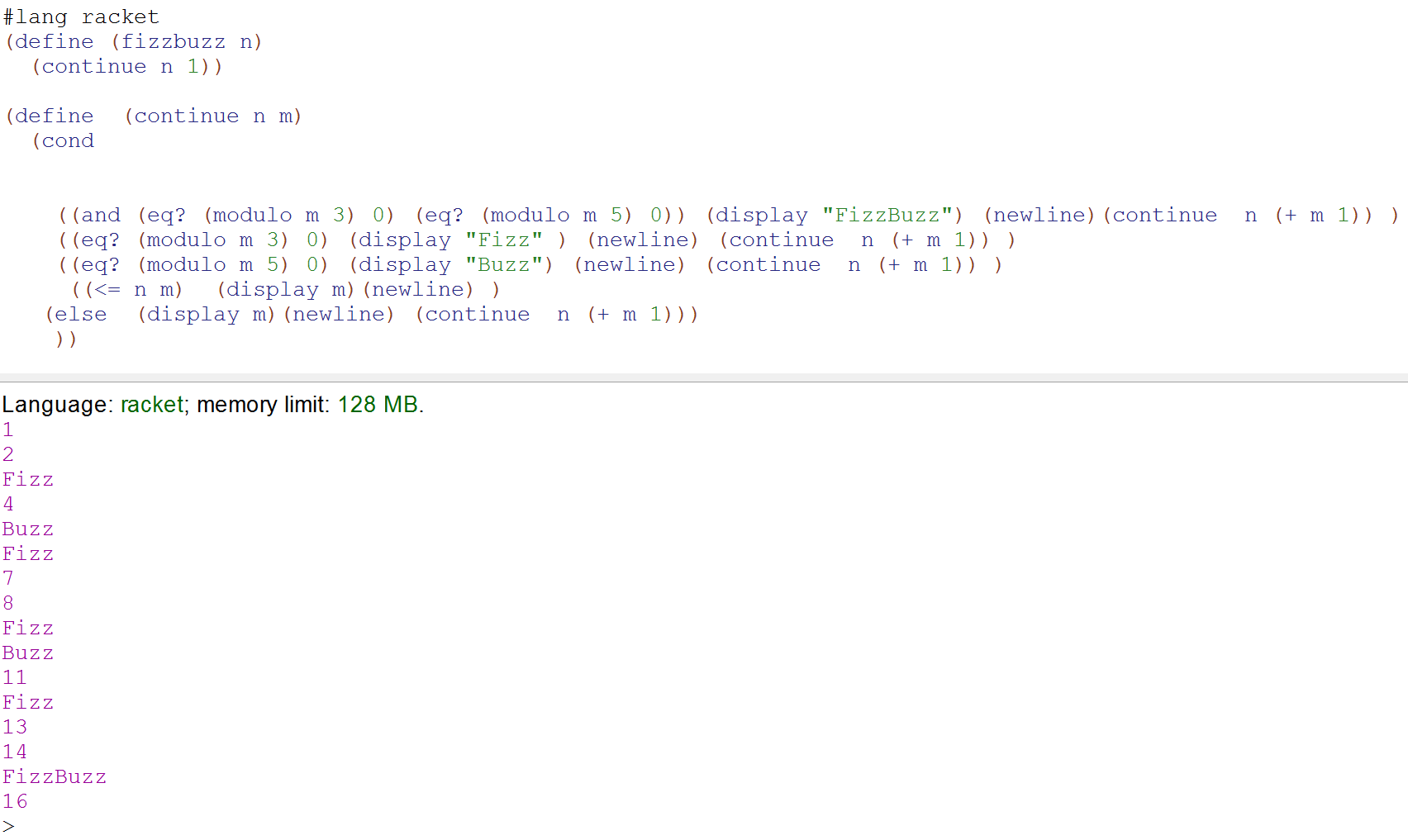
((eq? (modulo m 3) 0) (display "Fizz" ) (newline) (continue n (+ m 1)) )

((eq? (modulo m 5) 0) (display "Buzz") (newline) (continue n (+ m 1)) )

(else (display m)(newline) (continue n (+ m 1)))

))

(fizzbuzz 20)



**Scheme Program Language Math Translation**

**Code:**

#lang racket

(define chinese '(ling yi er san si wu liu qi ba jiu shi))

(define english '(zero one two three four five six seven eight nine ten))

(define testList '(one two liu jump wu) )

(define (translate x)

(cond

((or (eq? x 'ling) (eq? x 'zero) )0 )

((or (eq? x 'yi) (eq? x 'one) ) 1)

((or (eq? x 'er) (eq? x 'two)) 2)

((or (eq? x 'san) (eq? x 'three)) 3)

((or (eq? x 'si) (eq? x 'four)) 4)

((or (eq? x 'wu) (eq? x 'five)) 5)

((or (eq? x 'liu) (eq? x 'six)) 6)

((or (eq? x 'qi) (eq? x 'seven)) 7)

((or (eq? x 'ba) (eq? x 'eight)) 8)

((or (eq? x 'jiu) (eq? x 'nine)) 9)

((or (eq? x 'shi) (eq? x 'ten)) 10)

(else -1)

))

(define (ridBad2 list)

(filter positive? (map (lambda (x) (translate x)) list))

)

(define addText "Addition: ")

(define sum 0)

(define (showAdd list)

(cond

( (and (eq? (car list) null) (eq? (cdr list) null) ))

( (and (not(eq? (car list) null)) (eq? (cdr list) null) ) (set! sum (+ sum (car list))) (set! addText (string-append (string-append addText (number->string (car list))) " = ") ) (string-append addText (number->string sum)) (display addText) (display sum)(newline) )

( (not (eq? (car list) null) ) (set! addText (string-append (string-append addText (number->string (car list)))" + ")) (set! sum (+ sum (car list))) (showAdd (cdr list)) )

)

)

(define prodText "Multiplication: ")

(define prod 1)

(define (showProd list)

(cond

( (and (eq? (car list) null) (eq? (cdr list) null) ) )

( (and (not(eq? (car list) null)) (eq? (cdr list) null) ) (set! prod (\* prod (car list))) (set! prodText (string-append (string-append prodText (number->string (car list))) " = ") ) (string-append prodText (number->string sum)) (display prodText) (display prod)(newline) )

( (not (eq? (car list) null) ) (set! prodText (string-append (string-append prodText (number->string (car list)))" \* ")) (set! prod (\* prod (car list))) (showProd (cdr list)) )

)

)

(define (go lis)

(define transText "Translation: ")

(set! lis (ridBad2 lis))

(map (lambda (x) (set! transText (string-append (string-append transText (number->string x)) " "))) lis)

(display transText)(newline)

( showAdd lis)

(showProd lis)

)

(go testList)

