Uncovering Difficulties in Learning for the Intermediate Programmer

Adrienne Decker, David Simkins
School of Interactive Games and Media and RIT Center for Media, Arts, Games, Interaction and Creativity (MAGIC)
Rochester Institute of Technology
Rochester, NY - USA
adrienne.decker@rit.edu, dwsigm@rit.edu

Abstract— **Within computer science education, we have spent considerable effort on the introduction to the discipline (particularly to programming) and the teaching of novice programmers. However, we do not often think about the teaching and learning for the intermediate students. Having data about student's perceptions coming into a second year data structures course, it became of interest to systematically analyze the data to see what if any interesting patterns or results we could see in this information. Using the data from a first day of class general survey, we were able to gather information about student's perceived level of difficulty with some of the major topics that they have studied up until this point in the curriculum. The first half of the survey consisted of ten Likert-type items representing ten different topics in the previous courses. We are able to present quantitative analysis of the students perceived level of difficulty with these topics. The second half of the survey asked several open-ended questions about their learning. In this paper, we also provide analysis of the first of these questions which asked them to identify a topic that they studied previously that they still do not understand well. While not yet giving us the ability to generalize, this data shows some interesting information about students' perceived difficulty at a point in the curriculum that is not often studied, the intermediate years, and the results make an interesting case for further exploration of this problem with a larger group of students at more and diverse institutions.**

Keywords— Student learning; student misunderstanding; student difficulty; student misconceptions; intermediate student; intermediate programmer

I. INTRODUCTION

Within computer science education, we have spent considerable effort on the introduction to the discipline (particularly to programming) and the teaching of novice programmers. However, we do not often think about the teaching and learning for the intermediate students.

Having survey data from a course for intermediate students about their perceived difficulty with prerequisite material, it seemed like an opportunity to learn about the mindset of these students and their beliefs about their learning. The data comes from a first day of class questionnaire designed for the instructor to learn more about the prospective students in the class.

In this paper, we describe the first part of the analysis of this survey data and relate it to what the literature has shown us about student's perceived difficulties and misconceptions. However, due to the fact that this survey was not designed for the purpose of in-depth analysis of student learning and generalization, we view this as a first step and a starting point for further research on this group of students.

II. BACKGROUND

In the computer science education literature, we see studies of student misconceptions and various approaches to courses and curriculum that attempt to mitigate student difficulties.

A. Programming

In the 1990s, we saw a shift in introductory programming education to embrace object-oriented programming. While it is difficult to say if there was sufficient evidence to determine success or failure of this approach for better teaching in the introductory courses [1, 2, 3], concepts of object-oriented programming remain in latest curriculum revision and are still being taught at most institutions [4]. We do find evidence that students struggle with these concepts [5, 6], and the arguably advanced object-oriented concepts of inheritance and polymorphism make their way into the curriculum early on. Several have found that students struggle with these concepts and fail to understand them completely [7, 8].

There are other approaches to introducing students to programming that do not rely on object-oriented approaches or languages. Some of these programs [9, 10] follow a data- and test-driven approach and use Racket, a functional language in the first course. In Fisler's work [11], we see her examine Soloway's Rainfall Problem with students who have used one of these approaches. She determines that these students approach the problem and its solution differently than in previous Rainfall Problem studies particularly with regard to their use of high-level structures (loops and sentinels) in their code. Ramsey [12] gives a narrative of his observations of using this functional approach and the difficulties students face when programming in it. He observes that their difficulties are different than expected as well. As such, we wonder if student difficulties will be expressed for different topics if their first exposure to programming uses this

methodology, as was the case for the students surveyed.

B. Proofs, Induction, Recursion, DFAs

Another area where we find students struggling is in the area of mathematical proof, particularly induction proofs [13].

We are also aware that students have difficulty understanding and making mental models of recursion [14, 15, 16], a failure to comprehend recursion vs. iteration [17] as well as failure to recognize when to use recursion in problem solving [18]. There have been attempts to mitigate this in various ways [19, 20], including the marrying of induction and recursion [21, 22].

The study of automata theory, particularly DFAs, has been the focus of several initiatives to help students better understand the material. Students often express difficulty with this topic and instructors have looked for ways to make the material more relevant and engaging [23, 24, 25, 26].

C. Misconceptions

A number of studies have been done to look at student misconceptions about programming in the early stages of their academic career. We see work with the Delphi process to uncover difficult concepts [27], as well as continued development of concept inventories [28, 29]. There is the McCracken report [30] which indicates that students inability to problem-solve leads to their inability to program at the end of the first year, as well as the follow-up working group report from Lister et al. [31] that looks at lack of reading and tracing abilities with code as an explanation for student difficulty.

For data structures, we see think-aloud and "explain in plain English" techniques employed to help students learn and to help us understand what they have learned and understood about the concepts [32, 33].

D. Intermediate Students

Unfortunately, there is not a lot of published studies about work with intermediate students and their learning of computer science concepts. Kopec, Yarmish, and Cheung [34] study intermediate student programming errors and finds that nested loops and recursion are sources of difficulty and postulate that this is the barrier between novice and intermediate programmers. Further work by Yarmish and Kopec [35] looks into the errors of what they call "advanced novice programmers" with regards to nested loops, arrays, recursion and functions and creates a taxonomy to classify them. Shabo, Guzdial, and Stasko [36] discuss a cognitive apprenticeship approach to help the intermediate student navigate the path from novice to expert.

In the end, we are left with a population that is not well studied and one whose transition from novice to expert may arguably be one of the most important parts of their educational journey.

E. Apprenticeship, Identity, Practice

Each field has challenges associated with the transition from one level of knowledge to another. In disciplines such as the computing sciences, where knowledge instruction is paired tightly with procedures and best practices, the process of developing a learner from beginner to expert can be considered an apprenticeship [37]. In the apprenticeship model of instruction, methods are generally active and engage the learner in a process of identity development with the goal of integrating and affiliating the learner into a practice [38]. The introduction of students to programming can be seen as an apprenticeship into practice, although it is often not explicitly seen as such and instead focuses on introducing students to new knowledge. Nevertheless, the goal of teaching programming is to teach people how to use it, knowledge in practice.

Classic apprenticeship studies show how learners move from beginning through intermediate stages to expertise in practices such as architecture and navigation [39]. We are still lacking in rigorous description of the intermediate stages of apprenticeship when it comes to computer science education.

In looking at the areas of greatest difficulty according to the self-report of intermediate students, we can begin to gain insight not only into where they have struggled, but also in how they see themselves in relation to learning. Self-report about difficulties can provide insight into this intermediate group's current state of identity development, and can give warnings of potential places where theories of intelligence in development are helping or hindering the learning process. Most notably, it can help identify students who indicate an entity theory of intelligence, in which one believes that potential is purely or primarily innate, not learned. Such a theory of intelligence can be detrimental to student learning compared with incremental theories of intelligence [40, 41].

III. SURVEY DATA

The survey data was collected from an anonymous first day of class survey given the first day of the summer 2015 term at a large public west coast university. This instrument was given by the instructor to gauge perceived student difficulty of basic concepts taught by previous courses and to gain an understanding of common student difficulties in preparation for teaching the summer course, a course in data structures.

A. The Course

The *Data Structures* course would be typically taken during the second year of the program after courses that are roughly equivalent to CS1, some of CS2, and Discrete Math. The students often will have taken a course that covers basic logic circuits and DFAs before or while concurrently taking this course. *Data Structures* for summer 2015 covered: linked lists, stacks, and queues; Algorithm analysis (Big O, etc.); time and space complexity; induction and recursion; loop invariants and program correctness; sorting; binary search trees; tree rotation and B-trees; priority queues and heaps; hashing; graphs; counting.

The two direct prerequisites for the data structures course are *CS2* course and *Discrete Math*. The *CS2* course is a course in object-oriented software development and focuses on data abstraction, polymorphism, design patterns, and object-oriented design principles. Projects are completed in Java.

Prior to completing *CS2*, students must have completed an introductory course in programming that follows the Systematic Program Design curriculum [9] and projects are completed in Racket.

The *Discrete Math* course covers propositional and predicate logic, proofs (logic, direct, induction, contrapositive, and contradiction), sequential circuits, sets, functions, and DFAs.

B. The Survey

The survey consisted of ten 5-point Likert-type questions and three free response/short answer questions. The Likert-type questions were preceded by the following prompt:

> *Thinking back to your previous computer science courses, for each of the topics listed, indicate how easy/difficult you felt it was to learn.*

Students were then presented with the following topics:

- Learning to write programs in Racket (Q1)
- Learning to write programs in Java (Q2)
- Using variables in a program (Q3)
- Writing functions (Q4)
- Writing recursive functions (Q5)
- Writing proofs (Q6)
- Creating a DFA that solves a given problem (Q7)
- Polymorphism (Q8)
- Writing classes (Q9)
- Writing a program that has lists of data (Q10)

The possible responses were: *Very Difficult, Difficult, Neither Difficult nor Easy, Easy,* and *Very Easy*.

The three free response questions were:

- Give an example of something you studied in your previous computer science courses (does not have to be from the list above) that you feel you still don't understand very well.
- Why do you feel the topic you indicated was difficult for you to understand?
- Describe at least one topic from your previous computer science courses (does not have to be from the list above) that was difficult for you to learn and how you eventually learned it. (Use back of page if necessary.)

C. Results

In this paper, we will focus on the results of the Likert-type items and the examples given by students for the first of the free response questions.

1) Likert-Type Items

Table I shows the number of responses, mean, standard deviation, median, and mode for each of the ten Likert-type items. For calculating purposes, *Very Difficult* was coded as the value 1 and *Very Easy* was coded as the value 5.

There was a statistically significant difference between items as determined by one-way ANOVA ($F(9,1631) = 25.35$, $p < .001$). To determine where the differences were, Tukey's HSD was performed. Table II shows pairs of items whose differences were identified as significant according to the Tukey's analysis at a 95% confidence level. Pairs of items not listed in the table did not show significance. The table also indicates the direction of the difference based on the means. Since the value 1 was coded for *Very Difficult*, if there is significance in the difference and the mean is lower, there is evidence that the students perceived that topic as being more difficult.

Fig. 1 through Fig. 10 plot the number of survey responses for each option (*Very Difficult* to *Very Easy*) for each of the questions from this section of the survey.

TABLE I. DESCRIPTIVE STATISTICS FOR LIKERT-TYPE ITEMS

Topic	N	Mean	Std Dev	Median	Mode
Racket	156	3.19	1.03	3	4
Java	172	3.38	0.99	4	4
Variables	171	3.94	0.82	4	4
Functions	171	3.63	0.91	4	4
Recursive Functions	170	3.11	1.01	3	4
Proofs	168	2.54	1.06	2	2
DFA	143	3.08	0.96	3	3
Polymorphism	160	3.24	0.99	3	3
Classes	170	3.44	0.95	4	4
Lists	160	3.4	0.90	3	3

TABLE II. SIGNIFICANT DIFFERENCES IDENTIFIED BY TUKEY HSD

Topic	Perceived by students to be more difficult than
Writing programs in Racket	• Writing programs in Java • Writing Functions
Writing programs in Java	• Using variables
Writing recursive function	• Using variables • Writing functions • Writing classes
Writing Proofs	• Writing progams in Racket • Writing programs in Java • Using variables • Writing functions • Writing recursive functions • Creating a DFA that solves a … • Polymorphism • Writing classes • …program that has lists of data
Creating a DFA that solves a given problem	• Using variables • Writing functions • Writing classes
Polymorphism	• Using variables • Writing functions
Writing classes	• Using variables
Writing a program that has lists of data	• Using variables

Fig. 1. Responses for Q1

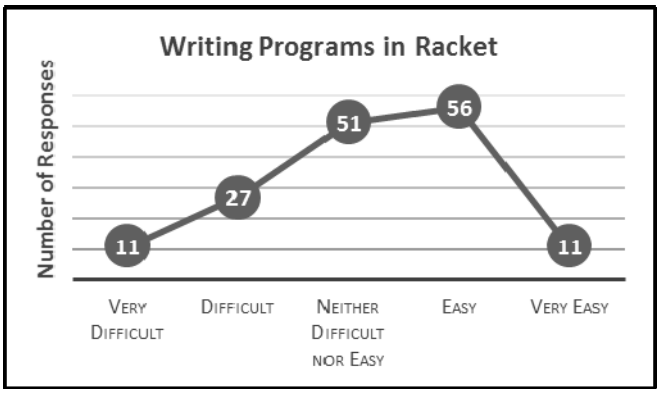


Fig. 2. Responses for Q2

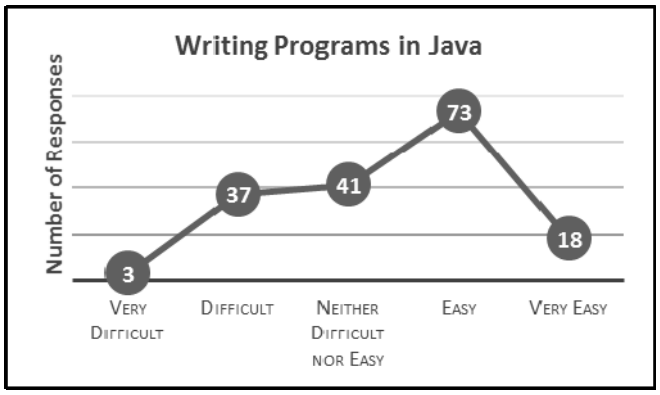


Fig. 3. Responses for Q3

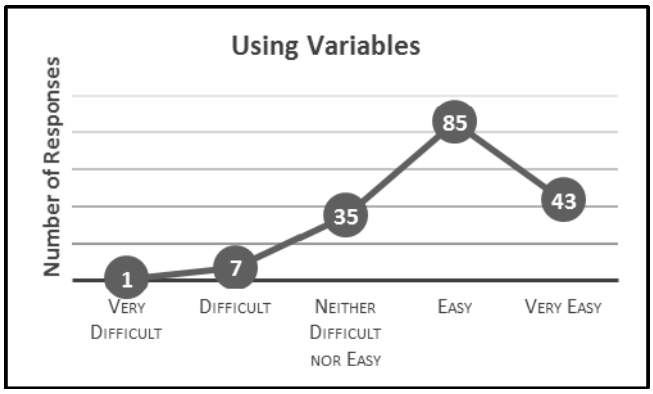


Fig. 4. Responses for Q4

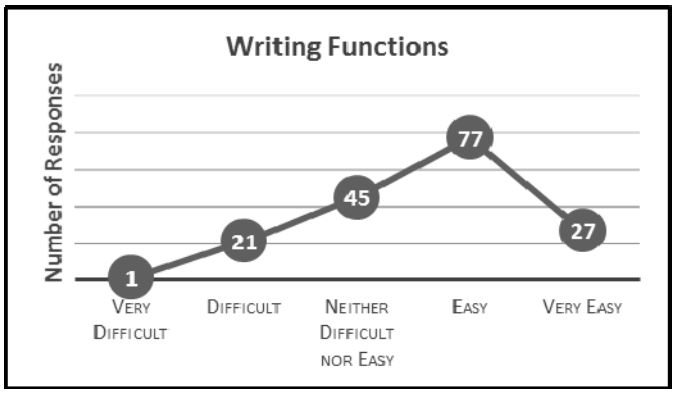


Fig. 5. Responses for Q5

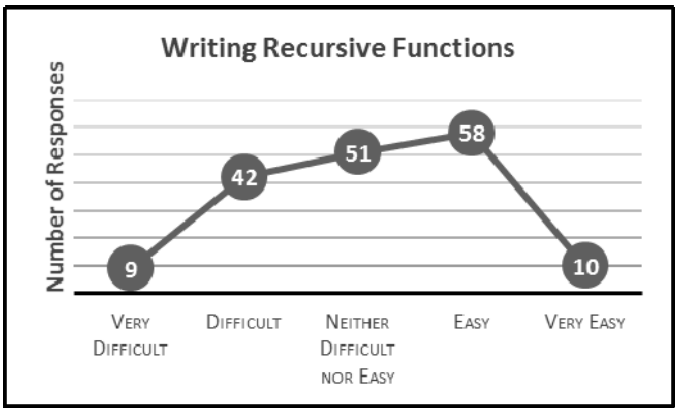


Fig. 6. Responses for Q6

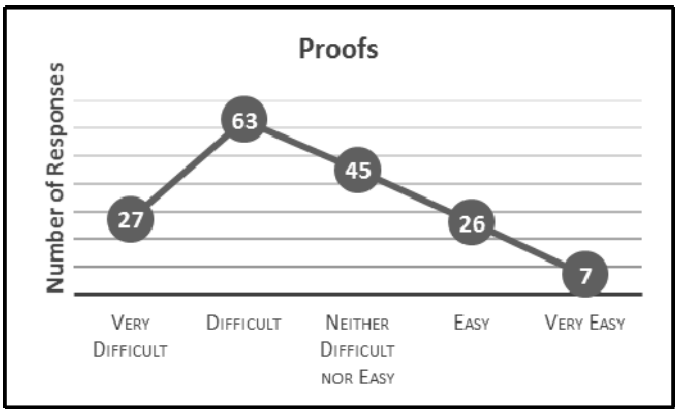


Fig. 7. Responses for Q7

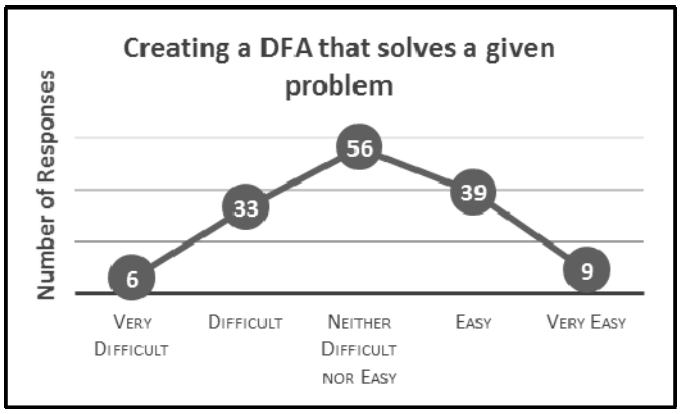


Fig. 8. Responses for Q8

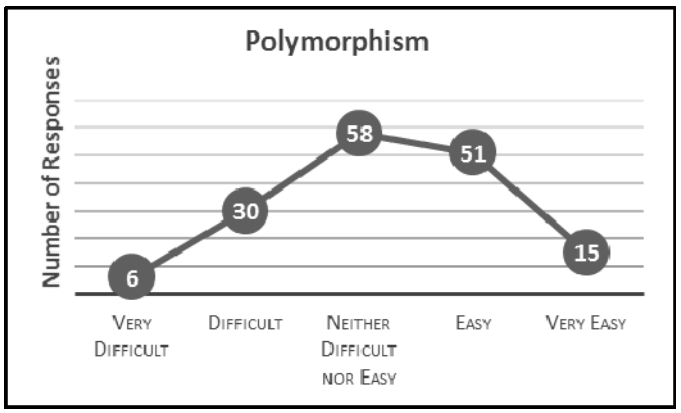


Fig. 9. Responses for Q9

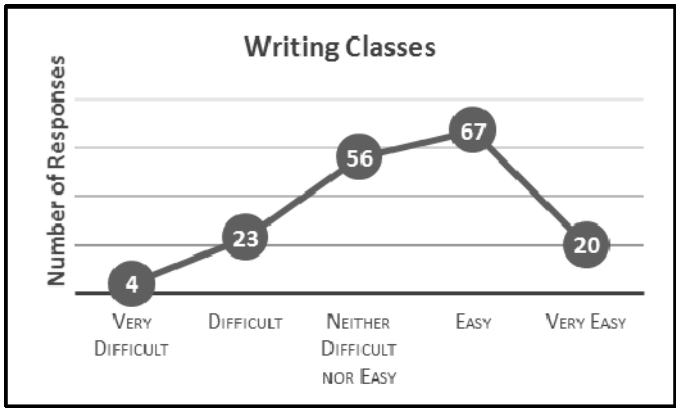
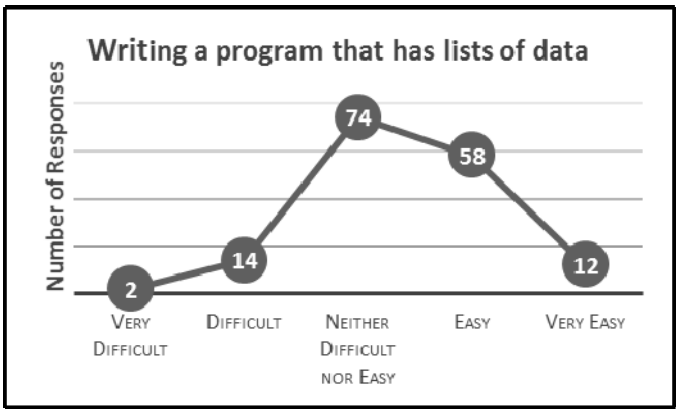


Fig. 10. Responses for Q10



2) Free Response Question 1

In analyzing the data for the first free response question: *Give an example of something you studied in your previous computer science courses (does not have to be from the list above) that you feel you still don't understand very well*, we first transcribed the items the students indicated on the survey forms. Responses that were sufficiently alike were combined as a category. There were 151 surveys that had a response to this question. Table 3 gives the responses/categories and the number of students who indicated the frequency column. The last row of the table gives an alphabetical listing of all responses with frequency of 1.

Students identified 96 unique topics for this question. At this stage, we were concerned with what effect, if any, the previous questions had as prompts to the answers for this question. Table 4 summarizes the analysis we did on the responses to code for the degree of difference between the prompts and the student responses. We can see that of the ten original topics, eight were repeated verbatim in the responses to this question, and there were several responses which we coded as "one-degree removed" from the original responses. To earn the designation of "one-degree removed", the student response either mentioned the original ten in some form or mentioned a term that is strongly associated with the original ten items. For example, "Accumulator in Racket" was considered a one-degree separation from "Writing Programs in Racket" because of the repeat of "Racket".

After this analysis, we have 68 topics that students identified as personal difficulties that are not the original ten items or one-degree removed from the original ten. There were 225 recorded responses and 136 were not repeated from the original ten or in the one degree removed category.

TABLE III. TOPICS IDENTIFIED BY STUDENTS FOR FREE RESPONSE Q1

Topic	Frequency
Proofs	27
Induction	14
Circuits/building circuits	11
Pointers/pointer arithmetic/dangling pointers	11
Search functions	8
Observer	6
Strong induction	6
Polymorphism	6
Big O	5
Binary trees, trees, binary search trees	5
Design Patterns	5
Recursion	5
Threading/concurrency	5
Iterator	4
DFAs	4
Racket	4
C	4
Iterable	3
Stacks	2
Graphs in Racket	2
Hashmaps	2
Observable	2
Data abstractions	2
Writing tests in Junit in Java	2
Writing programs in Java	2
Flip-flops	2
Tail recursion	2
Graph	2
Debugging	2
Iteration abstractions	2
Back-tracking	2
Numerical approximation and discretization	2
When to use certain data structures	2
[One response each] Abstract recursive functions; Abstractions; Accumulator in Racket; Advanced circuits; Arb-arity trees; Array vs list; Assembly; Back-tracking recursion; Balancing trees; Classes in Java; Code proofs; Compiler and virtual machine (Java vs C++); Composite Design Pattern; Entirety of XXXXXXX; Event driven programming; Exceptions; File I/O; Game solving; Generics; Graph traversal; Hash functions; How the classes/interfaces etc. relate to the program I'm making… I understand the concepts but have no experience putting it all together into a whole; How the fetch-decode-execute cycle works at the level of assembly language; How to improve efficiency; Indirect Proof; Inheritance; IntelliJ; JSON; JVM the process of reading code in Eclipse; Logic proof; Loops and switch statements in assembly; Making ALU from adders, multiplexers and flip-flops; Map structures; Memory alignments; Mutual exclusion; Network systems; NFAs; Nothing; Notify; OO Design; Parsing; Predicate logic proofs; Procedural abstractions; Proof by contradiction; Racket; Regular expressions; Robust classes; SAX Parser; Seeking the cache on virtual memory; Sets and functions; Some topics in AI; Structs; Synchronization; Technicalities of how things work; Typing in different computer languages; UI code; Use of APIs; Use of lambda in Racket; Using Comparable in Java for sorted sets; Virtual memory; Virtualization; Why use HashSet over TreeSet; Writing a problem that has lists of data; Writing tests	

TABLE IV. ANALYSIS OF RESPONSES RELATED TO PROMPTS

Topic	Repeated in List	One degree removed responses
Learning to write programs in Racket	Yes	Graphs in Racket, Accumulator in Racket, Use of lambda in Racket, Search methods in Racket
Learning to write programs in Java	Yes	Writing tests in Junit in Java, Classes in Java, Using comparable in Java for sorted sets
Variables	No	-
Functions	No	Search functions
Writing recursive functions	Yes	Tail recursion, Back-tracking recursion, Abstract recursive functions
Writing proofs	Yes	Induction, Strong induction, Code proofs, Predicate logic proofs, Proof by contradiction, Indirect proof, Logic proof
DFA	Yes	NFAs
Polymorphism	Yes	-
Classes	Yes	Classes in Java, Robust classes
Lists	Yes	-

IV. DISCUSSION

Looking at the Likert-type items, we see that the students identified topics at different levels of difficulty. The scoring on Q6 (*proofs*) tells us that students viewed this as harder than all of the other topics. It is also the case that using variables and writing functions were viewed by students as easier to understand than most of the other topics on the list. This confirms what others have found about topics that are difficult for students. This is further seen when looking at the graphs of the responses. Figures 3 and 4 show peak values on *Easy* and do not show a large number of responses in the *Difficult* or *Very Difficult* categories. This is a sharp contrast to Figure 6, which spikes at *Difficult* and shows very few responses in the *Easy* or *Very Easy* categories. For the rest of the graphs, we see shapes that are similar to normal distributions.

While initially concerned that the prompts may have influenced student response to the question: *Give an example of something you studied in your previous computer science courses (does not have to be from the list above) that you feel you still don't understand very well,* we now feel that the prompts possibly aided in recall, but did not bias it. There were 68 items given as responses to this question that were not in the original prompts or very closely related to the original prompts. Overall, this is positive for the instrument and shows us the breadth of topics that have been covered already by these students and how difficulties are in many cases, unique to the individual. While proofs and induction had a frequency of 41 (18% of responses), and are directly related to prompts, we also see that circuits and pointers each had a frequency of

11 (5% of responses), and are not directly related to the prompts. However, 80 of the 96 responses (83%) had a frequency of 3 or less, which means that 83% of the topics listed as difficult were found difficult by 1% or less of the respondents.

Further work is needed to determine if these results are typical and generalizable. From the limited data set we have, just one class from one institution, it is not clear that we can say for sure that any of these findings will hold for a more general population. If they did, the finding that there are so many different points of difficulty poses an interesting educational challenge for those instructors teaching this intermediate group. Because there isn't a single focus point of difficulty, or even a small set of high-frequency results, we are faced with the challenge of how to help these students overcome these difficulties when they are practically unique for each student.

It is important to note that what these results tell us is what the student's perceive as difficult. There is no measurement of performance, and further, a student's performance is not interesting in this context. A student could perform well on a task and still find it difficult. They could also perform poorly and find it easy. This data represented perceptions of students and points to topics they identified as difficult to them.

V. CONCLUSION

After undertaking a more thorough analysis of the data collected by a first day of class survey, we are left wondering what more we can learn from these students. The results from the data analysis largely confirm previous studies about what topics students find difficult. More work is needed to determine if these results are institution specific, or if students across institutions show similar results. Even within this data set we believe there is more we can learn. Given that this intermediate group is not very well studied, we would like to explore their answers to the free response questions on the survey in a more systematic way to see what it reveals to us about the nature of their learning of these difficult topics, and see how it aligns with the Likert survey data. Through understanding where students perceive difficulty, we can begin a more careful study of how learning occurs and where challenges lie for this middle group of learners of the discipline.

REFERENCES

[1] O. Astrachan, K. Bruce, E. Koffman, et al. "Resolved: objects early has failed" in *Proc of the 36th SIGCSE tech symp on Comp sci ed (SIGCSE '05)*. ACM, New York, NY, USA, 451-452, 2005.

[2] J. Bergin, E. Wallingford, M. Caspersen, et al. "Teaching polymorphism early" in *Proc of the 10th SIGCSE conf on Innov & tech in comp sci ed* (*ITiCSE '05)*. ACM, New York, USA, 342-343, 2005.

[3] S. Reges. "Back to basics in CS1 and CS2" in *Proc of the 37th SIGCSE tech symp on Comp sci ed (SIGCSE '06)*. ACM, New York, NY, USA, 293-297, 2006.

[4] Joint Task Force on Computing Curricula. (December 2013). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science* [Online]. Available: https://www.acm.org/education/CS2013-final-report.pdf

[5] A. Ehlert & C. Schulte. "Empirical comparison of objects-first and objects-later" in *Proc of the 5th int wkshp on Comp edu research (ICER '09)*. ACM, New York, NY, USA, 15-26, 2009.

[6] V. Ramalingam & S. Wiedenbeck. "An empirical study of novice program comprehension in the imperative and object-oriented styles" in *Papers presented at the seventh workshop on Empirical studies of programmers (ESP '97)* ACM, New York, NY, 124-139, 1997.

[7] I. Hadar & U. Leron, "How intuitive is object-oriented design?" *Commun. ACM* vol. 51, no. 5, 41-46, 2008.

[8] N. Liberman, C. Beeri, & Y.B. Kolikant, "Difficulties in Learning Inheritance and Polymorphism", *Trans. Comput. Educ.* vol. 11, no. 1, Article 4, 23 pages, 2011.

[9] edX. (2016). *Systematic Program Design* [Online]. Available: https://www.edx.org/course/systematic-program-design-part-1-core-ubcx-spd1x-0#!

[10] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, How to Design Programs. MIT Press, 2001.

[11] K. Fisler. "The recurring rainfall problem" in *Proc of the 10th conf on Int comp ed research (ICER '14)*. ACM, New York, 35-42, 2014.

[12] N. Ramsey. "On teaching *how to design programs*: observations from a newcomer" in *Proc of the 19th ACM SIGPLAN int conf on Functional prog (ICFP '14)*. ACM, New York, 153-166, 2014.

[13] B. Yu & K. Voll. "Probing student problem solving skills in mathematical induction using a scenario based think aloud protocol" in *Proc of the 16th Western Canadian Conf on Comp Ed (WCCCE '11)*. ACM, New York, NY, USA, 33-37.G, 2011.

[14] H. Kahney. "What do novice programmers know about recursion?" in E. Soloway & J.C. Spohrer, eds, *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1989.

[15] C.M. Kessler & J.R. Anderson. "Learning control flow: Recursive and iterative procedures", in E. Soloway & J.C. Spohrer, es, *Studying the Novice Programmer*. , Hillsdale, NJ: Lawrence Erlbaum Associates, 1989.

[16] T.L. Scholtz & I. Sanders. "Mental models of recursion: investigating students' understanding of recursion" in *Proc of ITiCSE '10*. ACM, New York, NY, USA, 103-107, 2010.

[17] R. McCauley, B. Hanks, S. Fitzgerald, et al. "Recursion vs. Iteration: An Empirical Study of Comprehension Revisited" in *Proc of SIGCSE '15*. ACM, New York, NY, USA, 350-355, 2015.

[18] D. Ginat. "Do senior CS students capitalize on recursion?" in *Proc of the 9th SIGCSE conf on Innov & tech in comp sci ed (ITiCSE '04)*. ACM, New York, NY, USA, 82-86, 2004.

[19] E. Lee, V. Shan, B. Beth, et al. "A structured approach to teaching recursion using cargo-bot" in *Proc of the 10thl conf on Int comp ed research (ICER '14)*. ACM, New York, NY, USA, 59-66, 2014.

[20] I. Sanders, V. Galpin, & T. Götschi. "Mental models of recursion revisited" in *Proc of the 11th SIGCSE conf on Innov & tech in comp sci ed (ITICSE '06)*. ACM, New York, NY, USA, 138-142, 2006.

[21] K. Brandt & M. Richey. "Studying mathematical induction and recursive programming together", *J. Comput. Sci. Coll.* vol. 19, no. 4, pp. 108-114, 2004.

[22] R.L.S. Drysdale. "Mathematical induction is a recursive technique" in *Proc of the 42nd ACM tech symp on Comp sci ed (SIGCSE '11)*. ACM, New York, NY, USA, 269-274, 2011.

[23] T. Hung & S.H. Rodger. "Increasing visualization and interaction in the automata theory course" in *Proc of the 31st SIGCSE tech symp on Comp sci ed (SIGCSE '00)*. ACM, New York, 6-10, 2000.

[24] A. Merceron. "Design patterns to support teaching of automata theory" in *Proc of the 14th ACM SIGCSE conf on Innov & tech in comp sci ed (ITiCSE '09)*. ACM, New York, NY, USA, 341-341, 2009.

[25] S.H. Rodger, A.O. Bilska, K.H. Leider, et al, "A collection of tools for making automata theory and formal languages come alive", *SIGCSE Bull.* vol. 29, no. 1, 15-19, 2001.

[26] R.M. Verma. “A visual and interactive automata theory course emphasizing breadth of automata” in *Proc of the 10th SIGCSE conf on Innov & tech in comp sci ed (ITiCSE '05)*. ACM, 325-329, 2005.

[27] K. Goldman, P. Gross, C. Heeren, et al. “Identifying important and difficult concepts in introductory computing courses using a delphi process” in *Proc of the 39th SIGCSE tech symp on Comp sci ed (SIGCSE '08)*. ACM, New York, NY, USA, 256-260, 2008.

[28] V.L. Almstrum, P.B. Henderson, V. Harvey, et al. “Concept inventories in computer science for the topic discrete mathematics” in *Working group reports on ITiCSE on Innov & tech in comp sci ed (ITiCSE-WGR '06)*. ACM, New York, NY, USA, 132-145, 2006.

[29] L.C. Kaczmarczyk, E.R. Petrick, J.P. East, et al. “Identifying student misconceptions of programming” in *Proc of the 41st ACM tech symp on Comp sci ed (SIGCSE '10)*. ACM, New York, 107-111, 2010.

[30] M. McCracken, V. Almstrum, D. Diaz, et al, “A multi-national, multi-institutional study of assessment of programming skills of first-year CS students”, *SIGCSE Bull.* vol. 33, no. 4, 125-180, 2001.

[31] R. Lister, E.S. Adams, S. Fitzgerald, et al. “A multi-national study of reading and tracing skills in novice programmers” in *Working group reports from ITiCSE on Innov & tech in comp sci ed (ITiCSE-WGR '04)*. ACM, New York, NY, USA, 119-150, 2004.

[32] N. Arshad. 2009. “Teaching programming and problem solving to CS2 students using think-alouds” in *Proc of the 40th ACM tech symp on Comp sci ed (SIGCSE '09)*. ACM, New York, USA, 372-376, 2009.

[33] M. Corney, S. Fitzgerald, B. Hankset al. 2014. 'explain in plain english' questions revisited: data structures problems. In *Proc of the* SIGCSE '14. ACM, New York, NY, USA, 591-596, 2014.

[34] D. Kopec, G. Yarmish, & P. Cheung, “A description and study of intermediate student programmer errors”, *SIGCSE Bull.* vol. 39, no. 2, 146-156, 2007.

[35] G. Yarmish & D. Kopec, “Revisiting novice programmer errors”, *SIGCSE Bull.* vol. 39, no. 2, 131-137, 2007.

[36] A. Shabo, M. Guzdial, & J. Stasko. “Computer science apprenticeship: creating support for intermediate computer science students” in *Proc of the 1996 int conf on Learning sci (ICLS '96)*. International Society of the Learning Sciences, 308-315, 1996.

[37] J. Lave & E. Wenger, Situated learning: Legitimate peripheral participation. Cambridge, UK: Cambridge University Press, 1991.

[38] P. Bourdieu, The logic of practice. Stanford, CA: Stanford University Press, 1992.

[39] E. Hutchins, Cognition in the wild. Cambridge, MA: MIT Press, 1995.

[40] D. J. Kennet, & K. Keefer, “Impact of learned resourcefulness and theories of intelligence on academic achievement of university students: An integrated approach”, *Educational Psychology,* vol. 26, no. 3, pp. 451-457, 2006.

[41] L. S. Blackwell, K. H. Trzesniewski, & C. S. Dweck, “Implicit Theories of Intelligence Predict Achievement Across an Adolescent Transition: A Longitudinal Study and an Intervention”, *Child Development*, vol. 78, pp. 246–263, 2007.