CS 131 Quiz 1 Practice Problems

Modules 1–3: Haskell, Functional Programming, Lists, Pattern Matching, and Data Types

This is a set of practice problems to help you prepare for Quiz 1. These cover the same Modules 1–3 material as Quiz 1 (Haskell basics, functional programming, lists and pattern matching, and data types). These problems are ungraded and optional.

Problem 1: Bindings Are Not Assignment

Rowan says:

> "Haskell variables are basically the same as variables in Python. `x = 4` stores 4 in `x`, and later we can change `x` to something else."

What is the important misconception in Rowan's explanation?

Your answer should distinguish a **binding** from an imperative-style mutable variable.

Problem 2: What Does Purity Mean?

Sam says:

> "A pure function is just a function that always returns the same type."

Explain what is wrong with this definition. What property do we actually care about when we call a function **pure**?

If useful, connect your answer to side effects or referential transparency.

Problem 3: Tika's Divergent Binding

Tika says:

> "If I bind `x = forever 0`, then my whole program immediately loops forever."

Is that necessarily true in Haskell?

Give one small example where a name is bound to a divergent expression, but the overall computation still terminates. Explain why the divergent expression is never evaluated.

Problem 4: `map` Versus `filter`

Without evaluating every element one at a time, describe the essential difference between:

```
map (< 3) [5,2,9,1,0,4]
```

and:

```
filter (< 3) [5,2,9,1,0,4]
```

Your answer should say something about both:

- what happens to the elements, and
- the shape or type of the resulting list.

Problem 5: : Versus ++

Morgan writes:

> ": and ++ both add things to lists, so `3 : [4,5]` and `[3] ++ [4,5]` are basically the same operation and should have the same type."

The two expressions do produce the same value.

Is Morgan right that the two operators have the same type? Explain the important difference between what appears on the left-hand side of `:` and what appears on the left-hand side of `++`.

Problem 6: Pattern Matching Is About Structure

Casey tries to write:

```
classify (n < 0) = "negative"
classify (n >= 0) = "nonnegative"
```

Why is this not how pattern matching works?

A. Haskell does not allow Boolean expressions.

B. Patterns describe the structure or literal form of a value; they do not run arbitrary tests.

C. Pattern matching only works on lists.

D. The operator `<` cannot be used with numbers.

If you choose an answer, briefly explain it.

Problem 7: Reading a Recursive List Function

Consider:

```
mystery [] = 0
mystery (_:xs) = 1 + mystery xs
```

What does `mystery` compute?

More importantly, explain how the two function cases correspond to the recursive structure of a list.

Problem 8: Find the Existing Abstraction

Nia wants to double every number in a list. She writes a recursive function using explicit calls to `null`, `head`, and `tail`.

Dev says:

> "This can work, but the recurring traversal pattern is already captured by a higher-order function."

What higher-order function is Dev referring to?

What part of Nia's program should become the argument to that higher-order function?

Problem 9: Constructors Can Be Functions

Given:

```
data Animal
  = Cat String Int
  | Bird String
  | Fish
```

What are the types of the following constructors?

```
Cat
Bird
Fish
```

No explanation longer than one sentence is necessary.

Problem 10: `type` Versus `data`

Eli claims these declarations do essentially the same thing:

```
type Name = String
```

and:

```
data Name = Name String
```

Explain one important difference between the two declarations.

Problem 11: Recursive Data Does Not Force Recursion

Given:

```
data IntTree
  = Leaf Int
  | Node IntTree IntTree
```

Avery says:

> "Any function on `IntTree` must be recursive."

Is Avery's statement literally true?

If not, give a more accurate statement about the relationship between recursive data structures and recursive functions.

Problem 12: `Show` Versus `Eq`

Given:

```
data Mood = Happy | Sad
  deriving Show
```

Which expression is guaranteed to work?

A. `Happy == Sad`

B. `show Happy`

C. `Happy < Sad`

D. All three

What additional derived capability would be needed for equality testing?

Problem 13: Type Class or Data Type?

Dani says:

> "`Eq` is a datatype containing values that can be compared."

What is wrong with this description?

Give a more accurate one-sentence description of what a Haskell **type class** is.

Problem 14: Two Models of Computation

Two students describe computation:

> **Mina:** "A program works mainly by changing stored state step by step."
>
> **Leo:** "A program works mainly by evaluating expressions to values."

Which description is closer to the functional-programming model emphasized in this course?

Explain the contrast without claiming that all real programming languages fit perfectly into only one category.

Problem 15: What Does Purity Buy Us?

Alex says:

> "Purity seems like an arbitrary restriction. Why care whether functions have side effects?"

Give one concrete reasoning advantage of purity that helps explain another Haskell feature studied in the course.

Possible ideas to connect include referential transparency and lazy evaluation.

Problem 16: Is This Course Just Weird Haskell?

A friend says:

> "So far this course just seems like learning weird Haskell syntax."

Give a short response using **two** ideas from Modules 1–3 that are really about programming languages more broadly, not merely about Haskell syntax.

You may draw on ideas such as:

- models of computation,
- representation choices,
- functions as values,
- evaluation strategy,
- type systems,
- data representation,
- programs as data,
- language design choices.

Problem 17: Programs as Data Before a Full Interpreter

HW3 uses a datatype or tree representation of arithmetic expressions and then translates or evaluates those structures with a stack machine.

Why is this already an example of the broader idea:

> **Programs can be data**

even though you have not yet built a full interpreter?

Problem 18: What Does the Type Variable Really Mean?

Quinn says:

> "Because `map` has a polymorphic type, it can take a list containing several unrelated element types. The `a` just changes type for each element."

Is Quinn correct?

Explain what repeated occurrences of the same type variable, such as `a`, mean within one use of a function.

Problem 19: Stack Machine: Why This Order?

Suppose the arithmetic expression:

$$(2 + 3) * 4$$

is translated into stack-machine instructions.

Why must the instructions corresponding to `2 + 3` occur before the final multiplication instruction?

No actual stack-machine code is required. Explain the dependency between the intermediate result and the later multiplication.