Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories

Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories

Cruz Izu*
The University of Adelaide
Adelaide, Australia
cruz.izu@adelaide.edu.au

Carsten Schulte*
Paderborn University
Paderborn, Germany
carsten.schulte@uni-paderborn.de

Ashish Aggarwal
University of Florida
Florida, USA
ashishjuit@ufl.edu

Quintin Cutts
University of Glasgow
Glasgow, UK
quintin.cutts@glasgow.ac.uk

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

Mirela Gutica
British Columbia Institute of Technology
Burnaby, Canada
mirela_gutica@bcit.ca

Birte Heinemann
Paderborn University
Paderborn, Germany
birte.heinemann@uni-paderborn.de

Eileen Kraemer
Clemson University
Clemson, SC, USA
etkraem@clemson.edu

Violetta Lonati
University of Milan
Milan, Italy
lonati@di.unimi.it

Claudio Mirolo
University of Udine
Udine, Italy
claudio.mirolo@uniud.it

Renske Weeda
Radboud University
Nijmegen, Netherlands
renske.smetsers@science.ru.nl

ABSTRACT

This working group asserts that Program Comprehension (*ProgComp*) plays a critical part in the process of writing programs. For example, this paper is written from a basic draft that was edited and revised until it clearly presented our idea. Similarly, a program is written in an incremental manner, with each step tested, debugged and extended until the program achieves its goal. Novice programmers should develop program comprehension skills as they learn to code so that they are able both to read and reason about code created by others, and to reflect on their own code when writing, debugging or extending it. To foster such competencies our group identified two main goals: (g1) to collect and define learning activities that explicitly address key components of program comprehension and (g2) to define possible learning trajectories that will guide teachers as they select and sequence those learning activities in their CS0/CS1/CS2 or K-12 courses.

Both goals were achieved as described in this report: after a thorough literature review, a detailed description of the Block Model is provided, as this model has been used with a dual purpose (p1) to classify and present a comprehensive list of ProgComp tasks and (p2) to define a possible learning trajectory for a complex task, covering different cells of the Block Model matrix. The latter will help instructors to decompose complex tasks and identify which aspects of ProgComp are being fostered.

*Leads

ITiCSE '19, July 15–17, 2019, Aberdeen , UK – Published article:
https://doi.org/10.1145/3344429.3372501

KEYWORDS

program comprehension; learning trajectories; CS1; novice programmers; K-12 computing

ACM Reference Format:
Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. 2019. Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories . In *ITiCSE '19: The 24th ACM Annual Conference on Innovation and Technology in Computing Science Education, July 15–17, 2019, Aberdeen , UK*. ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3344429.3372501

1 INTRODUCTION

> "... The aim of life is not to change the world but to understand it."
>
> Youth with Split Apple, Kenny Hunter, 2005[1]

The quote above highlights the split between changing/creating and understanding. This same split is also seen in debates about teaching and learning programming, e.g. [27, 81]. Learning to program is not only about mastering the syntax and semantics of each construct of a programming language. From the outset, Soloway identified two key issues on learning to program [97]: the ability to identify a *plan*, a stereotypical solution to a programming problem, and an understanding of how "the computer turns a static program written on a piece of paper into a dynamic entity that exists over time", in which the causal relationships between statements are important for understanding and describing how the program works. The behavior of the machine is often too complex for students to

[1] Inscription on a sculpture in front of the Working Group building in Aberdeen.

comprehend, beginners in particular. Instead, instructors present a pedagogically designed simplification of the machine behavior to students, a model described as the Notional Machine [27].

Multiple models have been proposed to analyze program comprehension in terms of types of information implied [73], mental representations [115], cognitive demand [30], or as a hypothesis-driven process [109]. The 2010 ITiCSE Working Group report [91] compared and contrasted how those models conceptualize program comprehension. Although this comparison was the main focus of the report, it also provided some insights into learning concepts and obstacles, effective learning tasks and teaching methods. Thus, this Working Group continues to explore and support the teaching insights given in [91] by collecting and categorizing suitable learning tasks for Program Comprehension (*ProgComp* for short).

The focus on how different tasks develop the thinking process of learners and how tasks should be ordered to support effective learning progressions is what distinguishes our Working Group (WG) from other approaches that have collected useful examples and tasks, e.g. [12, 58, 87]. In other words, we have focused not on how to assess program comprehension, but on how to foster it.

In order to achieve these goals, we followed a five step plan:

Step 1 – Review the current state of research and development by analyzing literature on proposed activities addressing *ProgComp*.

Step 2 – Concurrently, interview instructors at various institutions on their classroom activities to foster *ProgComp*.

Step 3 – Use the outputs from the literature review and instructors' interviews to define and conceptualize what is meant by *ProgComp* in the context of novice programmers.

Step 4 – Catalog learning activities with regard to their prerequisites, intended learning outcomes and additional special characteristics.

Step 5 – Develop a map of learning activities and thereby also models of probable learning trajectories.

Regarding step 2, we wanted to learn from educators which elements and kinds of mental representation they seek to convey when teaching introductory programming classes. We also asked them to report any abstract or concrete (e.g., code) examples/activities/exercises they use for teaching these representations and what challenges students usually encounter when comprehending programs. Therefore, we are building on what Lobato and Walters have named the *hypothetical learning trajectory*, the trajectory as seen through the eyes of the instructor [55, p. 84]:

> The starting point in teacher planning is the creation of conjectures regarding what students understand initially and what they may be able to learn next. Instructional tasks are selected, not only on the basis of generic task features, such as high cognitive demand or student interest, but also because of an inferred quality of being able to engender the next level of sophistication of student thinking.

As we implemented the plan, it became clear that a framework for classification was needed in step 4. The WG found out such a framework was also useful to analyze an individual task concerning its prerequisites and thereby allowing to design learning trajectories that can be customised to adapt to learners' prior knowledge.

The rest of the report is organised as follows. In the first part of this paper, sections 2 and 3, we revisit previous works related to *ProgComp* and its theoretical foundations. Section 2 provides a definition of comprehension that ties concrete activities fostering *ProgComp* to the goals we expect students to achieve. It also situates comprehension as part of programming knowledge, and describes how program comprehension is assessed and taught. Section 3 presents the Block Model [90], which is our chosen framework to reason about *ProgComp*. We also describe tasks to foster *ProgComp* found in prior work, using the Block Model to analyse and classify those sample tasks.

The second part of this paper has 3 sections, each investigating and reporting work done by the WG covering different aspects of *ProgComp*. In section 4 we present perspectives on program comprehension from our practitioners' points of view. In section 5 we provide an extensive list of *ProgComp* tasks, classified according to the dimensions of the Block Model. In section 6 we discuss how the Block Model can be used to support the design of a learning trajectory of instructional materials that promote *ProgComp*. For each investigation taken, the methodology is described at the start of its section. Finally, in section 7 we present our conclusions and discuss opportunities for future work.

2 BACKGROUND AND RELATED WORK

This section provides our definition of program comprehension inspired by the findings of the literature review. We have grouped them into four broad categories: the characterization of programming knowledge, the identification of some useful coding abstractions, the assessment of novices' understanding of programs, and the learning of code reading skills.

2.1 Program Comprehension Overview

Up to now, as a matter of fact, teaching to program has mostly been approached as a "code writing" activity, the main goal being to develop programs. By contrast, "code reading" is sparingly taught explicitly, although instructors need to provide guidelines for interpreting code when presenting new language constructs or problem solutions to their students. Program comprehension's focus is precisely on this latter type of approach. Let's start by providing our WG definition of two terms:

Program Comprehension (*ProgComp*) – it is usually conceptualized as a process in which an individual constructs his or her own mental model of a program.

***ProgComp* task –** in such task the learner encounters an artifact that represents the program. The task asks the learner to engage with the artifact in some way. Through this interaction with the artifact, the learner is stimulated to elaborate on and refine their mental model.

The artifact is typically source code, but might also be another form of specification such as the nodes in a Parsons problems [72] or a collection of blocks in a blocks-based programming language such as Scratch [59].

The model is expected to include features such as the elements and structure of the program (starting from the basic coding constructs), the execution behavior, and the purpose of the program and its blocks, both from the programmer's perspective and in the

domain context. In the process, the learner is required to also retrieve and put into action their prior knowledge about programming and/or the problem domain, and possibly consider other sources of knowledge, e.g. the programming language documentation or a verbal explanation of the program's purpose. As the learner interacts with the program, she may choose to create external representations such as notes, traces, sketches or diagrams to help overcome the limitations of working memory [20, 117] and further support the development of the mental model and the accomplishment of the program comprehension task.

Although what pertains to *ProgComp* is the ability to read, interpret and explain code, we can identify a range of code editing activities that combine reading and writing, such as debugging, refactoring, or extending the functionality of existing code. All of these are concrete programming tasks with a clear code comprehension phase that informs the changes of code written. Without comprehension, it is nearly impossible to debug or extend code by trial and error. An advantage of this type of tasks is that, by their applied nature, they are more motivating to students and can help them realize how important is comprehension for the customary writing and editing processes. Thus, we will consider *ProgComp* tasks in this wider context, with focus on the comprehension facet — be it the final goal or an *explicit* subgoal of the task at hand.

Typical *ProgComp* tasks exist along a continuum of engagement from *explaining* tasks to *annotation* tasks to *modification* tasks, all forms of active learning [36]. In explaining or articulating tasks the learner reads the code and then explains to themselves or a partner what they think the code (or parts of the code) is doing. In annotation tasks, students might be asked to add comments or highlights or to create secondary external representations such as trace tables or sketches. Modification tasks, on the other hand, are about adjusting or reworking the original code to, for example, correct a bug, make the program more readable or add a feature.

2.2 Characterisation of Programming Knowledge

A number of seminal theoretical works have attempted to categorise the programming *knowledge* implied in *ProgComp*, in particular:

- Linn's *chain of cognitive accomplishments* to learn programming [50]; besides code writing and problem solving skills, it covers the precursor stages that concern the learning of language features and the development of a repertoire of templates, i.e. "stereotypic patterns of code using more than a single language feature [...] employed as an entity in programs to perform commonly encountered tasks".
- Rogalski & Samurçay's general framework for knowledge representation in the programming field [82], where the difficulties faced by novices are subdivided into four areas: conceptual representations about the computer device, control structures that disrupt the linearity of program text, variables, data structures and data representation, and programming methods, i.e. aids for identifying suitable strategies to solve problems of a given class.
- McGill & Volet's conceptual framework for analyzing students' knowledge of programming [63], which "integrates three distinct types of programming knowledge identified in the Computing Education literature (*syntactic*, *conceptual*, and *strategic*) with three distinct forms of knowledge proposed in the cognitive psychology literature (*declarative*, *procedural*, and *conditional*)". The resulting knowledge categories are: (i) *declarative-syntactic*, representing knowledge of syntactic facts for a specific programming language; (ii) *declarative-conceptual*, relative to the understanding of the notional machine; (iii) *procedural-syntactic*, referring to the ability to produce syntactically correct code; (iv) *procedural-conceptual*, concerning the ability to write programs. (v) *strategic/conditional* ("conditional" stands for knowing when, in which conditions to use a given strategy), addressing the ability to design, code, and test a program that solves a novel problem.

More recently, following a similar line of research, Xie et al. [116] drew on prior work to address distinct programming skills that, according to the authors, "prior theories do not translate to concrete instruction that supports" their development in novices. Their theoretical framework distinguishes among four subsequent learning steps focused on: (i) knowledge of the operational semantics, demonstrated by being able to trace code; (ii) knowledge of the syntactic structures demonstrated via translation of accurate description in a natural language into syntax that "compile and execute as expected"; (iii) learning of reusable abstractions, or program "templates", demonstrated by the ability to identify the components of such abstractions as well as their purpose in a given program; (iv) problem-solving skills, demonstrated by being able to apply and/or combine program templates to solve the problem at hand.

2.3 Use of Abstraction in Program Comprehension

When expert programmers read code, they use abstraction or chunking to identify key components. In this section we will summarise a range of studies covering two significant abstractions in this respect: *plans* and *variable roles*.

Soloway uses the term *plan* to describe 'chunks' of knowledge incorporated in a 'canned'-solution [97]. He related the role of plans in reading computer programs to what cognitive psychologists refer to as *schemata* as units of mental organization. Robins et al. [81] explicitly link "structured chunk of related knowledge" to *schema* and *plan*. Rist [80] describes a plan as "the basic cognitive chunk used in program design and understanding". Brook's work [6, 7] showed that students who are able to recognize 'beacons' which identify plans, can reason about programs at a higher level.

De Raadt describes a set of elementary plans (programming strategies) that can be combined into a programming solution [22]. Examples of elementary plans are Initialization (variables), Average, Triangular Swap, Counter-Controlled Loop, Sum, and Count plans. He explains that the identification, selection, and application of plans can be seen as a representation for strategy (similarly described in [101], also known as patterns [110]) and should be taught and assessed explicitly [116]. Plans can be abutted (or concatenated), nested and merged — thereby their mastery is a high-level learning outcome. Similarly, plans can be decomposed into smaller units, or sub-plans, and used to distinguish pre-requisites.

Just as algorithmic thinking skills are needed to link plans together to create an adequate solution, programmers need to be able to identify and analyze plans and their parts to comprehend programs as a whole. Experts are good at recognizing, using and adapting plans, and as a result are faster, more accurate and employ effective strategies [81].

Spohrer and Soloway [100] analyzed difficulties related to plan-composition problems, and argue that these have a larger impact on programming success than language construct knowledge. Soloway's work [97] identified the need to teach plans as abstractions. Rist [79] explored this approach and found that when students know an appropriate high-level schema for solving a problem, they can reason adequately from plan to code.

More recent work by De Raadt et al. [23] has shown that explicitly teaching plans improves the learning outcomes. However, Rist also emphasizes that novice programmers are often unable to translate high-level plans into concrete program statements.

Even when students master those plans independently, constructing a working program where those plans need to be composed is not a trivial task. Fisler et al. [35] uses the Rainfall Problem as a benchmark to show that certain paradigms (e.g. functional), languages or methods of instruction could make this composition process more efficient. For readers not familiar with plans, refer to appendix A which presents an example of plan composition.

Although much work, especially on the ability to combine plans, focuses on program *construction*, we can think of it as providing lenses to make sense of a program's structure, in accordance with our *ProgComp* perspective. Learning about (combining) plans could be facilitated by focusing on program comprehension rather than composition. As such, novices could be required to recognise how these strategies are applied to create solutions to solve more complex tasks. In fact, Merriënboer [64] explored such an approach in a high school context with promising results.

Variables and the operations on them can be seen as beacons to identify a particular design pattern (see also [1, 31, 68]), or being used directly as having significant *roles* [11, 47] that help to abstract from the code to its goal. Only ten roles are needed to cover 99% of all variables at novice level. For example, a Sum plan can be seen as a program with a *stepper* variable (the control variable of the for loop) and a *gatherer* variable to accumulate the sum.

Sajaniemi et al [84] compared the impact emphasizing variable roles while teaching has on *ProgComp* relative to a group taught the traditional way. Both groups showed similar performance on a program prediction task and a program construction task, but the students that were regularly exposed to variable roles outperformed the others in the program comprehension task that asked them to describe the purpose of the program and how it worked.

In summary, recognizing beacons or variable roles, identifying plans, and understanding how they are used and combined are all an important part of program comprehension.

2.4 Assessment of Program Comprehension

Starting from pioneering studies by Mayer, Soloway, Spohrer, du Boulay and others [27, 60, 97, 98, 101] since the early '80s, much research work has been focused on the assessment of different aspects of novices' program comprehension.

Since then, several educators appear to agree about novices' difficulties to thinking at a "relational" level [54, 95]. In this respect, the ability to summarise the purpose of a program in a sentence (to "explain in plain English") has been investigated by Lister et al. [54], and later linked to the ability to write code [69]. Furthermore, the structure and goal of a program may have a significant impact on how students understand it. Duran et al. [30], in particular, investigated how different program structures and plan-composition strategies could lead to students' distinct perceptions of difficulty. Comprehending what makes a program complex and how it could be transformed or broken into smaller pieces to reduce complexity is an important ability that instructional designers should put into practice.

In general, novices' understanding of programs has been explored from a variety of perspectives, such as: to investigate the extent to which tracing, reading and writing skills correlate with each other [56]; to interpret students' ways of classifying code fragments based on perceived similarities and differences [105]; to categorise novices' mental models of the notional machine underlying imperative [5] and recursive [86] computations; to compare students' mastery of recursion vs. iteration [66]; to assess the understanding of conditionals, loops and nested loops [13, 42]; to analyse the relations between students' performance and their annotations in the exam papers [61]; to compare block versus textual representations of programs [111].

2.5 Fostering Program Comprehension

Most of the contributions cited above suggest interesting tasks in which students may engage. However, such tasks are usually meant as tools the instructor could use to evaluate students' achievements, whereas hardly any insight is provided as to how to attain learning progress in case of poor performance.

Early work by Deimel [25] provided relevant (but rarely cited) guidelines on teaching program reading as follows:

> Students should be encouraged to view programs at different levels of abstraction and in different frames of reference. We can show how a statement or group of statements may be understood in terms of the effect on particular variables, in terms of a change in a data structure, in terms of effecting part of an algorithm, or in terms of the problem which the program is supposed to solve. (To reinforce these ideas, we should assign exercises in which the students must interpret code at different levels ...) We must explain that programs may be read top-down or bottom-up, depending on one's reading objective, the program structure, and the nature of the comments.

However, only in the last ten years has there been a rising awareness and focus on program comprehension as part of *learning* to program. Researchers have proposed assessments that include or target aspects of program comprehension such as reading [10], tracing [70], explaining [69], or reversing changes to program state [41, 103]. Recurrent practice with similar *ProgComp* tasks should help with improved retention and transferability of the students' newly elaborated mental models to future program comprehension tasks [43].

More to the point, Sudol et al. [102] vindicated using code comprehension questions as learning events rather than as assessment items. In a similar vein, Shargabi et al. [94] selected 14 program comprehension tasks and surveyed practitioners to rank the tasks in terms of perceived effectiveness in developing novices' program comprehension.

In addition, recent pedagogical approaches propose to design courses addressing comprehension first:

- PRIMM (Predict-Run-Investigate-Modify-Make) [93], whose aim is helping teachers to organise programming lessons in which pairs of students are guided through reading and adjusting code prior to writing own code. In fact, PRIMM approach follows pattern not dissimilar to the one proposed by Deimel [25], which has 4 phases: run and investigate, read and trace, modify and extend, and finally write or make.
- PLTutor [70] models language execution and makes visible the causal relationship between syntax and machine behavior. Through observation of different aspect of program execution students learned about the semantic of language constructs and the notional machine prior to writing code.
- CS POGIL (Process Oriented Guided Inquiry Learning) [49], where groups of students construct their understanding about code through critical thinking questions that include reading, analyzing, adjusting code, and finally reflecting on what has been learned.

Note all three approaches are inquiry based, and exhibit many similarities to POE (Predict, Observe, Explain) [114], a well-known pedagogical approach to explore science topics at high school level.

Finally, it is worth observing that the tasks listed in later sections could be used in a variety of courses, as formative assessment in traditional courses, in active learning activities either using a comprehension-first approach or combining/replacing writing tasks with reading and editing tasks [64].

3 THE BLOCK MODEL

We complete the first part of the WG report by introducing the Block Model and explaining with the use of examples its role in analysing and classifying programming tasks.

The Block Model (BM) [90] is an educational framework that supports the analysis of core aspects of program comprehension. The merits of the Block Model have been attested by its application in different studies. In particular, it has been used in [87] with the aim of classifying tasks; moreover, in [112] the Block Model categorization was compared to Bloom's and SOLO taxonomy with the result that it leads to a more accurate categorization.

Block Model looks at a program from two perspectives:

(1) One lens looks at a program by considering different levels of zooming in and out: from single expressions or instructions to blocks, relations between blocks, and finally the whole program.
(2) Another lens, taking an orthogonal standpoint reminiscent of the SBF model [108], looks at a program as having three **dimensions** as follows:
   Text surface: the program code, a static entity;
   Program execution: the program in execution, a dynamic entity;
   Function/purpose: the program as an artefact with an extrinsic purpose.

These different perspectives are organised into a 4×3 matrix, where the rows represent a hierarchy of increasingly complex programming structures, whereas the three columns correspond to different dimensions of *ProgComp* — see Figure 1. Basically, stepping within the matrix upwards, from simple to complex, and rightwards, from surface to function, corresponds to achieving higher levels of abstraction.

At the bottom row, we have *atoms* or basic elements of the program, such as expressions or simple command lines. Next, we have *blocks*, such as a sequence of related assignments, like when swapping two variables, or a loop. Its upper level concerns the *relations* between blocks, which are implied, for instance, when a method is called with some arguments. Finally, the topmost *macro-structure* level takes into consideration the overall program.

Note the concept of an atom is relative to the instruction received: at the start every small element, an expression or a condition, is an atom; as code fluency increases, a full statement or a simple pattern (e.g. a swap) becomes an atom. This is why we need to situate the task in the learning process to match it at the appropriate level of comprehension. This will be done by indicating pre-requisites — so that those pre-requisites characterise the atom level.

The levels of increasing structural complexity can be seen as focusing on the sequence of steps in the comprehension process, which is conceptualized as flexible, context bound, bottom-up, and cyclic [90, 91]: textual information is perceived on a word-by-word basis and immediately incorporated in the mental representation. At the end of a block the capacity of the short-term-memory is reached, information needs to be transferred and integrated in working-memory so that short-term-memory is freed for the next cycle. In this integration process only some information (not all) of the former cycle is transferred, the mental representation is step-wise abstracted from the perceived material. This process is conceptualized as a hierarchic succession of mental representations, each hierarchy level being more abstract and independent from the details of the perceived information. During the process, the extracted information is connected or integrated with prior knowledge to build a coherent whole.

The columns reflect the different dimensions of a program: the leftmost column watches at the *text surface* of the program; the middle considers the program *when executed*; the rightmost is about the *purpose* or the *intention* of the program (and in some sense of the programmer who wrote it). In the following we discuss the programs' features pertaining to these different dimensions, and discuss them also in relation with programming knowledge categories from the literature review.

3.1 Knowledge Dualities in the Block Model

In this section we will explore in some detail two dualities related to programming knowledge and the Block Model.

Denotation versus Connotation. To begin with, we recall an important distinction, known from natural language, between **denotation** and **connotation**.

From the Block Model perspective, the text surface is the visible representation of the program text. This includes detecting the

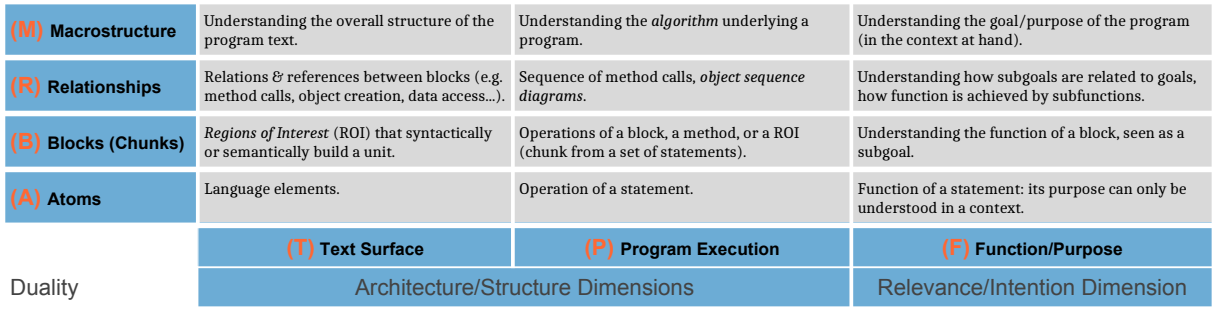

Figure 1: The Block Model Matrix.

beginning and the end of a given atom, and to discern the differences to other possible atoms, as well as to identify what kind of atom it is. For example, when reading the following code line:

```
i = i + 1
```

the reader discerns that "`i`" is one element (used twice), "+" is an operator, and "1" a literal. The attribution of "`i` is a variable" can be seen as first understanding. It can be called *denotation*, in contrast to connotation: denotation refers to the meaning of the element in a context-free sense, like the meaning of a word as described in a dictionary.

Connotation, in contrast, refers to discerning the meaning in the concrete context of the use. With regard to the variable `i` this can be to understand its role, e.g. as a "stepper" (see [85] for a list of roles). Identifying roles of variables is precisely *not* an example of task pertaining to the text surface dimension — in this dimension or column, understanding (if one would choose the term) is restricted to denotation.

ProgComp tasks in the Text dimension focus on the discernible features of the representational formats. In order to understand the text surface, some lexical and syntactical knowledge is required, e.g. where to put semicolons, where and how to declare a variable, and so on. Then, starting from identifying basic code elements at the atom level (AT), learners can be guided to discern structural information like the textual span of a block (BT), to recognise meaningful features that link atoms and/or blocks to each other (relational level – RT), and eventually to make sense of the macrostructure (MT) of the whole program.

Moving to the second dimension, understanding Program execution (or simply line execution) is based on the notional machine that introduces connotation as well. That is, execution usually depends on the context set up by additional elements. Let's refer to the example code shown before: understanding its execution would include working out the concrete value of variable `i` before and after the increment.

This dimension also exposes a crucial difference between natural and programming languages: while a natural language often leaves free room for subjective interpretation, the semantics of a program is univocal and is either correctly understood or not.[2]

[2]A program text can possibly be "interpreted", in some sense, to infer its operational meaning, for example by looking for *beacons* as clues of plans, as described in section 2.3. This is, however, a tentative interpretation, subject to verification.

The third dimension, Function (F), on the other hand, presupposes interpretation on a context *extrinsic* to the program itself.

Then, the related tasks require to link the program to some external purpose. For example, if the value of the variable `t` is *interpreted* as a temperature measure, the following expression:

```
1.8 * t + 32
```

can be meant as a conversion from Celsius to Fahrenheit degrees.

Programming versus Domain Knowledge. While the Block Model is organised around three different types of knowledge involved in *ProgComp*, another popular distinction is between program knowledge versus domain knowledge [91]. *Program knowledge* is required when extracting the appropriate information from the text surface and inferring the related operational semantics. *Domain knowledge* is used to make sense of the context, and thereby to understand the goals of a program. In short, the purpose (function dimension) is not an intrinsic property of a program but comes from an external source. A similar trait is referred to in a variety of formulations throughout different fields, as shown in Table 1.

Table 1: Different formulations of the structure vs. function duality.

Structure/Architecture		Function	Reference
Text	**Prog. exec.**	**Purpose**	Schulte [90]
Structure	Behavior	Function	SFB-Theories [108]
Mechanism		Explanation	Soloway [97]
Tracing		Reading	Lister [56] etc.
Plans		Goals	Soloway [101]
Program Model		Domain Model	Pennington [73]
Text base		Situation model	Kintsch [44]
Structure		Function	Kroes [46]
Proximate		Ultimate	Tinbergen [107][a]

[a]Originally used in Biology, now adapted for machine behavior [75]

From an educational viewpoint, it is interesting to observe that the three dimensions of the block model also correlate with categories characterising qualitatively different perceptions of the programming activity [106] and of the *learning* of programming [32], as attested by the outcome of empirical investigations of *variation* in novices' perception of programming within the framework

of phenomenographic research. The two cited studies, in particular, set forth a hierarchy of five categories in connection with (1) the textual representation of a program, (2) the action of a program, (3) the application addressed and (4) the problem solved by a program, (5) the contexts in which programming can be a valuable resource. From this perspective, the *text*, *program* and *function* dimensions can be seen as describing a hierarchy of knowledge and skills, where each subsequent stage presumes mastery of the concepts implied by the previous ones.

As a further related example, Bruce et al. [8] had also identified five categories, from which text, program and purpose emerge as distinct traits. More specifically, at the lowest level, students' "primary intent is to keep up with set assignments" to get enough marks. At the next level, learning to program is mostly seen as learning the syntax of the programming language (Text dimension). In the third category, the focus is on "the structure and logic of the language — in essence, how the language works" (Program dimension). Then, the programming language is not seen as an end in itself, but as a means to solve problems and achieve tasks (Function dimension). Finally, in the last category, programming is also experienced as a "culture", as "participating" in the programmers' community.

3.2 Using the Block Model to analyse programming tasks

The Block Model was designed to help educators in reasoning about the cognitive implications of program reading and comprehension, as well as in planning how to teach *ProgComp*. The Block Model's perspective seems apt to support what emerges from the experience of practitioners, i.e. the fact that program comprehension requires a variety of pieces of knowledge and skills to be mastered.

Indeed, in order to actually come to a full understanding of the program under consideration, one needs to understand all its different dimensions, at all different levels of complexity, as articulated by the Block Model. Consequently, several different kinds of activities are needed to foster *ProgComp*, each focusing on a different facet or endeavour.

Typical *ProgComp* activities ask the learner to explain what a piece of code does, annotate or comment the code, represent its execution with sketches or trace tables. Other tasks, such as Parsons problems [72] or debugging problems, are more connected to writing programs but still rely heavily on program comprehension.

Depending on the size and features of the piece of code under analysis, a certain kind of activity may help in achieving different learning goals, since it may activate and/or require different cognitive processes. For example, when asking to explain on one's own word what a piece of code does, one might aim at different kinds of answer. Often the task pertains to the comprehension of the code purpose or goal (Function dimension), as in the approach of Lister et al. [51]. However, the same type of task may focus instead on the mechanics of a counting loop (Program execution dimension), or even on the syntax of an assignment statement (Text surface dimension).

Analysing *ProgComp* activities within the the Block Model framework supports the identification of the learning goals and the prerequisites associated with the task, and may help teachers in understanding when and how to propose such different tasks in learning and assessment.

In the remaining of this section we will describe some representative *ProgComp* activities, namely tracing tasks, "explain in your own word" tasks, and Parsons problems, and analyse them by using the Block Model. We used the same method to analyse a number of *ProgComp* tasks we collected, which resulted in a classification that will be presented in Section 5. Furthermore, in section 6 we will show how *ProgComp* learning trajectories can be designed, by devising different learning activities that target particular cells in the block model such as AP or BF.

3.2.1 Tracing Tasks. Tracing is defined as following the execution of a program, atom by atom and line by line, in a sequential manner. Each tracing task is aimed at covering or assessing one particular aspect of the notional machine. While tracing, we follow the state of the variables after each line is executed, possibly using a tracing table. Hazzan's Guide to teaching Computer Science [38] describes the following variations for tracing:
"A tracing question can ask to follow (a) a complete program; (b) a single method; (c) a recursive method; (d) object creation. In addition, the following instructions can be used in each of the above variations: (1) follow the code execution according to a given input; (2) follow the code execution when learners choose the input; (3) follow the code execution according to several different specified inputs which are selected in a way that guides the learners to find what the given code performs; (4) find different sets of inputs so that each set represents a different flow by which the code is executed; (5) find a set of inputs that yields a specific output."

Figure 2 shows a range of tasks proposed in the literature as 'tracing tasks'. The first example focuses on assignment, while the other two tasks address loop iteration. Notice that, in fact, these exercises do not explicitly ask to trace code, but to write either the values of variables at the end of execution (first example) or the computed output (final value of *count* in the second example, or full trace of loop control variables "`i`" and "`j`" in the third example). Tracing is indeed the assumed strategy to obtain such values or output.

Also note that the second tracing task [87] is not given a starting value of `N`, instead the premise to the question says "assume that N is a positive integer", while providing 5 possible choices (`N`, `N/2`, `N/2+1`, `(N+1)/2`, or `0`). In line with Hazzan's guide, we could ask other tracing questions at different levels, such as

Q1 – If `N` has value `15`, what is the output of the program?
Q2 – What would be the value of *count* when `N` value is `10`? And when is `13`?
Q3 – For which values of `N` will the program print `0`?

In terms of the Block Model, the core of tracing activity is to follow the execution of atoms; therefore tracing focuses on cell AP. Understanding the execution of the current atom then includes understanding which atom is the next one to be executed. That is, there is no need to abstract from this perspective to some more general understanding of the relations of parts of the program —

T1: What do the variables v1, v2 and v3 hold after the following Python code is executed? Assume that they are all integer type variables .

```
v1 = 10;
v2 = 15;
v3 = v1;
v1 = v2;
v2 = v3;
```

T2: What is the output of the following code segment?

```
int count=0;
for (int i=0; i<N; i++) {
  if (i % 2 == 0) {
    count++;
}}
System.out.println(count);
```

T3: What is the output of the following code segment?

```
int i, j;
for (i = 1; i<=5; i++) {
  for (j = 1; j<=5; j++) {
    printf("%3d", i*j);
}}
printf("\n");
```

Figure 2: Three tracing tasks from the literature: T1 - trace swap [19], T2 - Count evens in range [87] and T3 - trace nested loops [13]

one atom simply denotes which atom is the next in the program flow. In summary, tracing can work at atom level only.

When the code to trace involves methods or functions calls, however, it is necessary to establish connections between states relative to both the caller code and the called procedural unit. Thus, in this case, tracing occurs at the Relational level (RP).

In a learning trajectory tracing seems to be a basic skill. Even when tracing is at the Atom level, the comprehension process need not be restricted to that level; the Block Model asserts then while tracing, the comprehension process will not stop, but some chunking or abstraction (going up the levels) will occur. So a higher understanding than at the atom level *may* also occur.

As Deimel explained [25] "students should not be discouraged from tracing code in order to understand it, but they should be made to realize that doing so is a means to an end, a source of data for the real task of interpretation."

Teague et al. [104] describe the development of 'abstract tracing' as reading the code without relying on concrete values — instead the reader can think about a set or a class of values and how those lead to specific traces (as in Example T2 above). This kind of abstract tracing doesn't have to be complete to allow inferences. Lister and Teague refer to inferring the 'purpose' of the code — but probably it is more inferring the algorithmic idea, hence MP (or perhaps BP) in the block model notation.

In summary, in later learning stages a student may not trace the code but, based on plan knowledge, get a quicker understanding of the code (probably at BP level). Likewise, students could take shortcuts in tracing, discussed as 'beacon' in [7]: by detecting some familiar elements a student might be tempted to infer from that beacon the overall plan of the lines of code to be traced, so that, instead of tracing, the value of the elements are inferred based on this intuitive understanding of the assumed goals or plans of the program. For example, a reader may anticipate that the first task in Figure 2 includes a swap, or that the second task is counting the

```
public void method10B(int iNum)
{
  for(int iX = 0; iX < iNum; iX++)
  {
    for(int iY = 0; iY < iNum; iY++)
    {
      System.out.print("*");
    }
    System.out.println();
   }
}
```

Figure 3: Example of "Explaining in your own words" task [56]

even numbers in the range. However, when the individual atoms are perceived as an integrated whole — e.g. task T2 as a "counter control" loop-plan [22] — it is possible that some of the details of the atom level get lost. For instance, the reader may not check whether the loop begins with `0` or `1`.

From a different perspective, tracing may also help to develop — besides construct knowledge (AT and AP) — also more general type of knowledge that is later needed to build a more abstract understanding of the program execution; that is knowledge about the notional machine. An example of this is reported by Nelson et al. [70] using a self-contained online course. Similarly, according to a study by Hertz and Jump [40], *program memory tracing* seems to be working in this proposed way.

In conclusion, the core learning effect of tracing is to address knowledge and skills pertaining to the AT and AP cells of the Block Model, even though this kind of activity can set the ground for knowledge at level of higher abstraction. As far as possible, novices tend however to avoid carrying out an accurate step-wise tracing process, and rather try — more or less successfully — to get a somehow abstract grasp of algorithmic patterns to comprehend BP, RP and eventually MP. Being able to develop viable and sound abstractions of program execution is of course required in order for the learner to progress in *ProgComp*. So, teachers and learners must be aware of the role, scope and aims of *concrete* tracing tasks.

3.2.2 "Explain in Your Own Words" Tasks. This type of task requires to explain in natural language the execution (Program dimension) and/or the function/purpose (Function dimension) of a program. Typical examples mostly focus on commenting on the Block level and the Macro level, because comments stick to the block they are commenting on, or to the program as a whole.

Lister and colleagues refers to questions at the MF level as reading questions: Describe the purpose of the program text in your own words , [54]. Figure 3 shows one of the code fragments used in [56]. They are also called "Explain in Plain English" questions [18, 69], although we prefer to called them "Explain in your own words" so that it applies to non-english speakers as well.

One interesting observation is that students often have difficulties to explain the purpose (MF), and although asked to describe the code goal, they explain the execution of the program (P dimension) instead, and this often step by step (AP).

While authors differ whether the ability to explain the behavior (tracing) necessarily precedes the ability to explain the purpose

Parsons Problem	Target code

`  if( l[i]>b ){`
`int b=l[0];`
`  }`
`    b=l[i];`
`int[] l={20,24,23,35,30,35};`
`for(int i=1;i<l.length;i++){`
`}`
	`int[] l={20,24,23,35,30,35};`
	`int b=l[0];`
	`for(int i=1;i<l.length;i++){`
	`  if( l[i]>b ){`
	`    b=l[i];`
	`  }`
	`}`

Figure 4: Example (left) of a Parsons Problem code given for the goal "this program finds the largest value in the array", and (right) its expected solution.

[18, 54, 56], these empirical studies support the rationale of the BM dimensions, or Pennington's idea of program models (program model vs. domain model), that program execution is qualitatively different from its function or purpose.

From the BM perspective difficulties in discerning MF for students focusing on AP is not surprising - there are many comprehension steps in between.

To make it easier one can structure the task and provide scaffolds. E.g. so that first students explain the function of blocks (BF) — helping them to derive MF. It should also be helpful to provide explaining tasks at the RF level: Requiring learners to figure out the goal based on given subgoals. We are not sure if there are specific examples of "explain in your own words" tasks relative to RF, or if this has been overlooked in practical examples.

Two more observations on these types of task. First, from an experience point of view, it is sometimes hard to distinguish between explaining/describing the operation (program execution dimension) and explaining/ describing the function (function/intention dimension), unless the two features differ greatly. Maybe here is the crucial task to watch out for when using this learning activity in the classroom. Second, complexity of the task can vary greatly based on the cognitive complexity of the program, defined by the number of blocks and the relations between them.

3.2.3 Parsons Problems. In a Parsons problem, the correct code to solve a particular problem is provided, but the code is broken into code blocks (in general lines of code) and mixed up. The task is to rearrange the blocks into the correct order for the code to run successfully thus achieving a given goal. An example of a Parsons problem (also called a Parsons Puzzle) is shown in figure 4.

Parsons problems allow to work on complex code, such as the loop control in figure 4, with a lower cognitive load. For instance the solver can neglect the syntactic aspects and focus only on reconstructing the order of code fragments so that the resulting program implements the plan required to achieve the given goal.

The standard version can be extended by either (1) including *distractors* where irrelevant lines are added to and mixed with all useful lines of code, or (2) providing choice using *paired options* where a selection between two highlighted lines of code needs to be made, or (3) removing the indentation obtaining *two-dimensional* Parsons puzzles (in the standard version, lines appear with the indentation that they should have when in place in the final solution). Such variations can be used to focus assessment specifically on misconceptions or areas that learners typically struggle with.

Ericson at al. [34] found that solving two-dimensional Parsons problems with distractors took significantly less time than fixing code with errors or than writing the equivalent code, whilst being just as effective. Another clear advantage is that marking is fast and objective. Denny et al [26] noted a direct correlation between Parsons-problem and code writing scores. Overall, Parsons Problems are somewhere between reading and writing tasks [26, 34].

In terms of the BM, in Parsons problems the Function (or purpose, or goal) of the target code is given in the task description. The text surface, at least in the standard version, is complete, in that all elements (lines of code) are there. However, they are in a shuffled order, so that it is not possible to infer by the features of the text structure which elements form a block and how these are related to enable the desired program execution. Reading and thinking about these elements require to discern atoms (AT) and, from hints in the text surface — e.g. code comments, meaningful names of methods and variables — or in the task description, to get an understanding of their role in the program (MF understanding). This can be done by proceeding top-down (from MF downwards), bottom-up (from AF upwards), or with a mixed approach.

Non-standard Parsons problems' difficulty is easily adapted [34] hence, from the Block Model perspective, they have a very high variation in complexity, and focus.

4 TEACHERS' VIEWS OF PROGRAM COMPREHENSION

4.1 Methodology

Study Design. As we intended to capture the current practices and perceptions of *ProgComp*, we employed first an exploratory stage in which we conducted structured interviews on the topic with teachers involved in secondary and/or post-secondary/tertiary education. During the second stage of our study we (a) collected and organized program comprehension tasks and classified them using the BM and (b) defined possible learning trajectories that can guide teachers as they select and sequence those learning activities in their CS0/CS1/CS2 or K-12 courses. Our approach is in-line with the didactic transposition theory of transforming academic knowledge with the purpose of contextualization in an educational context: didactic transposition refers to the transformations an object or a body of knowledge undergoes from the moment it is produced, put into use, selected, and designed to be taught until it is actually taught in a given educational institution. And these transformation "presuppose the decontextualisation of academic knowledge from the conditions within which it was created and its recontextualisation according to the terms and restrictions imposed by the educational context", Chevallard and Bosch [14].

The instrument. In order to collect comparable data, before carrying out the interviews the working group members debated the questions to ask and the format: survey or interviews. Our goal was to elicit aspects of the instructors' pedagogical content knowledge (PCK) about program comprehension. Shulman's PCK [96] is meant to integrate teacher's knowledge of the subject being taught as well as of how to teach it in concrete situations. It is then grounded in

the beliefs and practices of the teacher and covers conceptual and procedural knowledge, of a repertoire of activities, techniques and resources, of how to evaluate the learning outcomes.

Teachers' *PCK* is usually characterized precisely through interviews and, in this respect, the *CoRe*s provide a suitable model to ask about important ideas/concepts ("Big Ideas") [57]. Examples of application of the CoRe protocol to characterise teachers' PCK about introductory programming topics can be found, e.g., in [3, 9, 83].

However, due to time constrains and ethic approvals we postponed using in-depth interviews and eventually agreed on a protocol for a short structured interview (see appendix B) which could be collected prior to our conference's meeting.

Data collection. We interviewed 31 instructors (22 M/9 F) from institutions in 10 countries (Canada, Finland, Germany, Italy, Peru, Spain, The Netherlands, Turkey, UK and USA). The interviewees are secondary school (8) or university teachers (19), some having taught at different instruction levels (4), including primary school. 8 interviewees have been teaching for 20 years or more, 12 between 10 and 19 years, 7 at least for 5 years. Most of them have a background in CS; the others either in Mathematics or in Engineering fields. A large part of their students are learning CS or computing-related subjects. The interviews were conducted in person or by e-mail by the authors of this paper and consisted of four main questions and several sub-questions (see appendix B).

Interview coding and analysis. We used a team-coding method with several researchers participating in interview coding and analysis. The interview data is very rich; however, we focused on this study on three main themes: (1) definition of *ProgComp*, (2) what concepts and skills are the most important for students' learning, and (3) which teaching aspects described by our participants matched cells of the BM as presented in Figure 1.

Our approach to coding the first two main themes was inductive: based on the interview data, one of the authors of this paper proceeded with an initial coding and proposed the codes presented in Table 2 and Table 3. In the analysis of the activities reported by teachers we tried to understand the activities in the context of the Block Model theory, therefore our approach was deductive as we were using the Block Model categories and definitions.

Five authors participated in a second stage of coding. Each interview was coded by two researchers: one initial coder coded an interview and the second coder read the coding done by the first one and indicated in a rubric all aspects of agreement and disagreement. After that, the two coders discussed and exchanged messages until they agreed on the final coding. We assured the coding validity as all coders were familiar with this research, followed the same coding protocol and understood the meaning of codes in the same way.

Next, we will present out results.

4.2 Teachers' views of Program Comprehension

It is interesting to capture and explore the views and motivations of practitioners in regards to *ProgComp*. In this section we will provide a summary of the answers to the interview question "Explain in a few words what the term 'program comprehension' means to you".

Table 2: Practitioner's views of Program Comprehension

ProgComp description	Frequency
ProgComp as code reading ability	23
ProgComp as mental model of the NM	9
ProgCom as writing code	5
ProgComp as knowledge of prog. constructs	4
Other views of ProgComp	7

With the exception of one interviewee (*"I don't think I explicitly teach program comprehension, but rather writing code"*), most teachers are aware of one or multiple aspects of *ProgComp* as shown in Table 2. The range of coverage varied: 19 teachers (60%) provided only one view, while 4 (13%) of them gave very comprehensive definitions that included three categories. Next, we will provide detailed examples on each category, except the last one which is a mixed bag: two teachers were very generic and hard to be classified, e.g., "Be able to have a whole view of the program", while another pair talked about the goal of writing code to solve problems.

ProgComp as code reading ability. Most teachers think of ProgComp as being able to read and explain (possibly to themselves or to others) code. Here are a few examples that elaborate on this theme:

> *ProgComp is a skill that allows a student to read a meaningful segment of code and find out what is designed to do.*

or:

> *ProgComp means to grasp a program's purpose and to be able to explain the underlying algorithm accurately.*

This reading skill implies the ability to predict the outcome of executing the code:

> *Understanding code written by others; being able to predict the outcome of such code.*

but one teacher points out that such understanding is deeper than simply being able to trace code:

> *It is different from tracing: tracing focuses only on the operational aspects (how the notional machine works).*

ProgComp as mental model of the notional machine. From another perspective, ProgComp is meant as developing a mental model of the notional machine, of *"what is happening beneath the hood"*:

> *ProgComp is how the students understand programs by developing mental models of how computers work.*

or more precisely:

> *Developing a precise mental representation of the internal state of the program (variables, activation records and stack…) and how this state evolves.*

ProgComp as writing/developing code. Some teachers express the need to understand code in order to either write or modify code, e.g.:

> *ProgComp means being able to read and understand a program well enough that I can make subtle changes to the code and students will be able to describe the effects.*

Some teachers discuss the pragmatics of code developing such as editing, compiling, debugging, working in team and so on:

> *For me, ProgComp refers to the methods that developers use to maintain existing source code.*

ProgComp as understanding basic constructs. Finally, four interviewees provide definitions involving the understanding of the basic imperative constructs:

> *At the introductory level, I think it means understanding the basic programming constructs (such as loops and conditional statements, etc.) as well as understanding the logic required for a given program.*

possibly including language syntax features:

> *Being able to understand the (concrete) syntax of a language as well as its semantics.*

In addition, one of the teachers also feels the need to distinguish between different levels of understanding in connection with *"the progression of constructs"*.

4.3 Learning objectives linked to *ProgComp*

In this section we will explore the answers to the interview question "What concepts and skills do you want your students to learn in connection with program comprehension?".

Table 3 provides a summary of this analysis. Most teachers indicated either one (38%) or two (48%) learning objectives (LO), whereas 3 teachers indicate three LOs.

Developing a mental model of the notional machine. Nearly half of the interviewed teachers have as a LO the development of a mental model of the notional machine either explicitly (sometimes including the mechanics of procedure call and return back to the caller), or indirectly by referring to tracing tasks. Here are a few sample excerpts from the teacher interviews:

> *I want them to be able to trace through code and work to figure out how to fix their code by understanding rather than hacking their way through it. I would also like them to understand what is happening in memory and how it is modified as the program executes.*
>
> *For simple programs, being able to work out the output and contents of variables. Full tracing: the pathways through a program, what happens at branch points. Understanding what paths could be taken (static CF) and what paths are taken (dynamic CF).*

One teacher pointed out the role of visualisation as an important aid to build mental models of program structures. Two teachers cited advanced topics such as activation records, pointers and multi-file programs, while other describe simple tracing with pen and paper or being able to explain individual constructs.

Being able to chunk and explain programs. Another common theme was the ability to chunk code when reading. A detailed example of this LO is given below:

> *I want students to learn to think at different levels of abstraction. I would like them to think about the programming plans/micro-patterns that they know and to recognize them in the code they are reading and to think about how those plans are composed together to solve a domain problem.*

Table 3: Learning objectives linked to *ProgComp*

Learning Objective	Frequency
Develop a model of the notional machine	14
Being able to chunk and explain code	14
High level thinking and abstraction	12
Being able to write/modify/debug code	11

More often, however, their statements simply mention the ability to explain programs, e.g.:

> *To read and understand code so I have them read and explain many code examples.*
>
> *Ability to be accurate about in/out specifications, to explain program behaviour using a formal language.*

High level thinking and abstraction. Several teachers also explicitly addressed the role of high level thinking and abstraction both when reading and when writing code, as follows:

> *So I think [students] need a lot of practice [tracing etc.] to be able to abstract things.*
>
> *Students need to abstract the problem [...] and think about ways to bring the core elements which are needed to solve the problems together. There's a lot of "imagining" what the program should look like and the students need to be sure what they will be doing before writing the program. Abstraction, structuring would be the terms which come to my mind.*
>
> *Students must be able to explain programs, to provide arguments, to compare and assess...*

Being able to write/modify/debug code. As to the *ProgComp* learning objectives, a number of teachers mostly refer to code writing, modifying and debugging abilities, maybe also implying problem solving skills. The underlying idea is that *ProgComp* can be developed by practicing programming, by writing code, somehow as a by-product outcome of this practice. This is well expressed by the following excerpt from a teacher:

> *ProgComp is not an explicit topic, part of what they do anyway but no lecture time explicitly devoted to ProgComp in a focused manner...Mostly just teach them how to program but hope/assume that through osmosis they can look at other code. We are primarily concerned about design and authoring with the implicit assumption that if you can design and author you can look at some other code and figure out what it's doing.*

Or, said a little differently, but eliciting some of the tasks in which students are expected to engage:

> *In some way, code comprehension is embedded in code writing. Debugging is also a* ProgComp *activity, since students need to form a model of the program to be able to fix it. Extend code from someone else or code that students wrote a long time ago requires strong* ProgComp.

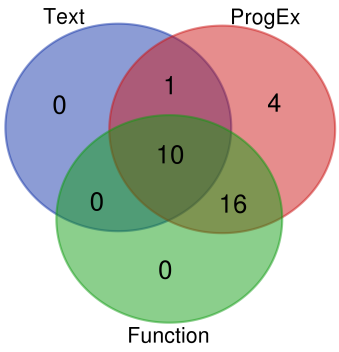

Figure 5: Venn diagram showing coverage of BM dimensions by teachers.

Other interviewees mention good coding practices such as program modularity, *"parsimony of code"*, problem decomposition, or recognition and adaptation of (recurring) program patterns.

4.4 Teachers views mapped into the BM

Finally, we will present the results of mapping interviews' content into cells of the BM matrix. Interestingly, although in the participants' perceptions of *ProgComp* "reading ability" was discussed more frequently than "the notional machine", the Programming execution (P) domain was the most frequent theme overall (74), followed by Function (F) as shown in Table 4. Even if the F dimension occupies the second place (61), it is important to note that AF is sparingly mentioned (coded for only 6 participants) and therefore the other three categories are very strongly represented (55).

Table 4: Block Model mapping numbers

	level				
Dimension	**A**	**B**	**R**	**M**	Total
Surface Text	7	4	3	5	19
Program Execution	20	17	19	18	74
Function	6	14	17	24	61

The Surface dimension has the lowest numbers at most levels. We interpret this result by considering many practitioners associate program comprehension with connotation instead of denotation. That is, recognizing/discerning elements in the text surface is not considered as understanding. An alternative interpretation is that teachers would cite more frequently those areas that students struggle with, as they require more scaffolding. Thus, it makes sense that the Surface dimension, which is considered to be easy, is cited less frequently.

The hardest areas, conceptual knowledge at the atom level (AP) strategic knowledge of relating goals to plans (RP and RF, MP and MF), are strongly represented.

We consider the final goals of achieving *ProgComp* are related to the cells found on the upper right corner of the Block Model. This view is shared by the teachers as the MF category (understanding the goal/purpose of the program) is the most frequently cited.

Interestingly, all interviewees have at least one reference to Program execution (P) as shown in Figure 5, most commonly in combination with Function (F) (51%). Note that one third of them talk about *ProgComp* in terms that matched all 3 dimensions (T/P/F), while none referred only to text surface (T).

As mentioned above, this is probably due to practitioners' experiences on the difficulty and hence importance of different aspects in teaching. In the following section we will present different learning activities, obtained – in part – in the interviews, and then in section 6.3 present some qualitative results of the interviews with regard to difficulties and hints for possible learning trajectories.

5 COLLECTION AND CLASSIFICATION OF *PROGCOMP* TASKS

Based on literature analysis, discussions within the working group, and examples of activities provided by our interview participants, we collected and categorised several types of activities that are intended to help students develop *ProgComp*.

The tasks listed in this section also include some common type of tasks analyzed in Section 3: tracing tasks (AP, RP), a Parsons problem (BP), "explain in your own words" tasks (BF, RF, MF).

5.1 Methodology

We categorized *ProgComp* tasks by using the Block Model framework. In other terms, we analysed each of the available tasks considering both at what level of complexity it focuses (atoms, blocks, relational, macro) and what dimension of the program it looks at (text surface, i.e. syntax; program execution, i.e. notional machine; function, i.e. purpose or intention of the code).

This approach allowed us to enrich the list of available *ProgComp* tasks. Indeed, during the analysis, we found that some parts of the block model matrix were not covered by any task, and this led us to devise new types of task with the potential to fill those gaps.

We will go through the block model column by column, presenting first task types that pertain to the Text surface, then the Program when executed and finally the Function of the program. For each column we will group the task types starting from the atom level upwards. These types of task then need to be further specified in relation with a particular code fragment. An example of this can be found in Figure 6.

5.2 Text Surface Tasks

The text dimension of the Block Model is based on the perceivable representation of a program. In terms of the comprehension process, reading and comprehending starts by perceiving, which implies identifying and discriminating between atomic elements in the text, then recognising their organisation into language structures of growing complexity, up to the overall program structure.

The types of tasks in this category are then focused on *statically detectable properties*, i.e. syntax as well as static typing. Even though we restrict our attention to program notational features, the inherent complexities of language constructs and dependencies may be overwhelming to students, as demonstrated by Luxton-Reilly et al. in their in-depth analysis [58]. Luxton-Reilly and colleagues also

provide valuable suggestions as to how to decompose a complex task (in a novice's perspective) into more focused components.

Other types of tasks considered in the literature to assess or develop novices' understanding of the static properties of programs include, in particular:

- Tasks requiring to fix compile-time errors introduced within the code in order to test students' ability to identify the actual sources of the problems [48].
- *Fill-in-the-gap* (e.g. choosing the right keyword) and *highlighting* (e.g. identifying the occurrences of a syntactic concept) tasks to test students' basic competencies on language syntax [45].
- Parsons-like puzzles involving only the language notation and tasks requiring to translate an accurate formal definition into code [65].

To be more concrete, we list a few specific task examples, either drawn from the literature or suggested by working group members and participants to the interviews. The examples are classified according to the rows of the Block Model:

Atom–Text (AT).

- Identify the keywords in a piece of code;
- Box all the assignment statements;
- List all integer variables;
- Box all arithmetic expressions (arithmetic expressions can be recognised from purely syntactic items);
- Box the headers of all methods/procedures/functions;
- Transform between alternative syntactic forms of atomic elements (e.g. from `i++` to `i=i+1`).

Block–Text (BT).

- Draw a box around the code of each conditional construct;
- Draw a box around the code of each loop;
- Box the body of each method/procedure/function;
- Check if the parentheses are placed correctly;
- Draw nested boxes to represent the structure of a complex expression.

Relational–Text (RT).

- Link each occurrence of a variable with its declaration;
- Identify the scope of a variable (assuming static binding);
- Identify where a particular function is called;
- Verify if all expressions are correctly typed;
- Verify if every potential flow path of a function's body ends with a return statement;
- Draw a box around the initialization/termination/increment expression of a `for` loop (relational for novices first learning about loop control).

Macro Structure–Text (MT).

- Represent the overall program structure by drawing a "block-nesting" tree;
- Restructure a program's code so that library links are at the top, followed by the definition of global variables and functions/methods, followed by the main program;
- Describe the overall program block structure by drawing nested boxes;
- Draw a diagram showing the overall program structure;
- Represent the overall program structure by drawing a tree of function/procedure dependencies (relative to invocations);

5.3 Program Execution Tasks

In order to deal with the dynamical aspects of execution, the information provided by the program text is not sufficient, but must be supplemented with a concept of machine *state*, establishing the context(s) in which the program is in action. Thus, the program dimension of the Block Model focuses on code execution, or, in technical terms the *operational semantics* of a program.

At the heart of any characterisation of the program dimension lies the construction of a viable mental model of the *notional machine* [27]. In this respect, Sorva [99] presents a comprehensive review of research threads "that have contributed to our understanding of the challenges that novice programmers face when learning about the runtime dynamics of programs and the role of the computer in program execution". When engaging with the task of tracing the execution of some piece of code, "sketching" is a common practice for students in order to overcome the working-memory load which would be implied by following a long progression of actions and states in their mind [20, 21, 117].

Several types of tasks designed to investigate on novices' mastery of programming pertain to this category. Here is a list of those most frequently encountered in the literature:

- Tasklets that focus on "atomic" aspects of the operational semantics, by taking a "reductionist" approach to novices' understanding and learning of programming [58].
- Tracing, predicting and "fill-in-the-gaps" (within code) tasks designed to assess novices' program comprehension [53, 66].
- *Proglets*, i.e. little programs aimed at reducing the learners' cognitive load by exploring a single programming concept [33], when used as the basis of tasks requiring to predict the program outcome, to modify the code, or simply to experiment freely with it.
- *Parsons programming puzzles* focusing on the understanding of the notional machine [26, 39, 72].
- Tasks requiring to trace recursive computations, [37, 65, 86, 89].
- Tasks requiring either to verify reversibility or to write reversing code [41, 52, 67, 103].

At a finer-grained level, by considering also the suggestions emerged within the working group and in the course of the interviews, we can classify a range of examples in terms of rows of the Block Model:

Atom–Program (AP).

- Trace the program execution for some given input data, where the program does not include procedural units (note that this task can be accomplished at the atom level, as a sequence of several atomic steps, each next step being determined by the previous one);
- Determine the program output (e.g. what is printed) for given input data, again where the program does not include procedural units;

- Determine the value of an expression for given values of the involved variables;
- Trace a particular sequence of statements for given values of the involved variables.

Block–Program (BP).

- Determine the number of iterations of a loop construct for a given initial state (here recognising which repeated step is to be counted implies reasoning at block level — the repeated block; in particular, think of a nested conditional in a loop);
- Identify recurring instrumental blocks such as that for swapping the values of two variables (the assumption is that the identification is based on reasoning about the execution of short sequences of statements);
- Identify the block(s) implementing some specific program pattern, e.g. among those catalogued by [2] or [74];
- Solve a Parsons puzzle for a specific programming pattern.
- Change a *for* loop into a *while* loop.

Relational–Program (RP).

- Identify the variable(s) playing a specific role (in the example of Listing 1: *stepper*, *most recent holder*, *most wanted holder*, *walker*);
- Trace the program execution for a given input, where the program includes calls to procedural units (this task requires to establish connections between states relative to the caller's code and to the called procedural unit);
- Verify whether some branches of a switch/case statement are redundant, i.e. can never be executed;
- Identify states, i.e. values of one or more variables, that could result in an infinite loop;
- Identify the scope of a variable.

Macro Structure–Program (MP).

- Verify if a program statement or block is ever reachable during program execution;
- Identify a comprehensive set of inputs to check *all possible* computation flows of a program;
- Select from given options the program that is computationally equivalent to a reference one, i.e. which gives rise to the same sequence of variable states for every admissible input data;
- Explain why two given programs are not computationally equivalent;
- Estimate the computational costs of the program.

5.4 Function or Purpose Tasks

Relative to the function dimension of the Block Model, a new context, introducing properties *extrinsic* to the program at hand, comes into play.

Drawing a borderline between (abstraction on) code execution features and purpose-driven features is not always straightforward, and it is likely to depend to a large extent on the knowledge assumed at a certain learning stage.

However, well-developed tasks exploring this dimension of program comprehension are more difficult to envisage. As pointed out by Begum and colleagues [4], "[v]ery little research has investigated the behavior of programmers from understanding the problem specification to computer program".

Among the tasks considered in the literature, in which novices are required to understand the program in connection with an extrinsic problem domain we can mention the following:

- Tasks asking to explain in words[3] [54, 66, 113] the purpose of a program.
- "Fill-in-the-gaps" tasks designed to assess novices' understanding of the relationships between a program and the problem being solved [53].
- *Parsons puzzles* focused on the problem to solve [26, 39, 72].
- Tasks requiring to choose more meaningful names for program functional units, or to chunk code segments and define semantically meaningful functions [65].

In more detail, again by integrating suggestions coming from the working group as well as the interview participants:

Atom–Function (AF).

- Identify the purpose of an expression or a simple statement, in connection with the problem domain (e.g. of an expression/assignment for converting Fahrenheit to Celsius)
- Identify the purpose of a condition w.r.t. the problem domain (e.g. divisibility for some positive integer);
- Rename a constant with an appropriate name from the problem.

Block–Function (BF).

- Choose an appropriate name for a simple procedural unit (method, procedure or function, where the unit body consists in a simple block);
- Summarise in a short sentence what the block goal is;
- Identify the program block(s) with a given function, described in problem-domain terms;
- Write comments explaining the purpose of a block and of the statements it is built from.

Relational–Function (RF).

- Choose an appropriate name for a variable (usually the function of a variable can be inferred by establishing relationships between different occurrences of it);
- Summarise in a short sentence the purpose of a simple block invoking one or more methods/procedures/functions;
- Solve a Parsons puzzle for a given code purpose by reordering simple blocks (it requires to identify the sub-purpose of each block and their relationships)
- Identify functionally equivalent blocks, i.e. blocks giving rise to the same overall state transformation (selection from a few predefined options).

Macro Structure–Function (MF).

- Choose an appropriate name for a program;
- Summarise in a short sentence what the program goal is;
- Select the sentence, from a few options, which most accurately summarises the program's purpose;

[3] called in the literature "Explain in plain English" but students may use their native language instead

- Create meaningful test cases for the allowed inputs and expected outputs (test cases are usually based on the program's purpose).

5.5 Towards a repository of Learning Activities

Due to time constrains, we were not able to set up an online repository for the collected task. However, it is a long-term goal to either create of join an open-source "live" repository where practitioners/teachers as well as researchers in the field of computer science education can find and contribute *ProgComp* resources. With this goal in mind, we have designed a template to be attached to each submitted *ProgComp* activity, which provides context and supports its use.

The template incorporates the following fields: the coding that describes the activity as in the a block model; pre-requisites (CS and *ProgComp*); materials provided by instructor; instructions for students; the new things that students will learn from this activity; how the activity can be designed as an individual or a team-based activity; and the perceived engagement as in the ICAP model [15].

To validate the template, which is included in appendix C, a subgroup filled a template form for four different types of activities : (a) identifying the role/purpose of variables, (b) commenting selected/key lines of code or code snippets, (c) tracing, and (d) debugging (finding and fixing an error).

6 MOVING FROM SINGLE TASKS TO LEARNING TRAJECTORIES

Learning trajectories (LT) have garnered the attention of math and science educators [55] because of their ability to model how the student's thinking about a specific topic evolves, which supports research-based curriculum development [88]. Such research-based curriculum development has taken place, for example, in the mathematics education community [17].

However, empirical knowledge about LT is largely absent in computer science education. One reason is that there is no established methodology to systematize and define learners' progression in CS disciplines. Some recent studies attempted to extract data from the literature to create learning trajectories for sequence, conditionals, and repetition [78]; abstraction [76] and debugging [77]. These LT provide a path for particular aspects for programming and comprehension, but to the best of our knowledge, there is no learning trajectory for *ProgComp* as a whole skill.

In the following sections, we present our methodology to create LT for *ProgComp*. This methodology could assist instructors in two ways. First, it provides practical examples for instructors of how to decompose a task that fosters *ProgComp* into sub-tasks that reduce the complexity with respect to the overall task, making it suitable for beginners, and later move to more advanced levels of complexity aimed to advanced learners, working on different aspects of *ProgComp*, as presented by the levels of the Block Model. Second, it provides a guideline that could help instructors identify where a specific task fits into the Block Model and what particular aspects of program comprehension are being fostered.

6.1 Methodology

Using the work of Lister and colleagues as a starting point, the "Leeds" ITiCSE working group Lister et al. [53] concluded that students lacked basic skills pre-requisite for problem-solving, such as comprehending program code. More recently, assessment tasks were found to be more complex than academics expected [58]. For example, tasks typically require both algorithmic thinking (for example initializing a variable before updating it), as well as a more advanced understanding of data representation (assigning a value to a property) [92]. In their research, Luxton-Reilly et al. [58] state that most assessments used in formal examinations combine numerous heterogeneous concepts, resulting in complex and difficult tasks. To develop tasks to determine a student's mastery of particular concepts, the Luxton-Reilly et al. 2017 ITiCSE working group decomposed complex assessments into atomic conceptual elements which can be assessed independently. Their work, which extends the McCracken et al. [62] research, shows that a single code-writing task often involves a plethora of conceptual knowledge.

Duran et al. [30] define these atomic elements as plans and subplans that can be extracted from concrete programs by analyzing the relationship of syntactic and semantic elements in the code and the respective cognitive actions leaners need to perform to comprehend the program. Our work uses Duran et al. model to provide cues on how to decompose a *ProgComp* task into sub-tasks and fit them in the Block Model. What becomes evident is that comprehending code too can be decomposed into multiple facets, each of which can be practiced independently.

The LT for *ProgComp* defines a spectrum of activities that will foster programming comprehension using as many levels in the Block Model as desired by the instructor. Different levels usually will use different activities (see section 5 for examples) that are better suited to achieve the desired learning outcome. Creating a LT is an iterative process where the instructor evaluate leaner's prior-knowledge in a particular context (e.g. using tests [71] or self-evaluation instruments [28, 29]) to identify a sub learning-outcome appropriate to learners' needs (extracted from the main task learning outcome), match this learning outcome to a given Block Model level and use an appropriate activity to foster *ProgComp* at that level. The process repeats until the instructor is satisfied with the granularity of the LT (the number of activities in different levels) or if the LT reaches the lowest level of complexity in the Block Model (the sub-task is already simple enough, e.g. uses an Atomic-Text-Surface activity) and no further refinements are required.

LT could be used by instructors in two different ways, depending on their goal. In the first one, the instructor iterates through one task's learning outcomes and evaluate if students' needs, prior knowledge, and granularity will be achieved with a proposed spectrum of activities. If tasks are too easy or too difficult the instructor can further decompose the tasks until a saturation point is reached. In a second way, the instructor follows an already defined learning trajectory, using the planned tasks to identify gaps in the existing set of activities and integrate new activities where needed. The LT could also work in tandem with diagnosing tools, where learners' difficulties in a particular level of the Block Model could be mitigated by using the appropriate activities.

```
1  int[] A = {20,24,23,35,30,35};
2  int c=1;
3  int b=l[0];
4  for( int i=1; i<A.length; i++ ){
5      if( A[i]>b ){
6        b=A[i];
7        c=1;
8      } else {
9          if( A[i]==b ){
10             c++;
11         }
12     }
13 }
14 System.out.println(c);
```

Listing 1: Activity example: summarise the goal of the following program.

In the next section, we provide an example of a walkthrough of the development of an LT where a task may seem too challenging for some learners or have some implicit assumptions that may not match the teacher's cohort. In this example, we will show how we can unpack a complex task into a set of possible class activities or support activities.

6.2 Using the BM to develop a trajectory

As an example of LT, we take a typical comprehension task, to summarise the goal of a program in a short sentence (i.e. an "explain in plain English" task, as described in section 5), and show how it can be decomposed into subtasks, each fitting into one of the levels in the Block Model matrix.

We consider the following problem: *Given an array A of temperature measurements (in degrees Celsius), summarize (in words) the goal of the Java program presented in Listing 1.*

A correct solution for this task would be a sentence similar to *"print the frequency of the highest temperature in the dataset"*. However, for learners to achieve the correct solution, many different sub-tasks of program comprehension have to be performed: comprehend syntactic elements of the programming language, comprehend the behavior (semantics) of these elements in the code, and comprehend the goal of each particular element in the code. These subtasks are merged to create increasingly complex plans, moving up in the Block Model, until the highest level plan which is the overall goal of the program itself.

As discussed in previous sections, the summarizing activity (Macro-Function level) might overwhelm learners without sufficient practice or prior knowledge. Therefore, we decomposed the task according to the 12 levels of the Block Model table. For each level, we used our PCK knowledge and the *ProgComp* activities and experiences from the teacher interviews to identify an activity which would most appropriately practice the corresponding aspect of *ProgComp*. We worked out an entire example, finding an activity for every level. Before describing the activities, we note that we did this for a second task format (to find good variable names for the variables in a program, picking from a list of suggestions), and were not able to find related, simpler, exercises for all the Block Model levels. Hence, the aim here is to show how a more complex task can be decomposed into simpler activities, rather than to suggest that one should be able to find a simpler activity for *all* Block Model levels.

Our decomposition of the program in Listing 1 is given in Figure 6. Considering the Block row in Figure 6, the Text Surface level now asks the learner to identify a *section* of code, consisting of a few lines, and with a higher-level naming — *the code belonging to the else statement*. While a student may do this quite mechanically, initially, by reading through line by line from else to the appropriate closing brace, with practice they will be able to abstract over the detail, and *see* the block of code as a unit.

An instructor can assign any of the *ProgComp* sub-tasks in the BM to a learner. If a learner is unable to complete the task, this can indicate that their knowledge gap or fragile knowledge resides in this or a prerequisite block, so the instructor could use activities in the previous levels of the BM to direct the learner to activities that could improve *ProgComp* and close their knowledge gap. A learner who can complete a task should be challenged with a task residing above or next to the currently accomplished one. In each case, there are many types of *ProgComp* tasks which an instructor can select from. We made a template to describe these tasks as well as what can be done to adjust them to make them more/less difficult.

6.3 Linking WG Outcomes to Practitioners' Views

The analysis of learning tasks with e.g. regard to their prerequisites, as well as their arrangement and hints for possible learning trajectories should help teachers (pre-service and in-service) to develop their PCK of teaching programming. In this final section we will described how these two outcomes relate to practitioners views and needs.

As one teacher phrased it: "we just give them something to understand, a program, we just don't tell them you know, break this down." The work done here can inform teachers how to break down a programming task, and also concrete examples and learning activities to support students also in solving those sub-tasks. This enriches the Block model in a way that it gets more approachable for teaching practice.

The need to teaching ProgComp. : As seen on related works and some interviews, the idea of including *ProgComp* into teaching programming is quite new, but is slowly spreading. As one interviewee put it:

> In the past, I always relied on the idea that it was my primary goal to teach students how to write programs rather than how to understand them. I somehow assumed that the latter skill would come as a consequence of the former. [...] Although the ability to write programs requires an understanding of the state-transition machine that works behind the scenes, the ability to read and understand programs has some additional complication on its own right, and requires some specific approaches and tricks that go beyond those that are required to write programs.

Another teacher was more aware of the BM and wanted to foster *ProgComp*, however could not find practical ways to do so"

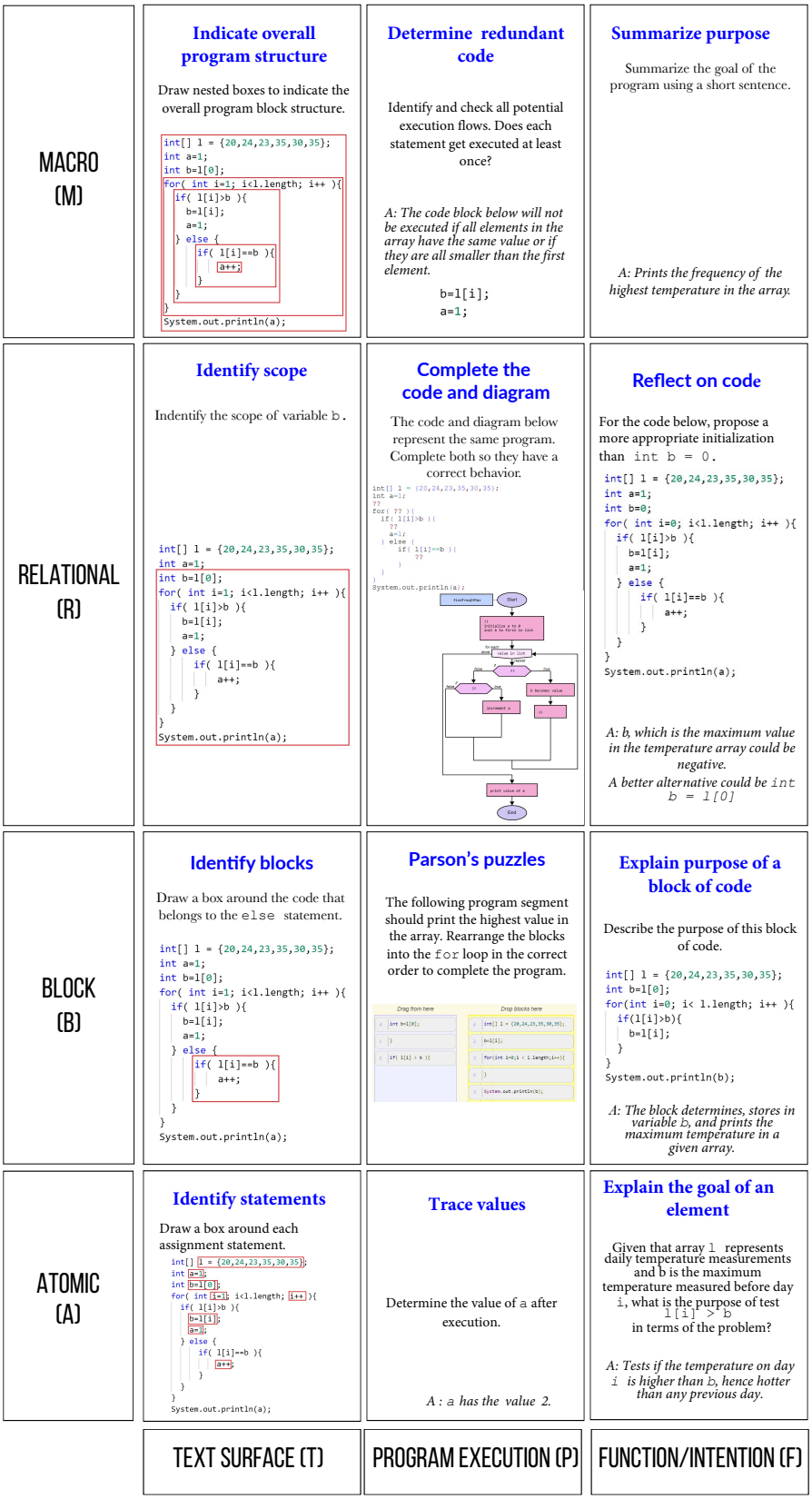

Figure 6: Exercise decomposition of the Listing 1 program. In each of the BM levels, the title (blue) describes the goal of the activity. Below the title there is an example statement of the activity, followed by the answer (A) of the activity.

> so I was studying about this model, the block model and ehm.. [...] but I never really understood how I would practise it as a teacher. [...] So what I would like is to have this model translated to exercises and tasks.

As shown in Section 4, most practitioners (with a few exceptions) provided partial definitions of *ProgComp* with emphasis on "ability to read code". However, their learning objectives and task descriptions provided a good coverage of the Program Execution (P) and Function (F) dimensions.

Progressions within the Block Model. Chunking and the ability to automate *schemas* is one of the most important mechanisms that allow students to progress in course content, moving upwards in the BM matrix towards more complex activities. One of our interview participants provides a good example of *upwards* progression from atoms to blocks:

> Most students will understand the low-level effect of single instructions, but fail to be able to promote this understanding to more complex structures. For example, when they get exposed to a for-loop for the first time, it is good that they understand the exact order in which things take place (initialization, test, loop body, update, test, etc.) but then they should move on and think of the whole structure more abstractly. In a way, they should forget the details and look at the overall effect of the loop as a whole.

On the other hand, phenomenographic investigations (see section 6) and instructional experience suggest that typical learning processes might progress from left to right in the BM matrix. The difficulty to progress rightwards (from Program-Execution to Function) is testified by another teacher as follows:

> Students tend to explain how a program works in terms of operations of the notional machine rather than in connection with the problem to be solved.

That is, they have problems to infer the purpose of the program from the mechanics in the text surface and program execution.

However, to the best of our knowledge, there are no studies that clarify if the knowledge and skills about the three dimensions are developed in sequence, or (maybe partly) in parallel. According to earlier work from Pennington [73], for instance, the type of knowledge inferred, and hence the program's understanding achieved, depend on the task at hand (e.g. debugging, refactoring, etc.).

7 CONCLUSIONS AND FUTURE WORK

The working group report has two major goals: (g1) to collect and define learning activities that explicitly address key components of program comprehension and (g2) to define possible learning trajectories that will guide teachers as they select and sequence those learning activities in their CS0/CS1/CS2 or K-12 courses.

Learning Activities for ProgComp*.* Section 5 has presented more than 60 different learning tasks/activities for *ProgComp*. Tasks have been mapped into cells of the BM matrix according to the intended learning goals. Figure 6, in particular, provides an overview of such classification, based on one example.

This list of learning tasks gives a first overview on different learning activities. In appendix C a sheet is shown, that outlines a comprehensive description of the learning activities.

To provide such a comprehensive description for each of the listed learning activities remains future work.

The work presented here gives some tools to tackle such a task in terms of a methodology or approach for an educational analysis of *ProgComp* learning activities. This approach has been outlined in sections 3.2 and following, as well as in section 6.2. In section 3.2 three learning tasks have been analysed with regard to their specific role for *ProgComp*. Such general analysis forms the basis to describe them according to the template in appendix C.

In addition, the general analysis of task types can be done accurately using one specific example, as shown in section 6.2, where the implications of the code in Listing 1 have been discussed in detail.

Interpretation: The most striking effect of this approach is a changed perspective on learning tasks and learning trajectories - based on the simple question: What does it mean, if a learner isn't able to solve the *ProgComp* task at hand? The outlined approach to describe tasks is also a methodology to answer this question: What can a teacher do in case of learning problems? One generic answer would be to provide the learner with easier tasks, and give the possibility for more practice with such easier tasks. Well, one difficult question to answer would b: what are *easier* tasks? And often such an answer is based on thinking about the task — with the approach at hand this is possible, but the fine grained analysis opens room to discern specific, probably individual learning issues of the learner — that is, there is room for several different types of easier learning tasks that would aim at different prerequisites. As described by Clear [16] the overall problem in learning programming (and *ProgComp* as related subtask) is the need to zoom-in and zoom-out:

> a process by which we move successively from the part to the whole and back again, to progressively develop a coherent and consistent conception of the software system we are developing or attempting to comprehend.

The 12 cells of the Block model describes the different steps or moves in this zooming in and out, and probably learners can have fragile, moderate, or deep knowledge for each cell, and need in correspondence different time on different learning tasks [16]. With the resources provided by this report, a teacher is supported in understanding what pertains to cells that need specific attention, and in ideas for learning activities that foster the skills and knowledge needed in connection with the specific cells, i.e. steps along a learning path.

So, in addition, the role of such learning activities was discussed in this report from two perspectives: its connection to "learning to program" in the literature, see Section 2.2; and the views of practitioners as elicited in interviews we have conducted, see Section 4.

It is difficult to give a comprehensive summary and interpretation of these, but we want to highlight two insights. First, *ProgComp* is a complex issues which needs specific learning activities. Among these, tracing tasks are interesting as learning activities for novices - they are probably really important to lay a foundation for more

complex steps, and thus it seems useful to train them so that it becomes automated. This might be a universal learning stage for all learners in which focusing on tracing activities seems to be important, as e.g. argued by Lister.

The other aspect comes from the interviews with practitioners: From this it seems that the text surface dimension is not so important, or: easier to master and hence doesn't need to get that much attention as the other learning steps.

Learning Trajectories for ProgComp and Programming. To help practitioners in designing successful learning trajectories in their classes is the second major goal of this report — but we didn't present one or more exemplary learning trajectories — why?

In order to be flexible and support different coverage of *ProgComp* depending on their learning needs, we focused on a different outcome: a toolset that supports customised design of learning trajectories, as cited before. Figure 6 outlines the core of this approach: A detailed analysis of the prerequisites of a task - which if done for all tasks gives an idea of possible learning trajectories.

Based on the literature review and the practitioner interviews, such learning trajectories are likely to have a general direction from the lower left corner of the Block Model to the upper right corner.

Future work. To develop comprehensive descriptions for each task is future work, and could be done in connection to (existing) learning repositories (see appendix C for a template that could be used when submitting task to such repositories).

We have presented an overview of the interviews with practitioners. There is however more interesting information we want to explore further in a fine grained analysis.

A major contribution in our point of view is the method for a fine-grained analysis of ProgComp tasks together with a collection of many different learning tasks. It seem very valuable to develop a teacher Professional Development (PD) kit to promote the use of *ProgComp* tasks in K-12, and to inform them on the ways to develop their own trajectories.

While we have collected so far a little more than 60 different learning tasks for *ProgComp*, it still seem useful to mine other tasks repositories, to analyse their program comprehension coverage. Together with such an overview of different approaches and learning tasks for ProgComp it is also useful to analyse in more detail the relevancy and need for each type. Probably there are some hot spots that need more attention than others.

Finally, these future steps will provide fertile ground to investigate possible learning trajectories in the classroom and towards more effective learning activities and progressions.

REFERENCES

[1] Owen Astrachan, Garrett Mitchener, Geoffrey Berry, and Landon Cox. 1998. Design patterns: an essential component of CS curricula. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*. ACM, New York, NY, USA, 153–160. https://doi.org/10.1145/273133.273182

[2] Owen Astrachan and Eugene Wallingford. 1998. Loop Patterns. In *Proceedings of the Fifth Pattern Languages of Programming Conference (PLoP'98)*. https://users.cs.duke.edu/~ola/patterns/plopd/loops.html

[3] E. Barendsen, V. Dagiene, M. Saeli, and C. Schulte. 2014. Eliciting computing science teachers' PCK using the Content Representation format: Experiences and future directions. In *Proceedings of ISSEP*. 71–82.

[4] Marjahan Begum, Jacob Nørbjerg, and Torkil Clemmensen. 2018. Strategies of Novice Programmers. In *Proceedings of the 41st Information Systems Research Seminar in Scandinavia: Digital Adaptation, Disruption and Survival (IRIS2018) (IRIS)*. http://hdl.handle.net/10398/9686

[5] Richard Bornat, Saeed Dehnadi, and David Barton. 2012. Observing Mental Models in Novice Programmers. In *Proc. 24th Annual Workshop of the Psychology of Programming Interest Group*. Article 6, 7 pages.

[6] Ruven Brooks. 1977. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies* 9, 6 (1977), 737–751. https://doi.org/10.1016/S0020-7373(77)80039-4

[7] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

[8] Christine Bruce, Lawrence Buckingham, John Hynd, Camille Mcmahon, Mike Roggenkamp, and Ian Stoodley. 2004. Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education* 3 (2004), 143–160.

[9] Malte Buchholz, Mara Saeli, and Carsten Schulte. 2013. PCK and reflection in computer science teacher education. *ACM International Conference Proceeding Series* (11 2013). https://doi.org/10.1145/2532748.2532752

[10] Teresa Busjahn and Carsten Schulte. 2013. The Use of Code Reading in Teaching Programming. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13)*. ACM, New York, NY, USA, 3–11. https://doi.org/10.1145/2526968.2526969

[11] Pauli Byckling and Jorma Sajaniemi. 2006. A role-based analysis model for the evaluation of novices' programming knowledge development. In *ICER '06: Proceedings of the second international workshop on Computing education research*. ACM, 85–96.

[12] Ricardo Caceffo, Steve Wolfman, Kellogg S. Booth, and Rodolfo Azevedo. 2016. Developing a Computer Science Concept Inventory for Introductory Programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 364–369. https://doi.org/10.1145/2839509.2844559

[13] Ibrahim Cetin. 2015. Student's Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective. *Canadian Journal of Science, Mathematics and Technology Education* 15, 2 (Feb. 2015), 155–170. https://doi.org/10.1080/14926156.2015.1014075

[14] Yves Chevallard and Marianna Bosch. 2014. Didactic transposition in mathematics education. *Encyclopedia of mathematics education* (2014), 170–174.

[15] Michelene T. H. Chi and Ruth Wylie. 2014. The ICAP Framework: Linking Cognitive Engagement to Active Learning Outcomes. *Educational Psychologist* 49, 4 (2014), 219–243. https://doi.org/10.1080/00461520.2014.965823 arXiv:https://doi.org/10.1080/00461520.2014.965823

[16] Tony Clear. 2012. The Hermeneutics of Program Comprehension: A 'Holey Quilt' Theory. *ACM Inroads* 3, 2 (June 2012), 6–7. https://doi.org/10.1145/2189835.2189837

[17] Douglas H. Clements and Julie Sarama. 2009. Learning trajectories in early mathematics–sequences of acquisition and teaching. *Encyclopedia of language and literacy development* (2009), 1–7.

[18] Malcolm Corney, Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. 2014. 'Explain in Plain English' Questions Revisited: Data Structures Problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 591–596. https://doi.org/10.1145/2538862.2538911

[19] Malcolm Corney, Raymond Lister, and Donna Teague. 2011. Early Relational Reasoning and the Novice Programmer: Swapping As the "Hello World" of Relational Reasoning. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114 (ACE '11)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 95–104. http://dl.acm.org/citation.cfm?id=2459936.2459948

[20] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 164–172. https://doi.org/10.1145/3105726.3106190

[21] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice Rationales for Sketching and Tracing, and How They Try to Avoid It. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 37–43. https://doi.org/10.1145/3304221.3319788

[22] Michael De Raadt. 2008. *Teaching programming strategies explicitly to novice programmers*. Ph.D. Dissertation. University of Southern Queensland.

[23] Michael De Raadt, Richard Watson, and Mark Toleman. 2006. Chick sexing and novice programmers: explicit instruction of problem solving strategies. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., 55–62.

[24] Michael De Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and assessing programming strategies explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education-Volume 95*. Australian Computer Society, Inc., 45–54.

[25] Lionel E. Deimel, Jr. 1985. The Uses of Program Reading. *SIGCSE Bull.* 17, 2 (June 1985), 5–14. https://doi.org/10.1145/382204.382524

[26] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a new exam question: Parsons problems. In *Proceedings of the fourth international workshop on computing education research*. ACM, 113–124.

[27] Benedict du Boulay. 1986. Some Difficulties of Learning to Program. *J. of Educational Comput. Research* 2, 1 (1986), 57–73.

[28] Rodrigo Duran, Jan-Mikael Rybicki, Arto Hellas, and Sanna Suoranta. 2019. Towards a Common Instrument for Measuring Prior Programming Knowledge. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 443–449. https://doi.org/10.1145/3304221.3319755

[29] Rodrigo Duran, Jan-Mikael Rybicki, Juha Sorva, and Arto Hellas. 2019. Exploring the Value of Student Self-Evaluation in Introductory Programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. ACM, New York, NY, USA, 121–130. https://doi.org/10.1145/3291279.3339407

[30] 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.

[31] J. Philip East, S. Rebecca Thomas, Eugene Wallingford, Walter Beck, and Janet Drake. 1996. Pattern Based Programming Instruction. In *1996 Annual Conference*. ASEE Conferences, Washington, District of Columbia.

[32] Anna Eckerdal and Anders Berglund. 2005. What does it take to learn 'programming thinking'?. In *ICER '05: Proceedings of the first international workshop on Computing education research*. ACM, New York, NY, USA, 135–142. https://doi.org/10.1145/1089786.1089799

[33] Carol Edmondson. 2009. Proglets for First-year Programming in Java. *SIGCSE Bull.* 41, 2 (June 2009), 108–112. https://doi.org/10.1145/1595453.1595486

[34] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research*. ACM, 20–29.

[35] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 211–216. https://doi.org/10.1145/2839509.2844556

[36] Scott Freeman, Sarah L Eddy, Miles McDonough, Michelle K Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences* 111, 23 (2014), 8410–8415.

[37] Tina Götschi, Ian Sanders, and Vashti Galpin. 2003. Mental Models of Recursion. In *Proc. of the 34th SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA, 346–350.

[38] Orit Hazzan, Tami Lapidot, and Noa Ragonis. 2011. *Guide to Teaching Computer Science: An Activity-Based Approach* (1st ed.). Springer Publishing Company, Incorporated.

[39] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. How Do Students Solve Parsons Programming Problems?: An Analysis of Interaction Traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. ACM, New York, NY, USA, 119–126. https://doi.org/10.1145/2361276.2361300

[40] Matthew Hertz and Maria Jump. 2013. Trace-based Teaching in Early Programming Courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 561–566. https://doi.org/10.1145/2445196.2445364

[41] Cruz Izu, Claudio. Mirolo, and Amali Weerasinghe. 2018. Novice Programmers' Reasoning About Reversing Conditional Statements. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, USA, 646–651. https://doi.org/10.1145/3159450.3159499

[42] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, New York, NY, USA, 251–259. https://doi.org/10.1145/2960310.2960324

[43] Jeffrey D Karpicke and Janell R Blunt. 2011. Retrieval practice produces more learning than elaborative studying with concept mapping. *Science* 331, 6018 (2011), 772–775.

[44] Walter Kintsch. 1998. *Comprehension: A paradigm for cognition*. New York: Cambridge.

[45] Matthias Kramer, Mike Barkmin, and Torsten Brinda. 2019. Identifying Predictors for Code Highlighting Skills: A Regressional Analysis of Knowledge, Syntax Abilities and Highlighting Skills. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 367–373. https://doi.org/10.1145/3304221.3319745

[46] Peter Kroes. 2012. *Technical Artefacts: Creations of Mind and Matter – A Philosophy of Engineering Design*. Springer, Dordrecht, Heidelberg, New York, London. https://doi.org/10.1007/978-94-007-3940-6

[47] Marja Kuittinen and Jorma Sajaniemi. 2004. Teaching roles of variables in elementary programming courses. In *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*. ACM, 57–61.

[48] Sarah K. Kummerfeld and Judy Kay. 2003. The Neglected Battle Fields of Syntax Errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20 (ACE '03)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 105–111. http://dl.acm.org/citation.cfm?id=858403.858416

[49] Clifton Kussmaul. 2012. Process oriented guided inquiry learning (POGIL) for computer science. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 373–378.

[50] Marcia C. Linn. 1985. The Cognitive Consequences of Programming Instruction in Classrooms. *Educational Researcher* 14, 5 (May 1985), 14–29. https://doi.org/10.3102/0013189X014005014

[51] Raymond Lister. 2007. The Neglected Middle Novice Programmer: Reading and Writing without Abstracting. In *Proceedings of the 20th Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07)*, S. Mann and N. Bridgeman (Eds.). 133–140.

[52] Raymond Lister. 2011. Concrete and other neo-piagetian forms of reasoning in the novice programmer. *Conf. Res. Pract. Inf. Technol. Ser.* 114 (2011), 9–18.

[53] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '04)*. ACM, New York, NY, USA, 119–150. https://doi.org/10.1145/1044550.1041673

[54] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. ACM, New York, USA, 118–122. https://doi.org/10.1145/1140124.1140157

[55] Joanne Lobato and C David Walters. 2017. *A Taxonomy of Approaches to Learning Trajectories and Progressions*. NCTM, 74–101.

[56] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proc. 4th Int. Workshop on Computing Education Research (ICER '08)*. ACM, New York, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[57] John Loughran, Pamela Mulhall, and Amanda Berry. 2004. In search of pedagogical content knowledge in science: Developing ways of articulating and documenting professional practice. *Journal of Research in Science Teaching* 41, 4 (2004), 370–391.

[58] Andrew Luxton-Reilly, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Simon, and Jacqueline Whalley. 2017. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '17)*. ACM, New York, NY, USA, 47–69. https://doi.org/10.1145/3174781.3174784

[59] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 16.

[60] Richard E. Mayer. 1981. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13, 1 (March 1981), 121–141. https://doi.org/10.1145/356835.356841

[61] Robert McCartney, Jan Erik Moström, Kate Sanders, and Otto Seppälä. 2004. Questions, Annotations, and Institutions: observations from a study of novice programmers. In *Proceedings of the 4th Koli Calling International Conference on Computing Education Research (Koli Calling '04)*. ACM, New York, USA, 11– 19.

[62] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bull.* 33, 4 (Dec. 2001), 125–180. https://doi.org/10.1145/572139.572181

[63] Tanya J. McGill and Simone E. Volet. 1997. A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education* 29, 3 (1997), 276–297. https://doi.org/10.1080/08886504.1997.10782199

[64] Jeroen J. G. Van Merriënboer. 1990. Strategies for Programming Instruction in High School: Program Completion vs. Program Generation. *Journal of Educational Computing Research* 6, 3 (1990), 265–285. https://doi.org/10.2190/4NK5-17L7-TWQV-1EHL arXiv:https://doi.org/10.2190/4NK5-17L7-TWQV-1EHL

[65] Claudio Mirolo. 2010. Learning (Through) Recursion: A Multidimensional Analysis of the Competences Achieved by CS1 Students. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10)*. ACM, New York, NY, USA, 160–164. https://doi.org/10.1145/1822090.1822136

[66] Claudio Mirolo. 2012. Is Iteration Really Easier to Learn Than Recursion for CS1 Students?. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. ACM, New York, NY, USA, 99–104. https://doi.org/10.1145/2361276.2361296

[67] Claudio Mirolo and Cruz Izu. 2019. An Exploration of Novice Programmers' Comprehension of Conditionals in Imperative and Functional Programming. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 436–442. https://doi.org/10.1145/3304221.3319746

[68] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*. ACM, New York, NY, USA, 151–155. https://doi.org/10.1145/1268784.1268830

[69] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in Plain English' Questions: Implications for Teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, USA, 385–390. https://doi.org/10.1145/2157136.2157249

[70] Greg L. Nelson, Benjamin Xie, and Andrew J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/3105726.3106178

[71] 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

[72] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *ACE '06: Proceedings of the 8th Australian conference on Computing education*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157–163.

[73] Nancy Pennington. 1987. Comprehension Strategies in Programming. In *Empirical Studies of Programmers: Second Workshop*, Gary M. Olson, Sylvia Sheppard, and Elliot Soloway (Eds.). Ablex Publishing Corp., Norwood, NJ, USA, 100–113.

[74] Viera K. Proulx. 2000. Programming Patterns and Design Patterns in the Introductory Computer Science Course. In *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education (SIGCSE '00)*. ACM, New York, NY, USA, 80–84. https://doi.org/10.1145/330908.331819

[75] Iyad Rahwan, Manuel Cebrian, Nick Obradovich, Josh Bongard, Jean-François Bonnefon, Cynthia Breazeal, Jacob W Crandall, Nicholas A Christakis, Iain D Couzin, Matthew O Jackson, et al. 2019. Machine behaviour. *Nature* 568, 7753 (2019), 477.

[76] Kathryn M. Rich, T. Andrew Binkowski, Carla Strickland, and Diana Franklin. 2018. Decomposition: A K-8 Computational Thinking Learning Trajectory. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. ACM, New York, NY, USA, 124–132. https://doi.org/10.1145/3230977.3230979

[77] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, and Diana Franklin. 2019. A K-8 Debugging Learning Trajectory Derived from Research Literature. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 745–751. https://doi.org/10.1145/3287324.3287396

[78] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, Cheryl Moran, and Diana Franklin. 2017. K-8 Learning Trajectories Derived from Research Literature: Sequence, Repetition, Conditionals. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 182–190. https://doi.org/10.1145/3105726.3106166

[79] Robert S Rist. 1989. Schema creation in programming. *Cognitive Science* 13, 3 (1989), 389–414.

[80] Robert S Rist. 1995. Program structure and design. *Cognitive Science* 19, 4 (1995), 507–561.

[81] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education* 13, 2 (2003), 137–172.

[82] Janine Rogalski and Renan Samurçay. 1990. Acquisition of programming knowledge and skills. In *Psychology of programming*, J.-M. Hoc (Ed.). Academic Press.

[83] Mara Saeli. 2012. *Teaching programming for secondary school: a pedagogical content knowledge based approach.* Ph.D. Dissertation. Eindhoven University of Technology, Eindhoven, The Netherlands.

[84] Jorma Sajaniemi, Mordechai Ben-Ari, Pauli Byckling, Petri Gerdt, and Yevgeniya Kulikova. 2006. Roles of Variables in Three Programming Paradigms. *Computer Science Education* 16, 4 (December 2006), 261–279.

[85] Jorma Sajaniemi and Marja Kuittinen. 2005. An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education* 15, 1 (2005), 59–82. https://doi.org/10.1080/08993400500056563

[86] Ian Sanders, Vashti Galpin, and Tina Götschi. 2006. Mental models of recursion revisited. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. ACM, New York, USA, 138–142.

[87] Kate Sanders, Marzieh Ahmadzadeh, Tony Clear, Stephen H. Edwards, Mikey Goldweber, Chris Johnson, Raymond Lister, Robert McCartney, Elizabeth Patitsas, and Jaime Spacco. 2013. The Canterbury QuestionBank: Building a Repository of Multiple-choice CS1 and CS2 Questions. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports (ITiCSE -WGR '13)*. ACM, New York, NY, USA, 33–52. https://doi.org/10.1145/2543882.2543885

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

[89] Tamarisk Lurlyn Scholtz and Ian Sanders. 2010. Mental Models of Recursion: Investigating Students' Understanding of Recursion. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10)*. ACM, New York, NY, USA, 103–107. https://doi.org/10.1145/1822090.1822120

[90] Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535

[91] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. 2010. An Introduction to Program Comprehension for Computer Science Educators. In *Proceedings of the 2010 ITiCSE Working Group Reports (ITiCSE-WGR '10)*. ACM, New York, NY, USA, 65–86. https://doi.org/10.1145/1971681.1971687

[92] Linda Seiter and Brendan Foreman. 2013. Modeling the Learning Progressions of Computational Thinking of Primary Grade Students. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*. ACM, New York, NY, USA, 59–66. https://doi.org/10.1145/2493394.2493403

[93] Sue Sentance and Jane Waite. 2017. PRIMM: Exploring Pedagogical Approaches for Teaching Text-based Programming in School *(WiPSCE '17)*. ACM, 113–114. https://doi.org/10.1145/3137065.3137084

[94] Amal Shargabi, Syed Aljunid, Muthukkaruppanan Annamalai, Shuhaida M Shuhidan, and Abdullah M Zin. 2015. Tasks that can improve novices' program comprehension. In *2015 IEEE Conference on e-Learning, e-Management and e-Services (IC3e)*. 32–37. https://doi.org/10.1109/IC3e.2015.7403482

[95] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L. Whalley. 2008. Going SOLO to Assess Novice Programmers. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08)*. ACM, New York, NY, USA, 209–213.

[96] Lee S. Shulman. 1986. Those who understand: Knowledge growth in teaching. *Educational Researcher* 15, 2 (1986), 4–14. https://doi.org/10.3102/0013189X015002004

[97] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. https://doi.org/10.1145/6592.6594

[98] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: an empirical study. *Commun. ACM* 26, 11 (Nov. 1983), 853–860. https://doi.org/10.1145/182.358436

[99] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 8:1–8:31 pages. https://doi.org/10.1145/2483710.2483713

[100] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[101] James C Spohrer, Elliot Soloway, and Edgar Pope. 1985. A goal/plan analysis of buggy Pascal programs. *Human–Computer Interaction* 1, 2 (1985), 163–207.

[102] Leigh Ann Sudol-DeLyser, Mark Stehlik, and Sharon Carver. 2012. Code Comprehension Problems As Learning Events. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. ACM, New York, NY, USA, 81–86. https://doi.org/10.1145/2325296.2325319

[103] Donna Teague and Raymond Lister. 2014. Programming: Reading, Writing and Reversing. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE '14)*. ACM, New York, USA, 285–290. https://doi.org/10.1145/2591708.2591712

[104] Donna Teague, Raymond Lister, and Alireza Ahadi. 2015. Mired in the Web: Vignettes from Charlotte and Other Novice Programmers.. In *ACE*. 165–174.

[105] Errol Thompson, Jacqueline Whalley, Raymond Lister, and Beth Simon. 2006. Code Classification as Learning and Assessment Exercise for Novice Programmers. In *Proceedings of the 19th Annual Conference of the National Advisory Committee on Computing Qualifications*, S. Mann and N. Bridgeman (Eds.). NACCQ in cooperation with ACM SIGCSE, 291–298.

[106] Michael Thuné and Anna Eckerdal. 2009. Variation theory applied to students' conceptions of computer programming. *European Journal of Engineering Education* 34, 4 (2009), 339–347. https://doi.org/10.1080/03043790902989374

[107] Niko Tinbergen. 1963. On aims and methods of ethology. *Zeitschrift für tierpsychologie* 20, 4 (1963), 410–433.

[108] Yasushi Umeda and Tetsuo Tomiyama. 1997. Functional reasoning in design. *IEEE expert* 12, 2 (1997), 42–48.

[109] A. Marie Vans, Anneliese von Mayrhauser, and Gabriel Somlo. 1999. Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies* 51, 1 (1999), 31–70. https://doi.org/10.1006/ijhc.1999.0268

[110] Eugene Wallingford. 1996. Toward a first course based on object-oriented patterns. *ACM SIGCSE Bulletin* 28, 1 (1996), 27–31.

[111] David Weintrop, Heather Killen, Talal Munzar, and Baker Franke. 2019. Block-based Comprehension: Exploring and Explaining Student Outcomes from a Read-only Block-based Exam. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 1218–1224. https://doi.org/10.1145/3287324.3287348

[112] Jacqueline Whalley and Nadia Kasto. 2013. Revisiting Models of Human Conceptualisation in the Context of a Programming Examination. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136 (ACE '13)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 67–76. http://dl.acm.org/citation.cfm?id=2667199.2667207

[113] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 243–252. http://dl.acm.org/citation.cfm?id=1151869.1151901

[114] Richard T. White and Richard F. Gunstone. 1992. *Probing understanding*. Falmer, London and New York.

[115] Susan Wiedenbeck and Vennila Ramaligan. 1999. Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies* 51, 1 (1999), 71 – 87. https://doi.org/10.1006/ijhc.1999.0269

[116] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J. Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235

[117] Benjamin Xie, Greg L. Nelson, and Andrew J. Ko. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 344–349. https://doi.org/10.1145/3159450.3159527

APPENDIX – ADDITIONAL BACKGROUND KNOWLEDGE

A PLANS AND PLAN COMPOSITION

This appendix describes plans and plan-composition strategies for the unfamiliar reader using an example.

Consider the following problem:

> Write a program that outputs the number of occurrences of the largest value in a given array of temperature measurements (in degrees Celsius).

Solving this problem requires three general tasks:

(1) determining the maximum.
(2) determining its frequency in an array.
(3) printing the result.

These three goals appear in many problems, and expert programmers have standard approaches (plans) in their repertoire for solving these [97]: 1) the Maximum plan, 2) Count plan, and 3) Output plan [24].

Thus, a solution can be written by combining those 3 plans The problem at hand can be solved by abutting the plans, i.e. merely executing the three in sequential order. In terms of tailoring, the programmer needs to ensure that the maximum is initialized to the first value in the array, and not zero for example because the values may all be negative values. In Java the result could look as shown in Listing 2.

```
1  int[] l = {20,24,23,35,30,35};
2
3  int b=l[0];
4  for( int i=1; i<l.length; i++ ){
5      if( l[i]>b ){
6        b=l[i];
7          }
8  }
9
10 int a=0;
11 for( int i=1; i<l.length; i++ ){
12   if( l[i]==b ){
13       a++;
14     }
15 }
16
17 System.out.println(a);
```

Listing 2: Abutment of plans: Maximum & Count & Output

The complexity of this approach can be measured by the number of components and the way they are connected, relative to the knowledge the student is expected to have at that particular moment [30, 42].

If we give the problem above to students towards the end of a CS1 course in which the three elementary plans (Maximum plan, Count plan, and Output plan) have already been taught, using the SOLO taxonomy [42] we will consider such elementary plans to reside at the uni-structural level . The abutment of two or more elementary plans is considered to coincide with a multi-structural level of understanding, as concatenation is the simplest way to

compose plans or lines. Thus as answer such as that shown in figure 2 is measured to be at the multi-structural level. In this solution, one can recognize the code 'chunk' for determining the maximum value followed by code block for determining the frequency of that value, and then outputting the value. Recognizing the chunks and abstracting from the details allows a reader to understand the meaning of the code as an entity (i.e. "see the trees through the forest") and construct a general description of its purpose as a whole.

The same goal can also be achieved by merging the two plans, since the data will be passed through only once. In Java the result could look as shown in Listing 3. In this case, the Maximum and Count plans have been merged. As an effect, recognizing the constituent plans becomes more difficult. The reader must search for beacons in the code, and relate these to the knowledge about individual plans in his or her repertoire. Furthermore, it requires an understanding about how plans can be merged in general.

The code shown in figure 3 will be classified as relational because merging 2 or more elementary plans is non trivial and requires a deeper understanding on how the plan fit together.

```
1  int[] l = {20,24,23,35,30,35};
2  int a=1;
3  int b=l[0];
4  for( int i=1; i<l.length; i++ ){
5      if( l[i]>b ){
6          b=l[i];
7          a=1;
8      } else {
9          if( l[i]==b ){
10             a++;
11         }
12     }
13 }
14 System.out.println(a);
15 }
```

Listing 3: Merging of plans: Maximum & Count abutted by Output

B INTERVIEW PROTOCOL

At the end of an introductory course, students are expected to have learned the rudiments about the nature, scope, general features of the language and basic methodologies of programming. It is then assumed that they have gained a more comprehensive understanding of small-scale programs, i.e., that they are able both to "read" — can interpret — as well as to "write" — design and develop — short chunks of code aimed at achieving a particular task. However, it is often less clear what the instructors exactly mean by program comprehension at an introductory level, and to what extent this skill is taught and assessed explicitly, not simply taken for granted on the basis of the accomplishment of some related programming activities.

The purpose of this interview is to explore your personal perspective about program comprehension at an introductory level, your teaching practice in order to develop program comprehension, and the major difficulties/misconceptions that may prevent students, according to your experience, from gaining a comprehensive understanding of small-scale programs.

Interview questions

A. Personal information
F/M
Country?
Institution?
What is your academic background?
How long have you been teaching programming?
B. Teaching context
- Instruction level / Student age - select ages as appropriate (pre-tertiary instruction)
primary school: 5 6 7 8 9 10 11
middle school: 11 12 13 14
high school: 14 15 16 17 18 19
university : CS1 CS2 (instruction level)
- Are your students specialising in a computing field? (university)
- How many hours per year are your students taught (attending a class) on computing topics?
- What programming language(s)/environment(s) do you use?
- Why did you choose this/these language(s)/environment(s)?
C. Open questions
C1. What Program Comprehension (ProgComp) means to you and your students
- Explain in a few words what the term"program comprehension' means to you
- What concepts and skills do you want your students to learn in connection with program comprehension?
-What are the major difficulties/misconceptions that conceivably prevent your students from comprehending small programs? Can you explain such difficulties/misconceptions?
C2. How to teach ProgComp
- Do you explicitly teach program comprehension? If so, could you briefly list and describe your instructional strategies to enhance students' program comprehension?
- Do you use any hands-on teaching activities that cover program comprehension?
If yes, a) describe shortly an example of a task you assign to enhance program comprehension
b) mention briefly reasons for assigning this task to enhance program comprehension
If no, briefly explain the reasons why you choose not to (limited time, embedded on examples, other)
C3 How to assess ProgComp
- Briefly describe what aspects of program comprehension you include in your assessment
- Discuss how the assessment is done (when/how/how much weight etc)
C4. Do you have any other suggestions to improve students' program comprehension?

C TEMPLATE TO DESCRIBE PROGRAM COMPREHENSION ACTIVITIES

Name of activity	***Identifying Expressions***
Block Model (trajectory)	*AT (atomic, text surface)*
Pre-requisite CS concept knowledge (What must they already know about CS concepts?)	*Literals, variables, operators* *(might also include function calls, collections, …)*
Pre-requisite PC skills **(What must they already know how to do re: Program comprehension skills?)**	*none*
What instructor provides	*Sample code, definition of expression, several examples*
What student is asked to do	*Read through the sample code and highlight or circle each expression (perhaps using different colors)*
What new thing(s) should students know or be able to do after this exercise? (learning outcomes)	*Recognize that expressions evaluate to a value, may be simple or compound, may evaluate to different types of values, may appear in different elements of the code (on RHS of assignment but not left, in conditional statements, in iterative statements, as parameters to functions, etc.)*
Social	*Students may work in pairs to stimulate engagement and equalize / promote sharing of prior CS concepts*
Engagement level (ICAP) (Chi 2014)	*I - Interactive if group discussion as described above follows the activity (else C- Constructive)*
Notes:	*Suggested to follow with breakout groups, then whole class discussion ;* *Was xyz on line X an expression? Why or why not?* *Where can expressions appear? On the RHS of an assignment statement? On the LHS of an assignment statement? Is a literal an expression? Does a function call evaluate to an expression? Is a single value an expression? Does an assignment statement evaluate to a value? Is that an expression?* *Do expressions occur in for loops? While loops? Conditional statements?* *NOTE: the code sample or samples must support all of these variations in order to address these questions, and should be ordered appropriately*