CS 550 Assignment 2 (Streams)

Due Wed. May 3, 2017

Name 1:\_\_\_\_\_\_\_\_\_\_\_\_\_\_Greg Matthews \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Drexel Username 1:\_\_\_\_\_gm453 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Name 2:\_\_\_\_\_\_\_\_\_\_\_\_\_Ke Yang\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Drexel Username 2:\_\_\_\_\_\_\_ky323\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Instructions: This assignment is to be done with your assigned group. Only one submission per group is required. Group members receive the same grade and it is the responsibility of the group to ensure that all members actively participate and that all group members understand the solution to all parts of the assignment.

This assignment explores the use and implementation of streams. In the first part you are to use Racket’s implementation of streams to perform the example from SICP that was discussed in class. In the second part, you are to modify mceval.rkt (metacircular interpreter for scheme) to support streams and test your implementation of streams by computing the same example from part 1. In the third part of the assignment, you are to use lazymceval.rkt (normal order implementation of the metacircular interpreter) to redo, yet again, the same example from part 1. Note that with lazy evaluation streams and lists are the same, though you will need to re-implement a non-strict version of cons. Rather than modify the interpreter to support non-strict primitives you can implement cons, car and cdr using functions as was done in the lambda calculus.

**Part 1.** Use Racket’s implementation of streams (require racket/stream) to compute and trace the computation (stream-first

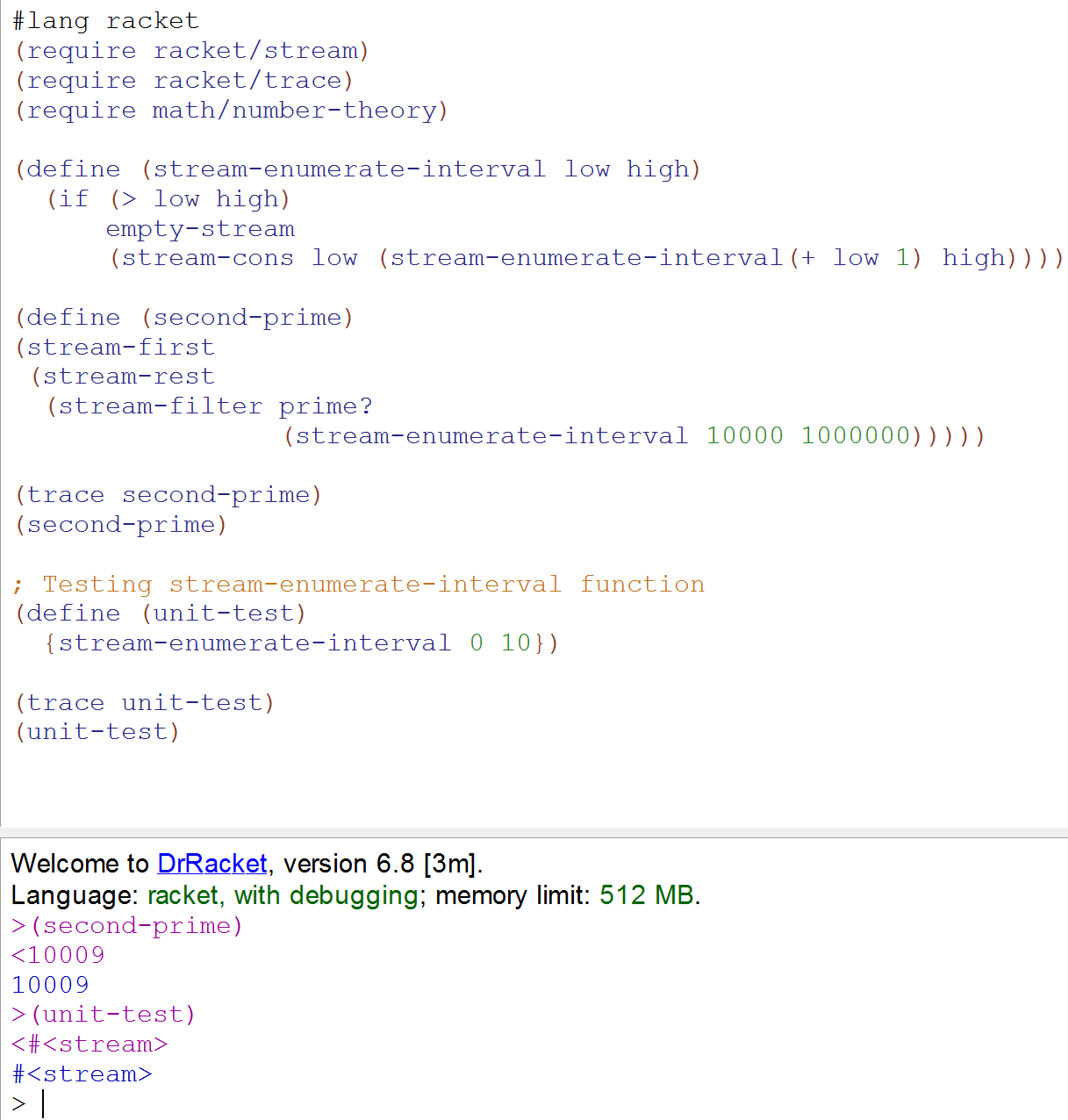
(stream-rest

(stream-filter prime?

(stream-enumerate-interval 10000 1000000))))

Verify that the correct answer is computed (use a unit test) and that only the necessary numbers are generated by stream-enumerate-interval (use the trace function from (require racket/trace) to trace the stream functions.

Note that Racket provides the prime? predicate in the math/number-theory package. Further note that the names of the stream functions in Racket differ from those used by SICP. You will need to refer to the documentation to find the appropriate function names. You will need to implement stream enumerate-interval.



**Part 2.** Modify mceval.rkt to support streams and redo the example from Part 1 using your implementation of streams. You will need the following functions (some will need to be special forms [which ones?], some can be added as primitive functions and some can be written using input to mceval [note that you can use eval-definition to add definitions in setup-environment].

1. force and delay
2. empty-stream, stream-empty?
3. stream-cons, stream-first, stream-rest 4. stream-filter, prime?

Your solution must use your own implementation of force and delay [you may NOT use these functions from Racket]. Use thunks as discussed in class and in SICP to implement force and delay. You should also use memorization to avoid forcing a function multiple times. A test file with unit tests for force and delay and the primitive stream functions is provided.

**1) Stream-cons and delay (normal and memoried version) all need to be implemented as special form. Other function can be defined normally.**

Special forms defined for stream-cons and delay in the mceval function:

(define (mceval exp env)

(cond ((self-evaluating? exp) exp)

((variable? exp) (lookup-variable-value exp env))

…….

**((delay? exp) (mceval (delay->app exp) env))**

**((stream-cons? exp) (mceval (stream-cons->app exp) env))**

……..

(else

(error "Unknown expression type -- EVAL" exp))))

**2) Definition of all the functions:**

(define (**stream-first stream**) (car stream))

(define (**stream-rest stream**) (force (cdr stream)))

(define **empty-stream** '())

(define (**stream-empty?** stream)(null? stream))

(define (**force** exp)(exp))

(define (**delay?** exp) (tagged-list? exp 'delay))

(define (**delay->app** exp)(list 'memo-exe (list 'lambda '() (second exp))))

(define (**stream-cons?** exp) (tagged-list? exp 'stream-cons))

(define (**stream-cons->app** exp) (list 'cons (second exp) (delay->app (list 'delay (third exp)))))

(define (**memo-exe** exp) ;; function to memorize the previous state if has been computed before

(let ((run-previous? false)

(result '()))

(lambda ()

(if (not run-previous?)

(begin (set! result (exp))

(set! run-previous? true)

result)

result))))

**3) Add all the definition besides special form into environment using eval-definition:**

(define (setup-environment)

(let ((initial-env

(extend-environment (primitive-procedure-names)

(primitive-procedure-objects)

the-empty-environment)))

…..

(eval-definition '(define **empty-stream** '()) initial-env)

(eval-definition '(define (**force** exp)(exp)) initial-env)

(eval-definition '(define (**stream-empty?** stream) (null? stream)) initial-env)

(eval-definition '(define (**stream-first stream**) (car stream)) initial-env)

(eval-definition '(define (**stream-rest stream**) (force (cdr stream))) initial-env)

(eval-definition '(define (**stream-filter pred stream**)

(cond ((stream-empty? stream) empty-stream)

((pred (stream-first stream))

(stream-cons (stream-first stream)

(stream-filter pred

(stream-rest stream))))

(else (stream-filter pred (stream-rest stream))))) initial-env)

(eval-definition '(define (**stream-enumerate-interval** low high)

(if (> low high)

empty-stream

(stream-cons low (stream-enumerate-interval(+ low 1) high)))) initial-env)

(eval-definition '(define (**second-prime**)

(stream-first

(stream-rest

(stream-filter prime?

(stream-enumerate-interval 10000 1000000))))) initial-env)

(eval-definition '(define (**memo-exe exp**)

(let ((run-previous? false)

(result '()))

(lambda ()

(if (not run-previous?)

(begin (set! result (exp))

(set! run-previous? true)

result)

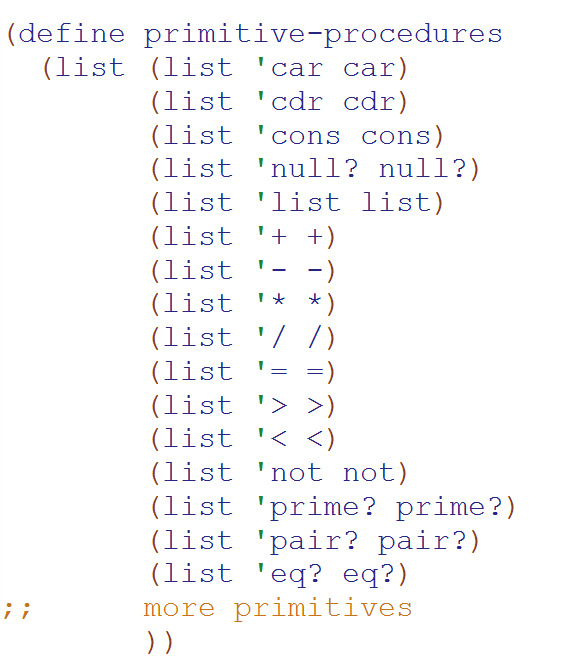
result))))

initial-env)

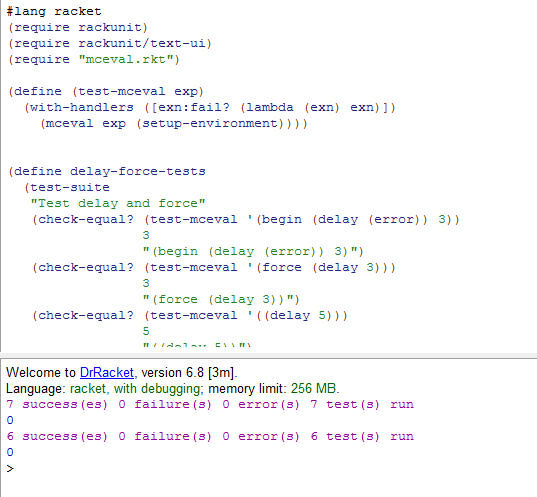
……..

initial-env))

**4) Add the necessary part into primitives:**



**5) Run the test2 file provided for testing all functions and memorization part:**



**Part 3.** In the lazy (normal order) version of the metacicular interpreter lists and streams are essentially the same thing (except lazy lists are lazy in both the car and cdr fields whereas streams are lazy only in the cdr field). Also the primitive function cons is strict since it just calls the built-in cons. To use lazy lists you must either provide a non-strict cons or use alternative cons, car and cdr functions implemented by mceval. This can be done by representing pairs by functions as was done in the lambda calculus.

(define (cons x y)

(lambda (m) (m x y)))

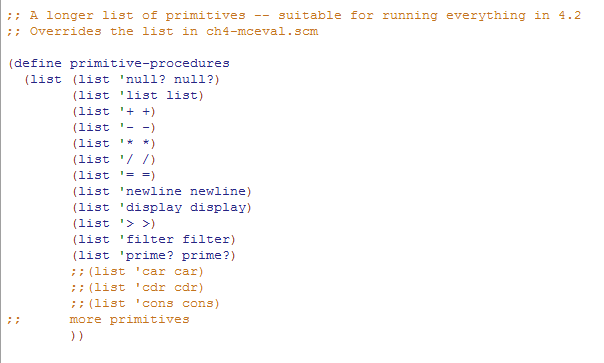
(define (car z)

(z (lambda (p q) p))) (define (cdr z)

(z (lambda (p q) q)))

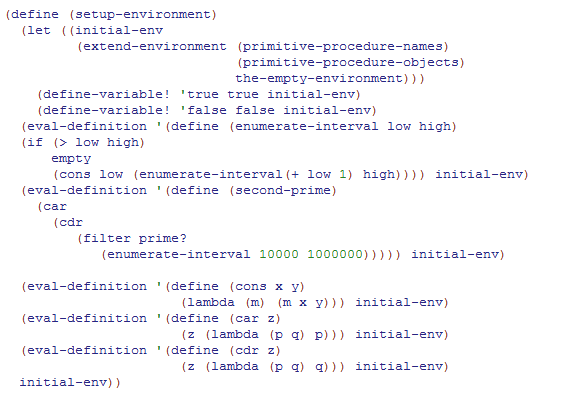
In this part of the assignment you are to redo the stream example from Part 1 using lists and the above implementation of cons, car and cdr.

**1) Comment out the primitive cons, car and cdr:**



**2) Add i) cons, car and cdr lambda function definition into lazymceval**

**ii) list version of part1 into lazymceval**



**3) Run the trace for second-prime using lazy interpreter:**

