# Assignment 4 Theoretical Questions:

Q 1.b :
Let's define the CPS equivalence of a higher-order function g and its CPS version g$ as:

For any CPS-equivalent parameters f1...fn and f1$...fn$: (g$ f1$...fn$ cont) is CPS-equivalent to (cont (g f1...fn))

We'll prove that pipe$ is equivalent to pipe by induction on the length of the input list.

Base Case: N = 1 (list with one function)

(pipe$ (list f1$) cont) = (cont (lambda (x cont2) (f1$ x cont2))) [by definition of pipe$] = (cont f1$) [by eta-reduction]

(cont (pipe (list f1))) = (cont f1) [by definition of pipe]

Since f1$ is CPS-equivalent to f1, these are equivalent.

Inductive Step: Assume the equivalence holds for lists of length k. We'll prove it for length k+1.

Inductive Hypothesis: (pipe$ (list f1$ ... fk$) cont) is CPS-equivalent to (cont (pipe (list f1 ... fk)))

For a list of length k+1:

(pipe$ (list f1$ ... fk$ f(k+1)$) cont) = (pipe$ (cdr (list f1$ ... fk$ f(k+1)$)) (lambda (res) (compose$ (car (list f1$ ... fk$ f(k+1)$)) res cont))) [by definition of pipe$] = (pipe$ (list f2$ ... fk$ f(k+1)$) (lambda (res) (compose$ f1$ res cont))) [simplifying]

By the inductive hypothesis, this is equivalent to: ((lambda (res) (compose$ f1$ res cont)) (pipe (list f2 ... fk f(k+1))))

Applying compose$: = (cont (lambda (x cont2) (f1$ x (lambda (res) ((pipe (list f2 ... fk f(k+1))) res cont2)))))

This is CPS-equivalent to: (cont (compose f1 (pipe (list f2 ... fk f(k+1)))))

Which, by the definition of pipe, is equivalent to: (cont (pipe (list f1 f2 ... fk f(k+1))))

Thus, we've shown that the equivalence holds for lists of length k+1 if it holds for lists of length k.

By the principle of mathematical induction, we conclude that pipe$ is CPS-equivalent to pipe for lists of any length.

Q 2.d :

1. reduce1-lzl: Best for complete, finite lazy lists.
2. reduce2-lzl: Useful for processing a limited number of elements from any lazy list, including infinite ones.
3. reduce3-lzl: Creates a new lazy list of cumulative reductions, ideal for running calculations on infinite sequences.

Q 2.g :

Advantage: Offers flexible precision levels, unlike pi-sum's fixed limit.

Disadvantage: Higher memory usage due to numerous closures created for lazy evaluation.

Q 3.1 :

1. unify[x(y(y), T, y, z, k(K), y), x(y(T), T, y, z, k(K), L)]

Steps:

- The outer functor f matches
- y(y) unifies with y(T),then T = y
- T unifies with T
- y unifies with y
- z unifies with z
- k(K) unifies with k(K)
- L unifies with y , then L = y

Result: Success, with : S={ y = T , L = y}

2. unify[f(a, M, f, F, Z, f, x(M)), f(a, x(Z), f, x(M), x(F), f, x(M))]

Steps:

- The outer functor f matches
- a unifies with a
- M unifies with x(Z), then M = x(Z)
- f unifies with f
- F unifies with x(M), then F = x(x(Z))
- Z unifies with x(F), then Z = x(x(x(Z)))

- f unifies with f
- x(M) unifies with x(M)

Result: Failure due to Z = x(x(x(Z)))

3.unify[t(A, B, C, n(A, B, C), x, y), t(a, b, c, m(A, B, C), X, Y)]

Steps:

- The outer functor t matches
- A unifies with a, then A = a
- B unifies with b, then B = b
- C unifies with c, then C = c
- n(A, B, C) fails to unify with m(A, B, C) , because the functors n and m are different

Result: Failure due to mismatch between n and m

4.unify[z(a(A, x, Y), D, g), z(a(d, x, g), g, Y)]

steps:

- The outer functor z matches
- a(A, x, Y) unifies with a(d, x, g), then A = d and Y = g
- D unifies with g, then D = g
- g unifies with Y (which is already bound to g)

Result: Success, with : S = {A = d, Y = g, D = g}

*The answer for Q3.3 is in the next page.

Q 3.3 :

```
          path(a, b, P)
         /            \
   edge(a, b)       edge(a, c)
      |               /     \
  path(b, b, P)  edge(c, b)  edge(c, a)
      |             |       /       \
     [b]        path(b, b, P) path(a, b, P)
                    |          /       \
                   [b]     [a, c, b]   [a, c, a, b]
                            /                \
                     path(c, a, P)      path(a, c, [c|P])
                            |                  \
                          [a, b]          ... (Infinite loop)
```

Is it a finite or an infinite tree?
The tree is <u>infinite</u> because the cycle in the graph allows for paths of infinite length to be constructed.

Is it a success or failure tree?
It is a <u>success</u> tree because it finds valid paths from 'a' to 'b'. In fact, it finds an infinite number of such paths, each one valid.