Study Guide: Midterm Exam

Overview

The Haskell exam will be written in class on Thursday, February 27 starting promptly at 8:30 a.m. Please ensure that you arrive early and are ready to start on time. You will have up to 80 minutes to complete the exam.

The exam is closed-book: students are not permitted to have any resources other than pencils and an eraser. This exam will constitute 15% of your overall course mark. Please see the course outline for more information.

Course Learning Outcomes to Be Assessed

The exam aims to assess the following learning outcomes:

- Explain, compare, and contrast declarative/functional and imperative programming paradigms;
- Explain how common algorithmic components, such as variables, decisions, and repetition, manifest themselves across these paradigms;
- Understand the λ-calculus, explain its relationship to functional programming, and manipulate and reduce λ-expressions;
- Read, explain, and modify Haskell code–this includes the ability to infer the most general type of an expression or function;
- Solve programming problems in Haskell by making use of:
    - functions, expressions, patterns, and guards
    - recursion
    - strong, static, polymorphic typing
    - tuples, lists, and other data structures
    - common patterns of computation over lists, including but not limited to folding, mapping, and filtering
    - lazy evaluation and infinite data structures
    - list comprehensions
    - algebraic data types
- Write generic code using parametric polymorphism;
- Hand-reduce functional expressions, especially using lazy evaluation.

The exam will assess all course content covered between the first day of class and the tutorial on Monday, February 24 (inclusive), whether listed explicitly above or not. This includes all material covered in lectures and tutorials, up to and including list comprehensions. Students should be able to complete all exercises on exercise sets one through four, individually and without the use of resources, as well as any problems of a similar nature. To be successful, students should also be able to complete all of the practice exercises below.

Practice Exercises

Solutions to these exercises will not be posted at present. Students are encouraged to attempt them and to present, discuss, and debate candidate solutions among each other. Some of these exercises or exercises like them may appear on the exam.

Questions 1 and 2 concerns the λ-calculus. The remaining questions concern Haskell.

1. Show how to represent the "nand" function in the λ-calculus using Church Booleans. Then, perform an example reduction to illustrate how it works.

2. It is not possible to directly define recursive functions in the λ-calculus. First, explain why. Second, as clearly but concisely as possible, explain how recursive functions can nevertheless be represented.

3. Write (declare and define) an `allSame` function that takes a list of integers and returns `True` if and only if the list does not contain distinct elements. Write it without using helper functions from the library.

   E.g.,

   ```
   ghci> allSame [5,5,5]
   True

   ghci> allSame [5,6,5]
   False
   ```

4. Write (declare and define) a `generate` function that behaves as illustrated below (again, write it without using helper functions from the library):

   ```
   ghci> generate 2 13
   [2,3,4,5,6,7,8,9,10,11,12,13]
   ```

5. Write (declare and define) a `filterOdds` function that behaves as illustrated below:

   ```
   ghci> filterOdds [1,2,5,6,2,1,9]
   [1,5,1,9]
   ```

   For this question, define this relation recursively *without* making use of list comprehensions.

6. Tracing practice! Use the following definitions to formally reduce the following expression. Show your answer precisely, using the format presented in this course. Follow the lazy evaluation strategy.

```
repeat x = x:repeat x

[]     ++ ys = ys                -- (++).1
(x:xs) ++ ys = x:(xs ++ ys)      -- (++).2

(x:_ ) !! 0 = x                  -- (!!).1
(_:xs) !! n = xs !! (n-1)        -- (!!).2

ghci> ([1,2] ++ repeat 3) !! 3
```

7. What is the (most general) type of the following function, as would be inferred by the Haskell compiler?

```
mystery (x:xs) (y:ys) n = (if x then Nothing else Just y, n):mystery xs ys (n+1)
mystery  _      _     n = []
```

8. Consider a function that takes two lists of integers and multiples each integer in the first list by the corresponding integer (i.e., at the corresponding position) in the second list.

   E.g., invoking:

   ```
   multiplyPairwise [2,4..10]  [10,20..50]
   ```

   … should produce:

   ```
   [20,80,180,320,500]
   ```

   Define this function recursively, without using any helper functions. Note: if the two lists are not of the same length, ignore any extra elements at the end of the longer list.

9. Write (declare and define) a `partition` function that takes an integer "pivot" as well as a list of integers. It produces a pair of lists where the first element is the list of all values strictly less than the pivot, and the second is the list of values greater than or equal to the pivot. Ensure that your solution is as efficient as possible by making only *one* traversal of the input list and by testing each element at most *once*.

   E.g., invoking:

   ```
   partition 4 [1,5,7,2,12,0]
   ```

   … should produce:

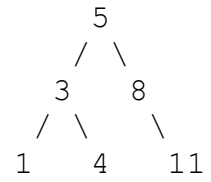
   ```
   ([1,2,0],[5,7,12])
   ```

10. The quicksort example provided in the lecture notes is not quite as efficient as possible. First, explain why. Then, redefine a version for sorting a list of integers, but in a more efficient way. Do so by making use of `partition`.

11. Consider the binary tree datatype defined in class:

    ```
    data BinTree a = Leaf | Node a (BinTree a) (BinTree a)
    ```

    Write a Haskell expression that represents the following binary tree structure:

12. Based on the datatype definition from the preview question, write (declare and define) an `insertBST` function that takes in integer and a binary search tree (BST) of integers and returns a BST with the first argument's value inserted at the appropriate leaf position. If that number is already in the tree, the operation has no effect.

13. Write (declare and define) a `buildBSTFromList` function that uses the function from the previous question to build a BST in the order of a given list. (Note: you are *not* being asked to implement tree balancing. If the input list is in ascending or descending order then the BST generated will be degenerate).

    Also: Is this function a fold, a map, a filter, or none of these? Explain your answer.

14. Write (declare and define) functions that each takes a BST and produces a list according to pre-order, in-order, and post-order traversals.

15. Write a function that takes two "index" integers and a list and produces the sub-list between the two indices. For example, the sub-list of `[10,20,30,40,50]` between indices `1` and `3`, inclusive, is `[20,30,40]`. Use the standard Prelude `take` and `drop` functions to accomplish this task.

16. Write a list comprehension that produces a list of all integer (x,y) pairs, with x and y each between 1 and 100, such that x raised to the power of y is less than 100,000.

17. Write a list comprehension that evaluates to the list of all numbers that are one less than the squares of the non-negative integers that are evenly divisible by either 3 or 5.

18. Declare the type of an `insertEverywhere` function that returns a list of all the ways that a given value can be inserted into a given list. For example, the character `'a'` can be inserted into the string `"bc"` in three different ways: `"abc"`, `"bac"`, and `"bca"`.

19. Write (declare and define) a `perms` function that returns a list of all the permutations of a given list. Define this recursively. In the recursive case, use a list comprehension and the `insertEverwhere` function from the previous question (you can assume a working implementation, even if you haven't defined it yet).

20. **Challenge:** efficiently define the `insertEverwhere` function from above.