2. Consider the following function definitions:

```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```

All the functions have this in common:
The name of function, and then a list.

Nothing special there.

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:

```
f1 [] = 1
f1 (x:xs) = x * f1 xs
f2 [] = 0
f2 (x:xs) = 1 + f2 xs
f3 [] = 0
f3 (x:xs) = x + f3 xs
f4 [] = []
f4 (x:xs) = x ++ f4 xs
f5 [] = 0
f5 (x:xs) = (x*x) + f5 xs
```

Each function has their own little element that they return on the first line.

How fun!

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:

```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```

Each function also has their own little internal function that takes one thing (e.g. an a) & another thing (e.g. a b) and returns a different thing (e.g. a c).

(a -> b -> c)

Wow! Is this going somewhere? :o

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

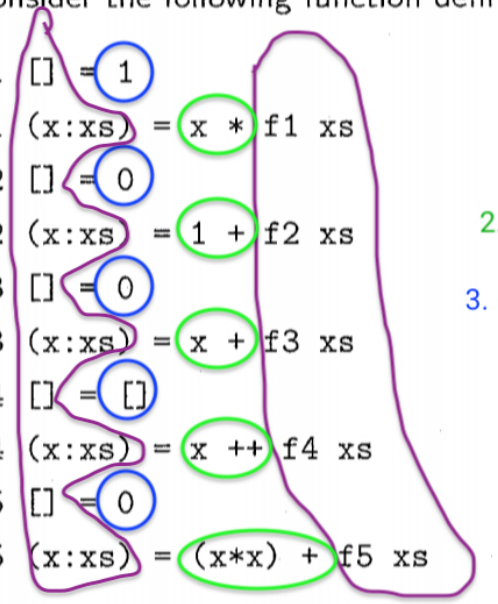
[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:

```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```



Summary: What do we know?

1. Every function takes a list
2. Every function has their own wacky internal function they do (that takes two things and returns one)
3. Every function has their very own special little element
4. Every function returns something (duh!)

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

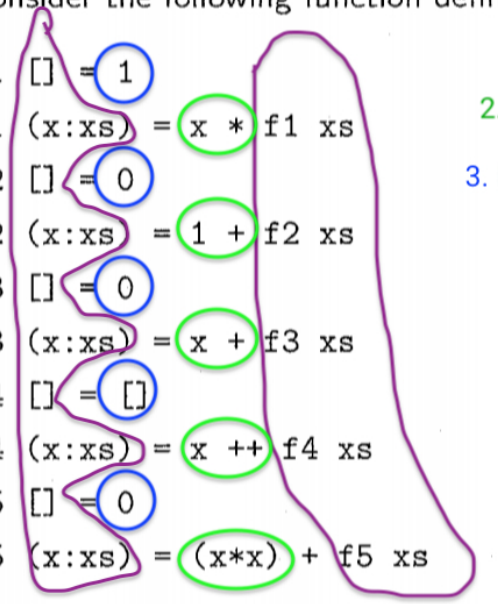
[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:

```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```



1. Every function takes a list
2. Every function has their own wacky internal function they do (that takes two things and returns one)
3. Every function has their very own special little element
4. Every function returns something (duh!)

We can therefore write this:

```
hof :: a -> (b -> c -> d) -> [e] -> f
```

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:

```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```

1. Every function takes a list
2. Every function has their own wacky internal function they do (that takes two things and returns one)
3. Every function has their very own special little element
4. Every function returns something (duh!)

Q. Hold up!

Why are they all like: a, b, c, d, e, f?
You tryna teach us the alphabet or somethin'?

```
hof :: a -> (b -> c -> d) -> [e] -> f
```

A. We haven't put any thought into the types yet, so it's always a good idea to lay out the function definitions like this before you do.

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

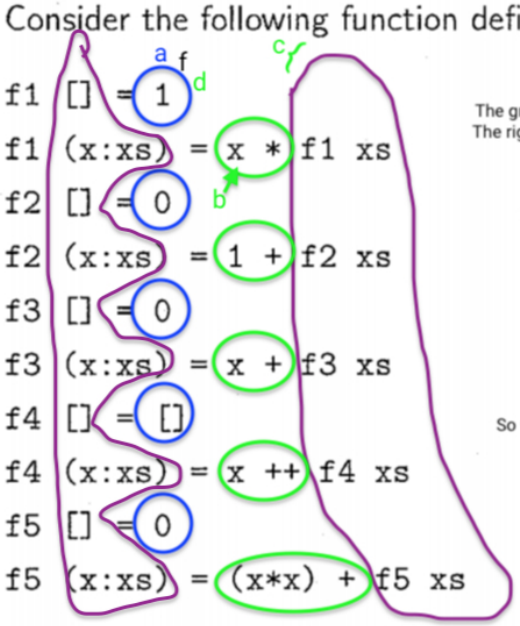
(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:



Sorry, I didn't mean to scare you with the letters.
Let's change them so there isn't as many.

The blue circles are the same type as what are returned, so we can rename 'f' to 'a'.

The green circles are functions that take the left hand side and righthand side and return something.
The righthand side of the green circles is gonna be what f1/f2..../f5 returns, which is the same type as the blue circles.
So we can rename 'c' to 'a' as well.

When the function of the green circles returns, the whole righthand side will be resolved.
This means that the return type of the green functions is the same as the blue circles.
So we can rename 'd' to 'a' as well.

x & xs have to be of the same type because they are put together in a list like '(x:xs)'.
So we can rename 'e' to 'b'.

Then finaaaally, we can see that each of the operators in the green functions:
*,+,++
are ones that take two things of the same type and return something of that type.
So we can change all occurrences of 'b' to whatever we changed 'c' to (we changed 'c' to 'a' :)).

```
hof :: a -> ( b -> c -> d ) -> [e] -> f
hof :: a -> ( a -> a -> a ) -> [a] -> a
```

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

Wow, that's a lot more difficult to explain in a tiny bit of text than it is to do in your head...

2. Consider the following function definitions:

```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```

1. Every function takes a list
2. Every function has their own wacky internal function they do (that takes two things and returns one)
3. Every function has their very own special little element
4. Every function returns something (duh!)

Q. Hold up again!

Why did you put the purple at the end instead of the start?
You doing somethin' fishy there?

```
hof :: a -> (a -> a -> a) -> [a] -> a
```

A. Because the purple list is the parameter to 'f1','f2','f3'....etc.
It has to come after what is unique to the functions for the hof to work.

This makes a bit more sense if you know how to do (or just know the answer to) part b of the question.

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

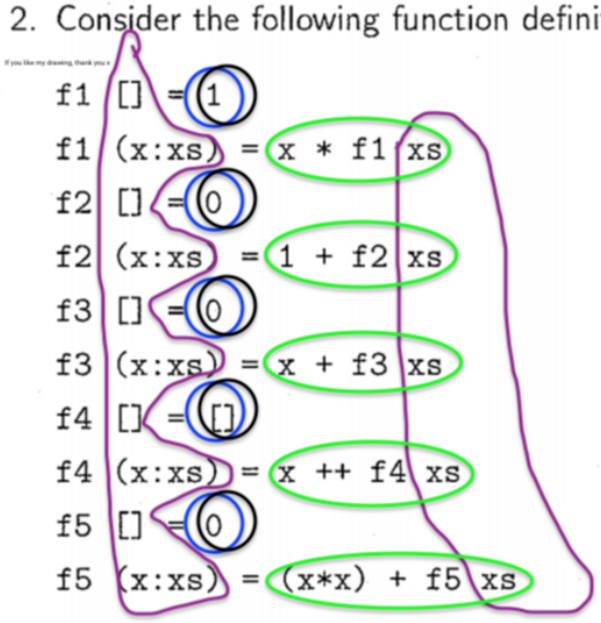
(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:




```
hof :: a -> (a -> a -> a) -> [a] -> a
```

Cool. x

Let's write out the rest, cause this is already taking pretty long...

If the purple list is empty, then we return the blue thing. Simple!

```
hof :: a -> (a -> a -> a) -> [a] -> a
hof e _ [] = e
```

They all have a common pattern of behaviour.

(I just called it 'e' for fun cause I'm a fun guy, but you can call it whatever. might be confusing if you call it 'a' though)

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

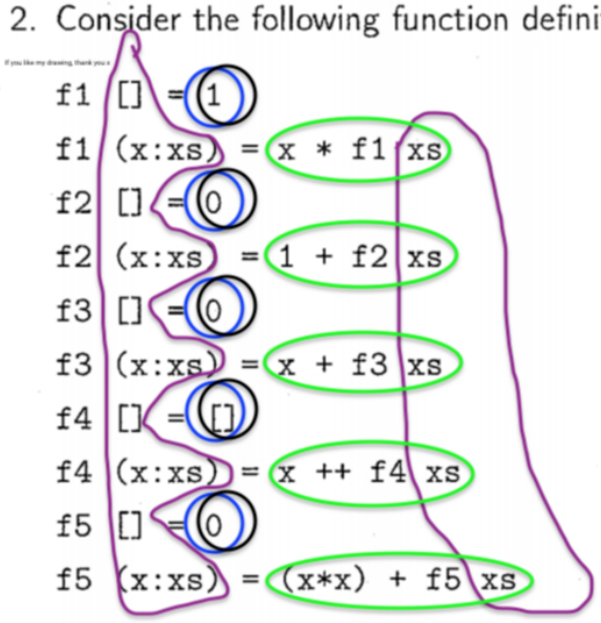
(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:



```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```

```
hof :: a -> (a -> a -> a) -> [a] -> a
hof e _ [] = e
```

Otherwise, the list is not empty.
So you apply the function to the head of the list
and the result of doing the function to the rest of the list:

```
hof e fx (x:xs) = fx x (hof e fx xs)
```

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:

```
f1 [] = 1
f1 (x:xs) = x * f1 xs
f2 [] = 0
f2 (x:xs) = 1 + f2 xs
f3 [] = 0
f3 (x:xs) = x + f3 xs
f4 [] = []
f4 (x:xs) = x ++ f4 xs
f5 [] = 0
f5 (x:xs) = (x*x) + f5 xs
```

```
hof :: a -> (a -> a -> a) -> [a] -> a
hof e _ [] = e
hof e fx (x:xs) = fx x (hof e fx xs)
```

Wow! What a perfect looking hof. You should be proud (of me lol)

Now for part b, we just have to call hof with appropriate arguments for each function.

For each function, this should look like:

f? = hof e fx

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:

```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```

e

fx

The main goal of part b is to figure out what these parameters of hof should be for each function.

1. hof f a -> (a -> a -> a) -> [a] -> a
2. hof e _ [] = e
3. hof e fx (x:xs) = fx x (hof e fx xs)

E.G:

(Spoiler alert!)

The answer for the first function will be:

'f1 = hof 1 (*)'

The '1' is the 'e' in line 2 and 3 from above.

The '(*)' is the 'fx' in line 2 and 3 from above.

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

2. Consider the following function definitions:

Remember,
'fx xs' will turn out to be just one thing (e.g. y)
that is the same type as x

```
f1 []     = 1
f1 (x:xs) = x * f1 xs
f2 []     = 0
f2 (x:xs) = 1 + f2 xs
f3 []     = 0
f3 (x:xs) = x + f3 xs
f4 []     = []
f4 (x:xs) = x ++ f4 xs
f5 []     = 0
f5 (x:xs) = (x*x) + f5 xs
```

```
hof :: a -> (a -> a -> a) -> [a] -> a
hof e _ [] = e
hof e fx (x:xs) = fx x (hof e fx xs)
```

Template:
f? = hof e fx

Answers:

```
f1 = hof 1 (*)
f2 = hof 0 myFunc2
        where myFunc2 x y = (1 + y)
f3 = hof 0 (+)
f4 = hof [] (++)
f5 = hof 0 myFunc5
        where myFunc5 x y = (x * x + y)
```

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]

That 'where' stuff is just Haskell syntax. I guess you just know that or you don't :/

2. Consider the following function definitions:

```
f1 [] = 1
f1 (x:xs) = x * f1 xs
f2 [] = 0
f2 (x:xs) = 1 + f2 xs
f3 [] = 0
f3 (x:xs) = x + f3 xs
f4 [] = []
f4 (x:xs) = x ++ f4 xs
f5 [] = 0
f5 (x:xs) = (x*x) + f5 xs
```

Ok Great! That was 9 hours well spent.

Let me wrap this up by telling you that the answer to part c is:

"Yes, hof is provided by the prelude under the name 'foldr'."

If this helped you learn something, why not pay it forward and teach someone else something to help them with their exams?

ok bye! x

They all have a common pattern of behaviour.

(a) Write a higher-order function `hof` that captures this common behaviour

[8 marks]

(b) Rewrite each of `f1`, `f2`, ... above to be a call to `hof` with appropriate arguments.

[20 marks]

(c) Is `hof` provided by the Haskell Prelude (under another name)?
If so, what is it called?

[5 marks]