1 Instructions of the Usual but Important Kind

Answer all questions. Unless we feel there is some ambiguity, we are not writing “What to turn in” sections on this assignment because we think it is obvious that you should answer the questions and turn in your answers. If you have questions about what to turn in please ask us.

Problem sets 1-3 were composed of a small number of fairly long problems. This problem set has a different style, and contains a larger number of relatively short problems.

You should write up and print out your solutions to this assignment, and hand them in as per turnin.html on the course materials page.

Cheating is a very serious issue, and we take it as such; please read over the Course Information and the Duke Honor Code.

You must write and sign a pledge on your assignment, that you acted honestly in completing the assignment.

2 Instructions of the Technical and Important Kind

.i) Warning: Do not use stream-iterate on this assignment. Such solutions will get 0 credit.

.ii) Are streams in Racket memoized? By the end of this problem set you should be able to conclude whether or not they are. But we want to make this problem set easier. Therefore, you do not, and should not, use or assume memoization in your analyses, proofs, or code. And you don’t need it for this assignment.¹

3 Assignment

Read the handout for this problem set, “Famous Mathematical Sequences and Series.”

3.1 Part One: Combinatorics and Infinite Series

1. The following stream represents an infinite series:

   (define fact (mul-streams integers (stream-cons 1 fact)))

¹It would be interesting to have another problem set where we ask all the questions in Section 3.1 assuming an implementation of Scheme in which streams are memoized. Maybe some day we will do that. But today, we are not asking for that, and since some of the answers could be different, you should not consider memoization.
where, as usual,

(define ones (stream-cons 1 ones))
(define integers (stream-cons 1 (add-streams ones integers)))

(define mul-streams
  (lambda ((a <stream>) (b <stream>))
    (cond ((stream-empty? a) b)
          ((stream-empty? b) a)
          (else (stream-cons (* (stream-first a) (stream-first b))
                              (mul-streams (stream-rest a) (stream-rest b)))))))

(a) What is the value of (last (stream->listn fact 20))?
(b) What is the value of (last (stream->listn fact 1001))?
(c) Now consider the following function:

(define factorial
  (lambda (n)
    (if (< n 2) 1
        (* n (factorial (- n 1))))))

(d) What is the value of (factorial 20)?
(e) Prove that (factorial n) = (last (stream->listn fact n)) for any natural number n > 0.

2. The following stream represents an infinite series:

(define fibs
  (stream-cons 0
              (stream-cons 1
                           (add-streams fibs (stream-rest fibs)))))

(a) What is the value of (last (stream->listn fibs 21))?
(b) What is the value of (last (stream->listn fibs 1001))?
(c) Now consider the following function:

(define fibonacci
  (lambda ((n <integer>))
    (if (< n 2)
        n
        (+ (fibonacci (- n 1))
           (fibonacci (- n 2))))))
(d) What is the value of (fibonacci 20)?

(e) Prove that (fibonacci n) = (last (stream->listn (fibs n + 1))) for any natural number n > 0.
   Comment: You may write (last (stream->listn (fibs n+1))) as (last (stream->listn (fibs (+ n 1)))) if you prefer.

(f) What happens when you try to evaluate (fibonacci 1000) in Racket? Describe what you observe.

(g) In general, as n grows large, which is faster, (fibonacci n) or (last (stream->listn (fibs n + 1)))?

(h) Why? Justify your answer.

   Hint: Derive the running time using $O(\cdot)$ notation for evaluating both expressions, (fibonacci n) and (last (stream->listn (fibs n + 1))). Write them both down. Justify your answers. Then compare them.

3. Triangular and Hexagonal Number Series.

   Read the handout for this problem set, “Famous Mathematical Sequences and Series” again.

   (a) Replace the “...” below with appropriate Scheme code to define an infinite stream representing the triangular numbers starting at 3. We do not want a solution that uses mul-streams, map-stream, scale-stream, stream-iterate, or map-streams. We want a solution with recursive streams. The only functions you may use are add-streams and stream-cons. You may use the streams ones and integers:

   (define triangular ...)

   Run your code to obtain:

   > (stream->listn triangular 20)
   ⟨3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 210 231⟩

   Turn in your code and also the results of testing it as specified above. You may turn in additional tests of your code if you think it is necessary to demonstrate that it works.
   Do not define any functions other than add-streams. Do not use anonymous lambda. Do not define any variables other than triangular. Do not use any explicit iteration or looping (e.g. letrec).
   Hint: use ones, integers, add-streams, and stream-cons. You do not need to use any other streams or functions other than these.
   Hint: The answer to this question is quite short. If you wrote a long complicated answer it is probably wrong, and even if it works, we will deduct points for unnecessary complexity. Search for a fairly short answer. We could do it in three lines of beautifully formatted readable code.
(b) Replace the “…” below with appropriate Scheme code to define an infinite stream representing the hexagonal numbers starting at 6. We do not want a solution that uses mul-streams, map-stream, stream-iterate, or map-streams. We want a solution with recursive streams. The only functions you may use are add-streams, scale-stream and stream-cons. You may use the streams ones and integers:

```
(define hexagonal ...)
```

Run your code to obtain:

```
> (stream->listn hexagonal 20)

(6 15 28 45 66 91 120 153 190 231 276 325 378 435 496 561 630 703 780 861)
```

Turn in your code and also the results of testing it as specified above. You may turn in additional tests of your code if you think it is necessary to demonstrate that it works.

Do not define any functions other than add-streams and scale-stream. Do not use anonymous lambda. Do not define any variables other than hexagonal. Do Not use any explicit iteration or looping (e.g. letrec).

Hint: use ones, integers, add-streams, scale-stream, and stream-cons. You do not need to use any other streams or functions other than these.

Hint: The answer to this question is quite short. If you wrote a long complicated answer it is probably wrong, and even if it works, we will deduct points for unnecessary complexity. Search for a fairly short answer. We could do it in three lines of beautifully formatted readable code.

(c) Give at least two natural numbers greater than 5 that are both triangular and hexagonal.

(d) Replace the “…” below with appropriate Scheme code to define an infinite stream representing the natural numbers greater than 5 that are both triangular and hexagonal:

```
(define triangular-and-hexagonal ...)
```

Turn in your code and also the results of testing it. You may turn in additional tests of your code if you think it is necessary to demonstrate that it works.

(e) In the handout for this problem set, “Famous Mathematical Sequences and Series” precise mathematical expressions are given for the triangular and hexagonal sequences.

Write down a mathematical expression for the sequence of numbers that are both triangular and hexagonal (the previous problem, above).

(f) Prove that your mathematical expression (immediately above) is correct.

(g) Replace the “…” below with appropriate Scheme code to define an infinite stream representing the natural numbers greater than 9 that are triangular but not hexagonal:

```
(define triangular-not-hexagonal ...)
```

Turn in your code and also the results of testing it. You may turn in additional tests of your code if you think it is necessary to demonstrate that it works.
3.2 Part Two: Logic and Computability

1. You have seen in lecture a proof that the Halting Problem is undecidable. This proof hinged on showing that the predicate safe? cannot exist, where safe? is a hypothetical Scheme predicate that we wanted to have the following properties:

(safe? prog arg) returns #t if the function call (prog arg) halts with an answer, and returns #f if it doesn’t.

After thinking about this for a while, you wonder if perhaps there is a simpler version of the Halting problem that is decidable. For example, maybe, you think, it is possible to decide for functions of no arguments. A function with no arguments is called a thunk.

Consider this (arguably simpler) version of the Halting problem. Namely, consider whether there exist a function:

(simple-safe? prog) that returns #t if the function call (prog) halts with an answer, and returns #f if it doesn’t.

Either prove that simple-safe? exists, or, prove that it does not.

2. After thinking about this more, you have another idea. You wonder if perhaps there is something in the middle, “between” the Halting Problem we discussed in class, and the “simpler” version immediately above.

You wonder if there is middle version of the Halting problem that is decidable. For example, maybe, you think, it is possible to decide when a given function is called on a specific, fixed argument, namely, the argument “1” (an integer).

For example, you reason, perhaps a function analogus to safe?, called one-safe? could be written for this purpose. And maybe one-safe? could be written to take advantage of the fact that in our queries, the argument to prog will always be 1. In other words, asking “Will prog halt on 1” is a very specific question: perhaps it could be solved.

On the other hand, you are cautious, after hearing Prof. Donald’s lecture on “Computability and the Halting Theorem.” So you want to know whether or not one-safe? is computable or not.

Consider this “Middle Difficulty” version of the Halting problem. Namely, consider whether there exist a function:

(one-safe? prog) that returns #t if the function call (prog 1) halts with an answer, and returns #f if it doesn’t.

Here, of course, “1” is the integer 1 in Scheme that you are familiar with.

Either prove that one-safe? exists, or, prove that it does not.

---

2This would be a good time to go back and study that lecture.
3. Suppose we are writing an interpreter or a compiler.

For example, suppose we have in the compiler:

```scheme
(let* ((t1 (f))
       (t2 (g))
       (t3 (+ t1 t2)))
  t3)
```

Now we’d like to optimize: if \(f\) and \(g\) compute the same value, then we can optimize out the call to \(g\) (ie, bind \(t2\) to \(t1\)).

So how can we tell if two functions \(f\), and \(g\), compute the same value? Sounds easy, and various cases seem pretty straightforward:

```scheme
(lambda () 3) \text{vs.} \ (\lambda () (+ 1 2)).
```

Can we do it in general?

Suppose there exists a function \(\text{Equiv?}\) that decides whether or not two functions are equivalent.

That is, suppose \(g\) and \(h\) are two Scheme functions. Then

\((\text{Equiv?} \ g \ h)\) returns \#t iff \(g = h\).

Note that \(g = h\) iff \((g \ x) = (h \ x)\) for every input \(x\).

\(\text{Equiv?}\) could be very useful, for example, in our compiler optimization:

```scheme
(\text{Equiv?} \ (\lambda (x) (+ x 3)) \ (\lambda (y) (+ y (/ (sqrt 9) (+ 1 2 2 -4)))))
```

\[\Rightarrow \#t\]

Suppose you are asked to write a Scheme function \(\text{Equiv?}\) that works for any pair of procedures of one argument \(g\), and \(h\) (not just for the example above).

In your answer, assume that both functions \(g\) and \(h\) passed to \(\text{Equiv?}\) are functions of exactly one argument.

Does \(\text{Equiv?}\) exist? Prove your answer. Either (a) write \(\text{Equiv?}\) in Scheme and show that it is correct, or, (b) prove that \(\text{Equiv?}\) does not exist.
### 3.3 Extra Credit

Are streams in Racket memoized? Justify your answer.

Hint: What evidence from this PS can you use? Can you design an experiment to test your hypothesis?

Grading:

No answer(blank) (0 points).

Yes/No answers with no justification will be graded as follows: correct (+1 point); incorrect (-1 point).

Yes/No answers with a justification will be graded as follows: on a 0 to +5 point scale. A good justification, which can be brief, is necessary to receive full extra credit (+5 points). A good justification can settle this question definitively so that no doubt remains.