Summary of “Why Functional Programming Matters” by Kushaal Manchella

This paper aims to show programmers how valuable functional programming can be by outlining its advantages and explaining how to leverage them. Functional programming is most known for several key characteristics such as having no assignment statements, no side effects, irrelevant order of execution, and referential transparency. However, these points highlight what functional programming lacks rather than what it offers. With examples and real-world analogies, this paper focuses on the strengths of functional programming that we can rely on to build simple and effective code. Specifically, we are shown how to break problems into modules and glue them together with higher-order functions and lazy evaluation.

By drawing an analogy between structured programming and functional programming, we can better understand how key modularity is in large-scale programming. Structured programming is known for having no goto statements and single-entry, single-exit blocks, making programs more mathematically tractable. However, the key advantage of structured programming is its modular design, which boosts productivity by allowing quick and easy coding of small modules, reusability of general-purpose modules, and independent testing of program parts. A crucial part of modular programming involves dividing a problem into sub-problems and gluing them together. Carpentry is a great example of how modularity helps solve problems in the real world. For example, Building a chair is much easier by first building the individual legs, seat, and back and then gluing them together. The glue is also a vital piece of the process without which there is no complete solution.

One such glue used by functional programming is higher-order functions. Functional programming allows simple functions to be combined into more complex ones using higher-order functions. This can be illustrated with a simple list-processing problem using Haskell. Below we will be creating a new list datatype and evaluating the sum of a given list of integers. We can start by defining ListOf:

```
data ListOf x = Nil
              | Cons x (ListOf x)
              deriving (Show, Eq)
```

We further define a reduce function to be used in summing this new ListOf type:

```
reduce:: (t1 -> t2 -> t2) -> t2 -> ListOf t1 -> t2
reduce _ acc Nil = acc
reduce f acc (Cons x xs) = f x (reduce f acc xs)
```

This reduce function allows us to recursively traverse our new list and apply our function f on it. It modularizes the computation of the sum function f on a list by glueing together a

general recursive pattern using the two pattern-matching cases above. Finally we can define a sumList that performs the summing of a list based on the reduce:

```
sumList :: ListOf Int -> Int
sumList = reduce (+) 0
```

By modularizing the sumList function, we can benefit from re-using the reduce to solve other problems such as multiplication of all elements in a list. We used a simple function (sumList) as a combination of a "higher order function" and some simple arguments to arrive at a part (reduce) that can be used to write down many more functions on our list data type with little programming effort. We can extend this strategy to other new datatypes we create such as trees; higher-order functions can be written for new data types for easy data manipulation.

Functional languages offer another type of powerful glue called "lazy evaluation" which enables programs to be composed together efficiently. When composing two programs f and g where the output of (f input) is used as an input for g, functional programming avoids impractically storing the intermediate output of (f input) in a temporary storage. Instead, f runs only as needed by g and can be suspended and resumed as required. If g finishes consuming all of f's output, f is aborted. This type of evaluation allows programs to additionally handle non-terminating programs. Lazy evaluation allows programs to be modularized into a generator and selector format, making it a very powerful tool to work with.

Lazy evaluation and its advantages can be illustrated with the example of Newton-Raphson square roots. This algorithm computes the square root of a number N by starting from an initial approximation a0 and computing increasingly better ones. Each approximation is derived from the previous one by using the function:

```
next :: Double -> Double -> Double
next n x = (x + n / x) / 2
```

next n allows us to map one approximation onto the next. To compute a sequence of approximations, we can define a function myRepeat that can be computed as shown below:

```
myRepeat :: (a -> a) -> a -> [a]
myRepeat f a = a : myRepeat f (f a)
```

This is an example of a function that utilizes lazy evaluation where a potentially infinite output is possible. However, no more approximations will be computed than the rest of the program requires. We will also need another function named relative which checks the relative difference between successive approximations:

```
relative :: Double -> [Double] -> Double
relative eps (a:b:rest)
  | abs (a - b) <= eps * abs b = b
```

```
  | otherwise                  = relative eps (b : rest)
relative _ _ = error "relative called with a list that is too short"
```

Finally we have a function which takes an initial approximation a0, your final relative difference and a number which we want to find a square root for:

```
relativeSqrt :: Double -> Double -> Double -> Double
relativeSqrt a0 eps n = relative eps (myRepeat (next n) a0)
```

Using this approach, we can use lazy evaluation to approximate to the extent we want. Furthermore, we can modularize by reusing our relative functions to perform computations that generate a sequence of approximations.

We can take higher-order functions and lazy evaluation to solve all sorts of interesting problems in a modular and efficient manner. For example, we can implement the alpha-beta heuristic algorithm in a game of Tic-Tac-Toe to estimate how good of a position a game player is in by looking ahead to see how the game might develop while avoiding unprofitable lines. Since we have to generate a game tree that represents all possible moves in a game, we could end up with an infinite number of options. Lazy evaluation ensures that we construct only the parts of the tree that we actually need for evaluation which saves time and memory. On the other hand, higher-order functions allow us to write general-purpose functions that can be reused across different parts of the algorithm. Additionally, recursive functions that build and evaluate the game tree can be expressed more concisely using higher-order functions.

In summary, modularity is essential for successful programming. Languages aiming to improve productivity must support modular programming beyond just new scope rules and separate compilation mechanisms. Effective modularity involves the ability to decompose problems into parts and glue solutions together seamlessly. Functional programming languages excel in this area by providing two powerful tools: higher-order functions and lazy evaluation. These tools enable programs to be modularized in innovative ways, resulting in smaller, more general, and reusable modules. Consequently, functional programs are typically smaller and easier to write than conventional ones. The authors encourage functional programmers to focus on modularizing and generalizing complex parts of their programs using higher-order functions and lazy evaluation.