COMP 333 Practice Exam # 2 Solutions

Swift

Pattern Matching

1.) Consider the following enum definition:

```
enum SomeEnum {
  case foo(Int)
  case bar(Int, Int)
  case baz(Int, Int, Int)
}
```

Write a function named `test` which takes a value of type `SomeEnum`. The function should do the following:

- If given a `foo`, it should return the value in the `foo`
- If given a `bar`, it should return the sum of the two values in the `bar`
- If given a `baz`, it should return the sum of the **first** and **last** values in the `baz`. You should **not** introduce a variable for the second (middle) value in the `baz`.

An example call to the function follows: `test(SomeEnum.baz(1, 2, 3))`

```
func test(_ value: SomeEnum) -> Int {
  switch value {
    case .foo(let x):
      return x
    case .bar(let x, let y):
      return x + y
    case .baz(let x, _, let y):
      return x + y
  }
}
```

Generics and Higher-Order Functions

2.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine<A, B>(a: A, b: B) -> (A, B) {
  return (a, b)
}
```

3.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine2<A, B>(a: A) -> ((B) -> (A, B)) {
  return { b in (a, b) }
}
```

4.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine3<A, B>(tup: (A, B)) -> A {
  let (a, _) = tup
  return a
}
```

5.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine4<A, B>(a: A, f: (A) -> B) -> (A, B) {
  return (a, f(a))
}
```

6.) Consider the following `enum` definition:

```
enum Something<A, B, C> {
  case alpha(A)
  case beta(B)
  case gamma(C)
}
```

6.a.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine5<A, B, C>(s: Something<A, B, C>) -> (A, B, C) {
  Impossible to implement.  s holds one of an A, B, or C, and the
return type requires all three
}
```

6.b.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine6<A>(s: Something<A, A, A>) -> A {
  switch s {
    case .alpha(let a): return a
    case .beta(let a): return a
    case .gamma(let a): return a
  }
}
```

7.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine7<A, B>(f: (A) -> B, b: B) -> A {
  Impossible to implement.  f needs an A, but we only have a B.
}
```

8.) Consider the following enum definition representing lists:

```
indirect enum List<A> {
  case cons(A, List<A>)
  case empty
}
```

8.a.) Write a function named `partition` which takes a predicate and divides a generic list into a pair of returned generic lists. The first element of the pair holds all elements for which the predicate returned `true`, and the second element of the pair holds all elements for which the predicate returned `false`. An example call is below:

```
let (matching, nonmatching) =
  partition(list: List.cons(1, List.cons(2, List.empty)),
            pred: { e in e > 1 })
// matching: List.cons(2, List.empty)
// nonmatching: List.cons(1, List.empty)
```

```
func partition<A>(list: List<A>, pred: (A) -> Bool) -> (List<A>,
List<A>) {
  switch list {
    case .empty: return (List.empty, List.empty)
    case .cons(let head, let tail):
      let (restMatch, restNonMatch) =
        partition(list: tail, pred: pred)
      if pred(head) {
        return (List.cons(head, restMatch), restNonMatch)
      } else {
        return (restMatch, List.cons(head, restNonMatch))
      }
  }
}
```

8.b.) Write a function named `takeWhile` which returns a list of consecutive list elements for which a given predicate `pred` returns `true`. Once `pred` returns `false`, the list is returned. `takeWhile` is generic. Example calls are below:

```
let list = List.cons(1, List.cons(2, List.cons(3, List.empty)))
let first = takeWhile(list: list, pred: { e in e < 3 })
// first: List.cons(1, List.cons(2, List.empty))
let second = takeWhile(list: list, pred: { e in e < 2 })
// second: List.cons(1, List.empty)
let third = takeWhile(list: list, pred: { e in e > 1 })
// third: List.empty
```

```
func takeWhile<A>(list: List<A>, pred: (A) -> Bool) -> List<A> {
  switch list {
    case .cons(let head, let tail):
      if pred(head) {
        return List.cons(head, takeWhile(list: tail, pred: pred))
      } else {
        return List.empty
      }
    case .empty:
      return List.empty
  }
}
```

Protocols and Extensions

9.a.) Define a protocol named `Equals` which defines an `equals` method. `equals` returns `true` if two values equal each other. Example calls are below, assuming `Int` implements the `Equals` protocol:

```
1.equals(1) // returns true
2.equals(3) // returns false
```

```
protocol Equals {
  func equals(_ other: Self) -> Bool
}
```

9.b.) Implement the `Equals` protocol for `Int`, using `extension`. As a hint, `==` can be used to test if two integers are identical, as with `1 == 2`.

```
extension Int: Equals {
  func equals(_ other: Int) -> Bool {
    return self == other
  }
}
```

9.c.) Consider the following enum definition:

```
enum Thing<A> {
  case thing1
  case thing2(A)
}
```

Implement the `Equals` protocol for `Thing`, using `extension`. As a hint, you'll need to tell the compiler that `A` needs to implement the `Equals` protocol. Two `thing1` values should equal each other, and a `thing2` value should equal another `thing2` value if both `thing2` values contain the same value of type `A`.

```
extension Thing : Equals where A : Equals {
  func equals(_ other: Thing<A>) -> Bool {
    switch (self, other) {
      case (.thing1, .thing1):
        return true
      case let (.thing2(leftThing), .thing2(rightThing)):
        return leftThing.equals(rightThing)
      case _:
        return false
    }
  }
}
```

10.) The following code does not compile. Why not?

```
protocol Foo { func fooMethod() -> Bool }
extension Int : Foo { func fooMethod() -> Bool { return true } }
true.fooMethod()
```

Booleans do not implement the Foo protocol, and the Foo protocol is needed to call fooMethod.

Parser Combinators

11.) Consider the following grammar, token `enum`, and function signatures:

```
exp ::= 'true' |
        'false' |
        '(' 'if' exp exp exp ')' |
        '(' 'while' exp exp ')'

enum Token {
  case leftParen
  case rightParen
  case ifToken
  case whileToken
  case trueToken
  case falseToken
}

func tokenP(_ token: Token) -> Parser<Unit>
func andP(_ lhs: @escaping @autoclosure () -> Parser<Unit>,
          _ rhs: @escaping @autoclosure () -> Parser<Unit>)
  -> Parser<Unit>
func orP(_ lhs: @escaping @autoclosure () -> Parser<Unit>,
         _ rhs: @escaping @autoclosure () -> Parser<Unit>)
  -> Parser<Unit>
```

Note that `andP` and `orP` have been modified from assignment 2 to operate on `Parser<Unit>`. As such, they won't parse in abstract syntax trees, but they will still succeed if they are able to parse, and fail if they are not. Write a parser which will accept this grammar, using `tokenP`, `andP`, and `orP`. You do not need to worry about producing the abstract syntax tree. You may assume you are defining a parser named `expressionP`.

```
let ifP = andP(tokenP(Token.leftParen),
          andP(tokenP(Token.ifToken),
          andP(expressionP(),
          andP(expressionP(),
          andP(expressionP(),
               tokenP(Token.rightParen))))))

let whileP = andP(tokenP(Token.leftParen),
             andP(tokenP(Token.whileToken),
             andP(tokenP(expressionP()),
             andP(tokenP(expressionP()),
                  tokenP(Token.rightParen)))))

orP(tokenP(Token.trueToken),
orP(tokenP(Token.falseToken),
orP(ifP,
    whileP)))
```

12.) Consider again the signatures provided in question #11:

```
func tokenP(_ token: Token) -> Parser<Unit>
func andP(_ lhs: @escaping @autoclosure () -> Parser<Unit>,
          _ rhs: @escaping @autoclosure () -> Parser<Unit>)
  -> Parser<Unit>
func orP(_ lhs: @escaping @autoclosure () -> Parser<Unit>,
         _ rhs: @escaping @autoclosure () -> Parser<Unit>)
  -> Parser<Unit>
```

12.a.) What does `@escaping` mean?

The function which has been annotated "escapes" into the returned function. That is, the returned function internally will hold the parameter function. Another way to phrase this is that the returned function "closes over" the annotated function.

12.c.) What does `@autoclosure` mean?

When called, the expression in this position will automatically be wrapped in a function known as a "thunk". For example, if `1 + 2` were the actual parameter to a function with a parameter annotated with `@autoclosure`, the function would get a function which evaluates `1 + 2` as a parameter, as opposed to `3`.

12.b) Why is `@autoclosure` necessary for parser combinators?

We typically define parsers recursively, and this allows us to effectively delay a recursive call until the exact point it is needed. Without this, we'd end up with infinite recursion when constructing parsers.

Prolog

Basic Procedures

13.) Define a procedure named `isFish` which encompasses the following idea: `goldfish`, `bass`, and `carp` are all fish. `isFish` should be defined as a series of facts.

```
isFish(goldfish).
isFish(bass).
isFish(carp).
```

14.) Assume the presence of a procedure named `isInstrument`, which lists various musical instruments. Define a procedure named `musicalFish`, which succeeds if the input is both a fish (according to `isFish`) and an instrument (according to `isInstrument`; `bass` are both).

```
musicalFish(Input) :-
  isFish(Input),
  isInstrument(Input).
```

15.) Consider the following procedure:

```
foo(0).
foo(1) :-
  X = 1.
foo(2) :-
  X = Y,
  X = 1,
  Y = 2.
foo(3).
```

What are the solutions to the following query?

```
?- foo(X).
```

```
X = 0 ;
X = 1 ;
X = 3 .
```