PPL 4 - Answers Sheet

Q1.b

By definition, a procedure f is CPS-equivalent to procedure f$ if for all input values x1, x2, …, xn, and for every continuation(cont) function the next equation is true:
(f$ x1 … xn cont) = (cont (f x1 … xn))

We'll use Induction on the number of functions given as arguments(n) to prove this CPS-equality:
Let's look at the base scenario which is:
pipe(f1 cont) and pipe$(f1) - the equation is now:
(pipe$ f1 cont) = (cont (pipe f1)), in this case:
a-e[(pipe$ f1 cont)] ⇒* a-e[(cont (f1))]
a-e[(cont (pipe f1))] ⇒* a-e[(cont (f1))]
These get the same result, so the base scenario is legit.

Now let's assume that this equation is true for n(The Induction Assumption):
(pipe$ f1 … fn cont) = (cont (pipe f1 … fn))

We'll now prove(Induction Step\Proof) that the next equation(for n+1) is true:
(pipe$ f1 … fn+1 cont) = (cont (pipe f1 … fn+1))

This equation can be further manipulated into:

(pipe$ (f2 … fn+1) (lambda (res) (cont (compose (f1) (res))))) =
(cont (compose (f1) (pipe (f2 … fn+1)))) ⇒

From the Induction Assumption(when used on the left side):

((lambda (res) (cont (compose (f1) (res)))) (pipe (f2 … fn+1))) =
(cont (compose (f1) (pipe (f2 … fn+1)))) ⇒

(cont (compose (f1) (pipe (f2 … fn+1)))) =
(cont (compose (f1) (pipe (f2 … fn+1))))

We can now see that both sides are equal also for n+1.
We proved the CPS-equality, Proof finished.

Q2

d.

reduce1-lzl: when given a finite lazy list and we want to use reduce (just like in typescript, to reduce the entire list to a single value).

reduce2-lzl: when we want to reduce up to the nth element in the lazy list (can be used for finite lazy lists to stop after n elements, can also be useful for infinite lazy lists).
reduce3-lzl: when we want to create a new lazy list that each element in index n is the original lazy list reduced up to the nth index (useful for finite/infinite lazy lists).

g.

advantages:

- to get a better approximation after getting an approximation, we don't need to re-do calculations previously made (for example, if we previously approximated with 3 elements from the series and now we want to approximate with one more element, for the lazy list we only need to calculate the 4th element, while in the pi-sum approach we'll need to calculate the first 4 elements).
- generate-pi-approximations is less prone to stack overflow, since it only needs to iterate through the list (each time only holding current value , and a function to calculate the next value), while the pi-sum holds all the elements in the series previously calculated up to the last element we want to calculate in the series.

disadvantages:

- In the lazy list, we must calculate all the elements that come before the one we want (this won't affect this algorithm since we do it in pi-sum too, but it can cause unnecessary overhead. for example: if we just want to iterate through 1 to 100, it won't make a difference doing normally or with integers-steps-from. but if we want the 100th number in the sequence, 1+99 is better than taking the first 100 elements in (integers-steps-from 1 1)).
- we can only move forward in the lazy list (can't reuse already calculated values unless saved somewhere else). In this case it might not matter (since there is the same calculation up to a nth element in both algorithms), but there are cases we may want to reuse calculated values.

Q3

3.1

.1.

we can make the next equation and have two complex expressions, one on each side, and their in the same form/format:
x(y(y), T, y, z, k(K), y) = x(y(T), T, y, z, k(K), L)

We can start the algorithm with this first equation:
equations: [x(y(y), T, y, z, k(K), y) = x(y(T), T, y, z, k(K), L)]
substitution: {} (empty at first)

We'll pick the first equation, which is the only one right now. We get the third scenario where there are complex expressions on both sides and they are in the same form, so we'll break the equation into smaller equations:
equations: [y=T, T=T, y=y, z=z, K=K, y=L]
substitution: {} (empty at first) and continue ⇒

We'll pick the first equation(y=T) and use the current substitution on it(after that, the equation is now T=y). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution. Also we'll remove the equations that are obvious like T=T.
equations: [y=L]
substitution: {T=y} and continue ⇒

Again, we'll pick the first equation(y=L) and use the current substitution on it(after that, the equation is now L=y). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: []
substitution: {T=y, L=y}

We finished going over the equations and got the result: {T=y, L=y}.

.2.
we can make the next equation and have two complex expressions, one on each side, and their in the same form/format:
f(a, M, f, F, Z, f, x(M)) = f(a, x(Z), f, x(M), x(F), f, x(M))

We can start the algorithm with this first equation:
equations: [f(a, M, f, F, Z, f, x(M)) = f(a, x(Z), f, x(M), x(F), f, x(M))]
substitution: {} (empty at first)

We'll pick the first equation, which is the only one right now. We get the third scenario where there are complex expressions on both sides and they are in the same form, so we'll break the equation into smaller equations:
equations: [a=a, M=x(Z), f=f, F=x(M), Z=x(F), f=f, x(M)=x(M)]
substitution: {} (empty at first) and continue ⇒

We'll remove the equations that are obvious like f=f. We'll pick the first equation(M=x(Z)) and use the current substitution on it(after that, the equation is now M=x(Z)). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: [F=x(M), Z=x(F)]
substitution: {M=x(Z)} and continue ⇒

Again, we'll pick the first equation(F=x(M)) and use the current substitution on it(after that, the equation is now F=x(x(Z))). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: [Z=x(F)]
substitution: {M=x(Z), F=x(x(Z))} and continue ⇒

Again, we'll pick the first equation(Z=x(F)) and use the current substitution on it(after that, the equation is now Z=x(x(x(Z)))). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: []

substitution: {M=x(Z), F=x(x(Z)), Z=x(x(x(Z)))}

We finished going over the equations and got the result: {M=x(Z), F=x(x(Z)), Z=x(x(x(Z)))}.
Because we got Z on both sides in the last equation, this is an <u>error</u>.

<u>.3.</u>
we can make the next equation and have two complex expressions, one on each side, and their in the same form/format:
t(A, B, C, n(A, B, C),x, y) = t(a, b, c, m(A, B, C), X, Y)

We can start the algorithm with this first equation:
equations: [t(A, B, C, n(A, B, C),x, y) = t(a, b, c, m(A, B, C), X, Y)]
substitution: {} (empty at first)

We'll pick the first equation, which is the only one right now. We get the third scenario where there are complex expressions on both sides and they are in the same form, so we'll break the equation into smaller equations:
equations: [A=a, B=b, C=c, n(A, B, C)=m(A, B, C), X=x, Y=y]
substitution: {} (empty at first) and continue ⇒

We'll pick the first equation(A=a) and use the current substitution on it(after that, the equation is now A=a). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: [B=b, C=c, n(A, B, C)=m(A, B, C), X=x, Y=y]
substitution: {A=a} and continue ⇒

We'll pick the first equation(B=b) and use the current substitution on it(after that, the equation is now B=b). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: [C=c, n(A, B, C)=m(A, B, C), X=x, Y=y]
substitution: {A=a, B=b} and continue ⇒

We'll pick the first equation(C=c) and use the current substitution on it(after that, the equation is now C=c). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: [n(A, B, C)=m(A, B, C), X=x, Y=y]
substitution: {A=a, B=b, C=c} and continue ⇒

We'll pick the first equation(n(A, B, C)=m(A, B, C)) and use the current substitution on it(after that, the equation is now n(a, b, c)=m(a, b, c)). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: [X=x, Y=y]
substitution: {A=a, B=b, C=c, n(a, b, c)=m(a, b, c)} and continue ⇒

We'll pick the first equation(X=x) and use the current substitution on it(after that, the equation is now X=x). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.

equations: [Y=y]
substitution: {A=a, B=b, C=c, n(a, b, c)=m(a, b, c), X=x} and continue ⇒

We'll pick the first equation(Y=y) and use the current substitution on it(after that, the equation is now Y=y). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: []
substitution: {A=a, B=b, C=c, n(a, b, c)=m(a, b, c), X=x, Y=y}
We finished going over the equations and got the result:
{A=a, B=b, C=c, n(a, b, c)=m(a, b, c), X=x, Y=y}

.4.
we can make the next equation and have two complex expressions, one on each side, and their in the same form/format:
z(a(A, x, Y), D, g) = z(a(d, x, g), g, Y)

We can start the algorithm with this first equation:
equations: [z(a(A, x, Y), D, g) = z(a(d, x, g), g, Y)]
substitution: {} (empty at first)

We'll pick the first equation, which is the only one right now. We get the third scenario where there are complex expressions on both sides and they are in the same form, so we'll break the equation into smaller equations:
equations: [a(A, x, Y)=a(d, x, g), D=g, Y=g]
substitution: {} (empty at first) and continue ⇒

We'll pick the first equation(a(A, x, Y)=a(d, x, g)) and use the current substitution on it(after that, the equation is now a(A, x, Y)=a(d, x, g)). Then, with this equation we get the third scenario where there are complex expressions on both sides and they are in the same form, so we'll break the equation into smaller equations, and add these equations to the equations pile.
equations: [A=d, x=x, Y=g, D=g, Y=g]
substitution: {} and continue ⇒

Now let's get rid of the duplicates and the obvious x=x and we get:
equations: [A=d, D=g, Y=g]
substitution: {} and continue ⇒

We'll pick the first equation(A=d) and use the current substitution on it(after that, the equation is now A=d). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: [D=g, Y=g]
substitution: {A=d} and continue ⇒

We'll pick the first equation(D=g) and use the current substitution on it(after that, the equation is now D=g). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: [Y=g]

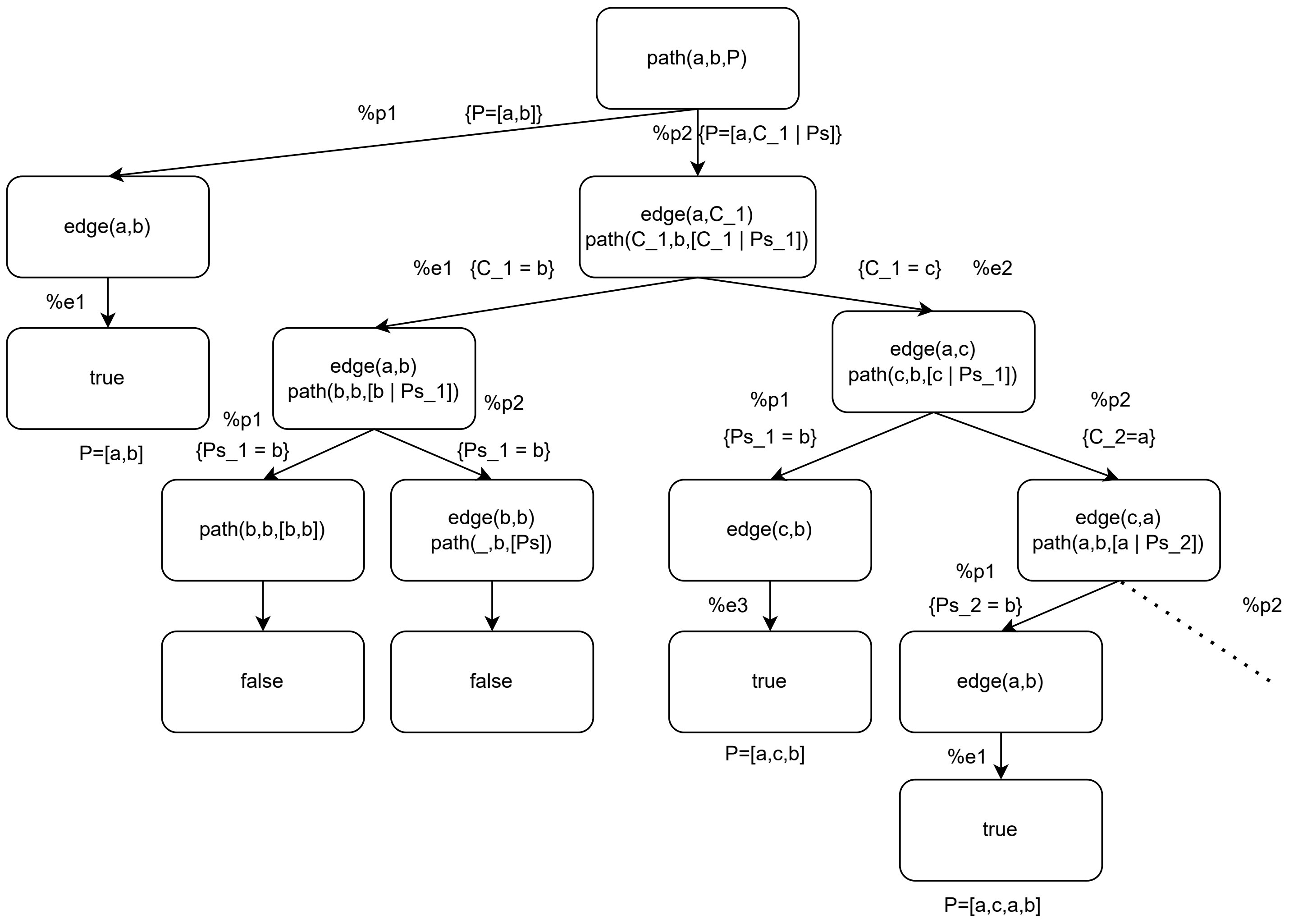
substitution: {A=d, D=g}   and continue ⇒

We'll pick the first equation(Y=g) and use the current substitution on it(after that, the equation is now Y=g). Then, with this equation we get the second scenario where one of the sides is a logic variable, so we'll add that equation to the substitution.
equations: []
substitution: {A=d, D=g, Y=g}
We finished going over the equations and got the result: {A=d, D=g, Y=g}

3.3

its an infinite tree, since $(ac)^{+}b,\ (ac)^{*}ab$ are valid answers for P(so there are infinite nodes that are true, [a,b], [a,c,b],[a,c,a,b],[a,c,a,c,b],[a,c,a,c,a,b], …)
it is a success tree because that it has success path (for example, P=[a,b])

(tree is on the next page).


path(a,b,P)
%p1
{P=[a,b]}
%p2
{P=[a,C_1 | Ps]}
edge(a,b)
%e1
true
P=[a,b]
edge(a,C_1)
path(C_1,b,[C_1 | Ps_1])
%e1
{C_1 = b}
{C_1 = c}
%e2
edge(a,b)
path(b,b,[b | Ps_1])
%p1
{Ps_1 = b}
%p2
{Ps_1 = b}
path(b,b,[b,b])
false
edge(b,b)
path(_,b,[Ps])
false
edge(a,c)
path(c,b,[c | Ps_1])
%p1
{Ps_1 = b}
%p2
{C_2=a}
edge(c,b)
%e3
true
P=[a,c,b]
edge(c,a)
path(a,b,[a | Ps_2])
%p1
{Ps_2 = b}
%p2
edge(a,b)
%e1
true
P=[a,c,a,b]