CS 131 Quiz 3 Practice Problems

Modules 6–8: Parsing, Functors and Monads, and the Lambda Calculus

These practice problems will help you prepare for Quiz 3. They cover the same material as the quiz: grammars, parse trees, and parser combinators (Module 6); functors, `Maybe`, bind, and `do` notation (Module 7); and β-reduction, Church encodings, and fixed points (Module 8). They are ungraded and optional. The problems are a bit longer than quiz problems, so working through all of them should leave you well prepared.

Lambda-calculus expressions are written fully parenthesized, as in the modules: `(λx.M)` for an abstraction and `(M N)` for an application. $\to_\beta$ means one β-reduction step and $\to^*$ means zero or more.

Problem 1: Concrete Syntax Trees and ASTs

Here is a tiny grammar from Module 6:

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

(a) Draw the concrete syntax tree (parse tree) for `45 + 6`.

(b) Using `data Expr = Num Double | Add Expr Expr`, write the AST for `45 + 6`.

(c) Which statements are true?

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

- ☐ A. Concrete syntax includes keywords, punctuation, and whitespace.
- ☐ B. An AST usually keeps the parentheses that appeared in the source.
- ☐ C. `1 + 2` and `( 1 + 2 )` can parse to the same AST.
- ☐ D. A parser for arithmetic computes the value of the expression it reads.
- ☐ E. A parse tree follows the grammar rules more closely than an AST does.
- ☐ F. `x + 1` in one language and `(+ x 1)` in another can have the same abstract syntax.

Problem 2: A Grammar for Lists

This grammar describes nested lists of numbers. `items?` means "optionally an `items`."

```
list  ::= "[" items? "]"
items ::= item ("," item)*
item  ::= number | list
```

(a) Which strings are valid `lists`?

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

☐ A. `[]`

☐ B. `[1, 2]`

☐ C. `[1,]`

☐ D. `[[], [3]]`

☐ E. `1, 2`

☐ F. `[[1, [2]]]`

☐ G. `[1 2]`

(b) Which names in the grammar are nonterminals, and which are terminals?

(c) Draw the parse tree for `[5, []]`.

(d) Change the grammar so that a trailing comma, as in `[1, 2,]`, is also allowed.

Problem 3: Ambiguity, Precedence, and Associativity

Consider this grammar:

```
expr ::= expr "+" expr | expr "*" expr | number
```

(a) Draw **two different** parse trees for `2 * 3 + 4`. What value does each tree give if we evaluate it?

(b) Why is a grammar with two parse trees for the same string a problem for a parser?

(c) Rewrite the grammar using two levels, `expr` and `term`, so that `*` binds more tightly than `+`.

(d) The piq grammar is below; repeated operators combine from the left.

```
expr   ::= term   (("+" | "-") term)*
term   ::= factor (("*" | "/") factor)*
factor ::= number | name | "(" expr ")"
```

Using constructors `Num`, `Var`, `Add`, `Sub`, and `Mul`, write the AST for `x + 5 * 2 - 1`.

Problem 4: Lambda Calculus as Data

Here is the grammar of the lambda calculus and a Haskell type for its abstract syntax:

```
expr ::= variable | (expr expr) | (λ variable . expr)

type VarName = String
data LambdaExpr = Var VarName
                | Lambda VarName LambdaExpr
                | Apply LambdaExpr LambdaExpr
```

(a) Which are valid, fully parenthesized lambda expressions?

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

☐ A. `x`

☐ B. `(λx x)`

☐ C. `((λy.(y y)) (λz.z))`

☐ D. `(f g h)`

☐ E. `(λx.(λy.(y x)))`

☐ F. `(λ(x y).x)`

(b) Write the abstract syntax for `(f (λx.y))`.

(c) Write the abstract syntax for `((λx.(x x)) (λy.y))`.

(d) Write the concrete syntax for `Lambda "x" (Apply (Var "x") (Var "z"))`. Which of its variables are free?

Problem 5: Parser Combinators

`digit :: Parser Char`	one digit
`number :: Parser Integer`	a whole number
`identifier :: Parser String`	a name
`text :: String -> Parser String`	exactly this text
`sym :: Char -> Parser Char`	one given symbol
`p <+> q`, `p <-+> q`, `p <+-> q`	sequence; keep both / the right / the left
`p <|> q`	try `p`; if it fails, try `q`
`p >>=: f`	apply `f` to `p`'s result
`p >>: v`	if `p` succeeds, return `v`
`many p`, `some p`	zero or more / one or more
`sepBy p s`	zero or more `p`s separated by `s`
`parens p`	`p` inside parentheses
`parse :: Parser a -> String -> a`	run a parser on the whole input

(a) What does each expression produce? Write "error" if it fails.

```
parse (number <+> number <+> number) "1 2 3"
```
_____

```
parse (sepBy number (sym ',')) "4, 5, 6"
```
_____

```
parse (many digit) ""
```
_____

```
parse (some digit) ""
```
_____

```
parse (text "dot") "dotty"
```
_____

```
parse (text "pen" >>: 7) "pen"
```
_____

(b) A procedure call looks like `box(20, 5)`, with the grammar rule `call ::= name "(" exprs? ")"`. Using `name :: Parser String`, `expr :: Parser Expr`, and `Call :: String -> [Expr] -> Stmt`, write `call :: Parser Stmt`.

(c) Module 7 says that `>>=:` is **not** bind. Which description is right?

Check exactly one box.

☐ A. `p >>=: f` is `fmap f p`: `f` is an ordinary function `a -> b`.

☐ B. `p >>=: f` is `p >>= f`: `f` returns another parser.

☐ C. `p >>=: f` runs `f` first and then `p`.

Problem 6: Functors

(a) Fill in the result of each expression.

```
fmap (*3) (Just 4)          _____
fmap (*3) Nothing           _____
fmap (== 42) (Just 0)       _____
fmap length ["hi", "there"] _____
```

(b) The `Functor` class is `class Functor f where fmap :: (a -> b) -> f a -> f b`. Why do we write `instance Functor Tree` and not `instance Functor (Tree Int)`?

(c) Someone writes this instance for their own copy of `Maybe`:

```
instance Functor MyMaybe where
  fmap _ MyNothing  = MyNothing
  fmap g (MyJust x) = MyNothing
```

It typechecks. Which functor law does it break, and on what input?

(d) Check the composition law `fmap (g . h) = fmap g . fmap h` on one example by evaluating both sides of:

```
fmap (show . (+1)) (Just 1)        (fmap show . fmap (+1)) (Just 1)
```

Problem 7: Composing Functions That Might Fail

```
safeSqrt :: Double -> Maybe Double
safeSqrt x = if x < 0 then Nothing else Just (sqrt x)

safeReciprocal :: Double -> Maybe Double
safeReciprocal 0 = Nothing
safeReciprocal x = Just (1 / x)
```

We want `reciprocalOfRoot x`, which computes $1/\sqrt{x}$ and fails if either step fails.

(a) Why doesn't `reciprocalOfRoot = safeReciprocal . safeSqrt` typecheck?

(b) Write `reciprocalOfRoot` using `case` and pattern matching on `Just` and `Nothing`.

(c) Write it again using `>>=`, and then a third time using `do` notation.

(d) Fill in the results, and for each `Nothing`, say which step failed.

`reciprocalOfRoot 4` _____ `reciprocalOfRoot 0` _____

`reciprocalOfRoot (-9)` _____

Problem 8: `do` Notation

Assume `safeSqrt` and `safeReciprocal` from Problem 7, and

```
atLeast :: Double -> Double -> Maybe ()
atLeast limit y = if y >= limit then Just () else Nothing
```

(a) Rewrite using `do` notation:

```
f x = safeSqrt x >>= \a ->
      safeReciprocal a >>= \b ->
      pure (a + b)
```

(b) Rewrite this `do` block using `>>=` (and `let ... in`), then give `g 1`, `g 4`, `g 8`, and `g 0`.

```
g x = do
  a <- safeReciprocal x
  let b = a * 4
  atLeast 1 b
  safeSqrt b
```

(c) This definition has two mistakes. Find and fix them.

```
bad :: Double -> Maybe Double
bad x = do
  a <- safeSqrt x
  doubled <- 2 * a
  doubled
```

(d) What is `[1, 2, 3] >>= \x -> [x, -x]`? _____

(e) True or false: in a `Maybe` `do` block, the `do` syntax itself checks for `Nothing`. _____

Problem 9: Church Encodings

Use these definitions:

```
True   = (λx.(λy.x))                  Pair   = (λx.(λy.(λb.((b x) y))))
False  = (λx.(λy.y))                  Fst    = (λp.(p True))
Not    = (λb.((b False) True))        Snd    = (λp.(p False))
And    = (λa.(λb.((a b) False)))      0      = (λf.(λx.x))
IsZero = (λn.((n (λx.False)) True))   Succ   = (λn.(λf.(λx.(f ((n f) x)))))
```

(a) Reduce `(Snd ((Pair p) q))`, one step per line. You may expand an abbreviation in a separate step.

(b) What do `((4 Not) False)` and `(IsZero 1)` reduce to?

(c) Reduce `(Succ 0)` to normal form. Which numeral do you get?

(d) Write `Xor`, which is `True` when exactly one of its two inputs is `True`. *Hint:* first write it in Haskell using only `if-then-else`, then translate.

(e) Let Ω = `((λx.(x x)) (λx.(x x)))`. What does `((And False)` Ω`)` reduce to if we always reduce the leftmost, outermost redex first? What happens if we reduce arguments first?

Problem 10: Fixed Points and Recursion

(a) Give a fixed point of $g(x) = 8 - x$ and show why it is one.

(b) Let `A = (λx.(g (x x)))`. Show one β-step of `(A A)`. Which part of the result is the expression you started with, and why does that matter for recursion?

(c) Here is a recursive Haskell function:

```
sumTo n = if n == 0 then 0 else n + sumTo (n - 1)
```

Write a **non-recursive** generator `sumToStep rec n`. What is `fix sumToStep 4`, where `fix f = f (fix f)`?

(d) `Y = (λf.((λx.(f (x x))) (λx.(f (x x)))))`. In your own words, what does `Y F` $=\beta$ `F (Y F)` say, and why does it give us recursion?

(e) Why does `Y` work when we reduce the leftmost, outermost redex first, but loop forever in a language that reduces every argument before calling a function?

(f) True or false: a language must have recursion built in (functions that can refer to their own names) in order to express recursive computations. _____