Functional programming, in the data structures course, in Java

John MacCormick
Dickinson College
Carlisle, PA
jmac@dickinson.edu

Abstract

We describe a new curriculum design, termed *functional embedding*, for embedding functional programming within a core sequence of courses taught in a non-functional programming language such as Java. We present evidence that functional embedding has been successful in practice, based on a survey of student perceptions and analysis of student exam performance. An analysis of college computer science curriculums demonstrates that at least 59% of colleges can benefit from the approach.

1 Introduction

There is wide agreement among computer science educators that it is important for students to acquire proficiency in a variety of programming paradigms. Of these paradigms, perhaps the most important are the object-oriented and functional paradigms. Modern computer science curriculums rarely have any difficulty in ensuring that all students are exposed to the object-oriented paradigm, but the same is not true of the functional paradigm. One relatively common curriculum design has a core sequence of three or four courses containing no functional programming. This core sequence of exclusively imperative/object-oriented content may comprise, for example, CS1→CS2→DSA (where DSA is a single course covering data structures and algorithms), or CS0→CS1→Data

Structures→Algorithms (where CS0 is a course skipped by students with prior programming experience). This curriculum design typically includes a separate Programming Languages (PL) course which contains a substantial functional programming component (say, a minimum of 30% functional—but we observe up to 100% functional programming in some PL courses). If the PL course is required for the computer science major, all CS majors are thus guaranteed reasonable exposure to functional programming.

It is not uncommon, however, for the PL course to be an elective. If the core sequence is exclusively imperative/object-oriented, this raises an important curricular problem: students who do not take the elective PL course can graduate with no awareness of the functional paradigm. The CS education community is divided on whether this should be considered a severe problem; section 5 gives more details on why it may be considered problematic that an undergraduate computer scientist can graduate with no knowledge of functional programming. In this paper, we propose a partial solution to this problem and report on our experiences with it.

The proposed solution is to insert a suitable amount of functional programming into the core sequence, without changing the (imperative, object-oriented) programming languages in which the core sequence is taught. Our department has experimented successfully with this approach since 2019, by making alterations to our CS2 data structures course, which is taught in Java. By reallocating material between our separate data structures and algorithms courses, we made room for a one-week unit of functional programming material. We do not introduce the overhead of studying a truly functional programming language. Instead, we use Java lambda expressions and other features in `Java.lang.function`, combined with practical applications for efficient parallel data processing via Java's Stream API.

This explains the title of this paper: we teach some aspects of functional programming, in the data structures course, in Java. The remainder of the paper will describe the implementation details and results of that approach. Nevertheless, we believe it would generalize to other situations. For example, it is certainly possible to create similar units of functional programming in courses that use Python or C++. And these units can easily be inserted into courses other than Data Structures, including CS1 or Algorithms. Indeed, one of our items of future work is to weave functional programming into most other required courses, now that we can be sure students have seen it before.

2 Related work and a taxonomy of functional programming curriculums

In this section, we briefly survey other approaches to incorporating functional programming within the computer science curriculum, and we propose a taxonomy of five possible approaches: functional-first, functional-required, functional-elective, functional-minimal, and functional-embedded.

Many previous researchers have investigated the use of functional programming within the core sequence of programming courses in a computer science major. One highly successful and much-studied approach is known as *functional-first* [7]: CS1, the first core programming course, is taught either entirely or largely in the functional paradigm. Famous examples of such curriculums include those of Grinnell College [1] and Brown University [4], amplified by the influential textbook *How to Design Programs* [3]. The functional-first approach has numerous well-documented advantages, but it also presents challenges. Empirically, we find that it has not been widely adopted. For example, in our sample of CCSC curriculums described later (section 4), we found that zero out of 29 institutions had adopted functional-first.

As described in the introduction, a common curricular approach is to include a substantial topic on functional programming within a programming languages (PL) course. If the PL course is required for the major, we refer to this as the *functional-required* approach; if the PL course is an elective, we describe the approach as *functional-elective.*

There are some computer science curriculums in which functional programming does not appear to be an explicit goal. Either it is not present at all, or it makes an appearance as a byproduct of certain elective courses not focussed on programming languages (such as an AI course that makes use of LISP). We refer to this curricular approach as *functional-minimal.*

In this paper, we advocate a new curricular approach to functional programming: a modest but nontrivial amount of functional programming is embedded into one or more of the courses in the core sequence, such as CS2/data structures. Moreover, embedded functional programming content is emphasized as an important topic that will be reflected in other parts of the curriculum. We refer to this approach as *functional-embedded.*

3 Method for embedding functional programming in a Java-based data structures course

In this section we give details of our current approach to embedding functional programming in a Java-based data structures course. Section 3.1 describes a low-overhead way of introducing functions as first-class objects, and section 3.2

describes a practical application for cementing student engagement via the Java Stream API. We provide these details for concreteness only; we believe many other approaches could achieve the same goals.

3.1 Rudiments of functional programming in Java

It can be argued that the most important concepts underlying functional programming are *immutability* (lack of side effects) and the treatment of functions as *first-class objects*. The brief survey of functional programming described here focuses on functions as first-class objects, mentioning immutability only as it applies to existing Java classes such as `String` and `Stream`. In practice, students learn that functions can be: (i) parameters of other functions; (ii) return values of other functions; and (iii) created and employed as local variables. Simple demos of these three features can easily be achieved using Java's functional programming package, `java.util.function`, which has been available since Java version 8 was released in 2014. For example, we can define the function $f(x) = 3x^2 + 5$ and then evaluate $f(2)$ via the following snippet of Java:

```
Function<Integer, Integer> f = x->3*x*x+5;
System.out.println(f.apply(2)); // prints "17"
```

There is one piece of syntactic ugliness that is unavoidable in the above snippet: to evaluate $f(2)$ we need to invoke the `apply()` method on the object `f`. Thus, we write `f.apply(2)` rather than using the more natural notation `f(2)`. This reflects the fact that Java is not a functional language: one cannot define a Java object that is a function. Instead, we create an object that implements a *functional interface*. The interface includes the `apply()` method, and that is how the function must be invoked.

To avoid this potential source of confusion, we prefer to first introduce the notion of functions as first-class objects using Python. We have found this works well, even in a course that is otherwise 100% Java and does not assume any prior knowledge of other programming languages. The syntax of Python is close enough to standard pseudocode conventions that students with no knowledge of Python can leap right in. In our experience, they can actively participate in an in-class, browser-based mini-lab on the topic of functions as objects after only a few minutes' explanation by the instructor. For example, the above Java snippet becomes

```
def f(x): return 3*x*x+5
print( f(2) ) # prints "17"
```

The above snippet can be formulated using a lambda expression in Python, and in our approach students will certainly learn how to do that. But we prefer to delay the introduction of lambda expressions until after the idea of functions as objects has already been demonstrated.

For example, the concept of passing a function as a parameter is always a challenging new abstraction the first time it is seen by a student. But a hands-on mini-lab, using snippets such as the following, can help students quickly adapt to this new idea:

```python
def isIncreasingOn123(f): return f(1)<f(2) and f(2)<f(3)
def add5(x): return x+5
print( isIncreasingOn123(add5) ) # prints "True"
def applyTo9(f): return f(9)
print( applyTo9(add5) ) # prints "14"
```

Once the idea of treating functions like any other data item is familiar, we can switch back to Java. We recommend omitting the details of functional interfaces, instead treating the `apply()` method as a required piece of Java syntax that is not explained further. Thus, the above Python snippets become:

```java
public static boolean isIncreasingOn123(Function<Integer, Integer> f) {
        return f.apply(1) < f.apply(2) && f.apply(2) < f.apply(3);
}

public static int applyTo9(Function<Integer, Integer> f) {
        return f.apply(9);
}

public static void main(String[ ] args) {
        Function<Integer, Integer> add5 = x->x+5;
        System.out.println( isIncreasingOn123(add5) ); // prints "true"
        System.out.println( applyTo9(add5) ); // prints "14"
}
```

Some further fundamentals of functional programming, including moderately advanced lambda expressions, can be introduced in a similar fashion. Further details are available in our publicly-available textbook chapter [9].

3.2 A practical application of functional programming: parallel processing of data streams

It is a well-known educational principle, not just within computer science, that learning outcomes for a theoretical concept are improved if students perceive the concept as applicable or useful [5]. We believe, therefore, that it is important for students to apply their knowledge of functional programming in Java to some real-world situations. For this, we use Java's Stream API; this is provided in `java.util.stream` and was introduced by Java version 8, at the same time as the functional programming facilities, in 2014.

In Java, `Stream<T>` is an interface for performing operations on sequences of objects of type `T`. There are two types of operations on streams: intermediate operations and terminal operations. In the brief coverage provided in our data structures course, we explore only eight of these operations: the four intermediate operations `filter()`, `map()`, `sequential()`, and `parallel()`; and the four terminal operations `count()`, `foreach()`, `reduce()`, and `sum()`. Any computation based on a stream performs a sequence of intermediate operations followed by one terminal operation. Because many of these operations accept lambda expressions as parameters, they are an excellent way to practice the application of functional programming. For example, the following snippet counts the number of words in a stream that begin with '`c`' and end with '`t`', by using two `filter()` operations followed by the `count()` operation; it employs two lambda expressions for the filters:

```
Stream<String> s = Stream.of("cat", "bat", "catch", "chat");
long numWords = s.filter(word -> word.startsWith("c"))
                 .filter(word -> word.endsWith("t"))
                 .count(); // returns 2
```

The Stream API provides opportunities for students to implement realistic experiments from the world of big data, based on only modest guidance from the instructor—certainly less than one class meeting, in our experience. For example, students can apply the map-reduce [2] framework to large data sets, and/or demonstrate the speed-up from using parallel versus sequential streams. This latter experiment is trivial to implement: one simply prepends the call "`.parallel()`" to the sequence of stream operations. Further details are available in our publicly-available textbook chapter [9].

4 Results

In this section we present results from three investigations which suggest that the functional-embedded approach (teaching functional programming in the data structures course and/or elsewhere in the core) is efficacious and suitable for incorporation in a substantial proportion of existing computer science curriculums.

Investigation 1: Student performance. We analyzed the final exam scores of students in the fall 2021 instance of the data structures course. A total of 25 students completed the course, and all are included in this analysis. The final exam included a three-part question requiring students to write code using the Java Stream API and employing lambda expressions. Students averaged

28/30 points on this question. On the most challenging part of the question, which required a non-trivial application of the map-reduce pattern, 22 of the 25 students (88%) scored 13/15 or higher. This demonstrates that a key application of functional programming was mastered by a strong majority of students.

Investigation 2: Student perceptions. We surveyed students who completed our functional-embedded data structures course, assessing their perception of the 9 topics in the course; 32 students participated in the survey. They rated each topic based on how "interesting" and (separately) "useful" they perceived it to be. Ratings employed a 5-point Likert scale, from 1 = "not at all interesting" to 5 = "extremely interesting" (and similarly for "useful"). Results are shown in figure 1. Functional programming was certainly perceived as interesting and useful: it averaged 3.7 and 3.5 respectively. That is, the average response places functional programming between "moderately interesting" and "very interesting" (and similarly for "useful"). The level of interest in functional programming was commensurate with most other topics; only the binary search tree/heaps topic received a statistically significant higher score. However, functional programming was ranked ninth out of the nine topics in terms of usefulness, substantially lower than classic data structures topics such as lists/stacks/queues (average 4.4) and graphs (4.3); these two differences have high statistical significance (t-test p-value $< 10^{-3}$). The low ranking of functional programming in terms of perceived usefulness may indicate that we could do a better job of demonstrating the importance of functional programming. Or maybe the student perception is correct: perhaps fundamentals such as stacks, queues, and graphs really are more useful than functional programming.

Investigation 3: Curriculum survey. We analyzed the curriculums for the computer science major of the 32 colleges and universities whose faculty contributed to the four most recent CCSC conference proceedings [8]. Three institutions that do not offer a four-year computer science major were excluded. For the remaining 29 institutions, we examined publicly-available requirements for the major and course descriptions to classify each institution's curriculum according to the taxonomy described in section 2. Figure 2 shows the results.

Notwithstanding the well-known examples of functional-first curriculums mentioned earlier, we find that, in this sample, none of these institutions employs the functional-first approach. Unsurprisingly, no institutions employ the functional-embedded approach either: this is the new approach used at our own institution and which we are advocating in the present paper.

The remaining three categories demonstrate that there is significant diversity in how these institutions cover functional programming: 79% include it in a programming languages (PL) course, split almost equally between the

topic	average Likert score for… "interesting"	"useful"
Binary search trees/heaps	4.2*	4.3**
Sorting algorithms	4.1	4.4**
Hash tables	4.0	3.9*
Graphs	3.9	4.3**
Lists/stacks/queues	3.8	4.4**
Functional programming	**3.7**	**3.5**
Recursion	3.7	4.1**
Algorithm Analysis	3.5	4.2**
Generics	3.1**	3.6

Figure 1: **Results of student perception survey.** Topics are sorted by the average rating for "interesting." Starred values indicate a statistically significant p-value for a two-tailed paired t-test comparing the given result with the result for functional programming: single * for $p < 0.05$; double ** for $p < 0.01$.

Approach	frequency	percentage
functional-first	0	0%
functional-embedded (proposed here)	0	0%
functional-required	12	41%
functional-elective	11	38%
functional-minimal	6	21%

Figure 2: Results of the functional programming curriculum survey.

functional-required and functional-elective approaches. That is, about half of the 79% with a PL course (41%) require the PL course for the major, and the other half (another 38%) offer PL as an elective. Finally, 21% of the institutions employ the functional-minimal approach, meaning they do not offer a significant amount of functional programming.

We believe these results demonstrate ample opportunity for the functional-embedded approach advocated here. Certainly, the 59% of institutions where CS majors can graduate with no exposure to functional programming could rectify this using functional-embedding. In addition, we believe that the 41% in the functional-required category would also benefit from embedding a certain amount of functional programming into one or more core courses such as CS2. This has the benefit of allowing the functional mindset to start earlier and resonate in other parts of the curriculum; perhaps students will view it as a more fundamental and integral technique, rather than an approach of mainly theoretical interest that has been sidelined to a challenging upper-level course. To summarize, 100% of the institutions in this sample may benefit from adopting the functional-embedded approach, and 59% would eliminate the possibility of students graduating with no exposure to functional programming.

5 Discussion

This paper makes the assumption that it is desirable for undergraduate computer scientists to be exposed to functional programming. It is beyond our scope to justify this claim, which has been the subject of debate for decades. Functional ideas have become increasingly important even within primarily imperative/object-oriented languages, and this is one argument for teaching functional programming explicitly. Another argument is that the ACM/IEEE 2013 Curriculum Guidelines ([7], page 156) require 7 core lecture hours of functional programming, equivalent to 2–3 weeks of classes. The more recent 2020 guidelines, known as CC2020 [6], express required content in terms of competencies rather than lecture hours. The guidelines list 84 competencies, of which 3 mention functional programming explicitly ([6], page 114; more details below). This can be converted very approximately to comprehensible units as follows. In a 10-course major, one of the courses would need to devote $3/84 \times 10 = 36\%$ of its time to covering the three functional programming competencies—equivalent to 5 weeks in a 14-week semester. This level of coverage is probably not a realistic goal for every CS program. But these curriculum guidelines imply that it is highly desirable for every computer science undergraduate to receive some exposure to functional programming ideas. As discussed in the previous section, our curriculum analysis shows that about 60% of institutions in our sample do not currently achieve this goal, and they could do so by using the functional-embedded approach.

One potential limitation of our functional-embedded approach is that we are not teaching the full range of functional thinking: our emphasis is on the practical use of lambda expressions. It lies beyond the scope of this paper to debate this in detail, but this is an interesting line of future research. If we are limited to a very small amount of time for functional programming, what are the most essential and useful ideas to convey? As a partial justification for the approach advocated here, we note that reasonable progress towards two of the three CC2020 functional programming competencies mentioned above can be achieved using only the material covered in our functional-embedded CS2 data structures course. In detail, the two competencies covered are PL-A (implement a function that takes and returns other functions) and PL-D (use operations on aggregates, including operations that take functions as arguments).

The competency not covered is PL-E (contrast the procedural/functional approach with the object-oriented approach).

Another potential limitation is that, at present, we devote only one week to functional programming in the data structures course. In fact, we have been satisfied with the level of conceptual coverage, but the one-week timeframe does seem short for a valuable topic. We do commit to functional programming as an important concept, listing it in the official bulletin description of the course and in one of the department-approved learning goals.

Hence, an area of future work is to thread some functional programming ideas into additional courses. At our institution, students also encounter lambda expressions in a software engineering course that relies on JavaScript web frameworks. We believe they could benefit from further exposure in the required algorithms course and in electives such as artificial intelligence. As a specific example of this, note that the `sorted()` function in the standard Python library accepts a functional parameter for extracting the key from each item. Hence, the instructor has an opportunity for a quick refresher on lambda expressions whenever sorting a list in Python. But an important issue for further investigation is, how can we embed deeper functional programming concepts, such as immutability?

6 Conclusion

We have described *functional embedding*, a new curriculum design that teaches a modest but meaningful amount of functional programming within a core course taught in a non-functional programming language. An analysis of CCSC college curriculums showed that about 60% of colleges in the sample do not currently ensure that CS majors have exposure to functional programming, and that they could achieve this with relative ease via functional embedding. We argued that the remaining 40% of colleges may also gain benefits from functional embedding. A survey of student perceptions shows that functional programming is perceived as interesting and useful, albeit to a lesser extent than classical topics in data structures. An analysis of student exam results demonstrates solid achievement on sophisticated learning goals such as the map-reduce paradigm, even when only one week of functional programming is embedded. We hope functional embedding will be adopted at other institutions and that future work can refine the embedding of functional programming throughout the typical computer science curriculum.

References

[1] Sarah Dahlby Albright, Titus H. Klinge, and Samuel A. Rebelsky. "A Functional Approach to Data Science in CS1". In: *Proc. SIGCSE.* 2018, pp. 1035–1040.

[2] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[3] M. Felleisen et al. *How to Design Programs: An Introduction to Programming and Computing.* 2nd edition. MIT Press, 2018.

[4] Robert Bruce Findler et al. "DrScheme: A programming environment for Scheme". In: *Journal of Functional Programming* 12.2 (2002), pp. 159–182.

[5] L. Dee Fink. *Creating significant learning experiences: An integrated approach to designing college courses.* 2nd edition. John Wiley & Sons, 2013.

[6] CC2020 Task Force. *Computing Curricula 2020: Paradigms for Global Computing Education.* Association for Computing Machinery, 2020.

[7] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.* Association for Computing Machinery, 2013.

[8] Baochuan Lu, ed. *Journal of Computing Sciences in Colleges.* Vol. 37. 6,7,8,10. Consortium for Computing Sciences in Colleges, 2022.

[9] John MacCormick. "Functional Programming and Streams". Available as https://arxiv.org/abs/2302.09403. 2023.