CSC 330 | Team 1

TEST 2
FUNCTIONAL PROGRAMMING WITH GO

Introduction

This report deals with the programming language Go. In particular, how it supports functional programming concepts, and its relation to lambda calculus are discussed. A mathematical overview of currying and partial application are explained along with examples.

The dataflow programming paradigm is detailed and related programming techniques using the GoFlow library are referenced.

Go’s support for higher-order functions and closure is elucidated. Examples are included of first class functions and closure through the use of anonymous functions.

Go has no map, reduce, or filter functions built in. This report seeks to clarify the reasoning behind this, showing "the Go way to do it" along the way. A method of emulating list comprehensions is also provided, along with a discussion of why list comprehensions are absent from the language.

Table of Contents

Question 1

1.1 Intro to Currying and Partial Application

Currying is the process of changing the arity of a function. In other words, transforming an n-ary function into a chain of unary functions that each take just one argument [1.3],[1.4]. The example below shows a mathematical function with curring applied to it.

```
f: X × Y → R can be curried into f': X → (Y → R)
```

While currying transforms the number of arguments a function takes, partial application refers to the process of taking a function and fixing a certain number of its arguments, resulting in a function with identical function yet smaller arity [1.1]. Using the same function example as above, partial application be also be expressed mathematically:

```
f: X × Y → R can be turned into f': Y → R
```

In this example, the new function f ' performs the same operations as f, however X becomes a fixed parameter, thus decreasing the arity of the function by one.

1.2 Support of Currying and Partial Application in Go

Go was designed primarily with concurrency in mind, and consequently left out many key aspects of functional programming, including native support for partial function application and function currying. This means that while it is possible to explicitly write a function to simulate currying functions, Go does not naturally curry its functions as Haskell does. The same applies to partial application of functions: partial applications are not supported by Go unless explicitly written.

In order to simulate curried functions, a function must be written such that it creates another function which is in itself curried when called.

```
func mkAdd(a int) func(...int) int {
```

This function `mkAdd` takes in a single integer parameter and returns a function that will accept an arbitrary list of integers as its parameters. The returned function is then curried when called in the second line shown below.

```
return func(b ...int) int {
```

Inside of this return call, a simple `for` loop adds each parameter `b`, to the initial parameter a.

```
for _, i := range b {

      a += i

}

return a
```

This can all be demonstrated by initializing some example functions, `add2` and `add3`, within the main function. Each function can take a list of parameters, all integers, and will be able to either add 2 or add 3 to each parameter in the list, respectively.

```
add2 := mkAdd(2)
add3 := mkAdd(3)
fmt.Println(add2(5, 3), add3(6))
```

The previous examples show how a currying function can be explicitly written, and the same could be done for partial applications functions. Even though currying or partial applications are not built into Go they can still be used explicitly, allowing the use of either of these two methods if they are needed.

1.3 Currying and Partial Application in Haskell vs. Go

Since Haskell is a functional language, it has native support for currying and partial application for all of its functions. In other words, by default Haskell functions are defined in their curried form [1.2]. Since Go does not support this, it is much less clear when attempting to write a curried function than it is in Haskell and must be explicitly written as its own function. For example, a simple function to add 2 to another integer can be written in Haskell as:

```
(+ 2) :: Int -> Int
(+ 2) a
```

In this case, the parameter a would be another integer and the result would be `a + 2`. To contrast this clarity, the same curried function in Go is notably more verbose:

```
func mkAdd(a int) func(int) int {
	return func(b int) int {

		return a + b

	}

}
```

This function performs the same operation as the first line of the Haskell. To call this function and get the same output as in Haskell, it would be written as follows in the main function:

```
	add2 := mkAdd(2)
	fmt.Println(add2(3))
```

Since Haskell's functions are curried by default, they require far less syntax and understanding of the actual process to write than the equivalent process in Go. Haskell allows the programmer to worry less about function returning, scope, and declaration of variables, which are all key components of any Go program.

While Go does not natively support function currying or partial applications it is still possible to recreate the same functionality with a little bit of work. Unlike Haskell, which provides a clear and concise way to have every function curried naturally, Go still allows the user to perform the operations if necessary.

Question 2

2.1 Intro to Dataflow Programming

Dataflow programming is a programming paradigm that models a program as directed graph that is similar to dataflow diagram. Operations are represented by a set of nodes (also called blocks) that has input and output ports. The set of nodes can be either sources or sinks, and they can also process information. The nodes are connected by directed edges that define the flow of the program. Computations are triggered when the input values are processed and passed to the dependent computation nodes. Dataflow programming focuses on the state of the program, rather than executing a specific order of operations. It has been used in a wide range of contexts. For example, it supports massive data computation and is sometimes used as the basis for visual languages providing end-user programming capabilities [2.1].

2.2 Using Infinite Lists to Implement Dataflow Programming

Dataflow programming is possible in many functional programming languages (e.g. Haskell) thanks to lazy evaluation. Lazy evaluation means that the evaluation of an expression is deferred until the results are needed. [2.2] This allows a program to ignore certain values and to produce an output that is relevant to the problem being expressed. For example, if a program were written to calculate and print all of the values of the Fibonacci sequence, a subset of the sequence would be printed, with the next subset available if the programmer has need for it. In the case of languages using eager evaluation, this type of problem would cause an error since any compiler handling the expression would attempt to evaluate the entire series, using up any available memory in the process before achieving the desired results.

While the Go language does not use lazy evaluation, the use of channels allows for some of the key concepts in dataflow programming to be implemented. Dataflow is a software model based on the idea of “disconnecting computational actors into stages (pipelines) that can execute concurrently.” [2.3] In Go, these pipelines are represented by channels that connect concurrent goroutines. For example, Go can produce the same Fibonacci sequence as the Haskell example referenced above through the use of a generator, which is a function used to generate the next value of a sequence. In the case of the Fibonacci sequence, the

Fibonacci function would be the generator responsible for deriving the next Fibonacci value. That value would be sent into a channel causing a stream of output values to be created, acting like a dataflow program [2.4].

2.3 Dataflow and Flow-based Programming Libraries

GoFlow is a dataflow and flow-based programming library for Go. It is a minimalistic implementation of flow-based programming that designs applications as graphs of components which react to data flowing through. The main properties of the proposed model are:

- Concurrent - graph nodes run in parallel.
- Structural - applications are described as components, their ports, and connections between them.
- Reactive/active - system behavior is how components react to events or how their lifecycle is handled.
- Synchronous/asynchronous - there is no determined order in which events happen, unless such an order is demanded.
- Isolated - sharing is done by message-passing, not shared memory [2.2].

There are several key concepts for GoFlow. The in ports and out ports are channels that pass data to/from components. There are no limitations for data types and quantities. Multiple in ports are used for information packets arriving at various times from different sources. However, a tuple or structure should be used instead if there are multiple elements being passed concurrently.

Components' behaviour is controlled by events. There are two supported event types: incoming packets received - when a new information packet (data element) is received on an input port - and connection closed - when the channel associated with in port has been closed by sender or network). Components handle events on their ports using methods. Events and handlers are asynchronous, however, handlers try to handle events concurrently [2.5].

Question 3

3.1 Higher Order Functions in Go

Although Go is not traditionally considered a functional programming language, it supports first class and higher order functions along with closure. Thus it supports a functional programming style [3.1].

Higher order functions are functions that either take one or more functions as arguments, and/or return a function [3.2].

In Go, functions and methods are first-class [3.3]. This means that the language supports the creation of new functions during execution of the program, storing those functions in data structures, passing them as arguments to other functions, and returning them as values from other functions [3.4]. These functions can be stored as a variable inside an object or an array as long as it is possible to pass them as arguments to other functions and return them as return values from other functions.

Below is example code of a higher order function in Go [3.1].

```
func Twice (f func(int) int, v int) int {

     return f(f(v))

}

func main() {

     f := func(v int) int {

           return v + 3

     }

     Twice(f, 7) // returns 13

}
```

The above function Twice takes as input a function f, (of type `int` that returns an `int`) and another `int v`, and returns an `int`. The main method has a function called f that takes as input an `int v`, and returns an `int`. This function f takes the input integer and returns the integer obtained after adding 3 to it.

Putting everything together, `Twice(f, 7)` takes the value 7, applies it to the function f twice, and then returns the output integer. Essentially, `Twice(f, 7)` performs the following calculation:

```
Twice(f, 7) = f(f(7)) = f(7+3) = f(10) = 10+3 = 13.
```

Twice is a higher order function because the function Twice takes a function f as a parameter and also returns a function.

3.2 Closure in Go

Within the context of programming languages, closure is a technique for implementing lexically scoped bindings and supporting first class functions [3.5]. In order to understand closure it is important to know about first-class functions which were defined above.

In essence, closure is a function which can passed around while maintaining the environment from which it was created . [3.6]. A closure function has access to these variables even outside of the first-class function's scope.

Go, in fact, does support closure through the use of anonymous functions (functions that are not bound by an identifier) [3.7]. For instance:

```
func intSeq() func() int{
	i := 0
	return func() int{
		i += 1
		Return i
	}
}
```

Here the function `intSeq` returns an anonymous function defined within the body of `intSeq`. The returned function "closes" over all the variables around it (in this case, the variable `i`). As a result, `i` is remembered by the closure and preserved alongside `func`.

After calling `intSeq` and assigning the result to `nextInt`, this function value holds its own value for `i`, which is updated and incremented each time `nextInt` is called as demonstrated below [3.7].

```
func main() {

    nextInt := intSeq()
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    // To confirm that the state is unique to that
    // particular function, create and test a new one.
    newInts := intSeq()
    fmt.Println(newInts())

}
```

After calling nextInt three times, it produces the result :

```
1
2
3
```

And to confirm that the state of the function is unique, `intSeq` is called and assigned to `newInts`, giving a new value:

```
1
```

3.3 Comparison between Go and Haskell

Haskell is a purely functional programming language while Go just has some aspects of functional programming such as its support for higher order functions and closure. Haskell being a functional programming language does support both higher order functions and closure [3.8].

In Haskell, functions can take functions as parameters and also return functions. Pure functions in Haskell can only take one argument however this restriction can be overcome by currying which is the process of manipulating a function that takes multiple arguments into a function that takes a single argument and may return another function if arguments are still needed [3.9].

Haskell also supports standard higher order functions in its library such as map, filter, and reduce, while Go does not (implementing these functions in Go is further discussed in Question 4).

Example: The twice function in Haskell:

```
twice :: (a -> a) -> (a -> a)
twice f = f . f

f :: Num a => a -> a
f = subtract 3

main :: IO ()
main = print (twice f 7) -- 1
```

Example: The twice function in Go:

```
func twice(f func(int) int, v int) int {
      return f(f(v))
}

func main() {
      f := func(v int) int {
            return v + 3
      }
      twice(f, 7) // returns 13
}
```

Closure is demonstrated in one of the fundamental features of Haskell: Lambdas, which are anonymous functions usually passed to higher order functions [3.10]. In Haskell, closure specifically makes use of free variables that were defined within their anonymous function, and only closes around these variables [3.11]. Functions with these free variables are considered as closures within Haskell. Because free variables appear so frequently in Haskell's functions, closure plays a key role within the language [3.12]. On the other hand, in Go, all variables are considered closed given that they are within the anonymous function's outer scope [3.13].

Question 4

4.1 Map, Filter, and Reduce

The Go language is not functional and thus does not natively include features such as Map, Filter, and Reduce. The static typing system in place is the largest barrier to functional a approach in Go. Languages like Haskell and JavaScript benefit from type inference systems that allow for changing the type of a variable on the fly. Thus functions such as Map, Filter and Reduce are not restricted to returning a single specific type. Go is limited in this regard. While its type system does not make it impossible to implement these functions, it does greatly increase the quantity of code required to implement such constructs.

Go is a strongly and statically typed language. Traditional map/reduce/filter functions must be polymorphic, either through generics or by "true" polymorphism. Since Go supports neither of these things [4.1], implementing these functions for all types requires some work:

- Some programmers write these functions using Go's "`interface{}`" type[4.2], [4.3], [4.5], the equivalent of using "`void *`" in C, or "`Object`" in Java.
- If the functions will only ever be used by one or two types, writing special, explicitly-typed versions just for those types is easy.
- Rather than defining separate functions for map, reduce, and filter, the idiomatic thing to do in Go is to simply write a loop that achieves the same goal [4.6].

The implementation of these functions for a given type is generally not difficult. The true problem surfaces once one wishes to create a map/filter/reduce function that can accept any list of an arbitrary type and also return any arbitrary typed list.

Below are examples of map, reduce, and filter functions, written for imaginary types T and S.

```
// Assume T is the input type and S is the output type:
func mp(l []T, f func(T) S) []S {
	out := make([]S, len(l))
	for i, e := range l {
		out[i] = f(e)
	}
	return out
}

func filter(l []T, f func(T) bool) []T {
	var out []T
	for _, e := range l {
		if f(e) {
			out = append(out, e)
		}
	}
	return out
}

func reduce(l []T, unit T, f func(T, T) T) T {
	out := unit
	for _, e := range l {
		out = f(out, e)
	}
	return out
}

// Or, without an initial "unit" argument:
func reduce2(l []T, f func(T, T) T) T {
	out := l[0]
	for _, e := range l[1:] {
		out = f(out, e)
	}
	return out
}
```

4.2 List Comprehensions

Go does not have native list comprehensions like Haskell or Python. It aims to be a "systems language" rivaling C++ for speed and efficiency[4.8], so list comprehensions contradict Go’s goals. They abstract away the memory model, and permit the programmer to ignore the data structures which underlie slices (Go's version of lists). Though Go is garbage-collected, it still aims to make it easy for a programmer to be aware of the memory costs of their program. As a result, many polymorphic and functional programming features are not included due to the taxes they incur on overall performance for system level programs.

Furthermore, Go aims to compile exceedingly fast [4.8], so many language features which might output efficient code but require heavy analysis by the compiler are not available in Go. List comprehensions fit this description; even if the compiler could probably output the same code from the two following snippets of pseudocode, it may take much longer to compile the list comprehension:

```
newL = [f(x) in oldL | x < 5]
```

vs.

```
newL = list of size len(oldL)
for x in oldL:
    if x < 5:
        append f(x) to newL
```

Note how in the second snippet, we define a specific capacity for our list "newL". This matches how Go works: slices are of fixed capacity, and a new slice must be created if we need to add more elements than will fit [4.7]. The compiler *could* choose a size for all new slices created using list comprehensions, but this choice may not be trivial for a sufficiently complex program.

Conclusion

Go is a lightweight, general purpose programming language. Though not a functional programming language, Go code can still be written with a largely functional style: Go supports first-class functions, and its for loop naturally iterates over arrays and slices; it supports anonymous functions with statically-bound variables (i.e. closures); and its channels are naturally suited to dataflow programming.

Go is not a functional programming language in the traditional sense, but functional programmers may still find themselves feeling at home working in the Go idiom.

References

[1.1] "Currying versus partial application (with JavaScript code)", 2ality.com, 2016. [Online]. Available: http://www.2ality.com/2011/09/currying-vs-part-eval.html. [Accessed: 08- Nov- 2016].

[1.2] "Partial application", En.wikipedia.org, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Partial_application. [Accessed: 08- Nov- 2016].

[1.3] "What's the difference between Currying and Partial Application?", Raganwald.com, 2016. [Online]. Available: http://raganwald.com/2013/03/07/currying-and-partial-application.html. [Accessed: 08- Nov- 2016].

[1.4] "Currying", En.wikipedia.org, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Currying. [Accessed: 08- Nov- 2016].

[2.1] T. B. Sausa, "Dataflow Programming Concept, Languages and Applications," [Online]. Available: https://paginas.fe.up.pt/~prodei/dsie12/papers/paper_17.pdf. [Accessed: 8-Nov-2016]

[2.2] "Lazy evaluation," 3 September 2015. [Online]. Available: https://wiki.haskell.org/Lazy_evaluation. [Accessed 8 November 2016].

[2.3] "Dataflow," 10 September 2016. [Online]. Available: https://en.wikipedia.org/wiki/Dataflow. [Accessed 8 November 2016].

[2.4] "On using Go channels like Python generators," 5 January 2014. [Online]. Available: https://blog.carlmjohnson.net/post/on-using-go-channels-like-python-generators/. [Accessed 8 November 2016].

[2.5] l. m. d. k. trustmaster, "goflow," [Online]. Available: https://github.com/trustmaster/goflow. [Accessed 8-Nov-2016].

[3.1] "Higher-order function". [Online]. Available: https://en.wikipedia.org/wiki/Higher-order_function. [Accessed: 8-Nov-2016]

[3.2] "First-class functions". [Online]. Available: https://rosettacode.org/wiki/First-class_functions. [Accessed: 8-Nov-2016]

[3.3] “First Class Functions in Go". (2011, June 30). [Online]. Available: https://blog.golang.org/first-class-functions-in-go-and-new-go. [Accessed: 8-Nov-2016]

[3.4] “First-class function” [Online]. Available: https://en.wikipedia.org/wiki/First-class_function. [Accessed: 8-Nov-2016]

[3.5] “Closure”. [Online]. Available: https://en.wikipedia.org/wiki/Closure_(computer_programming). [Accessed: 8-Nov-2016]

[3.6] “Closures in golang”. [Online]. Available: http://keshavabharadwaj.com/2016/03/31/closure_golang/. [Accessed: 8-Nov-2016]

[3.7] “Anonymous function”. [Online]. Available:

https://en.wikipedia.org/wiki/Anonymous_function#Go. [Accessed: 8-Nov-2016]

[3.8] “Higher Order Functions”. [Online]. Available: http://learnyouahaskell.com/higher-order-functions. [Accessed: 8-Nov-2016]

[3.9] “Currying”. [Online]. Available: https://wiki.haskell.org/Currying. [Accessed: 8-Nov-2016]

[3.10] “Higher Order Functions”. [Online]. Available: http://learnyouahaskell.com/higher-order-functions. [Accessed: 8-Nov-2016]

[3.11] “Closure”. [Online]. Available: https://wiki.haskell.org/Closure. [Accessed: 8-Nov-2016]

[3.12] “Closures (in Haskell)”. [Online]. Available: http://stackoverflow.com/questions/9088295/closures-in-haskell. [Accessed: 8-Nov-2016]

[3.13] “A little confused about closures”. [Online]. Available: https://www.reddit.com/r/golang/comments/3asu5u/a_little_confused_about_closures/. [Accessed: 8-Nov-2016]

[4.1] "Why does Go not have generic types?". [Online]. Available: https://golang.org/doc/faq#generics. [Accessed: 8-Nov-2016]

[4.2] CL. Liu, "Glow: Map Reduce for Golang", 2015. [Online]. Available: https://blog.gopheracademy.com/advent-2015/glow-map-reduce-for-golang/. [Accessed: 8-Nov-2016]

[4.3] MC. Castilho, "Cheap MapReduce in Go", 2015. [Online]. Available: http://marcio.io/2015/07/cheap-mapreduce-in-go/. [Accessed: 8-Nov-2016]

[4.4] "Why doesn't type T satisfy the Equal interface?". [Online]. Available: https://golang.org/doc/faq#t_and_equal_interface. [Accessed: 8-Nov-2016]

[4.5] R. Pike, "Simple apply/filter/reduce package", 2015. [Online]. Available: https://github.com/robpike/filter. [Accessed: 8-Nov-2016]

[4.6] R. Pike, "README for Simple apply/filter/reduce package", 2015. [Online]. Available: https://github.com/robpike/filter/blob/master/README.md. [Accessed: 8-Nov-2016]

[4.7] "Go Slices: usage and internals", 2011. [Online]. Available: https://blog.golang.org/go-slices-usage-and-internals. [Accessed: 8-Nov-2016]

[4.8] "What is the purpose of the project?". [Online]. Available: https://golang.org/doc/faq#What_is_the_purpose_of_the_project. [Accessed: 8-Nov-2016]