Here are some Haskell questions like those you might find on your final exam. (Of course, these exact questions are now quite unlikely to appear on your final exam.) Prolog questions would look very much like those you have already seen on your midterm exam.

Question 1: [20 marks]

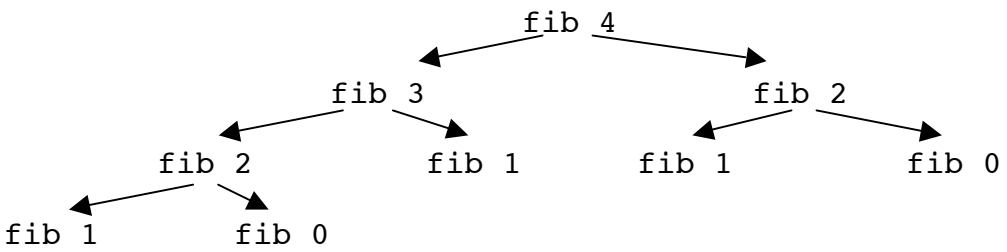
The Fibonacci numbers can be defined in Haskell like this:

```
fib 0 = 0
fib 1 = 1
fib n = (fib (n - 1)) + (fib (n - 2)) -- for integers n > 1
```

1a) [5] Using the `fib` function shown above, write another Haskell function called `generateAllFibs` that returns the list of all Fibonacci numbers in ascending order.

1b) [5] Now write yet another Haskell function called `isFib` that takes one argument, an integer, and returns `True` if the argument is a Fibonacci number and `False` otherwise.

1c) [10] The `fib` function shown on the previous page is amazingly inefficient. For example, just computing `fib 4` requires a call to `fib 3` and `fib 2`, but `fib 3` calls `fib 2` and `fib 1`, and so on. You could diagram the function calls like this:



Now imagine what that graph looks like if you want to compute `fib 30`...it staggers the imagination, and it certainly wouldn't fit on this page. Consequently, your answer to question 2a will also be horribly inefficient, assuming you followed the directions.

Your task now is to create a much more efficient Haskell function called `generateAllFibsFaster` that returns the list of all Fibonacci numbers in ascending order but does so without all the function calls required by your `generateAllFibs` function.

Question 2: [5 marks]

Recall that tail recursion is way of writing functions which avoids much of the computational expense associated with "ordinary" recursion (also known as "primitive recursion" or "augmenting recursion") by eliminating postponed computations. However, when we write Haskell functions using tail recursion, we don't seem to get the benefits that we might expect. Why is that?

Question 3: [10 marks]

Provide appropriate polymorphic type declarations for the following standard Haskell functions plus one non-standard Haskell function that you should be familiar with:

```
length ::  _____________________________________________________
```

```
reverse ::  _____________________________________________________
```

```
head ::  _____________________________________________________
```

```
zip ::  _____________________________________________________
```

```
oska ::  _____________________________________________________
```

(or whatever you called your top-level function in the second term project)

Question 4: [5 marks]

Write your own version of Haskell's `take` function. Be sure to provide the function's type declaration. The `take` function works like this:

```
Main> take 0 "abc"
""
Main> take 1 "abc"
"a"
Main> take 2 "abc"
"ab"
Main> take 3 "abc"
"abc"
Main> take 4 "abc"
"abc"
Main> take 2 [1,2,3,4]
[1,2]
```

Question 5: [10 marks]

Write your own version of Haskell's `zipWith` function, which combines the `zip` and `map` functions according to your textbook. Be sure to provide the function's type declaration. Here are some examples of the `zipWith` function in action:

```
Main> zipWith (++) ["a","b","c"]["d","e","f"]
["ad","be","cf"]
Main> zipWith max [1,2,3,4][4,3,2,1]
[4,3,3,4]
Main> zipWith (+) [1,2,3][4,5]
[5,7]
Main> zipWith min ["az"]["xbc"]
["az"]
```