Quiz Section 4: Functional Programming – Solutions

Task 1 – A Barrel of Halfs

Consider the following function, which calculates half when given an **even** number but also accepts other inputs (though it doesn't perform the same behavior in those cases):

$$\text{half} : (\text{undefined} \cup \mathbb{Z}) \to \mathbb{Z}$$

$$\begin{aligned} \text{half}(\text{undefined}) &:= 0 \\ \text{half}(n) &:= n/2 && \text{if } n \text{ is even} \\ \text{half}(n) &:= -(n+1)/2 && \text{if } n \text{ is odd} \end{aligned}$$

a) What would the declaration of this function look like in TypeScript based on the type?

```
const half = ( n : undefined | bigint): bigint => { .. };
```

b) What would the implementation of the body of this function look like in TypeScript?

```
const half = (n : undefined | bigint): bigint => {
    if (n === undefined) {
        return 0n;
    } else if (n % 2n === 0n) {
        return n / 2n;
    } else {
        return -(n + 1n) / 2n;
    }
};
```

Task 2 – The Rings of Pattern

Consider the following TypeScript code:

```
const maybeDouble = (t: {b: boolean, v: [boolean, bigint]}): bigint => {
    const [bool, num] = t.v;
    if (t.b) {
        if (bool) {
            return 2n * num;
        } else {
            return num;
        }
    } else {
        return 0n;
    }
};
```

How would you translate this into our math notation using pattern matching?

$$\text{maybeDouble} : \{b : \mathbb{B}, v : \mathbb{B} \times \mathbb{Z}\} \to \mathbb{Z}$$

$$\begin{aligned}
\text{maybeDouble}(\{b\colon \mathsf{T}, v\colon (\mathsf{T},\ n)\}) &:= 2n \\
\text{maybeDouble}(\{b\colon \mathsf{T}, v\colon (\mathsf{F},\ n)\}) &:= n \\
\text{maybeDouble}(\{b\colon \mathsf{F}, v\colon (d,\ n)\}) &:= 0
\end{aligned}$$

Task 3 – Sugar and Spice and Everything Twice

We are asked to write a function "twice" that takes a list as an argument and "returns a list of the same length but with every number in the list multiplied by 2".

a) This is an English definition of the problem, so our first step is to formalize it. Let's start by looking at examples. Fill in the blanks showing the result of applying twice to lists of different lengths.

nil _____

3 :: nil _____

2 :: 3 :: nil _____

1 :: 2 :: 3 :: nil _____

These should be nil, 6 :: nil, 4 :: 6 :: nil, 2 :: 4 :: 6 :: nil.

b) Now, let's write a formal definition that gives the correct output for **all** lists.

Write a formal definition of twice using recursion. The recursion can be identified by looking for a *pattern* in the answers to part (a).

$$\text{twice} : \text{List} \to \text{List}$$

$$\begin{aligned} \text{twice}(\text{nil}) &:= \text{nil} \\ \text{twice}(\text{a} :: \text{L}) &:= 2\text{a} :: \text{twice}(\text{L}) \end{aligned}$$

c) What would the implementation of the body of this function look like in TypeScript?

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

d) What is a set of test inputs that would meet all of our requirements?

First, we need to ensure statement coverage. The input [] will execute the first return statement. The input $[3]$ will execute the second return statement.

Next, we must consider branch coverage. The code also has one `if` statement, both of whose branches can be executed. In fact, the two example inputs above already execute both branches, so those two inputs give us branch overage as well.

Finally, since this function is recursive, the "loop coverage" heuristic requires that we pick tests that cause 0, 1, and many recursive calls. The list [] makes 0 recursive calls and the list $[3]$ makes 1 recursive call. If we add the $[1, 2, 3]$, which makes many (3) recursive calls, then we have loop coverage.

Hence, one acceptable set of tests inputs is: [], $[3]$, and $[1, 2, 3]$.

Task 4 – Put Your Mind To Test

a) For the following function, what is a set of test inputs that would meet all of our requirements?

```
const s = (x: bigint, y: bigint): bigint => {
  if (x >= 0n) {
    if (y >= 0n) {
        return x + y;
    } else {
        return x - y;
    }
  } else {
    return y;
  }
}
```

Statement coverage requires at least three cases. The input $(0, 0)$ executes the first return statement, the input $(0, -1)$ executes the second return statement, and the input $(-1, 0)$ executes the third return statement. Thus, the three inputs $(0, 0)$, $(0, -1)$ and $(-1, 0)$ give us statement coverage.

Since there are two `if` statements, we also need to ensure branch coverage. The first and third tests, together, execute both branches of the first `if` statement, and the first and second tests, together, exercise both branches of the second (inner) `if` statement. Thus, these three tests already have branch overage.

Finally, this code has no loops or recursion, so loop coverage is (vacuously) satisfied.

b) The following function allows only non-negative inputs. What is a set of test inputs that would meet all of our requirements?

```
const f = (n: bigint): bigint => {   // Note: requires n >= 0
  if (n === 0n) {
    return 0n;
  } else if (n === 1n) {
    return 1n;
  } else if (n % 2n === 1n) {  // n is > 1 and odd
    return f(n - 2n) + 1n;
  } else {                     // n is > 1 and even
    return f(n - 2n) + 3n;
  }
};
```

The input 0 exercises the first return statement, the input 1 exercises the second return statement, the input 3 exercises the third return statement, and the input 2 exercises the fourth return statement. Thus, the test inputs $0, 1, 2, 3$ give us statement coverage.

The code above has an `if` statement with 4 branches[1]. The four inputs above execute all four branches, so they give us branch coverage as well.

Finally, this function is recursive, so we also need to consider loop coverage. The inputs 0 and 1 make 0 recursive calls and the inputs 2 and 3 make 1 recursive call. If we add the input 4, which makes 2 recursive calls, then we would achieve loop coverage. Thus, the five tests $0, 1, 2, 3, 4$ meet all of our requirements.

Alternatively, we could change one of the inputs to give us loop coverage as well. For example, we could change 2 to 4. The latter executes the same return statement, so we would retain statement and branch coverage, but now we have an input that makes many recursive calls as well. Thus, $0, 1, 3, 4$ also meets our requirements, with only four tests. (That said, This is not a contest to write the fewest tests! Five test inputs is fine.)

c) The following function claims to calculate $|x|$:

```
const abs_value = (x: bigint): bigint => {
  if (x > 1n) {
    return x;
  } else {
    return -x;
  }
};
```

Testing it on the inputs 2 and -2 would meet our requirements, but it would not identify the bug.

Which input do we need to test to see the bug? Which if our non-required (but recommended) heuristics would have found this?

The bug appears only on input 1. The function returns -1 instead of $|1| = 1$.

The boundary testing heuristic would have told us to test 1.

[1] We can also think of this as 3 nested `if` statements. The branch coverage constraint would be the same.

Task 5 – Miami Twice

We are asked to write a function that takes a list as an argument and "returns a list of the same length but with *every other* number in the list, *starting with the first number*, multiplied by 2".

The first number in the list is at index 0, which is even; the second number in the list is at index 1, which is odd; the third number in the list is at index 2, which is even; and so on. Hence, we will call this function twice-evens because it multiples the numbers at even indexes by two and leaves those at odd indexes unchanged.

a) The definition of the problem was in English, so our first step is to formalize it. Let's start by writing this out in more detail. Fill in the blanks showing the result of applying twice-even to lists of different lengths.

nil	_____
4 :: nil	_____
3 :: 4 :: nil	_____
2 :: 3 :: 4 :: nil	_____
1 :: 2 :: 3 :: 4 :: nil	_____

These should be nil, 8 :: nil, 6 :: 4 :: nil, 4 :: 3 :: 8 :: nil, 2 :: 2 :: 6 :: 4 :: nil.

b) Now, let's write a formal definition that gives the correct output for **all** lists.

Write a formal definition of twice-evens using recursion. The recursion can be identified by looking for a *pattern* in the answers to part (a). If the answer for one input does not appear related to and the one immediately before it, it could be related to an even *earlier* answer.

$$\text{twice-evens} : \text{List} \to \text{List}$$

$$\begin{aligned}
\text{twice-evens}(\text{nil}) &:= \text{nil} \\
\text{twice-evens}(x :: \text{nil}) &:= 2x :: \text{nil} \\
\text{twice-evens}(x :: y :: L) &:= 2x :: y :: \text{twice-evens}(L)
\end{aligned}$$

c) What would the implementation of the body of this function look like in TypeScript?

```
const twice_evens = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else if (L.tl.kind === "nil") {
    return cons(2 * L.hd, nil);
  } else {
    return cons(2 * L.hd, cons(L.tl.hd, twice_evens(L.tl.tl)));
  }
};
```

d) What is a set of test inputs that would meet all of our requirements?

The three inputs [], $[1]$, $[1, 2]$ would execute all statements and branches. However, since there is recursion, we must also consider loop coverage. The first two inputs make 0 recursive calls and the last input makes 1 recursive call. Adding the input $[1, 2, 3, 4]$, which makes 2 recursive calls, gives us a total of 4 test inputs and meets our requirements.