Towards a Common Instrument for Measuring Prior Programming Knowledge

Rodrigo Duran
Aalto University
Finland
rodrigo.duran@aalto.fi

Jan-Mikael Rybicki
Aalto University
Finland
jrybicki@cc.hut.fi

Arto Hellas
University of Helsinki
Finland
arto.hellas@cs.helsinki.fi

Sanna Suoranta
Aalto University
Finland
sanna.suoranta@aalto.fi

ABSTRACT

Computing education researchers and educators use a wide range of approaches for measuring students' prior knowledge in programming. Such measurement can help adapt the learning goals and assessment tools for groups of learners at different skills levels and backgrounds. There seems to be no consensus on if and how prior programming knowledge should be measured. Traditional background surveys are often ad-hoc or non-standard, which do not allow comparison of results between different course contexts, levels, and learner groups. Moreover, surveys may yield inaccurate information and may not be useful due to lack of detail. In contrast, tests can provide much higher detail and accuracy than surveys about student knowledge or skills, but large-scale tests are typically very time-consuming or impractical to arrange. To bridge the gap between ad-hoc surveys and standardized tests, we propose and evaluate a novel self-evaluation instrument for measuring prior programming knowledge in introductory programming courses. This instrument investigates in higher detail typical course concepts in programming education considering the different levels of proficiency. Based on a sample of two thousand introductory programming course students, our analysis shows that the instrument is internally consistent, correlates with traditional background information metrics and identifies students of varying programming backgrounds.

CCS CONCEPTS

• **Social and professional topics → Computer science education**; *Model curricula*; Student assessment.

KEYWORDS

Self-evaluation, assessment, CS1, introductory programming, prior experience

ACM Reference Format:
Rodrigo Duran, Jan-Mikael Rybicki, Arto Hellas, and Sanna Suoranta. 2019. Towards a Common Instrument for Measuring Prior Programming Knowledge. In *Innovation and Technology in Computer Science Education (ITiCSE '19), July 15–17, 2019, Aberdeen, Scotland Uk.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3304221.3319755

1 INTRODUCTION

With the increase of computer science and programming related initiatives that seek to make computing accessible to all[1], an increasing number of students have some background in programming when they enter tertiary education. Computing education researchers share a general agreement that prior programming experience influences student performance in introductory programming courses [13, 29, 31, 37, 38, 40], while certain teaching approaches may reduce the difference [36]. Information on prior programming experience could be used to provide more challenging options for more experienced students [5].

Prior programming experience has been measured using numerous approaches [11]. Students have been asked, for example, if they have any prior programming experience [2]; how many programming languages they have used [13]; and what is the largest program they have written [25]. Furthermore, a range of approaches exist where students are asked to read code, answer a set of questions, and/or to write short programs [26, 34]; however, these are often used as tests and can thus be learned, as answers may be predictable. Researchers have also suggested using data from students' problem-solving processes to automatically infer students' programming experience [20, 35].

To address these issues, this article introduces and evaluates an instrument for measuring students' prior programming knowledge. Drawing inspiration from general scales used in language assessment, such as The Common European Framework of Reference for Languages (CEFR) [8], in which students' knowledge of a particular language can range from beginner to mastery level, we constructed an instrument for assessing elementary programming knowledge. Using the instrument, students can evaluate their programming knowledge from beginner to mastery level in topics typically taught in introductory programming courses.

While Feigenspan et al. [11] presented a work similar to ours, their work focuses on more advanced topics and self-ratings of experience, whereas our instrument is targeted at concepts within introductory programming courses. Their results, however, support our notion that self-evaluation questions can be useful for measuring experience. With our instrument, students can observe their gradual knowledge increase, which also allows them better understand course objectives and view their own progress – that is, our instrument can be used multiple times during a course. The instrument offers teachers information on students prior and current knowledge, which is useful in course design, decision making, and in guiding students towards particular content areas. In addition,

[1]Such as Code.org, CSforALL, CS Principles Project, Hour of Code, Khan Academy, and MOOCs.

the instrument can, in principle, be used at all educational levels, including open education.

This article conducts an exploratory analysis focusing on student responses in our data collection with the following research questions:

RQ1 Is the instrument internally consistent?
RQ2 What types of knowledge patterns does the instrument identify?
RQ3 How does the instrument compare with traditionally used ad-hoc measures of programming background?

This article is organized as follows. Section 2 presents the related work regarding instruments of evaluation, such as questionnaires in CS and other contexts as well as validated tests in CS. Section 3 discusses how our instrument is constructed. Section 4 describes the course and data collection methods. Section 5 outlines the results of our work, which are further discussed in Section 6. Finally, Section 7 concludes the article and suggests future research directions.

2 RELATED WORK

Self-evaluation have been researched in various contexts, including health professions [12], classroom teaching [27], language teaching [8] and MOOCs [19]. Although students with little experience in self-assessment may over- or underestimate their abilities or knowledge, self-assessment can contribute to higher student achievement, particularly when students are guided in this process. For example, rubrics provide a criterion-based self-assessment, which helps to understand the goals and requirements of a course [27].

Since self-evaluation is a skill, research has shown that the validity, accuracy, and effectiveness of self-assessment can also be improved using qualitative formative feedback, and triangulation between self-, peer- and teacher assessment [12, 19, 27]. Self-evaluation can also function as a self-reflection tool, supporting self-regulation of learning [41].

In Europe, educational institutions across different levels are encouraged to adopt the Common European Framework of Reference (CEFR) [8] for assessing the language ability of learners. The CEFR framework includes both extended descriptions of skills levels allowing instructors to reach a common understanding of language skills as well as practical self-assessment grids for common reference levels in CEFR for languages [8, p. 26-27].

In the CEFR, the language skills levels range between A1 (beginner) and C2 (mastery). The skills are categorized into separate areas, such as listening, speaking, reading and writing skills. These grids are typically translated into different languages, but skills definitions and ranges remain the same or similar. This allows both students to estimate their own skills and teachers to gain a quick understanding of each student's approximate skills. When offering standardized language tests, such as TOEFL [10] and IELTS [1], the test results can be linked with the CEFR levels, allowing instructors to understand the overall skills levels of students in different skills areas.

Prior knowledge is essential to learning, and CS is no different; perhaps, the gaps between experienced and inexperienced learners are even more pronounced. Research shows that at the beginning of the course learners with some prior knowledge in CS outperform learners with no experience by a fair margin [14, 39], and prior knowledge can be used (with moderate effects) to reduce the gap between experienced/inexperienced learners by introducing intermediate level CS1 courses [18]. Even if at the end of the course this gap is mostly closed, students with no prior experience face a great amount of psychological stress and pressure to keep the pace of experienced students. If this gap is not diagnosed at early stages, learners can feel demotivated or even drop-out from courses [16].

Most current methods used to evaluate prior knowledge in CS are questionnaires. Usually, programming experience is treated as a single construct evaluated by metrics associated with experience in generic terms [11, 15, 17]. There is little consensus on what and how to ask participants and how representative is this information. Feigenspan et al. [11] presented one of the most complete, and closest to our study, investigations of the relationship between prior knowledge and performance, examining programming experience, familiarity with programming languages and paradigms, and educational background. Their findings show that these metrics correlate highly with performance on programming tasks, suggesting that students are able to self-evaluate reliably.

While these questionnaires use metrics that show interesting correlations, richer and robust methods used in personalized learning [2] demand a deeper understanding of how prior experience translates into the comprehension of concepts present in CS1 courses. Recent work towards more reasonable and granular assessment questions [9, 24] in CS1 exams points to the importance of knowing in detail the concepts learners are familiar with. Detailed information could be used to set more realistic expectations and design assessment instruments matching learner's ability (as captured by Vygotsky's *zone of proximal development* [28]), with the difficulty of concrete programs used in assessment instruments [9]. This kind of personalized learning approach can assist learners with less prior knowledge by directing them to activities that enhance their comprehension of a particular topic.

Computer Science Education recently produced validated CS tests, such as the FCS1 [34] and SCS1 [26], which could accurately and reliably provide a high degree of insight into learner's knowledge. However, these tests still have some limitations. They are not generally available since making them public could decrease their validity. Longitudinal studies with repeated measurements of the test could be affected by its limited pool of questions and low flexibility in arranging them. Tests are time-consuming and hard to scale to very large courses. Although the FCS1 and SCS1 introduce some agnosticism with the pseudocode used in tests, these cannot be considered language independent. Students with different backgrounds (e.g. visual blocks or functional programming language users) could be disproportionately under-evaluated. These tests can also be considered difficult and not suited as a pre-test.

3 CONSTRUCTION OF THE INSTRUMENT

Our instrument for self-evaluation (at https://goo.gl/nGR9Th) was designed to resemble the self-assessment grid for common reference levels in CEFR for languages [8, p. 26-27]. For this first version of the instrument, we chose to focus on code comprehension and reading skills. While we believe that in the future it is possible to design an instrument aimed at programming writing skills, at this initial stage of development reading/comprehension can be considered a less demanding cognitive skill [9, 21, 23]. To some extent, the hierarchical structure of developing skills and knowledge also resemble Bloom's taxonomy of educational objectives [3].

To fill the gaps in earlier approaches, our instrument aims to provide a fast, easy to apply and widely available way to evaluate prior knowledge in programming that covers much of the common introductory programming course content. This instrument can be used by anyone several times, including longitudinal evaluations within the same cohort without losing its validity. The instrument can be language agnostic since it does not use or depend on concrete code written in a particular language and paradigm. Although the instrument can use of support information (e.g. guidelines, examples in concrete code, etc) that can be bound to a specific language or paradigm, its framework is flexible enough to be easily adapted and refined, accommodating different aspects of programming at different levels of ability. The self-evaluation also can benefit students at the beginning of the course by making expectations and self-reflection evident [27].

The instrument has two dimensions: concepts and levels. The vertical axis contains the concepts related to programming in a CS1 course, based on the list of content areas provided by FCS1 [34]: *variables and assignment (var), input and output (io), expressions and arithmetic operators (exp), conditional statements (sel), loops and iteration (loops), data collections (lists), functions and methods (funcs) and classes and objects (objs).* Although FCS1 does not explicitly order the areas in increasing levels of complexity, our instrument uses an order that is consistent with concepts' order of complexity of a program written in an imperative paradigm [9]. However, the order of the concepts do not necessarily impose a hierarchy (they can be used in any order) and learners with different backgrounds and knowledge in these concepts will not be affected.

The horizontal axis contains the estimated proficiency levels of programming concepts, inspired by the CEFR scale of proficiency in language skills [8] adapted to programming contexts. The levels were designed using the principles of the stage of cognitive development in the programming context [9, 22, 32]. Every level matches the description of a stage of cognitive development, complexity levels or familiarity with the concept. We defined complexity in terms of the number and quality of interacting elements [4, 9, 30], learner's expertise level and the number of available higher-order schemas [9] and ability to summarize a program [6, 9, 33]. Overall, the levels describe the path the learner takes to be fully proficient in a concept: from total unfamiliarity with the concept, to familiarity with the concept drawn from other contexts, being able to recognize and comprehend the syntax of the concept in a concrete program, tracing a program using concrete values with meaningful naming conventions and finally explain a program using abstract values in plain English without meaningful conventions. Each level is assigned with a letter to define its stage and a number for a degree. Each level has a statement (in quotes, below) to be evaluated by the learner, who self-assigns to a level.

- *"I am unfamiliar with this concept"*: unfamiliar level (A0);
- *"In general, I know what this concept means"*: At the beginner level (A1), learners are able to relate to a concept without being able to identify it in the code. For example, a learner can relate to the concept of objects without being able to recognize it in the code or be able to work with it.
- *"In general, I can recognize the syntax used by the programming language to represent this concept"*: At the elementary level (A2), learners can recognize the syntax related to a concept in the code without being able to comprehend it as a whole or transfer it overall meaning to other contexts. For example, learners can recognize the term "class" in the code without understanding how it works or be able to manipulate its functionality.
- *"I can read and trace code that uses this concept with a few elements. I can correctly predict the output of the code using concrete values, and the code uses descriptive naming conventions"*: At the intermediate level (B1), for example, learners can understand that `"for cont in range of 10"` will repeat 10 times, or the function `"uppercase('ITiCSE')"` will return the capitalized input.
- *"I can read and trace code that uses this concept with several different elements. I can correctly predict the output of the code when this concept uses concrete values, and the code uses descriptive naming conventions"*: At the upper intermediate level (B2), learners can cope with code having multiple abutted loops with inner conditional structures, for example.
- *"I can mostly recognize patterns of using this concept in different types of code even if the naming conventions are not always standard or clear. I am able to comprehend the purpose and behavior of code that uses this concept receiving a range of distinct inputs"*: At the advanced level (C1), for example, when tracing the following code snippet

```
1 def b(x):
2     n = []
3     for e in x:
4         if (e >= 0):
5             n.append(e)
6     return n
```

  learners can, after careful consideration of elements in the code, recognize lower-level patterns, such as traversing a collection (line3), checking for negative numbers (line4), and adding elements to a list (line 5).
- *"I can easily recognize and explain the logic of code using this concept even if the naming conventions are not always standard or clear. I can generalize the purpose and behavior of code using this concept receiving a range of distinct inputs. I am able to summarize the purpose of the code using this concept in plain English"*: At the mastery level (C2), using the above code snippet as an example, learners can immediately summarize in their own words that the code represents a filter function which takes a list as input and outputs all non-negative numbers.

4 CONTEXT AND DATA

This study was conducted during an open online course in programming offered at the University of Helsinki. The course is offered free as a MOOC and covers the principles of programming in the Java language. The course is taken both by degree students and students not affiliated with the University of Helsinki. At the beginning of the course, students fill in a research consent form, our concepts instrument, a questionnaire with age, gender and metrics of their prior programming experience, similar to Feigenspan et al. [11]: elementary level school courses *(elemC)*, secondary level school courses *(secC)*, programming courses attended not at elementary or secondary levels (including online) *(otherC)*, hours programmed in total *(hpt)*, hours programmed per week *(hpw)*, programming languages *(langs)*, programming languages you consider yourself a beginner *(beginL)*, names of the programming languages you consider yourself a beginner, programming languages you consider yourself an advanced user *(advancL)*, names of the programming languages you consider yourself an advanced user, programming languages you consider yourself an expert *(expL)*, names of the programming languages you consider yourself an expert and largest program code *(lp)*.

The original instrument and questions were created in English and later translated to Finnish by an expert language instructor and an expert programming instructor. Answering the questionnaire is voluntary but includes an incentive raffle of movie tickets. Students could skip any question, and they could also choose to not to give research consent and still be included in the raffle.

From a total of 2853 course participants, 2468 answered the questionnaire[2]. From respondents, 2249 gave research consent. To curate the data, we set as exclusion criteria responses in which more than 2 variables were left unanswered in the concepts instrument, or if a value considered unreasonable (e.g. hundreds of programming courses taken) was inserted in more than two variables. If only one field fitted the exclusion criteria, a value `NA` was assigned to it. Open questions with free text were cleaned to remove typos, and the programming language names had the nomenclatures standardized. The final data set contains 2196 responses, of which 54.05% reported to be male, 43.85% female and 2.09% other. The average age of respondents is 36.93 years (ranging from 11 to 88 years old).

5 RESULTS

Data was collected using an online form and aggregated into a data set with all 23 variables (n = 2196), which were divided into three main groups: 8 variables in the concepts group, 2 in personal information group, and 13 in traditional background questions. Data was then analyzed using R Studio version 3.5.0.

Due to the nature of the course, respondents have heterogeneous backgrounds and levels of expertise. Regarding the programming languages, respondents reported knowledge in (at least) Java (n=561, 25.54%), Python (n=437, 19.89%), C/C++ (n=332, 15.11%), JavaScript (n=208, 9.47%), C# (n=137, 6.23%), Scala (n=36, 1.63%), Basic (n=35, 1.59%), Assembly (n=25, 1.13%), Lua (n=10, 0.45%), Arduino (n=6, 0.27%), and Scratch (n=3, 0.13%). To investigate if the data is normally distributed, we performed the Shapiro-Wilk test for all variables, which are not normally distributed. The concepts variables have W values equal 0.9, p-value < 0.001, except for objects with W=0.8. In background variables, W ranged from 0.3 to 0.02; thus, all tests account for such limitation. Table 1 shows basic descriptive statistics for all numeric variables from the concept instrument and background information.

To answer **RQ1**, we first investigated standard metrics of consistency. The inter-item (II) correlation was estimated using the `corr.test` function from the *psych* package using the Spearman rank method. Although the data is not normally distributed, the

[2]We consider a participant being in the course if they complete at least one programming assignment.

Table 1: Descriptive statistics for each variable in the study.

	var	io	exp	sel	loops	lists	funcs	objs
median	2.00	2.00	2.00	2.00	2.00	2.00	1.00	1.00
mean	2.36	2.27	2.33	2.41	2.26	2.03	1.89	1.53
std.dev	2.06	2.03	2.00	1.97	1.97	1.85	1.82	1.71

elemC	secC	otherC	hpt	hpw	lp	langs	beginL	advL	expL
0.00	0.00	0.00	10.00	0.00	5.00	5.00	1.00	0.00	0.00
0.15	0.33	1.51	511.80	3.32	18119.81	7.66	2.06	0.61	0.12
0.50	0.93	4.00	3290.97	35.16	539470.64	22.90	4.15	1.29	0.59

variances and covariances of variables are finite, making this approach appropriate. Cronbach's alpha (CA) was calculated using the `alpha` function from the *psych* package. Table 2 shows II and CA values of the concepts instrument.

Table 2: Internal consistency metrics.

	var	io	exp	sel	loops	lists	funcs	objs	Mean
II	0.86	0.87	0.87	0.88	0.88	0.85	0.86	0.82	0.86
CA	0.98	0.98	0.98	0.98	0.98	0.98	0.98	0.98	0.98

While II correlation and CA metrics provide a good overview of the internal consistency of the instrument, we also analyzed its construct validity. To investigate the theoretical latent factors in the data, a Factor Analysis (FA) was performed using a Principal Axis Factor with Varimax rotation. The Kaiser-Mayer-Olking measure (KMO=0.944) and Bartlett's test of Sphericity (p<0.001) suggested that the sample is factorable. After three iterations, the analysis yielded a two-factor solution, when loadings less than .30 were excluded, as presented in Figure 1. The first factor contains 10 items, including all variables in the concept instrument, accounting for 65% of the variance with factor loadings from .300 to .923. The second factor contains four items, accounting for 12% of the variance with factor loadings from .362 to .708. The first factor can be labeled as "concepts" and second as "prior knowledge".

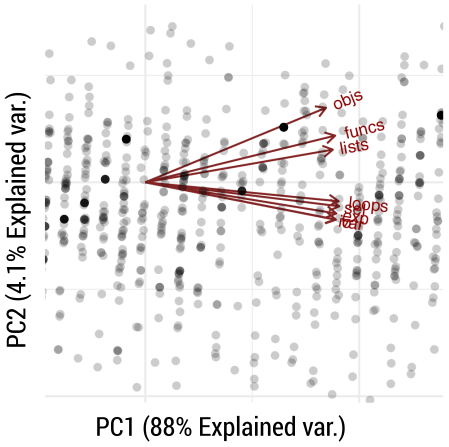
Figure 1: Factor Analysis.

		var	io	exp	sel	loops	lists	funcs	objs	otherC	hpt	advancL	expL	% of Variance
Factor	1	0.90	0.91	0.91	0.93	0.92	0.87	0.89	0.82	0.30		0.51		65.37
	2									0.53	0.71	0.36	0.62	11.95

The FA and the Principal Component Analysis (PCA) are techniques often applied in the construction of multi-scale tests to determine which items load on which scales [7]. While the FA tests a theoretical model of latent factors causing observed variables, the PCA reduces correlated observed variables to a smaller set of important independent composite variables. The PCA was performed using the `pcrcomp` function applied to all numeric variables in the data set. The first component (PC1) explains 44% of the variance, and the first nine components have a cumulative proportion of variance of 84%. The first component is composed by the concepts instrument, and the remaining components are a mix of background variables (e.g. elemC and advancL; age, expL, lp, hpt, hpw and otherC; beginL and secC).

Since the FA and PCA results suggest that the concepts instrument explains most of the variation in the data, we investigated if a set of components could emerge within the concepts instrument. The PCA of the concepts instrument shows a PC1 explaining 88% of the variance in the data, followed by a PC2 with 4.1% of the variance. Figure 2 shows the biplot of the components over the variables, indicating three distinct groups: *basic* (var, io, exp, sel, loops), *advanced* (lists, functions) and *expert concepts* (objects).

Figure 2: Component analysis of the concepts instrument. To improve clarity, the image is cropped to remove outliers.



Since the PCA and FCA suggest the existence of three distinct groups of concepts in the instrument, we investigated **RQ2** by analyzing patterns arising from responses. More specifically, since these groups suggest a hierarchy of concepts (basic, advanced and expert), we investigated if the patterns of responses supported such hierarchy. While it is not possible to generalize this hierarchy to every learner (e.g. Scala learners in a functional paradigm could show higher levels of proficiency in functions and objects than loops), we aimed to identify patterns that suggest a prevalence of groups identified in the FA and PCA analysis (perhaps more akin to traditional imperative paradigm instruction).

We filtered the original data set to remove all respondents with missing values in any concept (n = 2190) and evaluated for each respondent if the sequence of answers, analyzed by each concept (labeled as *single* in Table 3), could be classified as strictly increasingly monotonic, increasingly monotonic, strictly decreasingly monotonic, decreasingly monotonic, constant or other. We hypothesized that patterns of constant, strictly decreasingly monotonic and decreasingly monotonic responses could indicate a hierarchy of concepts (basic concepts with high scores, gradually decreasing to low scores in objects). Of the 538 respondents classified as constant, 269 (50%) evaluated themselves as A0 in all concepts, 98 (18.21%) as A1, 24 (4.46%) as A2, 36 (6.50%) as B1, 24 (4.46%) as B2, 31 (5.76%) as C1, and 56 (9.85%) as C2.

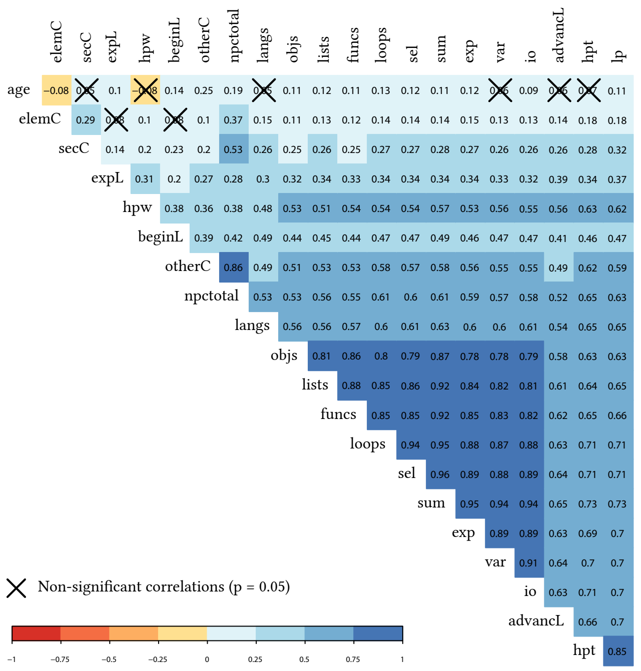
Given the high amount (56.34%) of non-classified responses, we investigated if the components described in the PCA could improve the classification methods. For each respondent, we calculated the mean of variables within each of the three groups (basic, advanced and expert) and observed if this sequence of three values followed an increasing or decreasing pattern. Respondents previously classified as constant were excluded from this analysis. In Table 3, the group column represents this second analysis. The results show a decrease in the number of non-classified responses (56.34% - 28.39%) and an increase in the ratio of responses that can be classified as part of a hierarchy (43.28% - 64.53%).

Table 3: Patterns in students responses, analyzed by concepts (single) or by groups of concepts (beginner, advanced, mastery). N = 2190.

	Single	Ratio	Groups	Ratio
Monotonically increasing	29	1.32%	119	7.2%
Strictly monotonically increasing	0		0	
Monotonically decreasing	389	43.28%	490	64.53%
Strictly monotonically decreasing	0		576	
Constant	538		538	
Other	1234	56.34%	469	28.39%

To answer **RQ3**, we calculated the correlations between the concepts instrument and traditional background variables. A correlation matrix was calculated for the overall data set. For correlations, we used the Spearman rank method in the `rcorr` function. Figure 3 shows the correlations among all variables, with non-significant correlations (at p=0.05) marked as crosses. All p-values were adjusted using the Bonferroni correction method. The strongest correlations are between concepts and the weakest with age.

Figure 3: Correlation matrix of the overall data, excluding names of PL variables.



	elemC	secC	expL	hpw	beginL	otherC	npctotal	langs	objs	lists	funcs	loops	sel	sum	exp	var	io	advancL	hpt	lp
age	−0.08	~~0.05~~	0.1	~~−0.08~~	0.14	0.25	0.19	~~0.05~~	0.11	0.12	0.11	0.13	0.12	0.11	0.12	~~0.06~~	0.09	~~0.06~~	~~0.07~~	0.11
elemC		0.29	~~0.08~~	0.1	~~0.08~~	0.1	0.37	0.15	0.11	0.13	0.12	0.14	0.14	0.14	0.15	0.13	0.13	0.14	0.18	0.18
secC			0.14	0.2	0.23	0.2	0.53	0.26	0.25	0.26	0.25	0.27	0.27	0.28	0.27	0.26	0.26	0.26	0.28	0.32
expL				0.31	0.2	0.27	0.28	0.3	0.32	0.34	0.33	0.34	0.34	0.34	0.34	0.33	0.32	0.39	0.34	0.37
hpw					0.38	0.36	0.38	0.48	0.53	0.51	0.54	0.54	0.54	0.57	0.53	0.56	0.55	0.56	0.63	0.62
beginL						0.39	0.42	0.49	0.44	0.45	0.44	0.47	0.47	0.49	0.46	0.47	0.47	0.41	0.46	0.47
otherC							0.86	0.49	0.51	0.53	0.53	0.58	0.57	0.58	0.56	0.55	0.55	0.49	0.62	0.59
npctotal								0.53	0.53	0.56	0.55	0.61	0.6	0.61	0.59	0.57	0.58	0.52	0.65	0.63
langs									0.56	0.56	0.57	0.6	0.61	0.63	0.6	0.6	0.61	0.54	0.65	0.65
objs										0.81	0.86	0.8	0.79	0.87	0.78	0.78	0.79	0.58	0.63	0.63
lists											0.88	0.85	0.86	0.92	0.84	0.82	0.81	0.61	0.64	0.65
funcs												0.85	0.85	0.92	0.85	0.83	0.82	0.62	0.65	0.66
loops													0.94	0.95	0.88	0.87	0.88	0.63	0.71	0.71
sel														0.96	0.89	0.88	0.89	0.64	0.71	0.71
sum															0.95	0.94	0.94	0.65	0.73	0.73
exp																0.89	0.89	0.63	0.69	0.7
var																	0.91	0.64	0.7	0.7
io																		0.63	0.71	0.7
advancL																			0.66	0.7
hpt																				0.85

6 DISCUSSION

It is still not usual to collect previous knowledge information of students in CS courses [5]. The results presented here could support a larger adoption of prior knowledge investigation methods by providing a richer and reliable set of instruments. Internal consistency analysis provides evidence to answer RQ1, showing that the instrument is highly internally consistent. PCA and FA analysis also show that the instrument is measuring similar but distinct components of knowledge. PCA also shows that within the instrument there is fair discrimination among three groups that align very well with established notions of basic, advanced and expert concepts (at least in an imperative programming paradigm).

In RQ2, we investigated patterns of responses in the instrument. More specifically, we wanted to examine if a hierarchy of concepts (assuming an imperative paradigm with a certain style of code composition) could be extracted from data. We were able to show that, overall, responses reflected patterns matching this hierarchy (a constant or decreasing pattern in scores), matching the FA and PCA analysis that yields three distinct groups of concepts. While these findings could be useful to instructional designers and supply additional information of students evaluations of certain topics, we are careful to make strong claims of its generalization. Table 3 shows that many responses could not be classified (56.34% and 28.39%). We speculate that some responses may be incoherent (e.g. responses where the var concept was marked as A0 but all other concepts marked as C2) and can be attributed to mistakes in filling the form. Some non-classified answers could reflect different learner backgrounds (e.g functional programmers rating functions and objects higher than loops). The instrument itself can be considered agnostic (programming language and paradigm), and its data allows the investigation of more diverse patterns.

To answer RQ3, we observe that traditional metrics still have a low to moderate correlation with each other and the instrument. Within the instrument, while all variables have a high correlation, objects can be regarded as a distinct element (also supported by the PCA). Given the diverse programming languages background of respondents and the high mean of their ages, we conjecture that, in general, learners were exposed to imperative paradigms with little (or at late stage) exposure to objects. Concerning the background variables, the largest code produced and total hours of experience still have high correlations with the instrument variables and background variables.

It is worth noting how little the courses at elementary and high school levels correlate with other variables. Given the age of some respondents, it is possible that a very low number of them were actually exposed to programming at these levels. Our data also indicates a very small number of students exposed to languages currently presented at ES or HS, such as Scratch or Arduino. A number of respondents reported a very high number of courses (up to 50 courses). It is likely that the emergence of micro-courses that may take just a few hours to complete biases these results. Interestingly, the number of programming languages in which learners consider themselves as advanced users have a strong correlation with the instrument. We conjecture that learners with these levels of exposure and ability have enough knowledge to self-evaluate more reliably.

In language education, these kinds of instruments have been adopted for administrative and educational purposes. In administrative contexts, universities may require that students need to prove a certain minimum language skills level in a foreign language before accepted to begin their studies, such as B2 or C1 level. The official estimation of language skills is provided by language professionals. However, students can use a CEFR grid to self-evaluate their skills and estimate whether they should participate in language courses to obtain required skills. In educational contexts, language courses can specify a prerequisite level that students should have before entering a course. This ensures students do not enroll in courses that exceed their skills level. In computing education, a widely adopted common framework for skills levels could offer similar benefits for learners, instructors and administrators.

As limitations of this work, we acknowledge that self-evaluation can suffer from poor reliability and accuracy. It is not clear how truthful students are in their responses, how well they comprehended statements in the instrument or how much effort they devoted to answering the questions. It is also possible that learners with distinct backgrounds have different standards and self-evaluate in different ways. Learners with high competency may underestimate and learners with low competency overestimate their knowledge.

7 CONCLUSION

This article has introduced and evaluated an instrument for language independent evaluation of prior programming knowledge for introductory programming courses. The instrument was inspired by language learning self-evaluation instruments, such as CEFR. Unlike traditional approaches and instruments measuring prior programming knowledge, our instrument focuses on core areas relevant to learning programming. The instrument can provide a teacher with an overview of students' prior knowledge on core areas and gives students an opportunity to evaluate and reflect on their current level of knowledge.

Our analysis with 2196 responses suggests that the instrument is internally consistent. Principal component analysis of the responses identifies three main components: (1) basic programming concepts, (2) advanced concepts (or perhaps, functional programming concepts?), and (3) object-oriented programming concepts. Metrics used by previous studies are still relevant and correlate well with the instrument.

Our future work aims to analyze how responding to the instrument influences student behavior in introductory programming courses and how these responses correlate with other instruments for assessing programming competency, such as course grades and the SCS1 [26]. We plan to investigate if this instrument allows the detection of distinct patterns in students answers, which could be matched to backgrounds in different paradigms and programming languages, yielding distinct hierarchies. Finally, we plan to investigate pertinent issues with self-reporting, focusing on how students' self-reported knowledge evolves as they proceed in introductory programming courses. We will refine the instrument by adding detailed descriptions and examples for each concept and each level, creating statements more contextualized. We will continue to extend the instrument to include measures of code writing ability in the future.

REFERENCES

[1] [n. d.]. The International English Language Testing System (IELTS). https://www.ielts.org

[2] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. 2015. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the eleventh annual International Conference on International Computing Education Research.* ACM, 121–130.

[3] Lorin W Anderson, David R Krathwohl, Peter W. Airasian, Kathleen A. Cruikshank, Richard E. Mayer, Paul R. Pintrich, James Raths, and Merlin C. Wittrock. 2001. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives* (abridged ed.). Addison Wesley Longman. 302 pages.

[4] Jens F Beckmann. 2010. Taming a beast of burden–On some issues with the conceptualisation and operationalisation of cognitive load. *Learning and instruction* 20, 3 (2010), 250–264.

[5] Janet Carter, Su White, Karen Fraser, Stanislav Kurkovsky, Colette McCreesh, and Malcolm Wieck. 2010. ITiCSE 2010 working group report motivating our top students. In *Proceedings of the 2010 ITiCSE working group reports.* ACM, 29–47.

[6] Tony Clear, Anne Philpott, Phil Robbins, and Simon. 2009. Report on the Eighth BRACElet Workshop: BRACElet Technical Report 01/08. *Bulletin of Applied Computing and Information Technology* 7, 1 (2009).

[7] Andrew L Comrey. 1988. Factor-analytic methods of scale development in personality and clinical psychology. *Journal of consulting and clinical psychology* 56, 5 (1988), 754.

[8] Council of Europe. 2001. *The Common European Framework of Reference for Languages: Learning, teaching, assessment.* Strasbourg. https://rm.coe.int/1680459f97

[9] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an Analysis of Program Complexity From a Cognitive Perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research.* ACM, 21–30.

[10] Educational Testing Service (ETS). [n. d.]. The TOEFL®Test. https://www.ets.org/toefl

[11] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on.* IEEE, 73–82.

[12] Michael J Gordon. 1991. A review of the validity and accuracy of self assessment in health professions training. *Academic medicine* 66, 12 (1991), 762–769. http://dx.doi.org/10.1097/00001888-199112000-00012

[13] Dianne Hagan and Selby Markham. 2000. Does It Help to Have Some Programming Experience Before Beginning a Computing Degree Program?. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSEconference on Innovation and Technology in Computer Science Education (ITiCSE '00).* ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/343048.343063

[14] Edward Holden and Elissa Weeden. 2003. The impact of prior experience in an information technology programming course sequence. In *Proceedings of the 4th conference on Information technology curriculum.* ACM, 41–46.

[15] Edward Holden and Elissa Weeden. 2004. The experience factor in early programming education. In *Proceedings of the 5th conference on Information technology education.* ACM, 211–218.

[16] Päivi Kinnunen and Lauri Malmi. 2006. Why students drop out CS1 course?. In *Proceedings of the second international workshop on Computing education research.* ACM, 97–108.

[17] Päivi Kinnunen, Maija Marttila-Kontio, and Erkki Pesonen. 2013. Getting to know computer science freshmen. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research.* ACM, 59–66.

[18] Michael S Kirkpatrick and Chris Mayfield. 2017. Evaluating an Alternative CS1 for Students with Prior Programming Experience. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* ACM, 333–338.

[19] Chinmay Kulkarni, Koh Pang Wei, Huy Le, Daniel Chia, Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R. Klemmer. 2013. Peer and self assessment in massive online classes. *ACM Transactions on Computer-Human Interaction* 20, 6 (2013), 1–31. https://doi.org/10.1145/2505057

[20] Juho Leinonen, Krista Longi, Arto Klami, and Arto Vihavainen. 2016. Automatic Inference of Programming Performance and Experience from Typing Patterns. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16).* ACM, New York, NY, USA, 132–137. https://doi.org/10.1145/2839509.2844612

[21] Raymond Lister. 2011. Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. *Proceedings of the Thirteenth Australasian Computing Education Conference* Ace (2011), 9–18.

[22] Raymond Lister. 2016. Toward a Developmental Epistemology of Computer Programming. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education.* ACM, 5–16.

[23] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research.* ACM, 101–112.

[24] Andrew Luxton-Reilly, Brett A Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Jacqueline Whalley, et al. 2018. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports.* ACM, 47–69.

[25] Matthias M Müller. 2004. Are reviews an alternative to pair programming? *Empirical Software Engineering* 9, 4 (2004), 335–351.

[26] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16).* ACM, New York, NY, USA, 93–101. https://doi.org/10.1145/2960310.2960316

[27] John A. Ross. 2006. The Reliability , Validity , and Utility of Self-Assessment. *Practical assessment, research and evaluation* 10 (2006), 1–13. https://doi.org/10.1016/j.aspen.2014.06.014

[28] Wolfgang Schnotz and Christian Kürschner. 2007. A reconsideration of cognitive load theory. *Educational psychology review* 19, 4 (2007), 469–508.

[29] Judy Sheard, Angela Carbone, Selby Markham, A J Hurst, Des Casey, and Chris Avram. 2008. Performance and Progression of First Year ICT Students. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78 (ACE '08).* Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 119–127. http://dl.acm.org/citation.cfm?id=1379249.1379261

[30] John Sweller. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational psychology review* 22, 2 (2010), 123–138.

[31] Harriet G Taylor and Luegina C Mounfield. 1994. Exploration of the relationship between prior computing experience and gender on success in college computer science. *Journal of educational computing research* 11, 4 (1994), 291–306.

[32] Donna Teague. 2015. *Neo-Piagetian theory and the novice programmer.* Ph.D. Dissertation. Queensland University of Technology.

[33] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136.* Australian Computer Society, Inc., 87–95.

[34] Allison Elliott Tew and Mark Guzdial. 2010. Developing a Validated Assessment of Fundamental CS1 Concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10).* ACM, New York, NY, USA, 97–101. https://doi.org/10.1145/1734263.1734297

[35] Richard C. Thomas, Amela Karahasanovic, and Gregor E. Kennedy. 2005. An Investigation into Keystroke Latency Metrics As an Indicator of Programming Performance. In *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42 (ACE '05).* Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 127–134. http://dl.acm.org/citation.cfm?id=1082424.1082440

[36] Phil Ventura and Bina Ramamurthy. 2004. Wanted: CS1 Students. No Experience Required. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04).* ACM, New York, NY, USA, 240–244. https://doi.org/10.1145/971300.971387

[37] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. 2014. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the tenth annual conference on International computing education research.* ACM, 19–26.

[38] Susan Wiedenbeck, Deborah Labelle, and Vennila NR Kain. 2004. Factors affecting course outcomes in introductory programming. In *16th Annual Workshop of the Psychology of Programming Interest Group.*

[39] Chris Wilcox and Albert Lionelle. 2018. Quantifying the Benefits of Prior Programming Experience in an Introductory Computer Science Course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* A, 80–85.

[40] Brenda Cantwell Wilson and Sharon Shrock. 2001. Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01).* ACM, New York, NY, USA, 184–188. https://doi.org/10.1145/364447.364581

[41] Barry J Zimmerman. 2002. Becoming a Self-Regulated Learner: An Overview. *Theory into practice* 41, 2 (2002), 64–70. https://doi.org/10.1207/s15430421tip4102