COMP 333 Practice Exam # 2 Solutions

Important: The final is cumulative, but it will biased towards Swift. This practice exam covers only the material since the last exam. You should also study the prior practice exam, in-class worksheets, and assignments.

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 **method** 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. As a hint, you'll need to use `extension`. Example calls are below:

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

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