Some Problems from HW5 – concerning higher order functions.

A&S 1.29 (Implementing Simpson's Rule)

The text states: "Using Simpson's Rule the integral of a function f between a and b is approximated as

$$\frac{h}{3}\left[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n\right]$$

where $h = \frac{(b-a)}{n}$ for some even integer n, and $y_k = f(a+kh)$

We are asked to code this.

Simpson's Rule uses parabolas instead of rectangles to approximate the integral

```
;assumes that n is even
(define (simpson f a b n)
  (simp-aux f a b n (/ (- b a) (* n 1.0) )))

(define (simp-aux f a b n h)
  (define (term k)
    (cond ((= 0 k) (f a))
      ((= n k) (f (+ a (* k h))))
      ((odd? k) (* 4 (f (+ a (* k h)))))
      ((even? k) (* 2 (f (+ a (* k h)))))
      ))

  (define (next k)
    (+ k 1))

  (* (/ h 3.0)
     (sum term 0 next n))

)
```

our sigma function

k is an integer! (want to avoid floating point issues)

Note that I am not using the yk values to control the loop.

What is to be proved?

Motivational

Aside on real-number pitfalls. I'd like you to write a loop with stopping condition x == 1, In x initially 0, which adds 0.01 to x with each iteration

```
x := 0 ;
while x <> 1.0 do
    x := x + 0.01
```

It does not stop!

CLEARLY we would NOT want to give a pf of Simpson's Rule itself. Rather - the proof burden would be to show that this code actually does IMPLEMENT Simpson's Rule.

A useful verification would start by showing (diagramatically) that the code does what Simpson's rule requires.

There is an easy way to avoid the repeated multiplication (* k h) - think about introducing a new variable to keep track of the right endpoints.

Note that Problem 1.30 - iterative sigma - has been discussed in class.

Problem 1.31 — Another example of the use of a specialized sigma term parameter in the computation of Wallis' product:

$$\frac{\pi}{4} = \frac{(2 \cdot 4) \cdot (4 \cdot 6) \cdot 6 \cdot 8 \cdots}{(3 \cdot 3) \cdot (5 \cdot 5) \cdot 7 \cdot 7 \cdots}$$

k=1 k=2

necessary to force to real number

```
(define (wallis n)

  (define (term k)
    (cond ((odd? k) (* 1.0 (/ (+ k 1) (+ k 2))))
          (else (* 1.0 (/ (+ k 2) (+ k 1))))))

  (define (next k)
    (+ k 1))

  (iter-prod term 1 next n))
```

where you can fill in the details for the function iter-prod.

Let's take a break from the homework problems with an Iterative Count Change program

A recursive divide and conquer approach is developed in the text (pp 40, 41 (in my copy)):

The number of ways to change amount a using n kinds of coins equals

- The number of ways to change amount a using all but the first kind of coin, plus
- The number of ways of changing amount a − d using n kinds of coins, where d is the denomination of the first kind of coin.

Divide & conquer: [One makes change either by using NONE of the highest value coins OR by using at least ONE of the highest value coins]

What might an iterative approach look like? One approach essentially solves the problem via an

extensive functional decomposition:

```
;; the only looping here is in sigma

;; GI: result = sum of term(i) from A to a-1 AND a <= b + 1

(define (sigma a term b)
  (define (iter a result)
    (cond ((> a b) result)
          (else (iter (+ a 1) (+ (term a) result)))))
  (iter a 0))

;; the max number of nickels is (quotient amt 5), and for each 0 <= i <= (quotient amt 5), there is
;; way of making change using i nickels (with pennies as the balance)

(define (pn amt)
  (cond ((< amt 5) 1)
        (else (+ 1 (quotient amt 5)))))

;; the max number of dimes is (quotient amt 10), and for each 0 <= i <= (quotient amt 10), there
;; (pn (- amt (* 10 i))) ways of making change

(define (pnd amt)
  (sigma 0
         (lambda (numberOfDimes) (pn (- amt (* 10 numberOfDimes))))
         (quotient amt 10)))

(define (pndq amt)
  (sigma 0
         (lambda (numberOfQuarters) (pnd (- amt (* 25 numberOfQuarters))))
         (quotient amt 25)))

(define (pndqh amt)
  (sigma 0
         (lambda (numberOfHalves) (pndq (- amt (* 50 numberOfHalves))))
         (quotient amt 50)))
```

No new invariant needed, since we've already proved that sigma works.

Suggested for you:

pn, pnd, pndq, pndqh

These last functions are so similar that our inner functional programmer wants to ask: can they be abstracted (as we did earlier for sigma)?

Well, back to work ...

Problem 1.34

```
(define (f g)
  (g 2))
```

What happens when we call

```
(f f)
```

?

This results in (f 2), which gives (2 2), which throws an error because (2 2) makes no sense as a function call.

(Continued Fractions)

Exercise 1.37

numerator function

denominator function

```
; iterative version

(define (cont-frac n d k)

  (define (cont-iter result count)
    (if (= count 0)
        result
        (cont-iter
         (/ (n count) (+ (d count) result))
         (- count 1))))

  (cont-iter (/ (n k) (d k))
             (- k 1)))

; test with golden ratio

(cont-frac
 (lambda (i) 1.0)
 (lambda (i) 1.0)
 1000000)


; guess-invariant: result =

   ;;
   ;;
   ;;
   ;;
   ;;
   ;;
   ;;
   ;;

; so now when count = 0, the first numerator included in result is
; N_(1).
```

?

N_(k)

D_(k)

initial value of result

or is it N_0?

Does the indexing here agree with that given in the text? How does the GJ change if you change the indexing from 0-based to 1-based?

Numbers a and b with a > b are said to be in the golden ratio if their ratio is the same as the ratio of their sum to a. That is, if a/ b = (a + b)/b

Approx 1.6180339 ...

what is the next value of result?

$$\frac{N_{K-1}}{D_{K-1} + \text{result}} \quad (1e) \quad \frac{1}{1+1}$$

and the next value?

$$\frac{N_{K-2}}{D_{K-2} + \text{Result}} \quad (1e) \quad \frac{1}{1+\frac{1}{2}}$$

and so on - we can see the emerging pattern, even if we might not see how to describe this pattern with anything other than a diagram

Suggested for you: if the terms above are the 0^{th}, 1^{st}, and 2^{nd} terms, what does the p^{th} term look like?

Does

$$\frac{N_{k-p}}{D_{k-p} + \left(\frac{N_{k-(p-1)}}{D_{k-(p-1)} + \cdots + \frac{N_k}{D_k} \right)}}$$

look right?

Perhaps this could be abbreviated as cont-frac $(N, D, p-1)$?

So then as a guess we might have

$$\text{result} = \frac{N_{k-p}}{D_{k-p} + \text{cont-frac}(N, D, p-1)}$$

?

But we've lost the variable count!
Can you complete this by relating expressing p in terms of count?

I'll leave the recursive solution to you. (It's easier.)

Moving right along:

A&S

```
; exercise 1.42

(define (compose f g)
  (lambda (x) (f (g x))))

(define (addone x)
  (+ x 1))

(define (square x)
  (* x x))

((compose square addone) 3)

; what is assumed about f and g?
```

(define compose
(lambda (f g)
(lambda (x) (f (g x)))))

"pf: assuming f and g are composable and that scheme works — this returns the composition of f and g."

f and g must be composable one-argument functions.

(square (addone 3)) =

(square 4) = 16

* as far as we have gone, our functions all produce a 1-dimensional output

Eg: we have no way now of (directly) writing a function

$h: \mathbb{Z}^2 \to \mathbb{Z}^2$

could we have (lambda (x y) ...) as the value of a call to compose? Yes!

g could take multiple args, with a singleton output. Eg: g could be 2-arg+ and the function rewritten

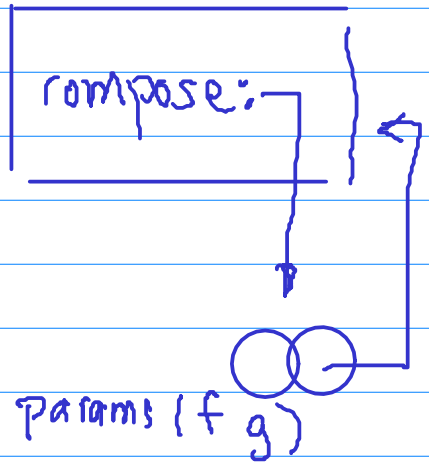
)

Of course, we could be clever and find a way to encode pairs of integers as single integers. (Eg consider the usual proof that set of pairs of integers is countable ☺...

see page after next

```
(define (new-compose f g)
  (lambda (x y) (f (g x y))))
```

What is the diagram for compose?



x, y need to satisfy precond for g and (g x y) needs to satisfy precond for f

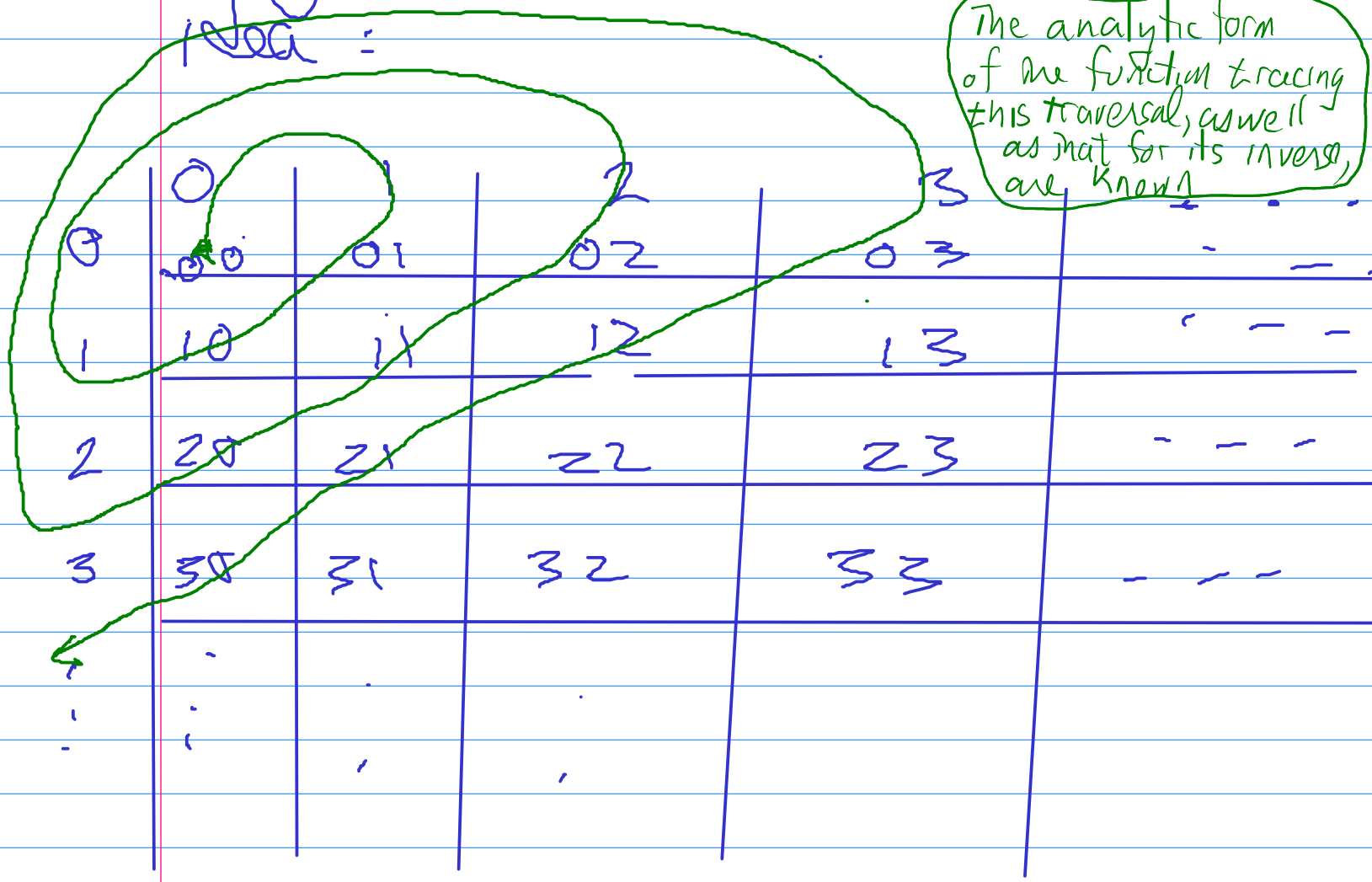
Practice: work out diagram for

```
((compose sq myadd) 2 3)
```

The proof for compose itself has nothing to do with the proofs of f and g

Is there truly no way of writing a function (with just what I know now) capable of returning multiple values? It is true that we can get only a single value (now), but we could use that value to encode (eg) pairs, triples, etc. One way: if we want outputs of 3 values p, q, and r, have the function compute $2^p 3^q 5^r$ — extracting p, q, r is then just a matter of (what amounts to) taking base 2, base 3, base 5 logs. Fancy way of saying: we can do repeated division by 2 to extract p (and so on)

There are other ways — recall the argument that the set of pairs (p, q) of integers has the same size as the set of integers itself. Here's the idea:



The analytic form of the function tracing this traversal, as well as that for its inverse, are known

call to compose deferred; this recursive

A&S

```
; exercise 1.43

(define (repeated f k)
  (if ( = k 1)
      f
      (compose f (repeated f (- k 1)))))

(define (compose f g)
  (lambda (x) (f (g x))))

(define (square x)
  (* x x))

((repeated square 2) 5)
```

eg ((repeated square 2) 5) = 625

What inductim? Inductim on k – the # of times f is to be composed.

divide & conquer: if we knew how to compose f with itself k-1 times, then we can obtain k-fold compositim as shown.

IH

basis step could be k=0

in that case, convention could be that f⁰ is (lambda (x) x)

; iterative solution:

Idesign idea – keep a variable result-so-far and another variable – count – and maintain

result-so-far = f composed with itself, count times

Hey! This is a function

Next: looks like count would most usefully increase from 0 up to count — stop when count = k, then result-so-far would be f^k.

(repeated sq 2) 2)

(sq (sq 2))

((repeated sq 3) 2) = (sq (sq (sq 2))) = (sq 16) = 256

; $k \geq 0$ is an integer

This is f^0

one possible start

```
(define (repeated f k)
  (iter 0 k f (lambda (x) x)))

(define (iter count k f result-so-far)
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ )
```

Aside:
(lambda (x) (f x)) is the same as f.

another

; $k \geq 1$ is an integer

```
(define (repeated f k)
  (iter 1 k f f))

(define (iter count k f result-so-far)
  (cond ((= count k) result-so-far)
        (else
         (iter (+ count 1) k f (compose f result-so-far)))))
```

make sure not to apply the adjustment for result-so-far

We want yet to check the invariant, but this thing was built with a specific GI in mind!

Suggested Practice: work out the down-counting version, in which k no longer needs to be passed as parameter.