Homework 3

The purpose of this assignment is to get practice with structural recursion and induction. Moreover, you will do some experiments with Node.js to understand JavaScript's type conversion mechanism.

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expressing computation currently unfamilar to you.

For submission instructions and the due date, please see the `README.md` file.

Problem 1 Structural Recursion and Induction (24 Points)

Recall the definition of the set N, whose elements represent the natural numbers:

$$\frac{}{\langle\rangle \in N} \qquad\qquad \frac{x \in N}{\langle x\rangle \in N}$$

On N we defined an addition function as follows

$$\begin{aligned} \oplus &: N \times N \to N \\ \langle\rangle \oplus y &= y \\ \langle x\rangle \oplus y &= \langle x \oplus y\rangle \end{aligned}$$

This definition is consistent with the definition of $\oplus$ given in the course notes and the definition of *plus* we discussed in class.

(a) Define a multiplication function $\odot : N \times N \to N$ that behaves like multiplication on natural numbers, i.e., your function should satisfy the following property:

$$\forall x, y \in N : D(x \odot y) = D(x) \cdot D(y).$$

You do **not** need to prove this property. You may use the function $\oplus$ in your definition. **(4 Points)**.

(b) Prove by structural induction $\forall x \in N : \langle x\rangle = x \oplus \langle\langle\rangle\rangle$ **(4 Points)**.

Recall the definition of the set *List* of all lists of integer numbers:

$$\frac{}{\langle\rangle \in \mathit{List}} \qquad\qquad \frac{\mathit{hd} \in \mathbb{Z} \quad \mathit{tl} \in \mathit{List}}{\langle \mathit{hd}, \mathit{tl}\rangle \in \mathit{List}}$$

Further, recall that *nil* stands for the empty list $\langle\rangle$ and that $\mathit{hd} :: \mathit{tl}$ stands for a cons cell $\langle \mathit{hd}, \mathit{tl}\rangle$.

(c) Define a function $\mathit{filter} : \mathbb{Z} \times \mathit{List} \to \mathit{List}$ that takes an integer number $n \in \mathbb{Z}$ and a list $\ell \in \mathit{List}$ and computes the list that is obtained by removing all numbers that are smaller than n from ℓ. For example

$$\mathit{filter}(0, 3 :: 5 :: 2 :: -1 :: 3 :: \mathit{nil}) = 3 :: 5 :: 2 :: 3 :: \mathit{nil}$$

(6 Points)

(d) Write a tail-recursive Scala function

```
def filter(n: Int, l: List): List
```

that implements the function *filter*. **Hint:** The code template contains two predefined tail-recursive functions `reverseLoop` and `reverse`. You may freely use these functions in your implementation of `filter`. **(3 Points)**

(e) Write a tail-recursive Scala function

```
def append(l1: List, l2: List): List
```

that behaves like the non-tail-recursive *append* function defined in class. **Hint:** Again, you may use the provided tail-recursive functions `reverseLoop` and `reverse` in your implementation. **(3 Points)**

(f) Prove by structural induction that for all $\ell \in \mathit{List}$ the following property holds:

$$\mathit{reverse}(\mathit{reverse}(\ell)) = \ell$$

In your proof you may use the following property without proving it: for all $\ell_1, \ell_2 \in \mathit{List}$

$$\mathit{reverse}(\mathit{append}(\ell_1, \ell_2)) = \mathit{append}(\mathit{reverse}(\ell_2), \mathit{reverse}(\ell_1))$$

Use the definitions of *reverse* and *append* provided in the course notes. **(4 Points)**

Problem 2 JakartaScript: Type Conversions (16 Points)

One aspect that makes the JavaScript specification complex is the presence of implicit conversions (e.g., string values may be implicitly converted to numeric values depending on the context in which the values are used). For example, consider the following Javascript expression:

```
3 * "4"
```

which multiplies the numeric value `3` with the string value `"4"`. Evaluating this expression using a JavaScript interpreter such as Node.js yields the numeric value 12. Thus, the string value `"4"` is first converted to the numeric value `4` before the actual multiplication operation is performed.

In this exercise, we will explore some of the complexity of JavaScript's type conversion mechanism by writing a few small JavaScript programs that illustrate how implicit conversion works. This exploration will inform us how to implement the next stage of our interpreter, which will take this additional complexity into account.

(a) The goal of this first part of the exercise is to understand how the implicit conversions between the different primitive types of JavaScript work. For now, we will focus on the primitive types for numeric values, Boolean values, and strings. To experiment with implicit type conversion, we can write simple JavaScript expressions using operators that expect values of specific types and then use these operators with values of other

types. For example, the multiplication operator `*` expects numeric values as arguments. So we can use this operator on string and Boolean values to experiment with the implicit conversion of these values to numeric values. Similarly, we can use the Boolean negation operator `!` to experiment with the implicit conversion of string and numeric values to Boolean values. Write at least 10 simple JavaScript expressions that illustrate the following implicit conversions:

- string values to numeric values
- Boolean values to numeric values
- string values to Boolean values
- numeric values to Boolean values

Evaluate your expressions using Node.js and report both the expressions and the result of the evaluation. Do not forget to consider corner cases where the conversion may not be obvious such as the numeric value `NaN` and the empty string `""`. Use your observations to infer general rules that describe how each of the considered implicit type conversions works.

Hint: avoid the operators `+`, `&&`, `||`, and `==` in your experiments. We will explore some of these operators in more detail below. **(6 Points)**

(b) One of the confusing aspects of implicit type conversion in JavaScript is how it interacts with operator overloading. For example, JavaScript's `+` operator can mean addition of numeric values but also concatenation of strings. Which of the two operations is performed for a specific occurrence of `+` in a JavaScript program depends on the types of the arguments of `+` that are observed at run-time. The same occurrence of `+` in a JavaScript program may sometimes be interpreted as string concatenation and other times as addition. Write at least six JavaScript expressions that involve the `+` operator and evaluate them using Node.js. Use string, numeric, and Boolean values as arguments and also mix the different argument types. Report the expressions and the results of their evaluation. Use your observations to infer a general rule that describes how `+` is interpreted in JavaScript. **(5 Points)**

(c) The semantics of the Boolean operators `&&` and `||` in JavaScript is complicated by so-called *short-circuit evaluation*. In particular, the type of the result value of these two operators depends on the type of their second argument. Write at least six simple JavaScript expressions where the operators `&&` and `||` are applied to a mix of string, numerical, and Boolean values. Evaluate your expressions using Node.js and report your results. Again, use your observations to infer general rules that describe how each of the two operators works. **Hint:** For `&&` and `||` type conversion is only performed on the first argument, but not on the second argument. **(5 Points)**