TLP **19** (2): 204–228, 2019. © Cambridge University Press 2018
doi:10.1017/S1471068418000431 First published online 14 November 2018

Introducing Computer Science to High School Students Through Logic Programming[*]

TIMOTHY T. YUEN
University of Texas at San Antonio, San Antonio, TX, USA
(*e-mail:* timothy.yuen@utsa.edu)

MARITZA REYES
University of Texas at Austin, Austin, TX, USA
(*e-mail:* maritza_reyes@utexas.edu)

YUANLIN ZHANG
Texas Tech University, Lubbock, TX, USA
(*e-mail:* y.zhang@ttu.edu)

submitted 29 September 2017; received 12 August 2018; accepted 15 August 2018

Abstract

This paper investigates how high school students in an introductory computer science (CS) course approach computing in the logic programming (LP) paradigm. This qualitative study shows how novice students operate within the LP paradigm while engaging in foundational computing concepts and skills: students are engaged in a cyclical process of abstraction, reasoning, and creating representations of their ideas in code while also being informed by the (procedural) requirements and the revision/debugging process. As these computing concepts and skills are also expected in traditional approaches to introductory K-12 CS courses, this paper asserts that LP is a viable paradigm choice for high school novices.

KEYWORDS: CS education, high school CS, declarative programming, logic programming, answer set programming

1 Introduction

The debate in paradigm selection in introductory computer science (CS) courses is often split between object-oriented (OO) and procedural paradigms, which also leads to discussions on programming language choice. This choice is difficult as there are no standard languages or paradigms to use in the field. Though both these approaches have been successful in introductory CS courses, some research has also shown only minimal differences when comparing the outcomes between paradigms (Chen *et al.* 2006; Vilner *et al.* 2007). However, research on the teaching of introductory CS courses also reveals the limitations of those approaches (Pears *et al.* 2007; Chakravarty and Keller 2004)

[*] The authors thank Cynthia Perez, Rocky Upchurch, and Edward Wertz for their contributions to this project, and thank Michael Gelfond for valuable discussions and sharing his teaching materials. This work is partially supported by NSF grant CNS-1359359. We thank the anonymous reviewers whose feedback helps to improve the quality of this paper.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

and that the suitability of Java, C/C++ for education is debatable. Often, these discussions of introductory courses are aimed at the university level. However, high school is a critical juncture when CS educators have one last opportunity to engage students in CS and provide adequate preparation before they select their majors and begin university. Thus, this paper emphasizes the importance of introductory CS courses, as well as paradigm/language selection, at the high school level. At this time, more research is needed on how to best teach CS in high schools.

Some research shows that teaching programming to middle and high school students is challenging (Sherin *et al*. 1993). By a recent survey on teaching CS in K-12 using programming (K-12 is a USA school system including Kindergarten, 1st grade (usually 6–7 years old) until 12th grade. Elementary schools usually include 1st to 5th grade, middle schools 6th to 8th grade, and high schools 9th to 12th grade), the majority of education research uses visual programming languages (Lye and Koh 2014). Examples of these languages include Scratch (Resnick *et al*. 2009), Toontalk (Kahn 2004), and Alice (Cooper *et al*. 2000). With these visual languages, students build programs using graphical objects and drag-and-drop interfaces, which significantly reduces the challenges for students to learn the syntax (compared to text-based programming). Another prominent feature of these environments is to encourage and facilitate tinkering and fit the needs of students who prefer *tinkering* to *logic and planning* (Papert 1980, Resnick and Rosenbaum 2013). On the one hand, these languages and environments have been very successful in reaching a large K-12 population. On the other hand, more research is needed to understand how computation thinking occurs as students are tinkering while using visual programming languages (Guzdial 2004; Lye and Koh 2014). While substantial efforts have been made in introducing CS to younger audiences through visual programming languages which are largely procedural or OO based, logic programming (LP)-based approaches have been largely ignored by the K-12 CS education research community (Pears *et al*. 2007; Lye and Koh 2014), despite continuing efforts in both teaching and research on LP.

1.1 LP and CS education

Prolog may be the most well-known instance in which LP was used to teach CS. In particular, Prolog was used to teach children in the 1980s (Kowalski 1982; Kowalski 1987; Nichol *et al*. 1988; Guzdial 2004). As a pioneer, Kowalski (1987) focused on the declarative aspects of Prolog and restricted the use of procedural aspects of Prolog to a minimum. Later, researchers and practitioners found that the procedural aspects of Prolog have been the main source of misconceptions and difficulties (Mendelsohn *et al*. 1990), while the benefits of its declarative aspects were acknowledged. In the last two decades, Prolog has made some appearances: listed in high school curriculum (Scherz and Haberman 1995), and taught to gifted and talented high school students (Stutterheim *et al*. 2013) as well as to general high school and undergraduate students (Levesque 2012; Beux *et al*. 2015) though it does not enjoy the same attention as procedural and OO paradigms. In the last two decades, a breakthrough in LP research was the establishment of the answer set programming (ASP) (Brewka *et al*. 2011; Gelfond and Kahl 2014; Kowalski 2014), a paradigm which has inherited the declarative nature of Prolog while fully removing its procedural features.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

ASP meets the expectations of a programming environment for young children (Papert 1980; Guzdial 2004; Resnick *et al*. 2009; Grover and Pea 2013): "low floor" (easy to get started) and a "high ceiling" (opportunities to create increasingly complex projects over time). ASP has a simple syntax. LP, in general, and ASP, in particular, are closer to natural language compared with other programming paradigms, allow natural modeling of problems and thus allows the focus on the essence of the problems. Also, ASP is a full-fledged programming language that can be used to solve NP complete problems or problems in a higher level of the complexity hierarchy.

As for CS education at the level of both K-12 and colleges, significant efforts in the last two decades were made to understand what CS is and what to teach about CS. One line of work is carried out under the notion of computational thinking (Allan *et al*. 2011; Barr and Stephenson 2011; Brennan and Resnick 2012; Wing 2011). Progress is being made on what computational thinking is (Grover and Pea 2013), but significant effort is still needed to have a full understanding of computational thinking (Hemmendinger 2010; CSTA 2012; Voogt *et al*. 2015). The study of computational thinking has led to the development of several national and international standards for teaching computational thinking at the secondary school level.

As shown in the recent curriculum standards and framework [e.g. Computer Science Teachers Association Standards (CSTA 2012) and K-12 Computer Science Framework (K-12 CS Framework Committee 2016)] and the AP course (an Advanced Placement high school course in USA) "Computer Science Principles" (CollegeBoard 2017), CS education should cover the following components: *abstracting*, *algorithms*, *programming,* and *communicating*.

ASP provides an environment for teaching these components. The objects, variables, and relations underlying ASP provide a rich structure for students to practice *abstraction* when modeling a problem. For example, students can practice abstraction from the very basic level involving a relation among concrete (constant) objects, to a higher level involving queries containing variables, and to the fullest level involving rules. There are also many finer abstraction levels in between those three levels.

As a well-established programming paradigm, ASP offers a setting for students to learn and practice all aspects of *programming*: design the model (program), edit the program, and learn the (informal yet rigorous to a great extent) syntax and semantics, recursion, coding and debugging. ASP allows a modeling methodology which starts from the description of knowledge (in the problem of concern) in natural language (e.g. English) and may need continuous refinement of the English description. The transition from English description to ASP rules provides scaffolds for students to understand both rules and English description in a more precise way. Therefore, ASP-based modeling trains students to practice *communicating information*.

As for *algorithms*, ASP can cover recursion. The training of students in abstraction and rigorous representation through LP is expected to increase students' capacity for specifying problems accurately, which is taken as desirable before studying algorithms (Kleinberg and Tardos 2006). However, as a pure declarative programming (DP) paradigm, ASP is hardly the best option for teaching algorithms. On the other hand, by skipping the need to know algorithms, ASP significantly reduces the gap between (interesting) problems and their computer modeling, which seems to serve better the initiative (Smith 2016) in the USA to teach CS to *all* students in K-12. See Section 7.1 for a more detailed discussion.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

1.2 Advantages of LP

From the existing work and practice in LP and general programming (Pears *et al.* 2007; Chakravarty and Keller 2004; Scherz and Haberman 1995; Ball and Zorn 2015), the main advantages of LP as related to computing education are: (1) simple syntax and intuitive/declarative semantics; (2) natural connection to abstraction, logical reasoning, and knowledge representation which form the foundation of computing and other disciplines; (3) early involvement of non-trivial, interesting, and challenging problems (e.g. Sudoku problem); and (4) the mathematical flavor of LP. LP also provides a rich context for embedding essential computing concepts and skills (as argued in the previous subsection). As for industrial relevance, LP and DP, in general, have seen their profound application and impact in database query languages, problem (formal) specification languages, and domain specification languages including popular web application development languages such as HTML, CSS and XSLT etc. Currently, ASP is a dominating LP formalism in knowledge representation.

1.3 Research question

Given the advantages of LP in CS education, more investigation is needed in how LP could be introduced to K-12 students. Although there is progress being made to expand LP to secondary schools [e.g. Dovier *et al.* (2016)], research on how ASP facilitates student learning in introductory CS is still absent. Therefore, the research question for this study is: *How does adopting ASP to teach an introductory CS course for high school students impact their understanding of computer science and computing?*

2 Methods

This study adopted a qualitative approach to investigate, in-depth, how high school students obtain understanding of CS and computing through LP, specifically ASP.

2.1 Setting and participants

Participants were recruited from the TexPREP program at Texas Tech University during Summer 2015. TexPREP is a summer enrichment program for 6th–12th graders and offers full day courses in science, engineering, math, and CS over several weeks. Students in these grades typically range from 11 to 18 years old. Programs such as TexPREP are often used to motivate students to attend university after the 12th grade.

TexPREP courses are typically taught by local school teachers as well as university faculty members and students. In the case of this study, the class was taught by the third author with some TAs (the second author was also a TA). TexPREP is a selective program: students had to have an A/B+ average, letters of recommendations, a written essay, and a personal interview for consideration. Prospective participants were in their fourth year of TexPREP taking Computer Science 4 (TexPREP-CS4). They had already taken TexPREP-CS2 with Scratch (Resnick *et al.* 2009) and TexPREP-CS3 with Alice (Kelleher and Pausch 2007). The TexPREP-CS4 course teaches the foundations of computing using ASP (Gelfond and Kahl 2014). This course emphasized problem solving and

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

knowledge representation through the ASP paradigm. The course met Monday through Friday from 1:00 pm to 1:50 pm from June 17 to July 9 of 2015.

Informed consent was obtained from both the participants and their parents prior to the start of class. There were sixteen ($N = 16$) participants for the study. There were seven females and nine males. Participants were students at high schools in the Texas Tech area in Lubbock, TX. In terms of rising grades, there were eleven 10th graders, three 11th graders, and two 12th graders.

2.2 Course organization

The teaching methodology employed is influenced by the existing work on teaching LP, for example, the work by Kowalski (1987), Sterling and Shapiro (1994), Clocksin and Mellish (2003), and Gelfond and Kahl (2014). Course topics were centered around building computer models, using objects and relations, for solving problems. This section describes how course activities are structured with respect to problem-solving methodologies within the LP paradigm.

2.2.1 SPARC

This course approached CS through SPARC[1] (Balai *et al*. 2013) which is the result of introducing sorts into ASP. A SPARC program consists of three sections: *sorts section* where sorts are defined, *predicates section* where the sorts of the parameters of each predicate is declared and *rules section* which consists of ASP rules. A full example is given in Section 2.2.2.

We use ASPIDE (Febbraro *et al*. 2011) as the programming environment for SPARC. ASPIDE[2] provides a graphical user interface for students to edit a program, ask queries and get answer sets of the program.

We choose SPARC, instead of popular ASP languages such as DLV (Alviano *et al*. 2011) or Clingo (Gebser *et al*. 2011), as our teaching language for the following reasons. First, the three-section structure provides a direct support of the modeling methodology (see next subsection). Second, the introduction of sorts helps avoid thinking about/teaching safety conditions on rules. Finally, sorts also help provide warning on type errors so that programmers can identify bugs early.

2.2.2 Explicit modeling methodology

Before the discussion of any modeling, we always first give the **problem description**, which consists of two components: the **knowledge in the problem** and the **questions and intended answers**.

[1] The manual for SPARC is available at `https://github.com/iensen/sparc/blob/master/User_Manual/Sparc_Manual.pdf`. It contains detailed information on the installation and use of a SPARC solver.

[2] ASPIDE is designed with advanced projects and users in mind. Its user interface is quite involved. Also the installation and maintenance of ASPIDE, a stand alone software, are a challenge in our teaching (Marcopoulos *et al*. 2017). In our more recent outreach to middle and high schools, we use onlineSPARC – an online programming environment for SPARC with simple and only necessary functionalities including an editor, query asking, obtaining answer sets, and rendering literals in answer sets that represent a drawing or animation (Marcopoulos *et al*. 2017).

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

Here is an example of a problem description:

- **Knowledge in the problem**: There is a family. John is the father and Joan is the mother. The children are Jim, Bill, and Sam.
- **Questions and answers**: Is John the father of Bill? Yes. Is John the dad of Bill? Yes. Is John a parent of Bill? Yes.

Students can be easily involved in understanding the problem, giving answers to the questions, as well as extending the knowledge and questions.

Note that we emphasize both questions and intended answers in problem description. Usually the knowledge in a problem description (particularly in a real life setting) is not complete or contains extra information. The questions and intended answers help the students to understand/decide the relevant knowledge to represent. The students are expected to understand what knowledge (and the reasoning process) is needed to obtain the intended answers. However, students are not encouraged to focus on "algorithmic" ideas to "find" a solution to a question. For example, to model the Sudoku problem (or word problems in algebra or physics), students are expected to use intended answer to help them to clarify knowledge needed and to "debug" if they miss any knowledge or use wrong knowledge. But students are not encouraged to figure out how to construct a final concrete answer manually or systematically. (In fact, a more specific modeling methodology for constraint satisfaction problems is needed for students to model the Sudoku problem.)

Students are expected to build computer models for the problems so that the questions in the problems can be answered by the computers correctly. The next part is a two-step explicit methodology to model a problem:

1. Identification.
   (a) **Objects in the problem**. Examples for the problem above include John.
   (b) **Relations in the problem**. Examples for the problem above include the dad and father relation between two persons. Students are expected to write the relation in the atom form with compulsory comments on its meaning. For example, `dad(X, Y)` denotes that person `X` is the dad of person `Y`, and `father(X, Y)` denotes that person `X` is the father of person `Y`.
   (c) **Knowledge in the problem**. For example, John is the father of Bill. Students are expected to list all knowledge (explicit in the problem description or implicit in our common sense) needed to answer the intended questions.
2. Translation/coding.
   (a) **Objects** are defined and grouped into sorts. For example,
   `#people = {john, bill, joan, sam}.`
   where `#people` is the sort name while the right side of = employs a standard set notation. Intuitively, the sort `#people` means the set of the four persons.
   (b) **Relations** are declared. For example,
   `dad(#people, #people).`
   declares that `dad` is a relation over `#people`.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

(c) **Knowledge** will be translated into rules. For example, the knowledge "for any person X and Y, X is the dad of Y if X is the father of Y" can be translated into a rule:
`dad(X, Y) :- father(X, Y).`

Here is an example of a full SPARC program:

```
sorts
    #people = {john, bill, joan, sam}.
predicates
    % father(X, Y) denotes that X is the father of Y
    father(#people, #people).
    % dad(X, Y) denotes that X is the dad of Y
    dad(#people, #people).
rules
    father(john, bill).
    dad(X, Y) :- father(X, Y).
```

In this course, students have a lot of opportunities to identify relations and their meaning [e.g. `dad(X, Y)` and its meaning] and to figure out what relation is used to represent a piece of knowledge (close to the declaration/specification of a function in an imperative language and the use of a function to solve a problem component). Once they are able to switch freely between the atom form of a relation and the English description of the relation, they are ready to write rules. Students are encouraged to always start from English description of the knowledge before writing any rules.

2.2.3 Course content

The sequence of major topics covered by this course is: an introduction to CS, modeling methodology, use of ASPIDE, applying methodology to model the family problem (facts on relations `father` and `mother`, rules on relations of `dad` and `parent`), queries to SPARC programs, reasoning (informal but in a manner as rigorous as possible) with facts and rules in the context of the family problem, writing rules through iterative refinement of English description, default negation (only in representing the closed world assumption), modeling the family problem with more relations such as `sister` etc., modeling the simplified battle field problem (a game used in middle school math teaching) and map coloring problem, recursion (defining one person is an `ancestor` of another using the `parent` relation), modeling a map problem (about the neighbor of a state and the color of a state), and modeling the map coloring problem and Sudoku problem. Details of the course content, except for the Sudoku problem, can be found in the lecture notes available online (Zhang 2015).

2.2.4 Course design

This section now discusses the course design. First, we explain the iterative refinement based methodology used to help students to develop rules from their English description

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

of knowledge they generated for a given problem. The class starts from facts. Students will practice writing English sentence(s) about simple facts. These sentences will then be translated into facts in SPARC. Depending on the problem description, this task might not be trivial. Guidance and examples are given to help students to get used to the methodology for writing facts. Next, we introduce rules with body using `dad`. We started from students' English description of the relationship between `dad` and `father`. The first version of the English description is usually hard to translate into a rule using the relation `dad(X, Y)` and `father(X, Y)`. The difficulty arises from the flexibility of natural language. Here, English allows us to employ an object and a relation related to the object. An example description of `dad` could be "Dad is the same as father." Instead of giving an idiomatic representation of such knowledge in ASP, we encourage students to apply the *iterative refinement methodology*, a fundamental method in CS, to refine their description until a rule can be written to their satisfaction. In the iterative refinement, students will recursively use our problem modeling methodology to identify the objects and relations in their description. Students will be asked what objects are involved in the description above and how they are related. With some hints, they can figure out the objects (implicit) in the description and relate them using the intended relation `dad` and `father`. This practice is expected to enhance students' reasoning capacity, clarity and preciseness in communication. Consider a refined description of grandparents: X is a grandparent of Y if X is a parent of a parent of Y. This English description involves the similar phenomenon in the example for dad. The second occurrence of parent is used to denote both an object and parent relation between this object and Y. By making the object explicit using a variable Z, the description can be further refined as: X is a grandparent of Y if (there is a person Z such that) X is a parent of Z and Z is a parent of Y. The translation of the refined description to a rule is now straightforward. We would also emphasize that our teaching materials are organized surrounding the modeling methodology. The methodology is practiced repeatedly in the family problem and its extension, the map problem, the simplified battle ship problem, the map coloring problem and Sudoku problem.

In the Sudoku problem, we were able to demonstrate the *top down designing approach*, another important problem solving skill in CS, when defining two squares in the same 3 by 3 block. Belt concept is introduced to define the same 3 by 3 block relation: two squares are in the same 3 by 3 blocks if the rows of the squares are in the same *belt* and the columns of the squares are in the same *belt*. We next define what is a belt. Two numbers are in the same belt if they are in first three numbers (i.e. 1–3), in the second three numbers (i.e. 4–6), or in the third three numbers (i.e. from 7 to 9). The key in the top down approach is to introduce and use a relation(s) as needed and define this relation(s) later.

Finally, as an introductory course to CS for high school students, the formal definitions of terms, rules or semantics (usually covered in most textbooks for college or graduate students) are not covered. However, by formulating definitions (e.g. for parent) in English and translating them into rules, the students are expected to experience and appreciate the rigor in representing the knowledge needed to solve a problem. Given the precise form of the rules and intuitive logical reasoning, the students are also expected to exercise their reasoning capacity to a great extent.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

2.2.5 Lab assignments

Four labs (closely related to topics discussed during class) were given, usually along with a program stub. However, students had the opportunities to modify and/or add: sort definitions, predicate declarations and rules. The first lab was based on the family problem and focusing on the basics of the problem description, modeling methodology and the programming in ASPIDE (for SPARC). The students were asked to define the relations of `father` and `mother` that were discussed during class and any new family relations they were interested in and model them in SPARC. They were also expected to submit a report containing the problem description and the result from the identification step of the problem modeling methodology. In the second lab, students were asked to define three specific family relations: `mom(X, Y)`, `uncle(X, Y)` and `sister(X, Y)`. The third lab is the standard map coloring problem, and the fourth lab is the standard Sudoku problem.

2.3 Data collection

Three main types of data were collected. Surveys were used to gather background information on what students learned before and after the class. The main data source came from clinical interviews which were used to examine students' conceptual understanding. Scores from lab assignments and the task in the clinical interview were then used to support the qualitative findings.

2.3.1 Surveys

Participants responded to surveys with open-ended questions at the beginning (pre) and end (post) of the course regarding their experiences with computing and DP. The purpose of the surveys was to provide some background information on participants' CS knowledge and to triangulate the findings from the clinical interviews. The survey refers to DP rather than LP or ASP even though this study focuses on the latter two paradigms. The rationale for referring to the more general DP is because: ASP belongs to the LP paradigm and LP is one paradigm of DP. So, knowing other DP paradigms (such as functional programming) may give an edge to the students in learning ASP. The pre-survey asked about participants' general knowledge of CS and DP. The pre-survey questions were:

- What is CS?
- Describe what you know about DP.

The post-survey asked about participants' knowledge of CS and the topics of this course. The post-survey questions specifically asked what participants learned in this course and how they understood the concepts; that is, rather than asking participants directly to explain DP, these questions asked how they perceived DP:

- What is CS?
- What have you learned, if anything?
- After having taken this course do you feel that you understood the subject of this course and the tasks that were asked of you? Why or why not?

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

2.3.2 Clinical interviews

Clinical interviews are task-based activities in which the researcher attempts to explore participants' understanding and cognitive process through active questioning and probing (Ginsburg 1997). Clinical interviews were conducted one-on-one by the research team and the participant. During the interview in the last week of class, participants were asked to work out one class-based problem while thinking aloud. Researchers asked probing questions in order to prompt participants to explain and clarify their thought processes. All researchers received the same training from the first and third author on conducting the clinical interviews. Clinical interviews lasted 30 min or until the participant completed their task – whichever came first. Clinical interviews were video recorded and transcribed.

We have collected interviews for 16 participants. All the other participants were out of town in the last week and thus no interviews can be done with them.

In the interview task, each student was asked to write a SPARC program to represent the knowledge about family relationships. This problem was an extension of the second lab and not a formal assignment in the course. Some relations, such as `father(X, Y)`, denoting `X` is the father of `Y`, and `mother(X, Y)` are given. A program stub including the declarations of these relations were also given. The students were asked to write rules to represent the knowledge: (1) Jon is the father of Matthew, and (2) to define the following relations: `grandparent(X, Y)`, `son(X, Y)`, `aunt(X, Y)`, and `descendant(X, Y)`. The results of the surveys and interviews are presented in the following sections.

2.3.3 Programming assignment grades

Grades for programming assignments were collected by the instructor of this course and used to support the qualitative findings of this course. Lab assignments were graded by a PhD student (called *the grader*) who was not a part of this study. The rubric graded projects for: completion of requirements, abstraction, correctness, coding style/readability, and documentation (commenting) on a 4-point scale. Completion refers to the extent in which a participant implemented all of the assignment's requirements. Correctness refers to the program's ability to produce the correct results for all test cases created by the teaching staff. Abstraction refers to the extent that participants were able to write generalized code that is reusable and adaptable. Participants' source code was graded on coding style in terms of organization, readability, and adherence to class coding style conventions. Documentation refers to the amount and quality of documentation (comments) that describes the source code.

3 Survey results

Survey responses were transcribed and analyzed by the researchers using an inductive coding approach in which responses were coded with descriptive labels (Corbin and Strauss 2008). That is, the researchers attempted to understand the meaning of each fragment and assigned a descriptive label. Similar labels were merged into larger categories. The main unit of analysis was a sentence. Longer sentences that included multiple ideas were sometimes separated for further coding. Similarly, some consecutive sentences were grouped together for analysis when they continued the same line of thought.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

Through this analysis, researchers found common themes across participants. Responses were grouped by common themes (or labels). The most frequent themes are presented below. Participants were given pseudonyms in the presentation of the results in the form of P#.

3.1 What students know about CS

Since this course was the third in the pre-engineering program sequence, participants already had two courses in computing and other courses in science and engineering. Most participants' definition of CS in the pre-survey centered on the creation of a program that does some task with the programming language being the means through which those tasks are completed [e.g. *The science behind how a computer functions and performs tasks* (P1); *Computer science is using different computer "languages" to get computers to perform tasks* (P13); *Understanding and being able to reproduce the code and programs that make a computer do its task* (P4)]. There was a strong connection between CS and programming with ten participants mentioning programming or the creation of software in their responses. Some participants acknowledged the importance of computers and technologies in the advancement of society [e.g. *The study [of] how to develop computers to help the world be an easier and more efficient place* (P10); *It has become extremely prominent in our daily lives and must always be improved for technology to come* (P11)].

At the end of the course, participants were asked the same question. In the post-survey, only three participants mentioned programming in their definition of CS. The participants' responses could be categorized as defining CS as problem solving [e.g. *Computer science is a study that involves mathematics and helps us understand the way a computer thinks* (P3); *Computer science is the study on how to make a computer work/think like a "human"* (P10); *It is how we get computers to solve problems and how we can use them for simplifying, numbers, solutions, or problems themselves* (P11)] and the study of technologies [e.g. *The science behind a computer and how it functions* (P1); *Computer science is the study of how and why computers work* (P16)]. Thus, by the end of the course, the participants' definition of CS had broadened a bit away from just programming; rather, it become more holistic and more on emphasizing problem solving in order to complete tasks and how technologies can support those processes.

3.2 What students know about DP

In the pre-survey, most participants did not know what DP was; however, many were familiar with the more "traditional" procedural programming and/or had prior programming experience. Eight participants said that they did not know what DP was. Some participants offered an explanation of what DP could be [e.g. *I would guess that it is a branch of computer science that has to do with making statements and receiving some form of output from the statements* (P12); *I believe that you give a description to of an object then the program runs it to find the item similar to what was described* (P8)] or explanation that they have done programming before [e.g. *In school, I have learned a little about Java and how Java is a declarative language, Other than that, I don't know*

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

much else about declarative programming (P3)]. In such explanations, participants were attempting to make connections to previous programming experiences, even though those were incorrect.

The remaining five participants have some understanding of what DP was: *Declarative programming is stating a statement and being able to program that statement. Giving the computer a "command" and programming it to accomplish that particular command* (P5); *It is a program where it does the things you type out* (P10); *Declarative programming is more of an elaboration off of algorithmic programming as it is developed for a broader use compared to Algorithmic programming. It is much more elaborates and is well suited for situations with similarities* (P12); and *It is the type of programming needs to be defined before any output action can be achieved* (P15). Participant 16 was the closest to having a correct answer: *I know that declarative programming works like telling a computer rules for what something is then having the computer rules for what something is then having the computer go find all the somethings [sic] in a set.*

From the participants' responses, they did not have experience with DP although have had some intuitive understanding of what DP may be.

3.3 What students have learned in this course

The survey data showed that most students reported learning how to program with the SPARC language within the context of their lab assignments. Though most did not explicitly articulate their understanding of LP, as they did with procedural-type languages from prior CS experiences, they were able to report how they solved problems using ASP methodologies and tools. Although the participants emphasized learning how to program with SPARC, 11 responses implied that the programming language provided the structure that required students to think about their problems through a deeper, more thorough perspective. P12's response was representative of how learning the programming language helped guide declarative understanding: *I've learned how to reevaluate my thinking process and consider the basis of my thought (so I can program). I've learned how to 'translate' – by writing my goals/rules/knowledge in English, then transferring and modifying it to fit SPARC code.*

Similar responses include: *I learned how to look at a problem in a different way* (P2); *I have learned how to sort objects, define relationships with predicates, use multiple variables, use multiple predicates/rules, use tex****[3] *rules with variables, and use if when for rules to help simplify solutions* (P11); *I have learned a new coding language to go along with the others I know, which is very valuable to me. I have also learned a new way to look [at] relations between objects* (P15); *New syntax – A new way to specify relations* (P9). These examples show that the rules and structure (i.e. syntax and semantics) that SPARC required and the associated methodologies helped guide participants in their problem solving. Four participants emphasized learning about the syntax and coding style. P8 also mentioned that this was the first course in which she had to type out her programming rather than through a drag-and-drop interface.

[3] The words in the original writing were not readable.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

3.4 How students understood the course topics and tasks

Eight participants stated that they understood the subject of this course within the confines of the course activities and three were indifferent. They understood that there was more to learn with respect to the programming language [e.g. *I think if given more time and work on the computer, I could use this language in the future* (P3); *The prompt and things that were asked were clear and easy to understand. They were easy to pick up on and gave a general understanding – making it relatively easy to complete a given task* (P5); *I understand the subject because it was easy. I'm pretty sure that if you wanted to program more complex things, it would be harder* (P13)].

Two participants stated they were able to understand the subject of the course, but were unsure how it was applicable in a real world or everyday setting: *I understand how, but not why. So, I don't understand the purpose of this type of program and how it will be beneficial to the world, other than artificial intelligence, but I feel as though this program requires more to get to that point* (P9) and *I do feel I understood the subject of the course as I did actually learn something. However, I did not always understand exactly what I was supposed to do and I was occasionally completely lost* (P4). Therefore, most participants knew that there was more to learn about LP beyond the contexts with which they were presented. Both P10 and P14 felt that the course was too repetitive with the simple topics and was not challenging enough.

3.5 Student feedback

Participants were asked to provide feedback on what they thought of the course and its LP activities, and 13 of them responded. Participants seemed to appreciate[4] the different approach that LP/ASP takes to programming as most of them have worked with either Java and/or a visual programming language like Scratch:

- *I like this program because it makes you look at code in a different way. ASPIDE is such a different set of code then the standard JAVA. I like the format of this code and the different symbols.*
- *I learned a lot about declarative programming and how to use it. The course was actually writing code and not drag and drop.*
- *…this course has a lot of potential and if executed right it can be very beneficial because the concept is cool and it is always great to comprehend a new language.*
- *It was a good course that has allowed me to learn a type of programing.*

They also found the ASP activities easy and interesting to work on, even if there was a small learning curve:

- *Once we got into actually working with the program, it was very interesting and I enjoyed playing around with it and figuring out just how simple it really was.*
- *The course structure and the SPARC program make it relatively easy for someone who has not spent much time doing computer science to understand the program and how to program…Following the "methodology" also makes it very easy to learn the program.*

[4] Some feedback overlapping with the positive comments reported in Section 3.3 is not repeated here.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

- *It was a bit easier to understand and even enjoy a little after a while, for the TAs went around and took the time to make sure it made sense.*
- *It was a little difficult at the beginning because I had no prior knowledge of programing. But it was a very enjoyable course and i would take it again.*

The main critiques of the course included the need for more specific instructions and expectations for the assignments, better pacing as some participants remarked that the teaching was occasionally too slow or too fast, more time for assignments. Most of these critiques had to do with the delivery of the course. Aside from one comment on the ASPIDE programming environment for SPARC, there were no explicit critiques of the LP paradigm.

4 Interview results

A grounded theory methodology was adopted to generate a theory that explained how students understood and approached computing through ASP activities based on the interview data. Details of the methodology can be found in Corbin and Strauss (2008). This qualitative approach generates a theory or explanation through systematic analysis of data. Grounded theory goes beyond describing and categorizing by finding the processes and relationships in the phenomenon being studied. This approach was conducted on the clinical interview data and served as the main analysis for this study. The unit of analysis was an utterance, which consisted of a complete thought or phrases spoken by the participant. In cases where utterances were rich with many potential units, they were broken down into smaller utterances. Utterances made by the researcher provided context to those made by the participant, but were not analyzed. There were 1452 utterances used for analysis.

The first author analyzed the data through an open-coding process coding each utterance with a descriptive label. This process required several iterations to ensure that coding was consistent across the entire data set. Then, the third author coded the data using the set of labels created by the first author. Through that process, the researchers debated the labels and coding of the utterances, which led to several iterations of recoding the data and revision of the initial labels. Memos were kept during this analysis process regarding the rationale behind labels and the formation of coding. Since the third author was the expert in the field of ASP and DP and taught the course, his coding was used for the remaining analysis. The resultant set of labels found in open coding are presented in Table 1 with their frequency.

The next step was creating categories that represented larger constructs that were happening in the data. Many of the merged labels were found to relate to each other as well with infrequent labels, which helped form these categories. The five major categories were: Abstraction, Representation, Reasoning, Revision, and Procedures. Axial coding analyzed the way these categories were related based on their properties; that is, this part of the analysis found the relationships between the major categories. The first author led the first stage of axial coding which led to these five major categories. Then, these were also debated and discussed by the researchers. The surveys, particularly the post-surveys, were used to guide the axial coding and the creation of the categories. The definitions of the categories also went through several revisions.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

Table 1. *Resultant labels from open coding*

Knowledge representation	137	Sorts	28
English definition	131	Predicate declaration	26
Reasoning	109	Commenting	22
Debugging	86	Relationship	19
Testing	71	Query	18
Rule	66	Thinking	17
Syntax	63	Translation	12
Strategy	60	Reusing	10
Refining	59	Prior knowledge	6
Problem understanding	44	Reflection	5
Problem solving	35	Variable	4

The next sections define each category and give example quotes.

4.1 Abstraction

This category describes instances where the participants understand the problem space in more general or abstract terms and are thinking on a higher level within the context of the code they are writing. This type of thought is in contrast to only discussing idea in concrete terms or with specific examples. In this generalization, participants are generally describing relationships with respect to these real-world terms and/or integrating their prior knowledge of the problem space.

Example quotes included: *The descendant part because the descendants can be anything that's... that was before that person, well before X in this case* (P8); *So, father of person and person, ok, parent, sister, same parent, gender* (P10); *I am thinking of like a family tree I guess sort of and so I am thinking of X being or I am thinking of Y being so I am thinking of Y being a person, whose descendants we are looking at...* (P12). *P03: Yeah um I am just trying to think of like a more general way of putting it instead of just listing it all down. (pop up appears and then is closed out) Parent XY and uh...* (P3).

Thus, Abstraction occurs when participants are able to explain and apply the LP concepts within the ASP environment as well as outside it using real-world examples. That is, students' understanding of the LP concepts is not limited to only working within the ASP environment.

4.2 Representation

This category refers to the process in which the participants are trying to understand the problem space in coding terms; that is, how their abstraction or understanding of the problem translates into the code. This category includes references to both the conceptual understanding and coding as well as the transition process between the two.

Example quotes included: *You can't say, X is the parent of Y if X, if Y is the son, because it's flipped. I hope I am explaining it right. So then I'm saying X is the son of Y if Y is the parent of X, and I'm putting gender because if you are a son you have to be male. And if X is a male. All right I am going to move on to aunt* (P1); *A descendant is like*

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

it's the same as a child it's umm like the child of two parents. That is their descendant that's their heir so that is what I am trying to say but I don't know how to say it (P14); *I put X is the son of Y. I put that in the predicates. This is it. The son is a person. And I know that we have the example here. I can put Matthew is the son of Jon...* (P15).

Representation is when participants map their understanding of the LP concepts into code; it is a translation between their conceptual understanding and coding. Rather than simplifying this process just as coding, Representation refers to the participants' process in creating a code-based representation of their understanding of the ideas.

4.3 Reasoning

This category is the problem-solving process. It describes instances in which participants were actively thinking about the problem, apply problem-solving strategies associated with ASP, and adapting existing code to solve the current problem. Reasoning was selected as the category name as that was the most frequently used label related to the problem-solving process. Example quotes included: *So grandchild Isaac of George, grandchild Joseph, George, grandchild Susie, George. Grandchild, those are the children, so, grandchild, grandchild, X is a grandchild of Y, if Y is grandfather, if Y is the grandfather of X and X is the, do we define gender? Yes, gender of X is male, because grandfather has to be a male* (P2); *I think I'm doing aunt. (Begins typing comment for aunt.) ... For any person X and person Y, X is the parent of Y. There is a person Z, Z is the sister of X such that X is the sister of Z... I mean Z is the sister of X and the aunt of Y.* (P5). *Nephew, so then you have Isaac, and actually you can create a rule* (P11).

Thus, Reasoning describes the entire problem-solving process as participants attempt to negotiate their understanding of the problem and the solution. It was this reasoning process – the internal discussions within themselves and the reflection – that was the most observed activity in the problem-solving process, which is why this category was named Reasoning.

4.4 Revision

Revision encapsulates a lot of the overall debugging process, but emphasizes the importance of participants asking questions in SPARC to see if their code was correct. This category includes the participants' process of questioning to see if they completed the tasks correctly, which was done through querying to see if their solutions were right. Example quotes included: *So I'll write aunt and then I'll write Rose comma Susie, and then I'll have to add a question mark, and then have to press execute* (P1); *Ok and now I need to write queries to see if my program works* (P14); *I am going to ask who are descendants of George because George is like the patriarch of the family* (P12). Simply put, Revision is the debugging process after the participants developed their initial solution.

4.5 Procedures

This category represents a large part of the data in which participants were either restating or interpreting facts or instructions. In some cases, participants were asking if they had the correct interpretation of the assignment instructions. In this category, participants frequently mentioned specific rules and strategies associated with declarative

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

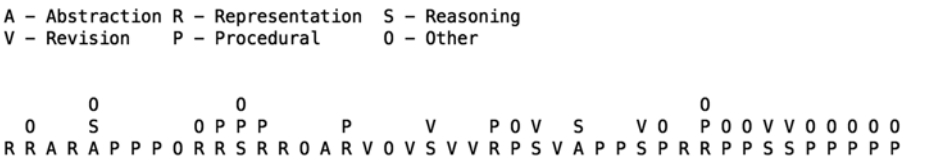


Fig. 1. Sample (Participant 12) of order of category occurrences during clinical interview.

programming. Often, these utterances were co-labeled with other more descriptive categories such as Representation and Reasoning. Example quotes included: *So I write. I have to comment first do that the people, errr, so that anyone else that is reading it knows what I am trying to say. . . . So I said that son X and Y denotes. Hmmm! It's all caps (small laugh). Denotes that X is the son of Y. And then I write the actual code and just put son, I put in parenthesis, uhh, . . .* (P1); *OK so X is the son of Y if Y is the parent of X and X is male. So. . . Gender um, Oh I was reading the English, . . .* (P3); *So now I am at son. Oh forgot to add the periods. Ok. Got it, got it, and got it. Now I need it aunt* (P8); *Jon is the father of Matthew, so that rule that I wrote works* (P16). So, in the case of ASP, the procedures category describes the participants' actions and ability in identifying the instructions and facts involved with the problem, which eventually helped them to develop a solution for their problem sets.

4.6 Order of category occurrences

As part of axial coding, there was an analysis of when these categories occurred during the clinical interview to learn more about the categories themselves and how they related to the other categories. A letter-coded graph plotted out the occurrence of each category in the order they appear with each participant. Figure 1 shows an example of this graph. When an utterance had more than one category associated with it, plots are stacked.

A visual analysis of all participants' graphs was conducted to describe when each category occurred and the relationships between those occurrences. Across all participants, interviewers had them all read the instructions once at the beginning, but these utterances were not coded. Since participants were asked to work on their assignments, some participants had already started on them prior to the interview. Thus, those graphs immediately started with non-procedural labels. Findings from a visual analysis of all participants graphs are presented in the following subsections.

4.6.1 Abstraction

Evidence of Abstraction mostly came from the beginning to the middle for most participants. For several participants, it occurred alongside Procedures (P3, P7, P8, P10, P11, P12, P13) as well as with Representation (P4, P9, P11, P12, P14, P16). Three participants had Abstraction after Revision, which may suggest that the questioning process may support part of abstraction abilities. Another three participants (P2, P3, P13) showed no evidence of Abstraction during the interview. However, the interviewer reported that P2 did not get a chance to test their program and P3 had technical issues with the computer. P13 was reported to feel confident about what they were doing, and started Revision early.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

4.6.2 Representation

Similar to Reasoning and Procedures, Representation also happened throughout the interview. For the most part, Representation started before clusters of Revision for 11 participants (P1, P3, P4, P5, P7, P8, P9, P11, P12, P15, P16). However, for three of these participants (P2, P11, P16), there were more revision afterward. For P13, Representation was intertwined into Revision throughout the interview. Most of P14's interview consisted of utterances labeled as Representation as he was mostly listing out what had to be represented in the knowledge base.

4.6.3 Reasoning

Reasoning was noted throughout the interviews for most participants, similar to the procedural utterances, which may suggest that it is an important recurring process. Reasoning occurred with and between instances of Revision (P1, P16), Representation (P2, P13), and Abstraction (P14, P15). Thus, Reasoning is also intertwined with other parts of the ASP process. There was no evidence of Reasoning from P6; however, there were not many coded utterances from that participant in general.

4.6.4 Revision

Most participants (P1, P2, P3, P4, P6, P7, P8, P9, P11, P15) were engaged in revision during the last half of the interview, which was expected as they spent the first half identifying the objects, relationships, knowledge, and coding before testing. P10 did not get as far during the interview to be engaged in revision. Other participants had already started their assignments before the interview, so they began to revise their solutions earlier than others.

4.6.5 Procedures

All participants had instances of utterances of procedure, which were distributed throughout the interview as well as simultaneously appearing with other categories. An interesting observation was that there were procedural moments throughout the interview, but not just at the beginning. That is, participants kept going back to the instructions and/or the facts presented in the assignment.

At first, the Procedures category did not seem to provide much insight into how the participants approached computing and CS. After all, these utterances were of participants reading and re-reading the rules and asking if they understood the instructions and information correctly. Procedural utterances were made throughout the entire computing process. In selective coding, analysis examined how one category could be connected to other categories (Corbin and Strauss 2008): the Procedures category was found as that central theme. Identifying objects and relations in the problem, through English definitions, by carefully reading the problem description and integrating one's common sense knowledge was a major component of the explicit methodology for solving LP problems. Thus, there was deep interaction between the Procedures and other processes. Students had also mentioned in their post-surveys that they had learned this methodology to solve

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

Table 2. *Average lab scores rounded to the nearest whole number: COMPLETE = completion, ABS = abstraction, CORRECT = correctness, STYLE = coding style, DOC = documentation, TOT = total*

Lab #	COMPLETE	ABS	CORRECT	STYLE	DOC	TOT
Lab 2 (N=16)	95%	88%	82%	100%	84%	90%
Lab 3 (N=14)	100%	92%	92%	95%	70%	90%
Lab 4 (N=11)	89%	98%	73%	100%	96%	91%
All labs	95%	92%	83%	98%	82%	90%

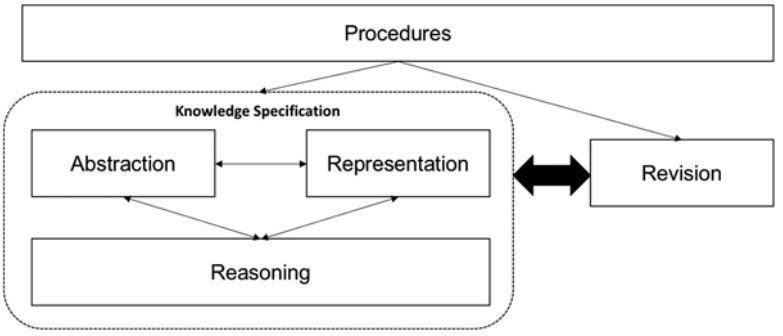


Fig. 2. Model of how participants approach LP through ASP.

the problems. Although that may not be surprising, it showed that they were able to adopt those strategies, which in turn, led to more reasoning and representation processes.

5 Achievement results

Each lab was scored using a rubric, which had five parts: completion of requirements, abstraction, correctness, coding style/readability, and documentation (commenting). Each component was scored on a scale from 0 (poor) to 4 (excellent) by the grader. Table 2 shows the scores and completion rate for Labs 2–4. Due to the logistics issues, we were not able to collect the submissions of Lab 1.

Overall, participants did well on all three labs across each component. The average score of completion was 95% (3.8 out of 4 points). This high average shows they were able to address the project requirements as well as follow the two-step methodology. The average score for abstraction was 91.9% (3.68 out of 4 points). The average score for correctness (e.g. getting the correct output) was lower than completion and abstraction: 83.1% (3.32 out of 4 points). For completion, the grader looked for attempts made by students to address each requirement. Thus, a participant could have made a reasonable attempt at all the requirements, but implementation did not fully match the specifications or yield correct results for all test cases. Similarly, abstraction scores would also be lower in those events. With respect to the model in Figure 2, this quantitative data supports the notion that students are able to engage in Representation (coding), Abstraction, and Reasoning (problem solving) within the LP paradigm. The lower correctness scores may demonstrate that the participants may need to improve in the area of coding. The ordering of the

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

categories showed that revision did not happen until the second half of the interview, if at all, which places some limitations on the revision (debugging) process. The high completeness score is also in line with the dominance of the Procedures category during the task, corresponding to the predictions of our model, since participants mostly attempted to address all the requirements. Coding style was exceptionally good across all the labs at 98.1% (3.92 out of 4 points). The lowest component was documenting code, which was 81.9% across the labs (3.28 out of 4 points). Generally, most students that submitted the assignment received high scores: the average lab score was 90% (18 out of 20 total points). The lower submitted rates in the later labs were attributed to student absences mainly resulted from family vacations (which are not rare in the summer in the USA).

6 Model of how participants approach computing through LP

Based on the grounded theory analysis, supported by the visual analysis of ordering of categories, a model was constructed that may explain how participants approach computing through LP as based on the data. It is shown in Figure 2. This figure highlights the foundational importance of following the strategies (i.e. explicit modeling methodology) of declarative programming. In the case of ASP, it allows for an explicit methodology, which guides students toward abstracting the concepts, representing it in code, and reasoning in their problem solving. Abstraction, Representation, and Reasoning often occurred together and are grouped separately from Revision, which mostly came toward the end of the task. Similarly, Abstraction and Representation happened more sequentially with Abstraction coming before Representation. Reasoning was observed throughout the interview, which is indicated by a larger rectangle in the figure.

The model shows the importance of Procedures in LP; in this case, students were able to follow the explicit methodology in LP. The students also demonstrated the general problem-solving skill of iterative refinement. There are two distinguishable big loop components: Knowledge Specification and Revision. The Knowledge Specification component is again an iterative refinement consisting of interleaving steps: Abstraction, Representation, and Reasoning. As an example, when a student tries to specify their knowledge about aunt, they try to figure out the objects (individual persons) and relations (e.g. who is whose sister, or who is whose parent) and then tried to define their knowledge about aunt using these objects and relations. Participants tried to make sure the definition captures their intended meaning, which requires them to reason with and understand the relations better. In fact, this model reflects the intended skills most researchers would like the students to obtain in an introductory course: the skills should be general but important in computing (and beyond). The skills include: the understanding and the application of the LP procedures (a general methodology for problem solving), iterative refinement in problem solving, programming, abstraction, rigorous high-level (logic) representation of knowledge, and (logic-based) reasoning with the knowledge.

LP can be thought of as allowing for an explicit problem-solving methodology: problem understanding, precise representation, and reasoning with knowledge. This methodology is almost universal in problem solving. It will particularly be useful as LP lays the foundation for problem understanding and knowledge representation. This foundation is expected to be helpful when students study algorithms in future.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

7 Discussion

The research question asked how using LP, specifically through ASP, to teach an introductory CS course for high school students impacts their understanding of CS and computing.

Prior to this course, most participants had limited experiences with CS, and certainly even more limited experience with declarative programming. Their perception of CS was mainly connected to programming and creating technologies for the world. Given that participants were unfamiliar with declarative programming, they attempted to explain within the context of a paradigm with which they were familiar: imperative and, to an extent, OO programming, because of their previous courses in Alice and Scratch. Interestingly, after this course, they were able to explain the tasks and procedures, even though they were unable to articulate what declarative programming was. The data suggest that students may not even understand what programming paradigms are.

The students had done very well in their lab assignments with an average score of 90% overall on all labs. Students also reported on post-surveys that they were able to learn how to program with SPARC and follow the modeling methodology in their computing activities. Such results are in line with other research that shows that LP can be easy to learn (Mendelsohn *et al*. 1990). The main reason is the simple syntax and intuitive/natural semantics of ASP which allow the students to focus less on language-specific feature and more on the problem-solving skills. The clinical interview data show further that the students were able to learn and apply the important concepts underlying computing. The model (Figure 2) shows that they were able to apply explicit LP methodologies in problem solving and iterative refinement in both big steps (Knowledge Specification and Revision) and small steps (Abstraction, Representation, and Reasoning), when solving the problems. Participants applied their knowledge and skills in Abstraction, Representation, and Reasoning heavily, in an interwoven manner, in understanding, extracting, and defining the knowledge needed in solving a given problem. Once they identified/defined the knowledge needed, they are able to carry out the standard programming tasks from coding to debugging.

The overall scores for completion (95%) showed that students were able to make reasonable attempts at meeting lab requirements. Though students were able to follow the two-step methodology, the lower correctness does bring attention to a need to help students address all specifications and provide more support during the revision process. The lower documentation score (81%) also suggests that students need to be encouraged to document their code. However, the instructional staff did refer to time being a limitation, which may explain higher completion, abstraction, and correctness (slightly) scores over documentation: that is, students may have focused on getting the project done and making it work first. This study shows that working within the LP paradigm yields similar results as within a traditional imperative or OO paradigm. The model generated shows that students can engage in computing skills, such as abstraction, problem solving (Representation), and debugging (Revision). These important LP skills are aligned with four of the six core computational thinking practices found in the AP CS Principles Framework (CollegeBoard 2017): connecting computing, creating computational artifacts, abstracting, analyzing problems, and communicating and collaborating.

Similarly, the study shows that participants were always engaged in cyclical process of Abstraction, Reasoning, and Representation. In fact, LP requires programmers to have

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

a solid grasp of the problem space before starting on the solution. The data also show that participants were engaged in abstraction – they were able to think about the major ideas in a more generalized context. In this course, they were also expected to explain and connect basic computing concepts. Indeed, the participants were able to explain computing concepts within the context of this course and tasks. Students were able to operate within the LP paradigm driven by the explicit methodology for LP; however, they may not immediately see how it is applicable outside the context of the assignments. But, they understood the LP concepts and processes and were successful in completing their assignments. More work needs to be done to make connections to how they see CS in the real world through LP: students understood the problem-solving nature of CS and believed LP was one way of solving problems.

7.1 Limitations

Participants were above-average students and were pre-screened for this program, and had some experience with Scratch and Alice. However, participants stated no prior experience with LP or Declarative Programming. A few students missed some assignments mainly due to family vacations. Lastly, this course met for four weeks, which provided limited exposure to LP as compared to regular secondary school or university courses. Future studies on LP in introductory CS courses could be conducted on full semester courses rather than short courses.

Also, it is worth of noting that *algorithms* are also taken as an important component of CS in K-12 education. As a pure declarative programming paradigm, ASP is hardly the best option for teaching algorithms. In our perspective, one reasonable sequence of CS courses would be teaching the foundations (except algorithms) of CS using ASP, and introducing algorithms in a subsequent course.[5] The advantage of this approach is to shorten the distance from problem understanding and representation to programming by skipping the algorithm component which is certainly non-trivial and significantly prolongs the learning process before students can solve interesting problems using computers. This advantage becomes more prominent given that the CS education in K-12 is expected to serve all students.

We do note that ASP can cover some important aspect(s) of algorithms. One example is recursion in ASP. It is also argued that a precise problem specification is desirable before the study of algorithms solving them (Kleinberg and Tardos 2006). As shown in our data, LP-based modeling helps to develop students' capacity on abstraction and representation (in a rigorous manner) which are key skills in precisely specifying problems.

8 Conclusions

This paper asserts that CS educators should take a closer look at using LP during the introductory courses. As suggested in Ball and Zorn (2015), future computer scientists should be equipped with foundational programming language principles involving logic and formal specification to design and implement complex software systems needed by

[5] If one really wants to introduce algorithmic ideas in the foundation class, one can employ the unplugged approach (Feaster *et al.* 2011) which is independent of a programming language.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

the society. Our results show that the students were able to focus on the key concepts of computing including abstraction, representation and reasoning when solving problems. The results show evidence that it is viable to teach an introductory computing course using ASP. It may also bring attention to how early we can teach students about different paradigms and if that would expand the students' conceptions of computing in order to engage in abstraction and problem solving within the declarative programming paradigm. This effort would later require educators to build the connections between these different paradigms and the rest of CS.

References

Allan, W., Coulter, B., Denner, J., Erickson, J., Lee, I., Malyn-Smith, J. and Martin, F. 2011. *Computational Thinking for Youth*. White Paper for the ITEST Small Working Group on Computational Thinking (CT).

Alviano, M., Faber, W., Leone, N., Perri, S., Pfeifer, G. and Terracina, G. 2011. The disjunctive datalog system DLV. In *Datalog Reloaded*. Springer, Berlin, 282–301.

Balai, E., Gelfond, M. and Zhang, Y. 2013. Towards answer set programming with sorts. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, Berlin, 135–147.

Ball, T. and Zorn, B. 2015. Teach foundational language principles. *Communications of the ACM 58,* 5, 30–31.

Barr, V. and Stephenson, C. 2011. Bringing computational thinking to k-12: What is involved and what is the role of the computer science education community? *ACM Inroads 2,* 1, 48–54.

Beux, S., Briola, D., Corradi, A., Delzanno, G., Ferrando, A., Frassetto, F., Guerrini, G., Mascardi, V., Oreggia, M., Pozzi, F., Solimando, A. and Tacchella, A. 2015. Computational thinking for beginners: A successful experience using Prolog. In *CILC*. 31–45.

Brennan, K. and Resnick, M. 2012. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association, Vancouver, Canada*. 1–25.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

Chakravarty, M. M. and Keller, G. 2004. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming 14,* 1, 113–123.

Chen, T.-Y., Monge, A. and Simon, B. 2006. Relationship of early programming language to novice generated design. *ACM SIGCSE Bulletin 38,* 1, 495–499.

Clocksin, W. and Mellish, C. S. 2003. *Programming in PROLOG*. Springer-Verlag, Berlin Heidelberg.

CollegeBoard. 2017. AP computer science principles including the curriculum framework. Advanced Placement Program, The College Board.

Cooper, S., Dann, W. and Pausch, R. 2000. Alice: A 3-d tool for introductory programming concepts. *Journal of Computing Sciences in Colleges 15*, 107–116 (Consortium for Computing Sciences in Colleges).

Corbin, J. and Strauss, A. 2008. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, Inc, Thousand Oaks, CA, USA.

CSTA. 2012. CSTA k-12 computer science standards. Computer Science Teachers Association.

Dovier, A., Benoli, P., Brocato, M. C., Dereani, L. and Tabacco, F. 2016. Reasoning in high schools: Do it with ASP! In *CILC,* 205–213.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

Feaster, Y., Segars, L., Wahba, S. K. and Hallstrom, J. O. 2011. Teaching CS unplugged in the high school (with limited success). In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ACM, 248–252.

Febbraro, O., Reale, K. and Ricca, F. 2011. ASPIDE: Integrated development environment for answer set programming. In *Logic Programming and Nonmonotonic Reasoning – 11th International Conference, LPNMR 2011, Vancouver, Canada, 16–19 May 2011. Proceedings*, 317–330.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T. and Schneider, M. 2011. Potassco: The potsdam answer set solving collection. *AI Communications 24,* 2, 107–124.

Gelfond, M. and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, Cambridge.

Ginsburg, H. 1997. *Entering the Child's Mind: The Clinical Interview in Psychological Research and Practice*. Cambridge University Press, Cambridge.

Grover, S. and Pea, R. 2013. Computational thinking in K-12 a review of the state of the field. *Educational Researcher 42,* 1, 38–43.

Guzdial, M. 2004. Programming environments for novices. *Computer Science Education Research 2004*, 127–154.

Hemmendinger, D. 2010. A plea for modesty. *ACM Inroads 1,* 2, 4–7.

K-12 CS Framework Committee. 2016. K-12 computer science framework. URL: http://www.k12cs.org . [Accessed on January 8, 2017].

Kahn, K. 2004. Toontalk-steps towards ideal computer-based learning environments. *A Learning Zone of One's Own: Sharing Representations and Flow in Collaborative Learning Environments*. IOS Press, Amsterdam. 253–270.

Kelleher, C. and Pausch, R. 2007. Using storytelling to motivate programming. *Communications of the ACM 50,* 7, 58–64.

Kleinberg, J. and Tardos, E. 2006. *Algorithm Design*. Pearson Education, India.

Kowalski, R. A. 1982. Logic as a computer language for children. In *ECAI*, 2–10.

Kowalski, R. A. 1987. *Logic as a Computer Language for Children*. Wiley-Interscience, New York, NY, USA.

Kowalski, R. A. 2014. Logic programming. *Computational Logic, Volume 9 (Handbook of the History of Logic)*. North Holland, Amsterdam.

Levesque, H. J. 2012. *Thinking as Computation: A First Course*. The MIT Press, Cambridge.

Lye, S. Y. and Koh, J. H. L. 2014. Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior 41*, 51–61.

Marcopoulos, E., Reotutar, C. and Zhang, Y. 2017. An online development environment for answer set programming. *CoRR abs/1707.01865*.

Mendelsohn, P., Green, T. and Brna, P. 1990. Programming languages in education: The search for an easy start. *Psychology of Programming*. Elsevier, Amsterdam. 175–200.

Nichol, J., Dean, J. and Briggs, J. 1988. *Prolog, Children and Students*. Nichols Publishing Co., New York, NY, USA.

Papert, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M. and Paterson, J. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin 39,* 4, 204–223.

Resnick, M. and Rosenbaum, E. 2013. Designing for tinkerability. *Design, Make, Play: Growing the Next Generation of STEM Innovators*. Routledge, London. 163–181.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. and Kafai, Y. 2009. Scratch: Programming for all. *Communications of the ACM 52,* 11, 60–67.

Scherz, Z. and Haberman, B. 1995. Logic programming based curriculum for high school students: The use of abstract data types. In *ACM SIGCSE Bulletin*. Vol. 27. ACM, 331–335.

Sherin, B., diSessa, A. A. and Hammer, D. 1993. Dynaturtle revisited: Learning physics through collaborative design of a computer model. *Interactive Learning Environments 3,* 2, 91–118.

Smith, M. 2016. Computer science for all. URL: https://www.whitehouse.gov/blog/2016/01/30/computer-science-all.

Sterling, L. and Shapiro, E. Y. 1994. *The Art of Prolog: Advanced Programming Techniques*. MIT press, Cambridge.

Stutterheim, J., Swierstra, W. and Swierstra, D. 2013. Forty hours of declarative programming: Teaching Prolog at the junior college Utrecht. arXiv:1301.5077.

Vilner, T., Zur, E. and Gal-Ezer, J. 2007. Fundamental concepts of CS1: Procedural vs. object oriented paradigm – a case study. In *ACM SIGCSE Bulletin*. Vol. 39. ACM, 171–175.

Voogt, J., Fisser, P., Good, J., Mishra, P. and Yadav, A. 2015. Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies 20,* 4, 715–728.

Wing, J. 2011. Research notebook: Computational thinking – what and why? *The Link Magazine*, Spring. Carnegie Mellon University, Pittsburgh, PA, USA. URL: http://link.cs.cmu.edu/article.php?a=600.

Zhang, Y. 2015. Lecture notes for teaching computer science using asp. URL: http://www.myweb.ttu.edu/yuazhang/TexPREP/ASP.pdf.

https://doi.org/10.1017/S1471068418000431 Published online by Cambridge University Press