

## CS 131 Quiz 2 Practice Problems

Modules 4–6: Code as Data, Evaluation, Environments, Scope, Closures, and Parsing

This is a set of practice problems to help you prepare for Quiz 2. They cover the same material as Quiz 2: Modules 4–5 (abstract syntax, evaluation, substitution, evaluation strategies, environments, scope, and closures), the ideas behind HW 4 and HW 5, and a little conceptual parsing from Module 6. These problems are ungraded and optional.

Unless a problem says otherwise, expressions in our small interpreted language use the datatype from Modules 4 and 5:

```
data Op = PlusOp | MinusOp | TimesOp | DivOp

type VarName = String

data Exp = Var      VarName
         | Num      Double
         | BinOp    Exp Op Exp
         | Let      VarName Exp Exp
         | FunDef   VarName Exp
         | Apply    Exp Exp
```

## Problem 1: Armani and Gal: Concrete or Abstract?

Here are four snippets of code related to HW 5.

(i) A piq program, as you would type it in a file:

```
x = 20
square x
move right 30
```

(ii) A Haskell value:

```
Program [] [Assign "x" (Num 20), Square (Var "x"), Move Rt (Num 30)]
```

(iii) A Haskell type declaration:

```
data Direction = Lt | Rt | Up | Dn
```

(iv) A Haskell helper function:

```
-- | The midpoint of two positions
midpoint :: XYPos -> XYPos -> XYPos
midpoint (x1,y1) (x2,y2) = ((x1+x2)/2, (y1+y2)/2)
```

(a) Classify each snippet. **Check exactly one box in each row.**

|       | Concrete syntax<br>for piq | Abstract syntax for piq,<br>written in Haskell | Neither                  |
|-------|----------------------------|------------------------------------------------|--------------------------|
| (i)   | <input type="checkbox"/>   | <input type="checkbox"/>                       | <input type="checkbox"/> |
| (ii)  | <input type="checkbox"/>   | <input type="checkbox"/>                       | <input type="checkbox"/> |
| (iii) | <input type="checkbox"/>   | <input type="checkbox"/>                       | <input type="checkbox"/> |
| (iv)  | <input type="checkbox"/>   | <input type="checkbox"/>                       | <input type="checkbox"/> |

(b) Armani says, “Snippet (iii) is clearly abstract syntax!” Gal says, “Snippet (iii) is clearly concrete syntax!” Whose statement is more accurate?

**Check exactly one box.**

- ☐ A. Armani’s
- ☐ B. Gal’s
- ☐ C. Both, depending on which language we have in mind
- ☐ D. Neither

(c) In one sentence, explain your answer to (b).

**Problem 2: Write the AST**

Write the abstract syntax tree, as a Haskell value of type `Exp`, for each expression. Use the usual precedence rules.

(a)  $10 - 2 * 3$

---

(b)  $(1 + 2) / (3 - 4)$

---

### Problem 3: Three Bad Representations

Each attempt below is a flawed way to represent arithmetic expressions as Haskell data.

(a) Rowan's attempt:

```
data Exp = Double | Plus Exp Exp | Times Exp Exp
```

Why can't Rowan represent  $3.14 + 1$ ?

**Check exactly one box.**

- ☐ A. `Double` here is just a constructor name with no fields, so it carries no number.
- ☐ B. Haskell does not allow a constructor named `Double`.
- ☐ C. `Plus` should take `Doubles`, not `Exps`.

(b) Jordan's attempt:

```
data Paren = LeftParen | RightParen
data Exp = Num Double | BinOp Exp Op Exp | Parenthesized Paren Exp Paren
```

Which statements are true?

**Check all boxes that apply** (more than one may be correct).

- ☐ A. It is faithful to concrete syntax rather than abstract syntax.
- ☐ B. The shape of a `BinOp` tree already records grouping, so the parentheses add nothing.
- ☐ C. It allows nonsense values such as `Parenthesized RightParen (Num 2) RightParen`.
- ☐ D. Without it, there is no way to represent  $(1 + 2) * 3$ .

(c) Avery's attempt:

```
type Op = String
data Exp = Num Double | BinOp Exp Op Exp
```

Write one value of type `Exp` that Avery's representation allows, but that our real `Op` type (`PlusOp | MinusOp | TimesOp | DivOp`) rules out.

---

## Problem 4: Should Parsing and Formatting Be Inverses?

Let `parse` turn source text into an AST and `format` turn an AST back into text.

Which property should we usually expect to hold?

**Check exactly one box.**

- ☐ A. `parse (format ast) == ast`, but not necessarily `format (parse source) == source`
- ☐ B. `format (parse source) == source`, but not necessarily `parse (format ast) == ast`
- ☐ C. Both
- ☐ D. Neither

Briefly explain why the other direction can fail.

## Problem 5: Step Through eval

Consider:

```
bigExp =  
  BinOp  
    (BinOp  
      (BinOp (Num 2.0) PlusOp (Num 3.0))  
        TimesOp  
        (BinOp (Num 5.0) MinusOp (Num 8.0)))  
      MinusOp  
      (Num 0.5)
```

(a) Write `bigExp` in ordinary infix notation.

---

(b) What is its value? Show the intermediate values.

*Hint:* it may help to draw the tree first.

## Problem 6: Implement `substitute`

Recall that  $e[x \rightarrow e']$  is implemented as `substitute e' (Var x) e`. Fill in the blanks.

```
substitute :: Exp -> Exp -> Exp -> Exp
```

```
substitute _ (Var v) (Num n) = _____
```

```
substitute replacement (Var v) (Var w) =  
  if v == w then _____ else _____
```

```
substitute replacement var (BinOp left op right) =  
  _____
```

```
substitute _ _ _ = error "bad substitution"
```

## Problem 7: When Call-by-Value Hurts

Suppose `loop` is an expression whose evaluation never terminates, and consider applying the constant function  $x \mapsto 10$  to it:

Apply `(FunDef "x" (Num 10)) loop`

Fill in what happens under each strategy.

Under call-by-name: \_\_\_\_\_

Under call-by-value: \_\_\_\_\_

## Problem 8: Currying in the AST

(a) Write the function  $x \mapsto (y \mapsto x * y)$  as a value of type `Exp`.

---

(b) Write an `Exp` that applies this function to 6 and then 7. (You may write `f` for your answer to (a).)

---

(c) Using the substitution-based evaluator, what is the result of the *first* application (to 6)?

---

## Problem 9: The Snag

Here is an early version of our substitution-based evaluator (`evalOp` and `substitute` are as in Module 4.2):

```
eval :: Exp -> Exp
eval (Num x)           = Num x
eval (BinOp left op right) = evalOp (eval left) op (eval right)
eval (FunDef varname body) = FunDef varname body
eval (Apply (FunDef varname body) arg) =
  eval (substitute arg (Var varname) body)
```

We use it to evaluate a curried two-argument function, applied to 10 and then 100:

```
eval (Apply (Apply (FunDef "x" (FunDef "y" (BinOp (Var "x") PlusOp (Var "y"))))
              (Num 10))
          (Num 100))
```

(a) What happens?

Check exactly one box.

- ☐ A. It returns `Num 110`.
- ☐ B. It fails at run time because no equation for `eval` matches (non-exhaustive patterns).
- ☐ C. It returns `FunDef "y" (BinOp (Num 10) PlusOp (Var "y"))`.
- ☐ D. Haskell rejects the program at compile time with a type error.

(b) Why?

Check exactly one box.

- ☐ A. The outer `Apply`'s function position is itself an `Apply`, not literally a `FunDef`.
- ☐ B. A `FunDef` has only one parameter, so curried functions cannot be represented.
- ☐ C. `substitute` has no case for `Num`.
- ☐ D. The argument `Num 100` is evaluated before the function.

(c) The fix is to evaluate the function position first. Fill in the blank:

```
eval (Apply fexp arg) =
  let FunDef varname body = _____
  in eval (substitute arg (Var varname) body)
```

## Problem 10: Why Environments?

(a) Why did Module 5.1 move from substitution to an environment?

**Check exactly one box.**

- ☐ A. Repeated substitution walks through large pieces of syntax again and again, which can cost  $O(N^2)$  work.
- ☐ B. Substitution gives wrong answers for all programs with variables.
- ☐ C. Haskell does not allow recursive functions over `Exp`.
- ☐ D. Environments make the evaluator use call-by-name.

(b) With `type Env = [(VarName, Exp)]`, did switching to an environment fix the cost? Name a better data structure for the environment.

## Problem 11: Three Uses of `let`

(a) What does each expression evaluate to?

```
let y = 100 in let x = 10 in x + y
```

```
let x = 100 in let x = 3 in x
```

(b) In Module 5.1, we gave `Let` a meaning using function definition and application. Fill in the three blanks:

```
eval (Let varname boundExp body) =  
    eval (Apply (FunDef _____) _____)
```

(c) `let` can also name a function. What does this evaluate to?

```
Let "f" (FunDef "x" (BinOp (Var "x") PlusOp (Num 1)))  
    (Apply (Var "f") (Num 100))
```

## Problem 12: Which Environment?

```
let n = 10 in
  let f x = n * x in
    let n = 3 in
      f 4
```

(a) Under **dynamic** scope, which binding of **n** does the body of **f** use, and what is the result?

---

(b) Under **static** scope?

---

## Problem 13: Can We Even Function?

Using the environment-based evaluator **without** closures, in which `eval env (FunDef varname body)`  
`= FunDef varname body`, we evaluate:

```
eval [] (Apply (Apply (FunDef "a" (FunDef "b" (BinOp (Var "b") PlusOp (Var "a"))))
                    (Num 100))
        (Num 42))
```

and get **\*\*\* Exception: unbound identifier: a.**

(a) Explain in a sentence or two why the binding for `a` is lost.

(b) What does the closure-based evaluator return instead?

---

## Problem 14: Using Maybe

In HW 5, `piq`'s environment is a `Data.Map`, and `Map.lookup :: Ord k => k -> Map k v -> Maybe v`. Fill in the two cases of `lookupVar`, which crashes with an error if the variable is not defined:

```
lookupVar :: Name -> Env -> Double
lookupVar name env =
  case Map.lookup name env of
```

\_\_\_\_\_ -> \_\_\_\_\_

\_\_\_\_\_ -> \_\_\_\_\_

## Problem 15: Random Art

In HW 4, Random Art expressions have this type, and `eval` computes the value of an expression at a point  $(x, y)$  with  $x, y \in [-1, 1]$ :

```
data Exp = X | Y
         | Times Exp Exp      -- product of e1 and e2
         | Avg   Exp Exp      -- average of e1 and e2
         | SinPi Exp          -- sin (pi * e)
         | CosPi Exp          -- cos (pi * e)

type Point = (Float, Float)

eval :: Exp -> Point -> Float
```

(a) Fill in the `Avg` case:

`eval (Avg e1 e2) point = _____`

(b) Every Random Art expression must evaluate to a value in  $[-1, 1]$  whenever  $x$  and  $y$  are in  $[-1, 1]$ . Suppose we added a constructor `Plus Exp Exp`, which evaluates to the sum of its two subexpressions.

Write two expressions of type `Exp` that use `Plus`:

(i) one whose value stays in  $[-1, 1]$  at **every** point with  $x, y \in [-1, 1]$ :

\_\_\_\_\_

(ii) one that **breaks** the rule. Also give a point  $(x, y)$  where its value falls outside  $[-1, 1]$ , and that value.

Expression: \_\_\_\_\_

Point  $(x, y)$ : \_\_\_\_\_ Value: \_\_\_\_\_

## Problem 16: Regular Expression Matches

In HW 4, `rexpMatches :: RegExp -> Text -> [(Match, Text)]` returns every way that a regular expression can match a *prefix* of the input, paired with the remaining text.

Write the result of each call.

(a) `rexpMatches (Letter 'a') "baaa"` = \_\_\_\_\_

(b) `rexpMatches (Concat (Letter 'a') (Letter 'b')) "abc"` = \_\_\_\_\_

(c) `rexpMatches (Concat (Star (Letter 'a')) (Letter 'b')) "aab"` = \_\_\_\_\_

## Problem 17: piq Environments

(a) What are the side lengths of the three squares, in order? And how many lines of Python does `x = x * 2` produce?

```
x = 10
square x
x = x * 2
square x
x = x * 2
square x
```

Side lengths: \_\_\_\_\_ Lines of Python: \_\_\_\_\_

(b) What is the side length of the square?

```
i = 100
for i from 1 to 3 { dot }
square i
```

\_\_\_\_\_

(c) How many times does the loop body run, and what is `n` after the loop?

```
n = 3
for i from 1 to n {
  n = n + 1
}
```

Iterations: \_\_\_\_\_ Final `n`: \_\_\_\_\_

## Problem 18: Which Scope Does piq Use?

In HW 5, a procedure body is compiled in a local environment made from **the caller's environment** plus the parameters. Consider:

```
define tile(s) {  
  square s + w  
}  
  
w = 10  
tile(5)  
w = 30  
tile(5)
```

(a) What are the side lengths of the two squares?

First square: \_\_\_\_\_ Second square: \_\_\_\_\_

(b) The free variable `w` in `tile`'s body is resolved using the environment where `tile` is *called*. Which scope rule is that?

**Check exactly one box.**

- ☐ A. Dynamic scope
- ☐ B. Static (lexical) scope

**Problem 19: Associativity**

Consider the input  $10 - 3 - 2$ .

(a) Subtraction is left-associative. Write the AST, as a value of type **Exp**, that a parser should produce.

---

(b) What value does your AST evaluate to? What value would we get if a buggy parser grouped the input from the right instead?

Left grouping: \_\_\_\_\_ Right grouping: \_\_\_\_\_

## Problem 20: Parse Trees and Keywords

(a) For the grammar

```
Expr    ::= Number "+" Number
Number  ::= Digit+
Digit   ::= "0" | "1" | "2" | ... | "9"
```

the input `12 + 3` has a concrete syntax tree (parse tree) and an abstract syntax tree `BinOp (Num 12) PlusOp (Num 3)`. Which of these appear in the **concrete** syntax tree but **not** in the abstract syntax tree?

**Check all boxes that apply** (more than one may be correct).

- ☐ A. Separate `Digit` nodes for 1 and 2
- ☐ B. The `"+"` token as a leaf
- ☐ C. The fact that the expression is an addition
- ☐ D. The numbers twelve and three

(b) In our language's concrete syntax, `let` is a keyword, as in `let x = 5 in x + 1`. A careless parser recognizes the keyword by checking whether the input *starts with* the characters `l`, `e`, `t`. What goes wrong on the input `letter + 1`, where `letter` is a variable?

**Check exactly one box.**

- ☐ A. It reads `let` as the keyword and then tries to parse `ter + 1` as the rest of a `let` expression, which fails.
- ☐ B. It rejects every expression that contains a variable.
- ☐ C. It evaluates `letter` before parsing the rest of the input.

(c) Which approach fixes it?

**Check exactly one box.**

- ☐ A. Read a whole identifier first, then check whether it is exactly `"let"`.
- ☐ B. Check for `let` only at the end of the input.
- ☐ C. Remove all whitespace from the input first.