P4612 WEEK 7 SOLUTIONS

1 a) $Cn[z, id^3_1]$ is a three place function, sending x_1, x_2, x_3 to $z(id^3_1(x_1, x_2, x_3))$. It is a total function, as z & id^3_1 are total, & the composition of total functions is total.

$z(id^3_1(x_1, x_2, x_3)) = 0$ for every input! It is the three input zero function.

b) $Cn[sum, id^3_3, Cn[z, id^3_1]]$ is a three place function, sending x_1, x_2, x_3 to $sum(id^3_3(x_1, x_2, x_3), Cn[z, id^3_1](x_1, x_2, x_3))$.

This is total, as the basic components are total & the composition of total functions is total.

$$sum(id^3_3(x_1, x_2, x_3), Cn[z, id^3_1](x_1, x_2, x_3)) = x_3 + 0$$

So this function is a complicated way of specifying id^3_3! Given three values as input, it returns the third value.
Eg. If we feed it $(5, 7, 8)$, the result is $8+0$, which is 8.

c) $Cn[prod, id^3_2, Cn[sum, id^3_3, Cn[z, id^3_1]]]$ is also a three place function. It sends x_1, x_2, x_3 to

$$prod(id^3_2(x_1, x_2, x_3), sum(id^3_3(x_1, x_2, x_3), Cn[z, id^3_1](x_1, x_2, x_3)))$$

It's total, for the same reasons as the previous two cases.

$$prod(id^3_2(x_1, x_2, x_3), sum(id^3_3(x_1, x_2, x_3), Cn[z, id^3_1](x_1, x_2, x_3))) = x_2 \times (x_3 + 0).$$

So, for example, the function sends $(5, 7, 8)$ to $7 \times (8+0) = 56$.

d) Let's break $Pr[Cn[s,z], Cn[prod, id^3_1, id^3_3]]$ down.

* $Cn[s,z]$ is a one place function (sending x to $s(z(x)) = s(0) = 1$).
* $Cn[prod, id^3_1, id^3_3]$ is a three place function, sending x_1, x_2, x_3 to $prod(x_1, x_3) = x_1 \times x_3$.
* So, $Pr[Cn[s,z], Cn[prod, id^3_1, id^3_3]]$ is a two-place function, defined by primitive recursion, in terms of those two functions.

Lets abbreviate $Cn[s,z]$ by f. So $f(x) = 1$.

& $Cn[prod, id^3_1, id^3_3]$ by g. So $g(x,y,z) = x \times z$.

Then $Pr[f,g]$ is a total function (as f & g are, & the primitive recursion of two total functions is total)

Let's abbreviate some more. We'll call $Pr[f,g]$ the function h for short.

Unwrapping the definition of h using the rules for primitive recursion we get:

$$h(x,0) = f(x) = 1$$
$$h(x,y+1) = g(x,y,h(x,y)) = x \times h(x,y)$$

So, let's check small values:

$h(2,0) = 1$

$h(2,1) = 2 \times h(2,0) = 2 \times 1 = 2$

$h(2,2) = 2 \times h(2,1) = 2 \times 2 = 4$

$h(2,3) = 2 \times h(2,2) = 2 \times 2 \times 2 = 8$

$h(2,4) = 2 \times h(2,3) = 2 \times 2 \times 2 \times 2 = 16$

And so on.

Similarly, we have:

$h(n,0) = 1 = n^0$

$h(n,1) = n \times h(n,0) = n \times 1 = n^1$

$h(n,2) = n \times h(n,1) = n \times n = n^2$

$h(n,3) = n \times h(n,2) = n \times n \times n = n^3$

$h(n,4) = n \times h(n,3) = n \times n \times n \times n = n^4$

In general, if $h(n,m) = n^m$ (which is true when $m=0$, since $n^0 = 1$), then $h(n,m+1) = n \times h(n,m)$

$= n \times n^m$

$= n^{m+1}$

So, for every natural number m, $h(n,m) = n^m$. The function h is the exponential function.

e) The function $Mn[Cn[rsub, id^2_1, Cn[f, id^2_2]]]$ is a function defined by minimisation.

The inner function $Cn[rsub, id^2_1, Cn[f, id^2_2]]$ is a two place function, sending the inputs x & y to

$rsub(id^2_1(x,y), f(id^2_2(x,y)))$

$= rsub(x, f(y))$

$= x \dot{-} f(y)$, where $a \dot{-} b = \begin{cases} a-b & \text{if } a \geq b \\ 0 & \text{otherwise} \end{cases}$

This function is total if f is total. Call this function g.

The outer function is $Mn[g]$, a one place function.

We cannot tell, yet, if this is total. Mn sometimes makes partial functions out of total ones. To see how this function works, lets unwrap the definition:

$Mn[g](x)$ = the smallest y such that $g(x,y) = 0$,
ie, such that $x \dot{-} f(y) = 0$

Lets take $f(y) = 2 \times y$

Then to calculate our function on the input x we look for the smallest y where $x \dot{-} 2 \times y = 0$.

	$x=0$	$x=1$	$x=2$	$x=3$	$x=4$	$x=5$	$x=6$
$y=0$ $x \dot{-} 2\times 0$	0	1	2	3	4	5	6
$y=1$ $x \dot{-} 2\times 1$	0	0	0	1	2	3	4
$y=2$ $x \dot{-} 2\times 2$	0	0	0	0	0	1	2
$y=3$ $x \dot{-} 2\times 3$	0	0	0	0	0	0	0

In this case the function is total, and performs division by 2, rounding up. It is finding a kind of 'inverse' to the function f. (Multiplying by 2 $\Rightarrow$ Dividing by 2)

In general, the function defined will be total if for every number n, there is some input y for f such that $f(y) \geq n$. In that case, the search for a least y where $x \dot{-} f(y) = 0$ will always succeed, whatever x is chosen.

2 a) To define the function t where $t(n)$ is the nth triangular number, we will format the definition to have the same shape as a definition using Pr.

$$\begin{cases} t(0) = 0 \\ t(y+1) = t(y) + y + 1 \end{cases}$$

← 0 place function for the base case — a number

← 2 place function used in the step function

↑ 1 place function

t is $Pr[0, g]$ where $g(y, z) = z + y + 1$

So, to put g in the appropriate format it is $Cn[sum, id^2_2, Cn[s, id^2_1]]$, since this function applied to y & z is

$$\begin{aligned} & sum(id^2_2(y,z), s(id^2_1(y,z))) \\ = \; & sum(z, s(y)) \\ = \; & z + y + 1 \end{aligned}$$

So, since sum is $Pr[id^1_1, Cn[s, id^3_3]]$, the triangular number function t is:

$$Pr[0, Cn[\underbrace{Pr[id^1_1, Cn[s, id^3_3]]}_{sum}, id^2_2, Cn[s, id^2_1]]]$$

Factorial is defined in just the same format:

$$0! = 1$$

$$(y+1)! = y! \times (y+1)$$

So, the function is $Pr[s(0), g]$ where $g(y,z) = z \times (y+1)$, so g can be expressed as $Cn[prod, id^2_2, Cn[s, id^2_1]]$.

So, since prod is $Pr[z, Cn[Pr[id^1_1, Cn[s, id^3_3]]]]$, the factorial function is

$$Pr[s(0), Cn[Pr[z, Cn[Pr[id^1_1, Cn[s, id^3_3]]]], id^2_2, Cn[s, id^2_1]]]$$

(sum: $Pr[id^1_1, Cn[s, id^3_3]]$; prod: $Pr[z, Cn[\ldots]]$)

3. Here are some thoughts about what procedure you can follow to compute the value of a given recursive function, let's see how we can specify how to compute each basic function, & functions specified using the constructors.

Successor: take the input, add 1, return that value.

Zero: take the input. Return 0.

id^n_m: take n inputs. Return the value in the mth

(Very easy if the values are stored as piles of stones, for example. Harder if a different method of representation is used.)

$Cn[f, g_1, \ldots, g_m]$

1. Make a copy of the n inputs, calculate g_1 on them, Store the result in safe place 1.
2. Make a copy of the n inputs, calculate g_2 on them, Store the result in safe place 2.
⋮
$m-1$. Make a copy of the n inputs, calculate g_{m-1} on them, Store the result in safe place $m-1$.
m. Take the n inputs, calculate g_m on them, Store the result in safe place m

Take the results from places 1 to m, and calculate g on them, & then return the result.

$Pr[f,g]$ to calculate this, on input $x_1,\dots,x_n,y$

take a copy of each value $x_1,\dots,x_n$ & store them in a safe place

Calculate $f(x_1,\dots x_n)$

Check if $y=0$. If it is, return the value you calculated.

If it isn't, store the value in THIS BOX, & let $i=0$.

*Calculate $g(x_1,\dots,x_n, i,$ THE VALUE IN THIS BOX$)$

Add 1 to i.

Check: is $i=y$

If it is return the VALUE you just calculated & stop.

If it isn't, put the VALUE in the box (replacing its previous value) & go back to step *.

Mn[f] To calculate $Mn[f](x_1, x_2, \ldots, x_n)$, do this.

Start with y set to 0.

Store $x_1, \ldots, x_n$ away safely so we can reuse them.

* Calculate $f(x_1, \ldots, x_n, y)$

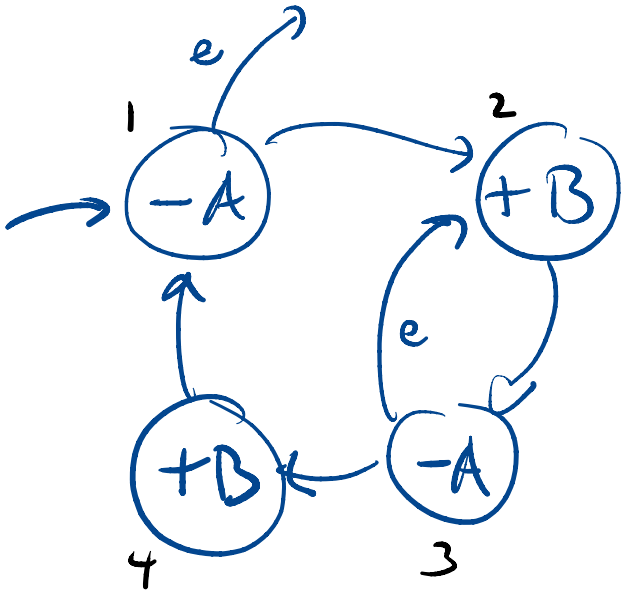
Is the result 0?

If it is, return the value y and stop.

If it isn't, add 1 to y and leap back to step *.

PH4612 Class Solutions Week 8

1a) Here is a flow graph for this register machine



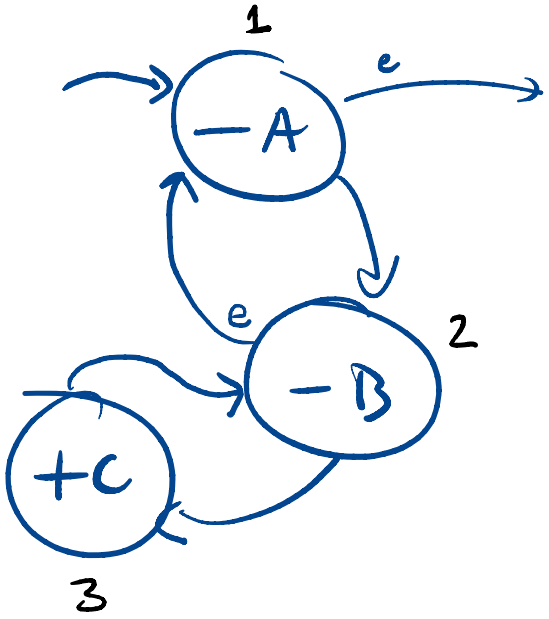
If you start the register machine with register A empty, it terminates immediately, with no change in either register.

If you start the registermachine with register A containing one rock, the program increments register B, tries to empty A, fails (it is already empty by then) & loops indefinitely, filling register B.

If you start with register A containing two rocks, the program will terminate after adding two rocks to register B.

In general, this program will
— Loop indefinitely if register A starts with an odd number of rocks
— If register A starts with n rocks & n is even, then the program will terminate with A empty & B containing n more rocks than it contained when the program started.

b) Here is a flow graph for the second register machine:



If this program starts with A empty, it terminates.

If the program starts with B empty, it terminates.

If A starts non empty, the number of rocks in B is transferred into C, & B is then empty, & A is emptied & the program terminates.

So, if the registers start

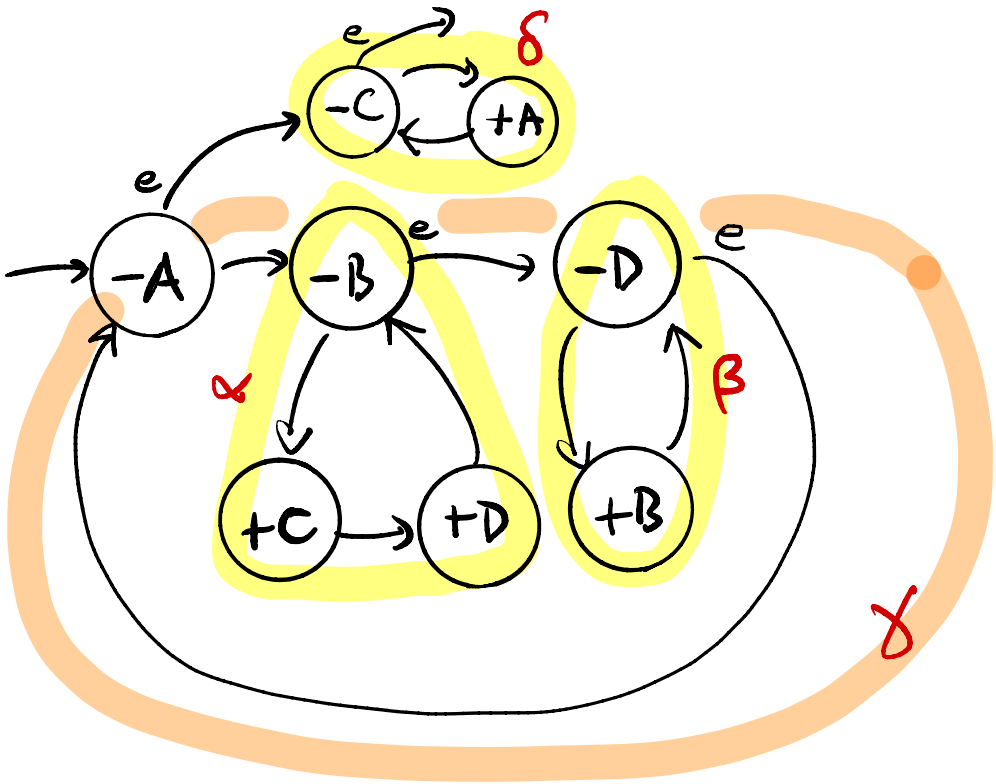
A [n] B [m] C [k]

the program finishes

$n=0$: A [–] B [m] C [k]

$n \neq 0$: A [–] B [–] C [k+m]

2.



This is how you can prove that the code calculates the product of A & B (given that C & D start empty

We break the code up into parts; labelled α, β, γ, δ.

We can prove by induction on the count in a register what each part does.

We break the code up into parts; labelled $\alpha, \beta, \gamma, \delta$.

We can prove by induction on the count in a register what each part does.

α: We prove that when the code in α executes on a machine starting in state $k\ \ell\ m\ n$ (A B C D) it ends in state $k\ 0\ \ell+m\ \ell+n$.

Prove this by induction on ℓ. If register B starts at zero, the code passes through. We start $k\ 0\ m\ n$ and we end $k\ 0\ 0+m\ 0+n$ as desired.

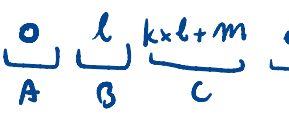
If, on the other hand when register B starts with ℓ, and the starting state is $k\ \ell\ m\ n$ it ends with $k\ 0\ \ell+m\ \ell+n$, the induction hypothesis, lets see what it does after it starts with $\ell+1$ in register B. $k\ \ell+1\ m\ n$

Here it starts by decrementing B, (successfully, since B is nonempty) and adding 1 to each of registers C & D. After this, the registers are $k\ \ell\ m+1\ n+1$. Now B has ℓ in it, & we already know what that does, by the induction hypothesis. & after this, we end with $k\ 0\ \ell+(m+1)\ \ell+(n+1)$, which is $k\ 0\ (\ell+1)+m\ (\ell+1)+n$ as we hoped to prove.

β: This is easy to prove: If we start β with the state $\underset{A}{k}\ \underset{B}{l}\ \underset{C}{m}\ \underset{D}{n}$, we exit this loop in state $\underset{A}{k}\ \underset{B}{n+l}\ \underset{C}{m}\ \underset{D}{0}$.
This is, formally a proof by induction on l, but it is obvious, so we declare this proved.

γ: Now we can tackle the loop γ. We will prove that if we start this loop in the state , we will exit the loop in state $\underset{A}{0}\ \underset{B}{l}\ \underset{C}{k\times l+m}\ \underset{D}{0}$.
We will prove this by induction on k.
If $k=0$, then we exit γ immediately. We started with $\underset{A}{0}\ \underset{B}{l}\ \underset{C}{m}\ \underset{D}{0}$ & end in the same state, which is $\underset{A}{0}\ \underset{B}{l}\ \underset{C}{0\times l+m}\ \underset{D}{0}$, as we wanted to. So, we make the hypothesis that when we start in state $\underset{A}{k}\ \underset{B}{l}\ \underset{C}{m}\ \underset{D}{0}$, we terminate γ in state $\underset{A}{0}\ \underset{B}{l}\ \underset{C}{k\times l+m}\ \underset{D}{0}$, for any choices for l & m. So, with that hypothesis, check what happens when we start with $\underset{A}{k+1}\ \underset{B}{l}\ \underset{C}{m}\ \underset{D}{0}$. Here is what happens. we decrement A. Then we go through α once, and the state now is $\underset{A}{k}\ \underset{B}{0}\ \underset{C}{l+m}\ \underset{D}{l}$. Then we go through β once, & then the state becomes this: $\underset{A}{k}\ \underset{B}{l}\ \underset{C}{l+m}\ \underset{D}{0}$. And we return to execute γ again. Now register A contains k, the induction hypothesis applies, & after going through the loop we terminate with the state $\underset{A}{0}\ \underset{B}{l}\ \underset{C}{k\times l+l+m}\ \underset{D}{0}$, which is $\underset{A}{0}\ \underset{B}{l}\ \underset{C}{(k+1)\times l+m}\ \underset{D}{0}$ as desired.

So γ transforms 

into 

δ: Clearly transforms 

into 

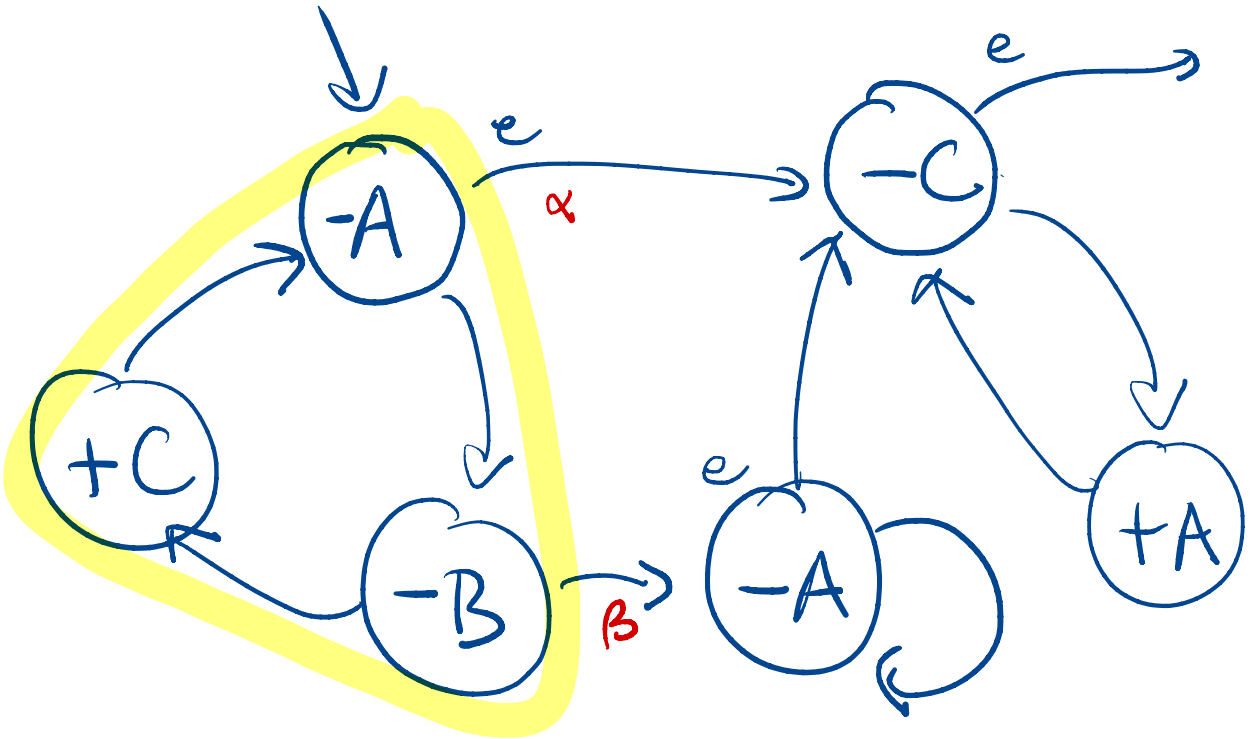
So, putting these together, if we start in state 

then after going through γ we exit in state 

and then, after going through δ, we end in state 

In other words, when we start with k & l in the first two registers, we terminate with $k \times l$ in the first register — it calculates the product function, as desired.

3. Here is a register machine that calculates $\min(A, B)$



If we start this machine with n in register A & m in register B,

* if $n \leq m$, we exit the yellow loop with register A empty & register C containing $n - \min(n, m)$. at α
* if $n > m$, we exit the yellow loop at β with register B empty & register C containing $m - \min(n, m)$

In either case, we have ended this part of the recipe with register C containing the smaller of the two values (or either, if they were equal). The rest of the recipe empties A (if it were not empty already) and returns the value in C back to A.

P44612 Class Solutions Week 10

1) Take an enumeration of all the (possibly) partial recursive functions $f_0, f_1, f_2, \ldots$ The diagonalisation argument constructs a diagonal function f^* by defining it like this: $f^*(n) = f_n(n)+1$. This is called a diagonal construction because if we set it out in a diagram, the values are found on the diagonal of the table:

	0	1	2	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	...
⋮	⋮	⋮	⋮	⋱

take these values and add 1.

If we apply this argument to all recursive functions — including the partial functions — we cannot conclude that the function f^* is not on the list $f_0, f_1, f_2, \ldots$ since we could have $f^* = f_n$ for some n.

Here is why: we can have

$$f_n(n) = f^*(n) = f_n(n)+1$$

if the value $f_n(n)$ is undefined — then $f^*(n) = f_n(n)+1$ is also undefined, so there is no contradiction. This argument cannot be used to show that there is a non-recursive function.

The next part of the question shows <u>why</u> there is no way that any argument like this is going to work.

We can define a recursive partial function $F: \omega \times \omega \to \omega$, where for each recursive function $f: \omega \to \omega$, there is a number n where $f(m) = F(n, m)$, for every m.

One way to think of F is as a "register machine compiler." Represent each register machine program in a single number, using some technique to encode the instructions in a single number (powers of primes, etc.) — then we can simulate the <u>state</u> of that register machine program in a single <u>number</u>, too, in the way we did in the argument that all register machine computable functions are recursive.

We then design a register machine program that will take the inputs n & m, & first encode m into the representation of the state, in a register of our machine, & then at each step we decode n to decide whether to simulate an increment or decrement and update the state number appropriately & is there is a next instruction, loop, & if not, extract the value from register A inside the encoded state & place that in register A, and stop.

This is a high-level description of why F is itself computable by register machine, and it is, therefore recursive.

Given this, we have a recursive enumeration of all partial recursive functions: $f_0(x) = F(0,x)$, $f_1(x) = F(1,x)$, $f_2(x) = F(2,x)$, ... etc.
The diagonalisation argument tells us that since $\lambda x\, F(x,x)$ is recursive, given that it is f_n for some n, we must have $F(n,n)$ undefined for this n.

2 When we define $A: \omega \times \omega \to \omega$ like this:

a. $A(0,n) = n+1$

b. $A(m+1,0) = A(m,1)$

c. $A(m+1,n+1) = A(m,A(m+1,n))$

then the reasoning can go like this ....

```
1.  A(3,1)  = A(2,A(3,0))   by c.
2.  A(3,0)  = A(2,1)        by b.
3.  A(2,1)  = A(1,A(2,0))   by c.
4.  A(2,0)  = A(1,1)        by b.
5.  A(1,1)  = A(0,A(1,0))   by c.
6.  A(1,0)  = A(0,1) = 2    by b.
7.  A(1,1)  = A(0,2) = 3    by 5, 6, a.
8.  A(2,0)  = 3             by 4,7
9.  A(2,1)  = A(1,3)        by 3,8
10. A(1,3)  = A(0,A(1,2))   etc...
11. A(1,2)  = A(0,A(1,1)) = A(0,3) = 4
12. A(1,3)  = A(0,4) = 5
13. A(2,1)  = 5
14. A(3,0)  = 5
15. A(3,1)  = A(2,5)
16. A(2,5)  = A(1,A(2,4))
17. A(2,4)  = A(1,A(2,3))
18. A(2,3)  = A(1,A(2,2))
19. A(2,2)  = A(1,A(2,1)) = A(1,5)
20. A(1,5)  = A(0,A(1,4))
21. A(1,4)  = A(0,A(1,3)) = A(0,5) = 6
22. A(1,5)  = A(0,6) = 7
23. A(2,2)  = 7
24. A(2,3)  = A(1,7)
25. A(1,7)  = A(0,A(1,6))
26. A(1,6)  = A(0,A(1,5)) = A(0,7) = 8
27. A(1,7)  = A(0,8) = 9
28. A(2,3)  = 9
29. A(2,4)  = A(1,9) = A(0,A(1,8))
30. A(1,8)  = A(0,A(1,7)) = A(0,9) = 10
31. A(2,4)  = A(0,10) = 11
32. A(2,5)  = A(1,A(2,4)) = A(1,11)
33. A(1,11) = A(0,A(1,10))
34. A(1,10) = A(0,A(1,9))
35. A(1,9)  = A(0,A(1,8))
36. A(1,8)  = A(0,A(1,7)) = A(0,9) = 10
37. A(1,9)  = A(0,10) = 11
38. A(1,10) = A(0,11) = 12
39. A(1,11) = A(0,12) = 13
40. A(2,5)  = 13
41. A(3,1)  = 13
```

↖ woo hoo! It does take a while.

Now, here is why the process of evaluating A will always terminate, even though it might take a very long time.

Draw a table like this:

$$\begin{array}{llllll}
A(0,0) & A(0,1) & A(0,2) & A(0,3) & \cdots & A(0,n) \cdots \\
A(1,0) & A(1,1) & A(1,2) & A(1,3) & \cdots & A(1,n) \cdots \\
A(2,0) & A(2,1) & A(2,2) & A(2,3) & \cdots & A(2,n) \cdots \\
\vdots & \vdots & \vdots & \vdots & & \vdots \\
A(m,0) & A(m,1) & A(m,2) & A(m,3) & \cdots & A(m,n) \cdots \\
\vdots & \vdots & \vdots & \vdots & & \vdots
\end{array}$$

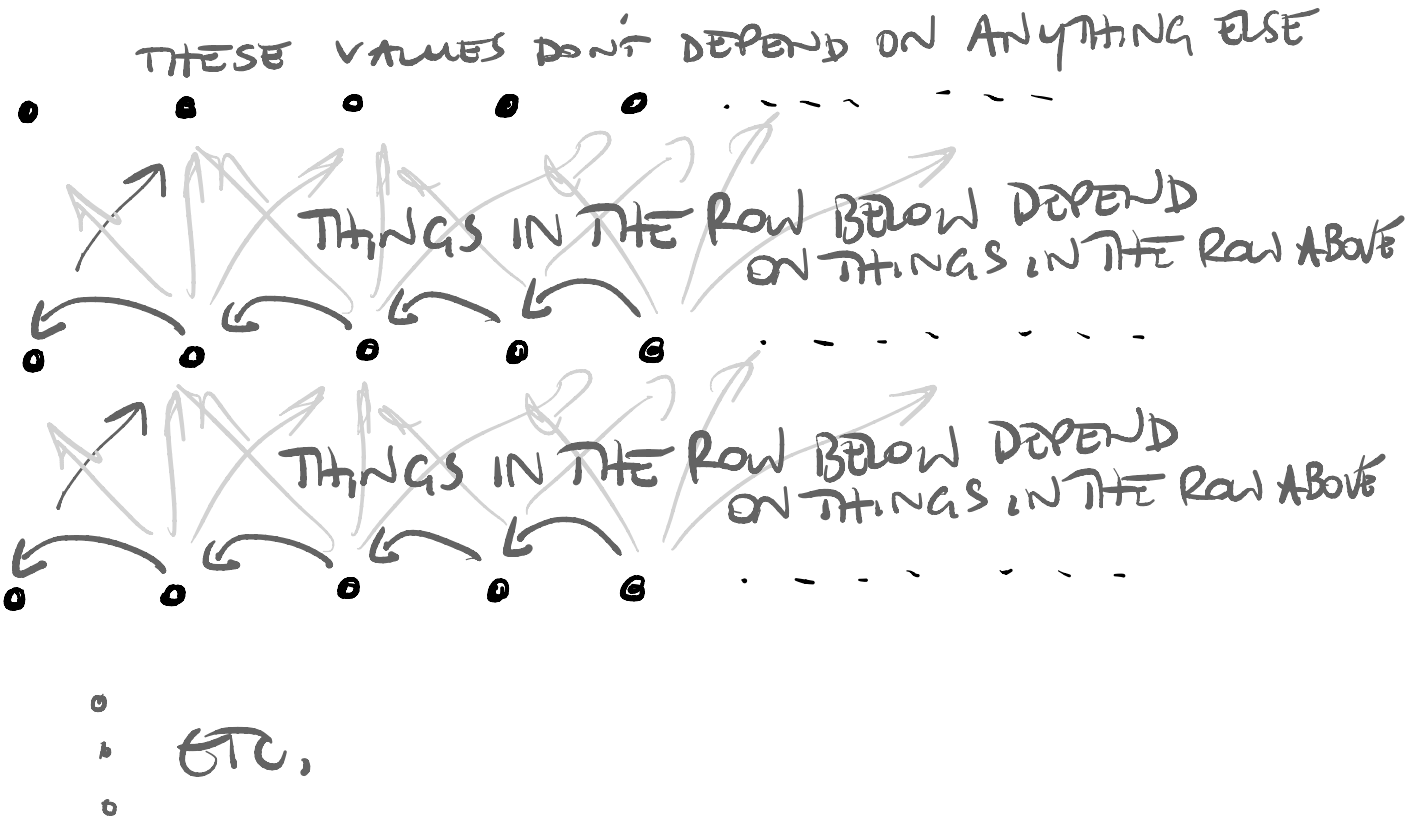
And take a look, for each spot on the table, look for what values you need to check first in order to calculate the value $A(m,n)$.

For values in the top row, we can calculate them immediately: $A(0,n) = n+1$.

For values in the first column ($A(m+1, 0)$), we can calculate them in terms of a value in the previous row.

for other values: ($A(m+1, n+1)$), to calculate that you need to calculate a value earlier in the same row ($A(m+1, n)$) and some value in the previous row: $A(m,$ whatever number $A(m+1, n)$ is)

The 'dependence' relation goes like this:



If we define a kind of 'alphabetical ordering' of pairs of numbers $\langle m,n\rangle \prec \langle m',n'\rangle$ if $m > m'$ OR $m = m'$ AND $n > n'$ then we have

$\langle 0,0\rangle \prec \langle 0,1\rangle \prec \langle 0,2\rangle \prec \langle 0,3\rangle \cdots$

AND ALL OF THESE COME AFTER ALL OF THESE

$\langle 1,0\rangle \prec \langle 1,1\rangle \prec \langle 1,2\rangle \prec \langle 1,3\rangle \cdots$

AND ALL OF THESE COME AFTER ALL OF THESE

$\langle 2,0\rangle \prec \langle 2,1\rangle \prec \langle 2,2\rangle \prec \langle 2,3\rangle \cdots$

AND ALL OF THESE COME AFTER ALL OF THESE

$\langle 3,0\rangle \prec \langle 3,1\rangle \prec \langle 3,2\rangle \prec \langle 3,3\rangle \cdots$

$\vdots$ etc.

And to calculate $A(m,n)$ we need only depend on values that come before $\langle m,n\rangle$ in a sequence of dependence.

Now: Start with (m,n) and try 'counting backwards'. You go $(m,n-1)$, $(m,n-2)$, ... $(m,0)$. Now find a value earlier than $(m,0)$. This is impossible if $m=0$, but if $m \geq 1$ we have infinitely many to choose from. But if forced to choose an earlier one, it has the form (m', l) where $m' < m$, and l is some number. We can then count down: $(m', l-1)$, $(m', l-2)$, ..., $(m', 0)$, and do it again ... and again m times at most. This cannot go on forever. Every descending chain from (m,n) is finite. So, no path of calculating $A(m,n)$ could ever go on forever. A is a total function.

(This special kind of ordering is the shape of the ordinal number $\omega \times \omega$ — it is ω many copies of ω. There are many other ordinals & they all have the property that descending chains in the ordinal are finite.)

3 Here is why every nonempty recursive set S is recursively enumerable. Take some recursive set S. It has a recursive characteristic function f_S. Define an enumeration e of S like this: Recall that e must be a surjective function $e : \omega \to S$ and it must be recursive. Since S is nonempty, it has some member $a \in S$. Define e by setting

$$e(n) = \begin{cases} n & \text{if } f_S(n) = 1 \\ a & \text{if } f_S(n) = 0 \end{cases}$$

This is recursive (actually, $e(n) = n \times f_S(n) + a \times (1 \dot{-} f_S(n))$) and it is a surjective function $\omega \to S$, since if $n \in S$ then $e(n) = n$.

(You cannot just construct your enumeration by making a list of all members of S, deleting from $0, 1, 2, 3, \ldots$ the non members. If S is finite this list runs out & is not a function from ω.)

Since for each number n we can test whether it is prime or not in a finite number of steps (by trying to divide the already discovered primes $2, 3, \ldots, m \leq \sqrt{n}$ into it evenly), the set of primes is recursive.

To recursively enumerate the prime gaps, go like this: Start with 1 $(= 3-2)$, then

find the next prime after 3, and subtract 3 from that, & generate that value. Remember that prime, generate the next prime, by checking each number along for primeness, then calculate the prime gap, etc... This is an easy enumeration.

But to check whether a number is a prime gap or not, in general, is very difficult, & maybe impossible. (I do not know if the set of prime gaps is recursive, or merely recursively enumerable.)

(It is, apparently, an open question as to whether every even number n is a prime gap. If this is true, the set of prime gaps is

$\{1, 2, 4, 6, 8, 10, \ldots\} = \{1\} \cup \text{Evens}$,

and this is recursive.)

PY4612 Class Solutions Week 11

1 • 5 This is a **number**.

• $\underline{0}'''''$ This is a **term** in the language of arithmetic. In the standard model of arithmetic it denotes the number 5. It is the **numeral** for 5.

• $\exists y(y+\underline{0}'=x)$ this is a **formula** in the language of arithmetic. It has one free variable, x. This open formula represents the set of non zero numbers, the set $\{1,2,3,\ldots\}$.

• The Gödel Number of $\exists y(y+\underline{0}'=x)$ is, as it says, a number. Which number it is depends on the numbering scheme. Typically the number is found by combining the Gödel numbers for the symbols $\exists, y, (, +, \underline{0}, ', =, x$ in some systematic way.

• $\ulcorner\exists y(y+\underline{0}'=x)\urcorner$ is the **numeral** for the Gödel number of the open formula $\exists y(y+\underline{0}'=x)$. It is a term in the language of arithmetic — a $\underline{0}$ followed by a big number of $'$s.

• The diagonalisation of $\exists y(y+\underline{0}'=x)$ is a formula in the language of arithmetic. It is the formula

$$\exists x(x=\ulcorner\exists y(y+\underline{0}'=x)\urcorner \wedge \exists y(y+\underline{0}'=x)),$$

which is equivalent to

$$\exists y(y+\underline{0}'=\ulcorner\exists y(y+\underline{0}'=x)\urcorner),$$ which is almost certainly

provable in Q. — It's provable if the Gödel number of $\exists y(y+\underline{0}'=x)$ is not 0 — which it almost certainly isn't.

- diag is a function $\omega \to \omega$ — it's a recursive function, defined as follows: $\mathrm{diag}(x) = 0$ if x is not the Gödel number of a formula.

  = the Gödel number of the diagonalisation of the formula with Gödel number x, otherwise.

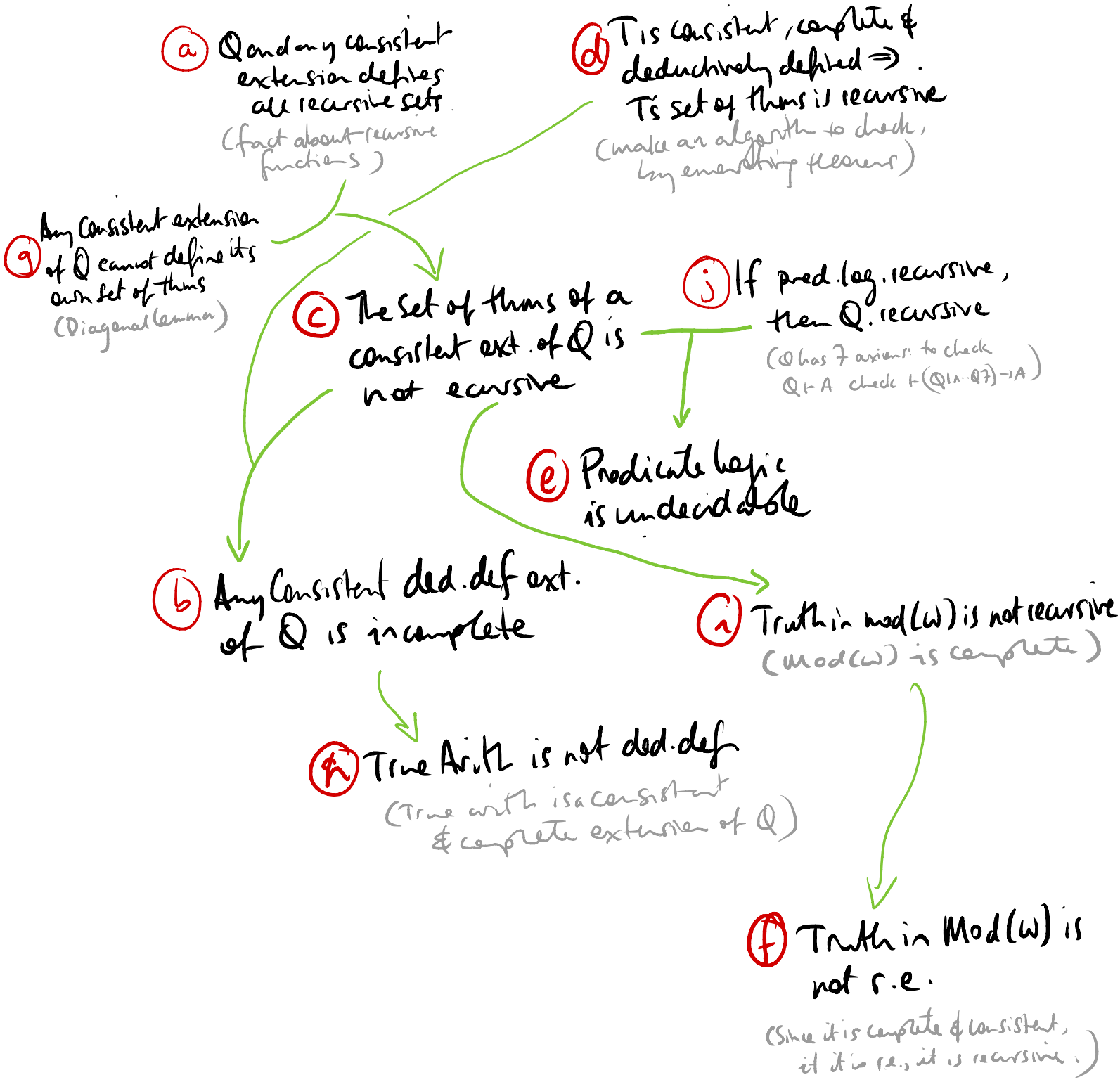
- A formula representing diag in Q is a formula in the language of arithmetic. It has the form $\phi(x,y)$ (it has two variables free), and we have for every number n.

$$Q \vdash \forall y\,(\phi(\underline{n},y) \leftrightarrow y = \underline{\mathrm{diag}(n)})$$

(numeral, numeral)

There is such a formula $\phi(x,y)$ since diag is recursive and Q represents every recursive function.

2 Here is my tree of connections.



PY4612 CLASS SOLUTIONS WEEK 12

1. There is some formula $Pr(x,y)$ that represents the relation

x is the Gödel number of a PA-proof of a formula with Gödel number y.

in PA. This relation is clearly recursive — it is a simple matter of proof checking.

This representability fact means that if n is the gn of a PA-proof of a formula with gn m, then $PA \vdash Pr(\underline{n}, \underline{m})$, and if n isn't the gn of a PA-proof of a formula with gn m, then $PA \vdash \neg Pr(\underline{n}, \underline{m})$.

Does $\exists x Pr(x,y)$, which intuitively means "there is a PA-proof of a formula with Gödel number y" would represent the provable formulas if

$PA \vdash \exists x Pr(x, \underline{m})$ if m is the gn of a provable formula, and

$PA \vdash \neg \exists x Pr(x, \underline{m})$ if m is the gn of an unprovable formula.

The first part holds. If m is the gn of a provable formula, it has some proof, with some gn (say n), where we have $PA \vdash Pr(\underline{n}, \underline{m})$, and hence $PA \vdash \exists x Pr(x, \underline{m})$.

The second part does not. If m is the gn of an unprovable formula, then indeed we have $PA \vdash \neg Pr(t, \underline{m})$ for every numerical term t, but this does not mean that $PA \vdash \neg \exists x Pr(x, \underline{m})$, and by Gödel's 2nd incompleteness theorem, this has no proof

2. PA – this theory is consistent & incomplete

$PA \nvdash Con(PA)$ (by Gödel's second incompleteness theorem)

$PA \nvdash \neg Con(PA)$ (since $\neg Con(PA)$ is false in ω)

So, since these two facts hold,

$PA + Con(PA)$

$PA + \neg Con(PA)$

are <u>both</u> consistent theories – & since they are also deductively defined consistent extensions of Q, they are both <u>also</u> incomplete.

$PA + \neg Con(PA)$ <u>is</u> consistent, even though it states that PA is inconsistent. This theory is <u>incorrect</u>, but consistent. Since $\neg Con(PA)$ is false in ω,

$PA + \neg Con(PA)$ is not true in ω, while

$PA + Con(PA)$ is true in ω.

Since $PA + Con(PA)$ is a consistent deductively defined extension of Q, it too is incomplete, and

$PA + Con(PA) \nvdash Con(PA + Con(PA))$

$PA + Con(PA) \nvdash \neg Con(PA + Con(PA))$

& the incompleteness phenomenon continues....