CSC/MAT-220: Discrete Structures
Final Summary

Thomas R. Cameron

December 6, 2017

Abstract

This summary is intended to help guide you in a review process that is both edifying and promotes success on the final. We outline a list of topics along with corresponding books sections and assignments. This review is by no means comprehensive and you should consider it your responsibility to map out how all the topics in this course are interconnected. As one last piece of advice, be sure that you know definitions and major theorems.

i. Recurrence Relations.
   - Section 23
   - EFYs 8–9
   - Homework 5
   - Lab 5 and 7 (cost analysis)

ii. Functions
   - Sections 24–25
   - EFYs 10 and 12
   - Homework 5
   - Every Lab (functional expressions)

iii. Cardinality and Infinity
   - Sections 10 (cardinality of finite sets) and 25 (cardinality of infinite sets)
   - EFY 12 and story of Cantor from Week 11
   - Homework 5
   - Infinite Sentinel from Week 11.

iv. Permutations
   - Sections 27–28
   - Dihedral Group from Week 12
   - Homework 6

v. Probability
   - Sections 30–34
   - EFY 9 and 13
   - Homework 6
   - Lab 7 and Binary Search from Week 13

vi. Graphs
   - Section 47–50
   - EFYs 14–15
   - Lab 8

vii. Trees
   - Section 50
   - Lab 8
   - SML Trees from Week 15

Functional Programming and SML Summary. In essence, functional programming is a programming paradigm in which the elements of a program is broken into the evaluation of mathematical functions that avoids changing state and mutable data. In contrast, the object-oriented programming paradigm is based on the concept of objects and having these objects interact with one another. In Java, objects are instances of classes. Python is a multi-paradigm programming language that supports both object oriented programming, along with imperative and procedural programming. The goal of this section is to see how the structure of SML forces us to work within the functional programming paradigm, and to understand the benefits of doing so.

From the beginning we have seen that SML is a statically typed language; that is, all bindings that occur are static in nature. The expression *val x = 2* followed by *x=3* evaluates to false, it does not update the value of *x* which is what would happen in an imperative language such as Python. The important takeaway is that SML is avoiding changing state; therefore, limiting the number of times memory is overwritten during the life of the program.

Another interesting feature of SML is that functions are datatypes of the form $typ \ \rightarrow \ typ'$, where the *typ* is the domain type and typ' is the co-domain type. Functions in SML are *first-class* datatypes; meaning, they may be computed as the value of an expression, bound to a variable, and passed as an argument of a function. For instance, the following function *comp* takes as an argument a tuple of two functions and returns the composition of these two functions.

$$\begin{aligned} &val\ comp\ :\ ('b\ \rightarrow\ 'c) * ('a\ \rightarrow\ 'b)\ \rightarrow\ ('a\ \rightarrow\ 'c)\ = \\ &\quad fn\ (f,g)\ \ =>\ \ fn\ x\ \ =>\ \ f(g\ x) \end{aligned}$$

For example, the expression *comp(Real.fromString,Int.toString)* evaluates to a function of type $int \ \rightarrow \ real\ option$. This function takes on an integer, converts it to a string, and then converts the string to a real option.

It is essential to note that the above expression is a binding of the functional expression $fn\ (f,g)\ \ =>\ \ fn\ x\ \ =>\ \ f(g\ x)$ to the variable *comp*. As soon as this binding is created, the state of SML associates the pattern *comp(f,g)* with the expression $fn\ (f,g)\ \ =>\ \ fn\ x\ \ =>\ \ f(g\ x)$. Note that this is very different from a function in C, where variables are passed by reference or value, local copies are made, memory is modified, and a result is returned and stored in memory. The “call” *comp(f,g)* simply evaluates the associated expression. Nothing more, nothing less.

Because SML makes functions first class datatypes, the modularization of a program is natural. This allows us to break up a large task into smaller independent and interchangeable functions. Modularization is very important for large scale projects and can lead to task parallelization. For instance, suppose that we have a function that depends on 4 independent and interchangeable helper functions. Then, we could create 4 threads and run these helper functions in parallel.

I encourage you to read through all of our old labs, just for the edification of watching for how SML enforces the paradigm of functional programming and to consider the potential benefits this has.