CS 230 Discrete Math for Computer Science — Eddy Lin (ELL22) and Nathan Kim (NJK24)
Duke University — 1 November 2019

Problem Set 4

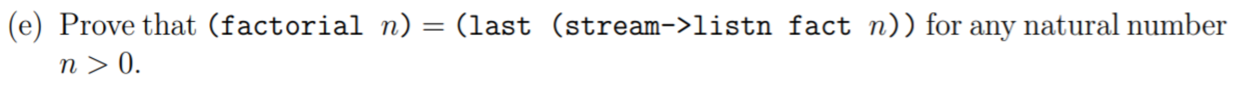
3.1 Part One: Combinatorics and Infinite Series

Problem 1

a) The value is 2432902008176640000, which is equal to 20!
b) The value is equal to 1001!, but is the following number:

```
402789647337170867317246136356926989705094239074925347176343710340368450911027649612636252695456374205280468598807393254690298539867803
367460225153499614535588421928591160833678742451354915921252299285456946271396995850437959540645019696372741142787347450281325324373824
456300226871609431497826989489109522725791691167945698509282421538632966523376679891823696900982075223188279465194065489111498586522997
573307838057934994706212934291477882221464914058745808179795130018969175605739824237247684512790169648013778158661520384916357285547219
660337504067910087936301580874662367543921288988208261944834178369169805682489420504038334529389177845089679546075023305854006141256288
633820079940395329251563788399404652902154519302928365169452383531030755684578503851488154092323576150311569325891190105926118761607100
286827930472944913272420825078912158741589850136017030887975452922434889688775883386977825215904423682478943313806072144097432418695807
412571292308739802481089407002523955080148184062810447564594783139830113821372260474145316521647368313934670783858482781506915288378941
348078689691815657785305896912277993200639858696294199549107738635599538328374931258525869323348477334798827676297868823693023377418942
304272267800509765805435653787530370118261219994752588866451072715583785495394684524593296728611334955079882857173250037068541860372512
693170819259309411027837176612444692649174536429745421086287708588130082168792750697158901737130221751430550976429258055277255676893874
108456870904122902259417224707137723406125811549952159629766771063079472679280213882978523785424760309678138268708239764925768714349554
665438389311198715040908077757086900159389712443987670244241787904585093011546861502058550090914877900852701619648229332192401075747543
562989953271508977501771085759521631427816116191761031257454497039673414248149210836002497114107565960458576525212556159634975715552638
678172137468172843066451093984443636560722213668172225585711566558134467392654185460222589723312097599987253417831473939565071006344352
518096564427781204200068323913056897090916602712260306869786107237077757244586657294576097772163940833843000997602897053915082236553856
613962747814621747092348996915755983464741082000337526945990059365493439921937093368896754791416759604324895514660325913157843796039917
819613717350380997781225472000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000
```

d) The value is 2432902008176640000, which is equal to 20!



(e) Prove that (`factorial` n) = (`last` (`stream->listn fact` n)) for any natural number $n > 0$.

e)

Claim: `(factorial n) = (last (stream->listn fact n))` for any natural number *n* > 0.

Proof: Let P(n) be the claim that `(factorial n) = (last (stream->listn fact n))` = n! for any natural number *n* > 0. Proof by induction on n. Assume that the helping functions mul-streams, integers, etc. used below are correctly defined.

Base Case:

For n=1, use substitution model to evaluate and compare `(factorial n)` & `(last (stream->listn fact n))`.

(factorial {1})
[PROC {1}]
(if (< {1} {2}) 1 …)
(< {1} {2}) is #t, so evaluate first argument afterwards.
{1}
(factorial 1) evaluates to 1

(last (stream->listn fact 1))
(stream->listn fact 1)
[PROC {fact} {1}]
(stream-first fact) => {1}
(last {1})

{1}
(last (stream->listn fact 1)) evaluates to 1
Thus, `(factorial n) = (last (stream->listn fact n))`, so P(1) is correct.)

Induction Hypothesis:
For a fixed integer n, where 1<k<n, assume that P(k) is true, where `(factorial k)` & `(last (stream->listn fact k))`.

Induction step:
NOS that P(n) is true. Determine if `(factorial n) = (last (stream->listn fact n))`. Using SM,
(factorial {n})
(if (< n 2) … …)
(< {n} {2}) evaluates to #f, because n>k>1. Evaluate second statement
(* {n} (factorial (- n 1))
(* {n} (factorial {n-1})) Because IH, (factorial {n-1}) correctly returns (n-1)!
(* {n} {(n-1)!})
{n!}

Similarly, compare this to (last (stream->listn fact n)). Using SM and the contract of last and stream->listn, this is equivalent to the nth element of the stream fact, which is equivalent to
(* {n} (last (stream->listn fact {n-1}))). From IH, the second argument of the multiplication expression is equal to (n-1)!
(* {n} {(n-1)!})
{n!}

In both cases, `(factorial n) = (last (stream->listn fact n))` = n!.
QED

Problem 2

a) The value is 6765

b) The value is equal to
434665576869374564356885276750406258025646605173717804024817290895365554179490518904
038798400792551692959225930803226347752096896232398733224711616429964409065331879382
98969649928516003704476137795166849228875

d) The value is 6765

e)

Claim: `(fibonacci n) = (last (stream->listn fact (+ n 1)))` for any natural number *n* > 0.

Proof: Let P(n) be the claim that `(fibonacci n) = (last (stream->listn fact (+ n 1)))` = the correct nth fibonacci number fib(n) starting with fib(0) = 0. for any natural number *n* > 0. Proof by induction on n. Assume that the helping functions add-streams, etc. used below are correctly defined.

Base Case:

For n=1, use substitution model to evaluate and compare `(fibonacci n)` & `(last (stream->listn fact (+ n 1)))`

(fibonacci {1})
[PROC {1}]
(if (< {1} {2}) {1} …)
(< {1} {2}) is #t, so evaluate first argument afterwards.
{1}
 (fibonacci 1) evaluates to 1

(last (stream->listn fibs 2))
(stream->listn fibs 2)
[PROC {fibs} {2}]
 (stream-cons {1} {1})) => `(1, 1)
(last `(1, 1))
{1}
 (last (stream->listn fibs 2)) evaluates to 1

Thus, `(fibonacci n) = (last (stream->listn fibs n))`, so P(1) is correct.

Induction Hypothesis:

For a fixed integer n, where 1<k<n, assume that P(k) is true, `(fibonacci k)` & `(last (stream->listn fact (+ k 1)))`.

Induction step:

NOS that P(n) is true. Determine if `(fibonacci n) = (last (stream->listn fibs n))`. Using SM,
(fibonacci {n})
(if (< n 2) … …)
({n} {2}) evaluates to #f, because n>k>1. Evaluate second statement
(+ (fibonacci (- n 1)) (fibonacci (- n 2)))
(+ (fibonacci {n-1}) (fibonacci {n-2})). From IH, this correctly sums up the previous two fibonacci numbers, so this returns the correct fib(n) from the definition.
{fib(n)}

Similarly, compare this to (last (stream->listn fibs {n+1})). Using SM and the contract of last and stream->listn, this is equivalent to the nth element of the stream fact, which, from its definition, is equivalent to
(+ (last (stream->listn fibs {n})) (last (stream->listn fibs {n-1})))
From IH, this correctly sums up the previous two fibonacci numbers, so this returns the correct fib(n) from the definition.

In both cases, `(fibonacci n) = (last (stream->listn fibs n)) =` fib(n).
QED

f) The program runs continuously for a long time without arriving at the answer.
g) In practice, as n grows large, (last (stream->listn fibs n + 1)) is much faster. However, assuming that streams are not memoized in Scheme, the runtime should be the same order $O(2^n)$.
h) We can analyse the recurrence relation T(n) of a call (fibonacci n):

   $T(0) = O(1)$
   $T(n) = T(n-1) + T(n-2) + O(1)$, which is approximated as
   $T(n) = 2T(n-1) + 1$
   This is equivalent to:
   $T(n) = 2(2(T(n-2)) + 1) + 1$
   $T(n) = 4(T(n-2)) + 2 + 1$
   $T(n) = 8(T(n-3)) + 4 + 2 + 1$
   After k steps, this is
   $T(n) = 2^k(T(n-k)) + 2^{k+1} - 1$
   Let k = n
   $T(n) = 2^n(T(0)) + 2^{n+1} - 1$
   Thus, $T(n) = O(2^n)$

Similarly, for streams (assuming no memoization), the runtime to get each value in the stream fibs, it will have to do the equivalent calculation, having to make two recursive calls. Again, this is $O(2^n)$ for each call, where n is the position of the element in the stream. Thus, the runtime is $O(2^0)+O(2^1)+O(2^2)+...+O(2^n) = O(2^{n+1}) = O(2^n)$.

So, both of the implementations for calculating the 1000 index fibonacci number are the same runtime order of growth!

Problem 3

a)

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

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

b)

```
(define hexagonal
  (stream-cons 6 (add-streams (add-streams (scale-stream ones 5)
(scale-stream integers 4)) hexagonal)))

(stream->listn hexagonal 20)
; ==> (6 15 28 45 66 91 120 153 190 231 276 325 378 435 496 561 630
703 780 861)
```

c)

6, 15, 28, 45, 66…

d)

The numbers which are hexagonal are also triangular. Thus, the same expression for hexagonal numbers,

$$h_n = 2n^2 - n = n(2n-1) = \frac{2n \times (2n-1)}{2}.$$

is the same expression for triangular and hexagonal numbers

f)
Claim:

The expression $h_n = \frac{2n*(2n-1)}{2}$ represents all triangular and hexagonal numbers, where $h_n = T(n)$ represents the nth element of the hexagonal numbers.

Proof:

From the handout, the triangular numbers are defined as follows:

$$T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

, where t_n = T(n) represents the nth element of the hexagonal numbers. If we can show that ALL of the hexagonal numbers are also triangular numbers, then we can say that all of the hexagonal numbers represent the intersection between triangular and hexagonal numbers.

Thus, NOS that T(n)=H(m) for some integer *n, m*.

$\frac{n(n+1)}{2} = \frac{2m*(2m-1)}{2}$

$n(n+1) = 2m(2m-1)$. From this, we get

$n = 2m - 1$, so

T(2m-1) = H(m).

Because H(m) encompasses all the integer indexes of the hexagonal numbers, it includes all of the hexagonal numbers, which each corresponds to a triangular number.

QED.

g)

```
(define triangular-not-hexagonal
  (stream-cons 10 (add-streams (add-streams (scale-stream ones 7)
(scale-stream integers 4)) triangular-not-hexagonal)))

(stream->listn triangular-not-hexagonal 20)
; ==> (10 21 36 55 78 105 136 171 210 253 300 351 406 465 528 595
666 741 820 903)
```

3.2 Part Two: Logic and Computability

Question 1
Claim: For a procedure call `(simple-safe? prog)`, it is impossible to write a procedure that correctly determines whether the "prog" procedure with no arguments will halt.

Proof: first, assume that *simple-safe?* is implemented and correctly returns #t halts with an answer, and #f otherwise. Thus, a program like the following would work:

```
(define forever (lambda x) x)

(define contradiction
        (if (simple-safe? contradiction)
                (forever forever)
                `halt))
```

If *simple-safe?* is computable and correct, it must be that *contradiction* halts because the second computation, (forever forever), will only run if *simple-safe?* on *contradiction* returns #t. Thus, *contradiction* must always halt with an answer. Thus,

Now, let us compute `(forever forever)` using substitution model.

```
(forever forever)
(if (simple-safe? (forever forever))
(simple-safe? (forever forever)) : returns #t, as stated above. Evaluate next
expression:
(not (forever forever))
```

Thus, (forever forever) = (not (forever forever)). This is a contradiction, proving that *simple-safe?* cannot possibly be correctly implemented!

Question 2
Claim: For a procedure call `(one-safe? prog)`, it is impossible to write a procedure that correctly determines whether the "prog" procedure, with the argument 1, will halt.

Proof: first, assume that *one-safe?* is implemented and correctly returns #t halts with an answer, and #f otherwise. Thus, a program like the following would work:

```
(define forever (lambda x) x)

(define contradiction-1
    (lambda (x)
       (if (one-safe? contradiction-1)
              (forever forever)
              'halt))
```

If *one-safe?* is computable and correct, it must be that *contradiction-1* halts because the second computation, (forever forever), will only run if *one-safe?* on *contradiction-1* returns #t. Thus, *contradiction-1* must always halt with an answer. Thus,

Again, let us compute `(forever forever)` using substitution model.

```
(forever forever)
(if (simple-safe? (forever forever))
(simple-safe? (forever forever)) : returns #t, as stated above. Evaluate next
expression:
(not (forever forever))
```

Thus, (forever forever) = (not (forever forever)). This is a contradiction, proving that *one-safe?* cannot possibly be correctly implemented!

Question 3

It is not possible to write a function that checks whether or not two functions are equivalent. This is due to the fact that functions are units of abstraction, meaning they are created arbitrarily in order to perform a specific task. Although two functions may perform the exact same task, they are not implemented in exactly the same way. Here is the proof:

Claim: For a procedure call `(equiv? func1 func2)`, it is impossible to write a procedure that correctly determines whether the func1 procedure and the func2 procedure return the same value given the same parameter.

Proof: first, assume that *equiv?* is implemented and correctly returns #t if func1 and func2 are equivalent functions and #f otherwise. Thus, a program like the following would work:

```
(define funky1
  (lambda (x) 1))


(define funky2
    (lambda (x)
       (if (equiv? funky1 funky2)
              0
              (funky1 1)))
```

If e*quiv?* is computable and correct, it must be that funky2 outputs 0 when (equiv? funky1 and funky2) correctly outputs #t. In this case, funky2 will return 0, which is not equivalent to funky1 (which returns 1 always), so there is a contradiction.

In the other case, where (equiv? funky1 funky2) correctly returns #f, funky2 evaluates (funky1 1), which returns 1. This is equivalent to the output funky1, so funky1 and funky2 are equivalent! This, again, is a contradiction!

Because all cases of (equiv? funky1 funky2) result in a contradiction, equiv? cannot possibly exist and be correct.

QED

Extra Credit
Streams in Racket ARE memoized!

Justification:
In general, as n grows large, (last (stream->listn fibs n + 1)) is MUCH faster than its non-memoized counterpart, (fibonacci n).

As the example (fibonacci 1000) and (last (stream->listn fibs 1001)) shows, the stream implementation is extremely quick (almost instantaneous), while the (fibonacci 1000) runs without arriving at an answer for a long time. This is likely because a call to (fibonacci n) has to make two recursive calls to (fibonacci (- n 1)) and (fibonacci (- n 2)), and each of these calls has to make two recursive calls. This is significantly slower than the stream implementation, which uses stream addition to add the two streams together, which takes an amount of time linearly proportional to the number n.

(fibonacci n) runs with runtime $O(2^n)$, as we have shown in Part 3.1, problem 2. However, in practice, (last (stream->listn fibs n+1)) runs with a much faster runtime than (fibonacci n), of runtime $O(2^n)$. If streams in Scheme are memoized, the runtime of (last (stream->listn fibs n+1) runs with runtime O(n).

This is because, the way add-streams is implemented, each step to create the next value in the stream is adding up the previous two values already found (using memoization), which is an O(1) operation. Thus, this O(1) operation occurs O(n) times while iterating through stream->listn, so O(1)*O(n)=O(n).

Thus, (fibonacci n) is about $O(2^n)/O(n) = O(2^n/n)$ times faster than (last (stream->listn fibs (+ n 1))), which is well-evidenced by the fact that running (last stream->listn fibs 1001)) is almost instantaneous, while (fibonacci 1000) does not halt, for a long time. Thus, we can say that there is good evidence that memoization exists for streams in Scheme!