10.Classical programming

The classical programming paradigm involves solving problems via prescriptive algorithms and utilising a well-defined sequence of instructions. Inputs are defined and constrained, developing algorithms to process inputs via defined programming paradigms (Procedural, Object-Oriented, Functional and Logical) to produce outputs.

Discovering algorithms to solve temporally and spatially efficient problems is notoriously challenging. In computational complexity theory, problems are either P (Polynomial), NP (Non-deterministic Polynomial), NP-Complete or NP-Hard. Polynomial problems can be solved and verified quickly, while non-polynomial problems cannot. NP problems are difficult to solve via classical algorithmic approaches.

An example of a P problem is finding the shortest path between two points, while an example of an NP Problem is the travelling salesman problem where when given a series of locations, what is the optimal path to visit all sites travelling the shortest possible distance. It just so happens that many of the problems that are currently required to be solved computationally fall into the NP category. In most cases, for practical purposes, an approximate solution is often sufficient. When approaching daily life, humans always come up with approximate solutions, i.e. navigation.

There are implementation challenges in solving problems under a classical algorithmic paradigm in an efficient manner. Algorithmic complexity is a branch of computer science that deals with how efficient algorithms run in the temporal (time complexity) and spatial domains (space complexity).

While there are many different ways to implement algorithms, an obvious but potentially non-efficient way to implement an algorithm is often referred to as a “naive” algorithm. Most efficient algorithms are not easy to discover, and Computer Scientists have been working for many years to find the best algorithms to solve classes of problems. Algorithms in production systems are implemented in temporally and spatially efficient ways. The largest software companies ensure the highest algorithmic standards via peer-programming, similar to peer-review. It takes many years of training and a high degree of skill to implement algorithms efficiently.

In a practical sense, while programming techniques have evolved and programming languages (i.e. C, C++, Java, JavaScript, PHP, Python) have varied in popularity, the fundamentals have remained the same since Bell Labs, in the 1970s, wrote the first C modules. There are some practical ways to make theoretically challenging problems work in production, such as using distributed systems. Advanced approaches come at a higher cost and are outside the capabilities of all but the most technologically progressive organisations.

Given the complexity of classical algorithmic programming approaches, organisations have long since found it difficult to build effective teams to produce such systems internally. Further, many have also struggled to find external providers to solve niche challenges that are theoretically possible to implement. Designing a specific algorithm becomes significantly more complex as the problem complexity increases. It is especially true for multivariate problems. It is best to use classical algorithm approaches for situations where an algorithm can quickly solve a problem and works for most cases.

The first article in this series defined the term classical programming languages. The longevity of three programming languages, Fortran, COBOL and Pascal (more than five decades) is the reason for calling them classical programming languages. COBOL is an acronym for Common Business-Oriented Language, which is almost exclusively used for programming business applications. Let's find out why it is relevant even today.

COBOL is still relevant for a number of reasons. First, it is a simple language and easy to learn. If you need to write a program to control the launch of a rocket into space, you will definitely be evaluating integrals, Fourier transforms and a whole lot of other difficult mathematical functions. COBOL is not suitable for such applications. But if your sole aim is to perform a simple task like payroll management, for which all that is needed are basic arithmetic operations, then the use of COBOL is justified. You don't need a cannon to kill a fly. But if merely being simple or easy to learn was the sole advantage of COBOL, then it would have definitely died out a long time ago.

The second advantage of COBOL is its legacy. During the sixties, when trained programmers were very rare, a simple language like COBOL was in high demand because of its shallow learning curve. At a time when computers were

a scarce resource, it was economical to train people in a relatively easy-to-learn language like COBOL. The relatively large number of COBOL programmers naturally led to a lot of COBOL code and applications being developed. Thus, even today, a large number of well-written COBOL code is in existence. Financial institutions all over the world still have huge amounts of COBOL code being executed from powerful mainframes on a regular basis. The existence of such a large amount of legacy code makes COBOL still relevant.

Notice that nobody repairs a smoothly running car. Similarly, nobody is going to port the still smoothly running COBOL code into some other programming language, even if it is the latest in the programming world. I feel some of the claims by members of the COBOL programmer community are a bit exaggerated. My research for this article made me come across statements like 'more than 200 billion lines of COBOL code are still active', '80 per cent of all the code written is in COBOL', etc. I do not want to refute these claims as I don't know where to look for evidence to prove or disprove these statements. But I have heard similar remarks about C, C++, Java, Python, etc, and the math doesn't add up. Whatever be the case, there definitely is a really large COBOL code base still doing service to mankind.

Data intensive operations are often carried out by mainframe computers. Thus, it is highly likely that you will see some COBOL code if you are working with mainframes.

I could also argue that there is a third point in favour of the relevance of COBOL. With even 5-year old children learning Python, what do you think is the average age of a Python programmer? I believe it to be in the early twenties. (No sources to substantiate this claim, though.) What about the average age of a COBOL programmer? It could be well over forty, I believe. Again, it is just a guess. So here comes the third advantage of having familiarity with COBOL. One day you might be a member of a very small elite group that can maintain the large amount of COBOL legacy code. Like a traditional art form like Kathakali, which still attracts admirers and performers, COBOL will also attract young professionals to learn and code, time and again.

The history

Now let us learn some history. COBOL was designed in 1959 and its development was guided by a consortium called the Conference/Committee on Data Systems Languages (CODASYL) created for the development of a standard programming language for business processing. No single individual

can be given the entire credit for the development of COBOL. But the name of one person must be mentioned, Grace Hopper. She was a pioneer in programming and one of the greatest programming language designers. As a side note, the illustrious career of Grace Hopper is inspiring and worth a few Internet searches to obtain more information. A programming language called FLOWMATIC developed by her has influenced the development of COBOL a lot.

Like all other programming languages, COBOL has also gone through a lot of revisions and upgrades over the years. Some of the standards of COBOL that made an impact include COBOL-60, COBOL-61, COBOL-65, COBOL-68, COBOL-74, COBOL-85, etc, with each introducing much awaited features. Finally, the long awaited object-oriented dialect of COBOL came in 2002, thus making COBOL a truly modern language. The latest standard of COBOL is COBOL 2014. Though COBOL is modernising almost as fast as any young programming language, it is still lagging behind them. A lot of features are missing but none of them are critical to the domain in which COBOL applications are deployed. Though of the least concern, it was disheartening to know that COBOL does not have a mascot or a logo. May be that is something the COBOL programming community should come up with soon, and the earlier the better.

Running COBOL programs in Linux

The next important question is: how to run COBOL programs in Linux? An open source COBOL compiler called GnuCOBOL helps us in this aspect. GnuCOBOL was earlier known as OpenCOBOL. GnuCOBOL can be installed in Ubuntu with the command *sudo apt install open-cobol.* The installation is simple in other Linux distributions also.

Now let us see a simple COBOL program so that we can test our compiler. Consider the program cob1.cob shown below. It is the classical program of printing a message on the screen. COBOL programs can also have the extensions *.cbl and .cpy:*

Let us go through the code for better understanding. First, let me make a comment. Don’t you feel that COBOL programs look like English text? I thought about showing one more COBOL program that almost looks like English prose but then decided not to do that. This lack of mathematical notations might be a hindrance to the mathematically oriented programmers of today but this was not so in the sixties, seventies and eighties where the bulk of the COBOL code was produced.

Now, let us come back to the program *cob1.cob,* which can be executed with the command *cobc -x -free cob1.cob.* The option -free tells the compiler that the program is written in free-form style. Initial versions of COBOL had strict regulations regarding the specific rows and columns in which a particular code should be written. The option -x makes sure that an executable file named cob1 is created. This executable file can be run using the command *./cob1.* Figure 1 shows the compilation and output of the program *cob1.cob.*

This program also tells us that COBOL programs are divided into different divisions. Some of these are mandatory and some others voluntary, and might be included in the program only when the need arises. Of course, COBOL does have its quirks and drawbacks. Imagine the horror of a seasoned Python programmer, who can find the sum of integers less than 100 by just using the one-liner print *sum(range(100)),* coming to know that he has to write half a page of meta-code to set up the environment even before writing a single line of actual code in COBOL. Yes! COBOL is really old and it does have problems due to its old age. Whatever the case may be, millions of lines of such code are still in existence and might be in need of maintenance. If a young programmer decides to learn and work with COBOL, he should think of himself as a mechanic repairing vintage cars. To the best of my belief, such mechanics earn well.

In this article we have seen why COBOL is still relevant and have learned the bare minimum to begin with COBOL. Before we part ways, let me attempt predicting the future of COBOL. I believe that as long as there are COBOL applications running in mainframe computers, the language will be relevant — and that will be happening for a long time from now. In the next and final article in this series we will discuss Pascal, which is another time tested, weather-beaten programming language that is still making an impact in the programming world.

In introductory programming the choice of first programming language and the corresponding first programming paradigm has a very important role. Based on the foundation of programming paradigms several strategies for teaching introductory programming exist, such as the algorithmfirst, imperative-first, functional-first, hardware-first, etc. approaches [1]. In Hungarian literature a slightly different naming convention is present [2] but the core concepts are basically the same. Based on our investigation of the curricula of the most renowned universities in Hungary that have some form of Computer Science

(CS) of Computer Engineering (CE) program1 , in Hungarian practice an algorithm-first approach is used in most cases starting with basic input-output operations and basic control structures and algorithms. This is usually followed up by the learning about the object-oriented paradigm. Some universities also introduce the functional paradigm early, along with the imperative and the object-oriented paradigms. The training programmes of the top US and European universities with CS or CE programs2 follow a similar pattern, however courses on functional programming are more prevalent. This shows that alongside the usual imperativefirst and algorithms-first approaches [3] the functional-first approach is also relevant [4,5] in current practice. The goal of this paper is to show that it is possible to use the tools of functional programming to introduce various topics of the classical algorithm-first and object-oriented-second methodologies, namely programming theorems and enumerators, and to show the analogies between the classical

Classic Hungarian programming education primarily uses the algorithm-first, object-orientedsecond approach based on the determinative works of Szlávi and Zsakó [6] that were also published in the booklet series Mikrológia [3,7,8]. One of the core concepts of this so called “Systematic programming” methodology is the usage of programming theorems. Programming theorems are a group of basic algorithms that provide solutions to generic group of algorithmic problems [7,9,10,11]. They also provide a framework for problem-solving by making the specification, planning and implementation of algorithms straightforward by using an analogy-based approach [10]. Szlávi and Zsakó identify eleven such programming theorems [7,11], the system of Gregorics has seven [9,10]. While there is a large intersection between the two lists, the names of theorems can vary. There is also a third source of naming conventions to consider: the vocabulary (keywords) of programming languages. In this paper we follow the naming conventions used by most programming languages to refer to the programming theorems. All programming theorems can be defined as a function that takes one or more collections of values as input. The type of the values in these collections can be arbitrary. Hereinafter we will use the generic types X (and Y) to refer these types. Szlávi and Zsakó group the programming theorems into two categories based on the number of their inputs and outputs [7,11] Based on this categorization they identify the following groups: In the first category there are programming theorems that take a single collection of values as an input and output a single value:

• Reduction (also called Folding): Returns a single value by applying a combining function systematically to a cumulative value and each element in the input collection. The combining function takes two values of type X and produces a single value of type X.

• Maximum Selection: Returns a single element (or its index) from the input collection that is the maximum (or minimum) based on some comparison between elements.

• Counting: Returns a single integer value that is the number of elements in the input collection that meet a certain condition.

• Decision: Returns a single logical value that shows whether the input collection has any values that meet a certain condition.

• Selection: Returns the first element (or its index) from the input collection that meets a certain condition (existence of such item is guaranteed by the precondition).

• Search: Returns the first element (or its index) from the collection that meets a certain condition if there is any such element. The second category has programming theorems that take one or more collections of values as an input and output one or more collections of values:

• Mapping: Returns a new collection in which each element is a transformed version of the value of the element with the same index in the input array. The transformation is done by some transforming function that takes a value of type X and returns a value of type Y.

• Filtering: Returns a new collection that contains those and only those elements in the input collection that meet a certain condition. Partitioning: Returns two collections. The first contains elements from the input collection that meet a certain condition, the second contains those elements that do not.

• Intersection: Takes two or more sets with elements of type X and returns a new set that contains those elements of the input sets that are present in all of them. • Union: Takes two or more sets with elements of type X and returns a new set that contains all values that are present in any of the input sets.

In classical algorithm-based programming we use the core concepts of imperative programming (i.e. statements, loops, and conditions) to implement programming theorems. In the following sections we provide the formal signature for each programming theorem and two implementations for each:

the classic implementation using imperative programming and an implementation using functional programming with the concept of folding. The goal of this comparison is to prove that the programming theorems can be introduced through not only imperative but functional programming as well.

For the following code examples a programming language or algorithm/function-description method had to be chosen [1,2,12,13]. For the purpose of the demonstration we chose the TypeScript programming language3 as it is a multi-purpose programming language that supports many programming paradigms (including imperative, functional, and object-oriented); thus, we can use it in all examples. This helps with the transition between the concepts that we present in the following sections. The syntax of TypeScript follows the conventions of C-style programming languages making the code easier to read. TypeScript also has support for static type annotations and template functions and classes. This makes the following code snippets generic for any input and output value types. All the implementations below are written as template functions and classes (using generic types X and Y). Another reasoning for choosing the TypeScript programming language for the demonstration is that TypeScript (and the JavaScript language that it is a superset of) has a lot of properties that are beneficial from an educational point of view. [14,15] The TypeScript programming language has a special value called undefined that represents a missing on uninitialized value. We used this undefined value for the Search theorem to represent “no output”. Implementations of some programming theorems assume a non-empty array as an input. Checking for this precondition is not present for more concise code. We also used some aliases for built-in types and values for the same reason. The list of these aliases can be seen in Table 1. For the functional implementations we use the array destructuring4 ([x0, ...xs]) and conditional statement (a ? b : c) features of TypeScript. All functional code snippets with the exception of the Selection theorem are based on pattern matching with conditional statements: if the input collection has exactly one element/has no elements then return some value, otherwise use a recursive formula to calculate the result. If a programming theorem cannot accept an empty input, then the first check

The Reduction (a.k.a. Folding, Sequential computing [11] or Summation [10]) programming theorem has two variants. The first takes the first value of the input collection and uses it as the starting value for the cumulation. The second variant uses a starting value (often named f0) for the reduction. This second

version is more general but, in most cases, we use a value that is neutral for the f function so that the starting value does not change the result (e.g. 0 for addition, 1 for multiplication). This means that most of the time this starting value can be omitted, and we can start the reduction with the first element of the input. In the following example we show the implementation for the first variant of the Reduction theorem that does not use the starting value. This means that only non-empty inputs can be accepted for the theorem. The Reduction theorem starts with the first element of the input then applies the f function to the current result and the next element of the collection until all elements of the input have been processed. The Maximum Selection theorem [10,11] takes the function m as an input which is a function that returns the maximum of two values based on some ordering. The theorem then uses this function to calculate the "larger" value of the current maximum and the next element in the input.

In addition to using folding and recursion, another way of implementing programming theorems with functional programming would be by using basic higher-order functions. The higher-order functions reduce, map, and filter are present in practically every programming language that supports functional programming (names may vary). Using only these three higher-order functions, it is possible to create easy "one-liner" solutions for all programming theorems (see figure below). This means that introducing only the concept of higher-order functions and these three basic functions are enough to easily solve problems that require the usage of programming theorems.

As seen in the code snippet above, reduce, map, and filter are exactly equivalent with the corresponding Reduction, Mapping and Filtering programming theorems. All the other theorems can be implemented using only these three higher-order functions. Many functional programming languages provide built-in functions for more than only these three theorems (e.g. in TypeScript the some method for the Decision and the find method for the Selection and Search theorems). However, reduce, map, and filter are very common in real-life programming and are enough to provide an "easy-enough" solution to the rest of the theorems. Szlávi and Zsakó group programming theorems based on whether their output is a single value or one or more collections of values [11]. Another way to group them could be based on their form in functional programming. It is possible to group the theorems into three distinct categories based on which higher-order function can be used for their implementations.

One of the classic ways to follow up an introductory, algorithm-first programming education is to continue with the means of data encapsulation and proceed towards object-oriented programming. This usually leads to the introduction of abstract data structures, classes, and objects. Another approach for proceeding towards object-oriented programming is with enumerators. Classic (imperative) enumerators are data types that have the following properties [9,17]:

Collections are a subtype of enumerators that store values of a specific type that can be enumerated [9]. They extend the Enumerable interface by a method that allows the addition of an element into the collection. This is required to implement programming theorems that output not only a single value but one or more new collections (e.g. Mapping, Filtering).

It is also possible to create enumerators for the functional implementations of programming theorems as well. This requires creating a new definition for functional enumerators to suit our requirements. The requirements for a functional enumerator are the following: • It is possible to decide whether it is empty, • ask for the first (head) element, • ask for the rest of the elements (tail) – i.e. elements except for the first.

Programming theorems for such enumerators can be implemented as standalone functions or as methods of an enumerator class itself [18]. If we use the latter method, we must use an abstract class instead of an interface to be able to implement the theorems as class methods. As for the methods of the Enumerable interface, we leave them as abstract methods, showing that these must be implemented for each specific enumerator (e.g. sequential input file enumerator, range enumerator, etc). The theorems can still use these abstract methods in their implementations.

Programming theorems for functional enumerators can be implemented either as standalone functions or methods on the enumerator with the folding method that is shown in Section 2.1. The main difference between the two approaches is that if a theorem is implemented as a method then it does not have to take the collection as an input parameter, it is automatically passed via the this reference. As shown in Section 2.2, we only need to implement three methods, reduce, map, and filter to have access to all programming theorems. As with these enumerators we target functional programming it is important that the theorem implementations work in accordance with the main principles of functional programming. This means that none of the theorems can change

the internal state of an enumerator (they should be immutable) and must return a new enumerator when necessary.

In classic programming education we usually follow an algorithm-first, object-oriented-second approach. With this method the focus in early programming is on the low-level concepts, i.e. how things work and how to implement basic algorithms. With a functional-first approach the same can be said if we start with the low-level concepts first, like pattern-matching or folding. This approach results in a bottom-up learning process. However, there is the possibility to start programming education with a higher-order-first approach. This would mean that programming is introduced with high-level, functional-style programming: using collections and the reduce, map and filter functions as shown in Section 2.2. This would allow students to easily solve most data-processing tasks easily using high-level tools. Based on our experiences in various introductory and web programming courses and our experience with teaching pupils in summer camps [15] this top-down learning approach can emphasise problem-solving, giving student early satisfaction and quick success to keep their motivation high. Principles of object-oriented programming could also be introduced early by data-encapsulation with classes and objects and data-processing methods. It would also be possible to use this approach combined with web technologies, web programming in the browser, and the principles of the constructionist learning theory to create a motivating and efficient framework for learning programming [15]. In later stages of a higher-order-first educational approach, it could be possible for students to learn about the internal operation of the functional theorems by creating custom enumerable data structures. The implementation of the theorems on custom enumerators could be either in a functional or imperative style thus focusing on how things work. This approach could help students to learn about various programming paradigms and how to combine them. As such higher-order-first approach would initially only use the Reduction, Mapping and Filtering theorems as shown in Section 2.2, some other theorems will be less efficient than some other implementations. One example would be the Partitioning theorem that is solved by applying the Filtering theorem twice in succession (thus iterating through the input two times), however on a lower level it can be solved by a single iteration over the input collection. In a top-down learning approach this lack of efficiency is not necessarily an issue, as the focus is on solving problems. Most of the times we do not require highly efficient programs and working with the occasional sub-optimal theorem

implementation is perfectly fine. Also, some programming languages provide many built-in higher-order functions that solve more theorems efficiently similarly to how the reduce, map and filter functions solve the Reduction, Mapping and Filtering theorems. In later stages of the learning process the efficiency aspect can be covered in more detail as well, and more effective solutions can be implemented with imperative or low-level functional programming.

Conclusions

Programming theorems and enumerators form a solid foundation for classical algorithm-first, objectoriented-second programming curriculums that are very popular all over the globe. This approach emphasises understanding the low-level concepts of programming and how to use those concepts to build more and more complex algorithms and data structures to solve problems. It is possible to use functional programming to implement the same programming theorems and enumerators. This means that it is possible to create a functional-first programming curriculum that is analogous to this classic method as it uses the same programming theorems and enumerators as its foundation. With functional-style programming and higher-order functions three programming theorems are enough to provide concise albeit form an efficiency perspective sometimes sub-optimal solutions to the rest of the theorems. Practically every programming language that supports functional programming have these three programming theorems available as higher-order functions out of the box. Based on these functions it could be possible to create a higher-order-first curriculum for teaching programming. This approach would facilitate a problem-solving centred learning process and would focus less on the low-level inner workings of programming theorems in the early stages of learning programming. In later stages it could also be possible to work our ways towards implementing new enumerators and other data structures that give a deeper understanding of the underlying algorithms or functional constructs. The browser and web programming combined with a constructionist learning methodology could provide a suitable environment for learning activities using this higher-order-first approach, thus they could hold great potential as a platform for learning programming using this methodology.