THE ROLE OF ARTIFICIAL INTELLIGENCE IN EDUCATING NOVICE PROGRAMMERS

Jack Weakley

TC 660H

Plan II Honors Program

The University of Texas at Austin

May 5, 2020

Roger Priebe, Ph.D.

Department of Electrical and Computer Engineering

Supervising Professor

Mary Eberlein, Ph.D.

Department of Electrical and Computer Engineering

Second Reader

ABSTRACT

Author: Jack Weakley

Title: The Role of Artificial Intelligence in Educating Novice Programmers

Supervising Professor: Roger Priebe, Ph.D.

Programming is an inherently difficult skill to acquire and develop. Those who attempt to learn programming may be easily discouraged. The current landscape for computer science education does not address the needs of every novice programmer. Literature reports a discrepancy between student misconceptions and instructors' perceptions of those misconceptions. Those who can afford a one-on-one human tutor perform on average two standard deviations better than those who learn via conventional methods, suggesting there is a need for a comparable, cheaper replacement. As a result, a number of intelligent tutoring systems have been developed for the purpose of teaching introductory programming concepts and replicating the benefits of one-on-one human tutoring. In this thesis, we analyze and discuss the literature pertaining to student misconceptions, selecting five fundamental misconception categories for introductory programming to demonstrate the effectiveness of existing intelligent tutoring systems. The features of existing intelligent tutoring systems are discussed and analyzed with respect to their effectiveness in addressing student misconceptions. Finally, we highlight the current gap in research on intelligent tutoring systems, hypothesizing the architecture and features of an ideal intelligent tutoring system for introductory programming.

Acknowledgments

Thank you to my family. Dad, Mom, Ryan, Dylan, Maddie, and Brynn have continually supported me in every facet of my life. I would not be where I am now without any of you.

Thank you to my advisors. Dr. Priebe and Dr. Eberlein have been instrumental in the creation of this thesis. Thank you for taking the time and care to nurture this project as it has developed. The positive impact of your guidance and insights cannot be overstated.

Table of Contents

Chapter 1: Introduction

Over the last several decades, technological innovations like the internet have fundamentally changed industries across the board. Beyond commerce, our daily lives have been altered in such a way that there is no denying we now live in a digital world. The sheer volume of information bouncing between devices each and every second can be hard to believe even today. This evolution has understandably led to a much higher demand for software engineers across the globe, a trend that is expected to continue. The increased demand for software engineers has in turn increased salaries and therefore student interest in software engineering as a career.

Today, instructors are faced with a larger than ever pressure to effectively educate larger number of students. Additionally, recent research shows that there is a disparity between what instructors think their students struggle with and what actually causes them issues (Kaczmarczyk et al., 2010). Because instructors are faced with teaching such a large number of students, they cannot spend a significant amount of one-on-one time with all their students to address their misconceptions. Those who have the resources to consult a one-on-one tutor perform on average two standard deviations better than students who learn via conventional methods (Bloom, 1984). This disparity, known as Bloom's 2 sigma problem, was initially reported in 1984 by educational psychologist Benjamin Bloom. Obviously, not everyone has access to a one-on-one human tutor, suggesting there is a need for a comparable, cheaper replacement.

In the last few decades, the level of research into Intelligent Tutoring Systems (ITSs) has substantially increased as researchers continue to look for an alternative to human tutors across all disciplines. ITSs are systems designed to replicate the effectiveness of human tutoring within a digital environment with features that adapt to user inputs in order to personalize instruction (Crow et al., 2018).

This thesis analyzes the viability and feasibility of ITSs for educating novice programmers against the backdrop of current tools and approaches for computer science education. The key research goals are to demonstrate that existing ITSs offer concrete help on specific topic areas within introductory programming and to formulate the structure of a more general ITS that addresses all the misconceptions of a novice programmer.

The following chapter of the thesis provides the necessary context for our question: what is the current state of computer science education? We outline the distinct educational options afforded to introductory programming students. Additionally, we compare the feasibility of these options for the average novice programmer to highlight the current shortcomings of computer science education. Ultimately, this section will reveal that there is a need for a cost-effective replacement for a one-on-one human tutor.

The third chapter of the thesis provides a detailed literature review on two topics: student misconceptions and ITSs. First, this chapter elucidates novice programmers' misconceptions and errors so that we can better understand the topic areas that trouble students. In order to address the effectiveness of any approach or tool, we must first identify the target problem area. Next, we provide further details about ITSs for programming education, more specifically the categories of AI-supported tutoring approaches that can be applied: example-based, simulation-based, dialogue-based, program analysis-based, feedback-based, and collaboration-based. Beyond employing one or more of the aforementioned approaches, existing ITSs for programming education primarily differ in whether they adapt feedback, navigation, or both, as well as any supplementary features they may have. We discuss in detail the various types of adaptive feedback. The second section of this chapter provides a review of existing ITSs for programming, specifically how they can address specific student misconceptions, the degree of adaptability of the systems, and any shortcomings they may have.

The fourth chapter demonstrates the effectiveness of existing ITSs in addressing specific misconceptions. Based on the findings from the literature review, we select five specific topics or categories of misconceptions that we believe to be fundamental for introductory programming. Next, we discuss how specific features of existing ITSs are well suited to address each of these misconception categories. Additionally, we hypothesize features for ITSs that would be particularly useful in addressing these specific misconceptions. Ultimately, this chapter demonstrates how existing ITSs are able to provide individualized instruction that address specific misconceptions.

The final chapter looks to the future promise of ITSs for introductory programming. What would the ideal ITS for introductory programming actually look like in terms of structure and features? What additional research should be conducted to get there? Ultimately, we can't answer that question currently, but as the literature review will reveal, there are features and structures of existing ITSs that provide the basis for an ideal ITS for programming education. Based on our analysis of existing ITSs for programming education and the generic architecture outlined by Pillay (2003), we hypothesize the structure and features of an ideal ITS for introductory programming education.

Chapter 2: The Current State of Computer Science Education

Education – like the demand for computer scientists – has dramatically changed as a result of technological innovations that allow us to more conveniently receive and distribute information. Prior to the invention of the internet, there were few media through which educational materials could be presented or taught. Ultimately, those who wished to know how to program had to learn in a traditional classroom setting or learn by scouring through lengthy and dense textbooks. If you were lucky, you might have been able to afford a professional one-on-one tutor, which - as mentioned in the previous section – would have provided a significant advantage over traditional methods.

Today, the educational landscape for programming differs quite significantly. While the aforementioned options remain available to a novice programmer, the internet as an educational medium has become increasingly attractive for those with substantial motivation and for those who may understandably balk at the cost of higher education in the 21st century. From unstructured online platforms like YouTube or Stack Overflow to structured online courses like Codecademy or Coursera, novice programmers now have the opportunity to learn programming fundamentals outside the confines of a traditional classroom at zero to minimal cost. However, there are certainly drawbacks associated with self-teaching programming on the internet. As mentioned, motivation is key. There are simply fewer motivating factors in an online learning environment compared to a traditional classroom environment (Guyan, 2013). When using either unstructured or structured online resources, there is no requirement that you be physically present and attentive as is typically the case in traditional classroom settings. With unstructured resources, there is no due date by which you must learn whichever topic you may be trying to tackle. While it requires motivation to learn anything, the traditional classroom setting places a greater burden of responsibility on the student, which may further motivate some students while discouraging others.

It is important to make a distinction between the efficacies of an unstructured online approach and a structured online approach. When using unstructured resources, students may have a difficult time ensuring that they are covering all the fundamental topics of an introductory programming class. Further, introductory programming topics typically build on top of one another, suggesting that there is a wrong order in which to learn topics. For example, you wouldn't learn how recursion works before you learned how a simple function call works. An unstructured online approach can often lead to a student attempting to learning topics in the wrong order. Additionally, a recent study indicates that the use of structured resources is more beneficial to a novice programmer than unstructured resources (Hsu & Hwang, 2017). With these concerns in mind, the unstructured online approach seems best suited as a supplement to either a structured online approach or the traditional classroom experience. However, every student is different; this is not to say that no novice programmer can learn how to code by only watching YouTube videos. Simply put, the expectation that the average novice programmer be able to teach themselves how to program with unstructured resources appears unreasonable.

While a structured online approach appears overall more effective than an unstructured one, it shares one of the drawbacks of a traditional classroom approach: a lack of significant one-on-one time between the professor and his or her students. However, if a student fails to understand a specific topic within a structured online approach, there is no physical professor with which to communicate his or her misunderstandings. For the student who may require additional assistance on difficult topics, a purely online approach may fall short. While the average novice programmer may be able to learn programming solely within the confines of a structured online approach, not every student will find the same success.

While the traditional classroom setting has remained a viable educational medium for many novice programmers, as previously mentioned, the amount of one-on-one time an instructor can spend on every student has decreased. However, the same innovations that have led to this problem have significantly changed the approaches and tools instructors use in their courses. To address grading the programs of a large number of students, researchers have developed a number of Automatic Assessment (AA) systems that both evaluate students' programs and provide instant feedback on their errors (Ullah et al., 2018). Despite the fact that AA systems are typically used for evaluation rather than instruction, they are included here because - like ITSs for programming – they provide automated feedback on student submitted programs.

Additionally, pair programming has become increasingly popular, as more and more professors and professional engineers share their success stories on the web. Pair programming refers to the software development technique where two developers work together to solve a development problem (Plonka et al., 2015). Typically, one person – the 'driver' – will sit at the machine and write down the code, while the other person – the 'navigator' – will actively perform code review on the driver's code. The roles are reversed every thirty minutes to an hour. Pair programming has been implemented in large classes so that students can talk through their thought process with each other when trying to solve a development problem. Additionally, there is evidence that pair programming provides significant knowledge transfer benefits (Plonka et al., 2015). However, what happens if neither of you knows the answer to some problem?

As we have seen with all of the aforementioned educational paths, at some point the student may have exhausted all potential approaches to learn how to program such that a one-on-one tutor appears like the only option left. As Bloom notes, those who learn with a one-on-one tutor perform far better than those who learn with conventional group methods of instruction. This statistic highlights that most students have the potential to reach a high-level of learning, but unfortunately, most students don't have access to teaching-learning conditions that enable such great performance (Bloom, 1984). This leads to the fundamental question of improving education: How do we create the teaching-learning conditions that maximize student success accessible and affordable to all? Clearly, we can't make human tutors accessible to everyone who would like assistance, but perhaps the cause of our fundamental problem – technological innovation – can assist in solving. However, before we look to innovations that could fundamentally improve the pedagogy of programming, we need to first understand the problem. What aspects or topics of introductory programming are causing novices difficulty?

In the following chapter, we will analyze student misconceptions and errors through a combination of large-scale data and student interviews, as well as discuss the body of literature for ITSs for programming education. What topics are inherently challenging for novice programmers and how do student misconceptions manifest themselves in tangible errors? The discussion of student misconceptions will inevitably reveal the effectiveness of ITSs for specific problem areas. Additionally, data on student misconceptions can be used to train AI algorithms used in ITSs and help us begin to understand why students are making certain mistakes. Following the literature review for student misconceptions, we define Intelligent Tutoring System (ITS). More specifically, what are the components that comprise an ITS for introductory programming? What are the various AI-supported tutoring approaches that an ITS can employ?

In what ways can ITSs individualize instruction? Lastly, I will move into a review of existing ITSs for programmers, explaining their differences and how they address specific misconception categories.

Chapter 3: Literature Review

3.1 Student Misconceptions

In this section, we will uncover misconceptions and errors frequently encountered by novice programmers. We will begin by describing and discussing the learning objectives that students are expected to achieve in an introductory programming class, so that we can contextualize misconceptions and errors. Next, we will discuss the common misconceptions that computer science students have about the execution of a program. Lastly, we will discuss the most frequent errors computer science students encounter and the severity of those errors, more specifically, how long it takes students to fix errors. At the end of this section, we will gain a greater understanding of what aspects of computer science are particularly challenging for novice programmers. More importantly, we hope to begin to understand why these aspects are challenging for novice programmers, which will inform our analysis of the feasibility and viability of ITSs.

The set of learning objectives for a subject is necessary to create a "concept inventory" (CI), which is a multiple-choice test designed to determine whether a student has an accurate knowledge of a specific set of concepts. The first CI, known as the Force Concept Inventory, was created to identify students' misconceptions about Newton's Laws of Force. CIs allow us to more deeply understand the thought process and knowledge of students. Moreover, they illuminate students' misconceptions of specific learning objectives, providing important feedback for instructors. Defining the general set of learning objectives that a student is expected to achieve in an introductory programming course is inherently difficult due to the variety of languages and pedagogical approaches that can be used. While certain approaches are widely used (e.g., Objects-first), these approaches still depend on the mechanics and environment of the selected programming language. We are concerned with programming fundamentals independent of language, so it is necessary we define a set of learning objectives applicable to all novice programmers.

The development of CIs for introductory programming is still beginning. In their pioneering work, Goldman et al. (2008) attempt to identify important and difficult concepts in programming fundamentals using a Delphi process to form the foundation for an introductory programming concept inventory. “A Delphi Process is a structured [multi-step] process for collecting information and reaching consensus in a group of experts” (Goldman et al., 2008, p. 257). Using the scores of importance and difficulty for learning objectives provided by the experts, Goldman et al. (2008) found 11 topics on which the expert group reached a consensus as being important and difficult. Of the 11 topics, only one “is exclusively related to object-oriented paradigms or languages” (Goldman et al., 2008, p. 258), suggesting the other 10 topics could be used to create a programming fundamentals concept inventory that is broadly applicable. The 10 topics are listed below:

1. Parameter scope, use in design
2. Procedure design
3. Issues of Scope, local vs. global
4. Abstraction/Pattern recognition and use
5. Recursion, tracing and designing
6. Memory model, references, pointers
7. Functional decomposition, modularization
8. Conceptualize problems, design solutions
9. Debugging, Exception Handling

10. Designing Tests

Here we have the first clear example list of learning objectives as a foundation for an introductory programming CI. Using these topics as the basis for an introductory programming CI would allow instructors to gain insight into their student's misconceptions, which can provide important feedback on the effectiveness of their methodologies. However, it should be noted there are inherent difficulties in developing an introductory programming CI independent of syntax, which has led researchers such as Goldman et al. (2008) to suggest the development of "topic-specific sub-inventories, as complements to a universal CI" (p. 258). In the remainder of this section, we will discuss the background of research into student misconceptions, highlighting recent research that employs a concept inventory approach.

While the subject of an introductory programming CI is relatively new, student misconceptions of programming fundamentals have been studied for some time. Early research conducted by Bayman and Mayer (1983) examined students' misconceptions about specific program statements in BASIC. Additional work by Bonar and Soloway (1985) examined student understanding of programming using more general groups of statements. However, as Kaczmarczyk et al. (2010) note, "these studies touched upon misconceptions, but their primary focus was on larger mental models and theories of cognitive representation, or in some cases discoveries of misconceptions were not fully followed up pedagogically" (p. 108). Spohrer and Soloway (1986) looked at programming errors and bugs to determine if they result from student misconceptions, concluding that bugs are more likely to arise due to a misunderstanding of problem specifications. Recent research has mostly focused on a narrow subset of programming such as parameters, mental models of assignments, and Objects.

None of the above research was able to comprehensively define student misconceptions about programming fundamentals. While there are certainly shortcomings with the current body of programming fundamental CIs, they provide the most robust and ranging characterization of student misconceptions. Kaczmarczyk et al. (2010) conducted ten student interviews at the University of California, San Diego in spring 2009 to reveal misconceptions on the ten topics identified in the previous section. Additionally, they hoped to validate the findings of Goldman et al. (2008) that these topics were indeed challenging. While not all questions required actual code, it was necessary that the researchers choose a language for code examples despite their goal to develop a CI that is as language neutral as possible. They chose Java for three reasons: (1) it is one of most widely used introductory programming languages, (2) the volunteer interviewees had been taught Java, and (3) the Delphi experts recognized Object Oriented Programming (OOP) contained a number of challenging concepts. In their results, Kaczmarczyk et al. (2010) identify four themes among the students' misconceptions but focused on student misunderstanding of memory usage. The four themes are listed below:

1. Students misunderstand the relationship between language elements and underlying memory usage.
2. Students misunderstand the process of while loop operation.
3. Students lack a basic understanding of the Object concept.
4. Students cannot trace code linearly.

However, the key data for creating a CI are the specific misconceptions, as they are used to create distracter answers for the multiple-choice questions.

Kaczmarczyk et al. (2010) focus on three misconceptions that fall under the first theme. "The first misconception, 'Semantics to semantics,' … occurred when the student inappropriately assumed details about the relationship and operation of code samples, although such information was neither given nor implied" (Kaczmarczyk et al., 2010, p. 109). Examples include assuming a relationship between variables when examining a list of Java variable definitions whose relationships are unstated, making incorrect assumptions about connections between variables leading to type errors, ignoring a variable that does not coincide with the student's assumption of how the variables should relate, among others. Interestingly, Kaczmarczyk et al. (2010) note that when their assumptions led to a contradictory conclusion, students were unable to realize their incorrect assumptions were the source of the contradiction. One student went so far as to assume that the syntax must be logically incorrect rather than walk back their previous semantic assumptions. This event where students use outside assumptions to tackle problems is well documented and not uncommon but ultimately impedes the learning of programming fundamentals. Miller and Settle (2019) conducted a study asking students to perform a simple code-writing task that required constructing references to objects and their attributes. When the nature of the attributes in the tasks where changed from identifying attributes to descriptive attributes, students routinely omitted the name of an identifying attribute while attempting to reference its value. This mistake is consistent with the use of metonymy in the context of human language. For example, if someone said "open the chips and put them in a bowl", most people infer that the speaker is asking for the bag of chips to be opened, not the chips themselves.

"The second misconception, "Primitive no default," … relates to lists of instance variables" (Kaczmarczyk et al., 2010, p. 110). This misconception is Java specific and manifests in one example where a student assumes Boolean variables without assigned values have no value. "The third misconception, "Uninstantiated memory allocation," … reveals itself when students think that memory is allocated for Objects which have been declared, but not instantiated” (Kaczmarczyk et al., 2010, p. 110). While both previously mentioned misconceptions relate to OOP, they reinforce the concept of memory usage as particularly challenging. Moreover, these misconceptions confirm the Delphi experts’ choice of the difficult concepts in introductory programming.

In a more recent study, Caceffo et al. (2016) provide a more robust list of misconceptions using the findings from two course exams and multiple educator interviews. They looked at an introductory programming course at the State University of Campinas (Brazil) where students were taught to program in C, so no OOP-related misconceptions were addressed. They identify seven specific misconception categories containing one or more concrete misconceptions after analyzing student responses from two course exams taken individually by 66 and 60 students. Following this analysis, the researchers conducted interviews with five course instructors, where they uncovered an eighth misconception category. The misconception categories are listed below:

1. Function and Parameter Use and Scope
2. Variables, Identifiers, and Scope
3. Recursion
4. Iteration
5. Structures
6. Pointers
7. Boolean Expressions
8. Syntax vs. Conceptual Understanding

The largest source of misconceptions came from the first category: Function Parameter Use and Scope. Within that category, Caceffo et al. (2016) identify three distinct misconceptions. The first and most common misconception was a lack of understanding of the difference between call by reference and call by value. In C programming, parameters are call-by-value, meaning that the value of the parameter and not the reference is passed into the function scope. This led to students thinking they could change the content of a formal parameter inside a function, believing this would also change the actual parameter variable outside the function. The second and third misconceptions relate to students failing to understand the initialization of formal parameters and that parameters are local variables. In addition to scope, these misconceptions directly relate to students' misunderstanding of the underlying memory mapping of variables similar to the findings of Kaczmarczyk et al. (2010). Most importantly, these findings illuminate the bridge between theory and practice in introductory programming. Students may fully and completely understand the algorithm that they should use to solve a problem, but still they lack the knowledge to implement a working solution within the boundaries of a given programming language. The second category (Variables, Identifiers, and Scope) uncovers similar misconceptions. Students do not understand that variables declared within a function are limited to the scope of the function. Like the misconceptions in the last category, students struggle to understand the scope of variables in a program.

The third category of misconceptions relates to recursion. Students developed recursive solutions that led to an infinite loop. Students failed to select the correct stopping condition. Students did not even implement self-reference. While none of these misconceptions comprised more than 15% of students, it reinforces the Delphi experts' categorization of recursion as a difficult concept. Additionally, the errors and misconceptions here are entirely language independent, suggesting student misconceptions about recursion persist across languages and perhaps teaching methodologies.

The fourth category (Iteration) includes specific misconceptions about basic loop behavior and the logic of loop counting. Students called functions inside of the loop instead of outside. Students failed to write a solution that would loop the correct number of times. However, these misconceptions were present among only 14% of students. While iteration is a fundamental topic for introductory programming, the Delphi experts did not recognize it as particularly challenging. The fifth category of misconceptions relates to C-style structures. Because the misconceptions here are very language specific, we will not address them.

The sixth category of misconceptions (Pointers) again highlights issues related to memory management and mapping. Students failed to return values through pointers and assign values to pointers. Again, we see the event of students applying incorrect outside assumptions (e.g., metonymy) to programming problems, ultimately leading to a contradictory and incorrect answer.

The seventh category relates to Boolean expressions. Students failed to understand Boolean expressions' meaning or their evaluations. This misconception appeared when students were unable to translate a sentence into a Boolean expression. Similar to iteration, Boolean logic is a language-independent fundamental of introductory programming but not classified as particularly challenging by the Delphi experts.

In their interviews with the course instructors, Caceffo et al. (2016) uncovered an eighth category of misconceptions: Syntax vs. Conceptual Understanding. As mentioned previously, some misconceptions seem to arise due to a lack of knowledge about language-specific syntax rather than failing to understand a concept. The nature of programming makes it difficult to remove the language-specific elements of a CI. In the remainder of this section, we will discuss research into the specific errors made by students in introductory programming classes with the hope that they will verify the findings about student misconceptions.

Previously, we discussed the important and challenging concepts that a student is expected to learn in an introductory programming class. Then, we discussed research that analyzed student misconceptions relating to those concepts. However, the work of Kaczmarczyk et al. (2010) and Caceffo et al. (2016) focused on a small number of students. In order to verify that these misconception categories and specific misconceptions are not anomalies, we must look to large-scale studies of student errors. It should be noted there is inherent difficulty in gleaning student thought process, as researchers analyzed code submitted by students rather than interviewed students to learn what they are thinking.

Brown and Altadmri (2017) analyzed findings from their Blackbox data collection project, which at the time had "captured data from almost 100 million compilations across more than 10 million programming sessions from users learning Java in the beginner's programming environment BlueJ" (p. 2). From their previous work, Brown and Altadmri (2017) identified 18 mistakes informally categorized into syntax, semantics, and type errors. While most of the errors relate to syntax, we believe that a lack of knowledge of syntax is less pressing than semantic misconceptions, as there are typically quick fixes for syntax errors. With the data from the Blackbox project, Brown and Altadmiri (2017) ranked these 18 mistakes by severity, which is a function of frequency multiplied by the median time-to-fix. For mistakes that took longer than 1000 seconds to fix or were never fixed, a ceiling of 1000 seconds was used.

The most severe error was syntactic (unbalanced parentheses, curly or square brackets), but this was due to the sheer frequency of the error. The median time-to-fix was only 17 seconds.

However, two of the top five errors were semantic with both having a median time-to-fix of 1000 seconds. Students used “==” instead of “.equals” to compare strings. Students ignored or discarded the return value of a method that has a non-void return type. The first error occurs when students do not understand that Strings are non-primitive variables in Java. Ultimately, it is difficult to map either of these errors (along with syntax and type errors) to misconceptions outlined in the previous section, because they are so language-specific. However, this article does uncover interesting patterns about instructor opinions of student mistakes when compared to the Blackbox data. Educators have only a weak consensus about the error frequencies and times-to-fix, and the educator’s level of experience had no effect on the accuracy of their opinions. Large-scale data provides valuable information about student syntax errors that could help train algorithms used by ITSs, but it reveals little about specific student misconceptions. Large-scale data does however provide valuable information to educators about the accuracy of their opinions on student misconceptions.

In this section, we focused on learning objectives and student misconceptions in introductory programming. Ultimately, our discussion of large-scale data did not reinforce the misconception categories and specific misconceptions mentioned in part two due to the language-specificity of the errors. However, large-scale data did illuminate a discrepancy between educators’ beliefs of student errors and the errors themselves.

Among the misconceptions discussed, memory mapping of variables and scoping of parameters and variables have proven to be particularly challenging topics for students in introductory programming. Additionally, Caceffo et al. (2016) and Kaczmarczyk et al. (2010) verify that the learning objectives mentioned are in fact important and challenging. During our discussion of the body of work in student misconceptions, researchers commonly noted that there is inherent difficulty in determining whether or not an error arose due a lack of syntactic knowledge or due to a lack of conceptual knowledge. Further research is needed in order to categorize student errors as syntax or semantics, so that we can understand the source of these errors. In the following section, we discuss the body of literature on Intelligent Programming Tutors (IPTs) and Intelligent Tutoring Systems (ITSs).

3.2 Intelligent Tutoring Systems

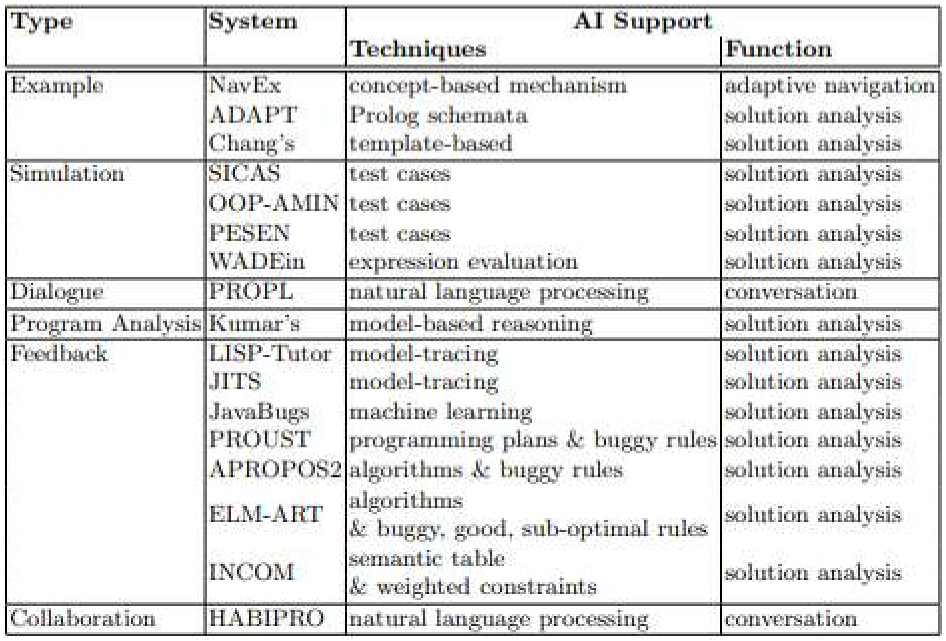
Intelligent Tutoring Systems (ITSs) aim to replicate the effectiveness of one-on-one human tutoring within a digital environment (Crow et al., 2018). In order to replicate these benefits, ITSs include features that adapt to users to provide personalized instruction and feedback. A number of ITSs have been developed for programming education. For the remainder of this thesis, we will refer to these specific ITSs as Intelligent Programming Tutors (IPTs). According to a meta-analytic review of 50 such systems, ITSs demonstrate benefit effect sizes similar to that of human tutors (Kulik & Fletcher, 2016). Nesbit et al. (2014) published a review of the effectiveness of ITSs. While this review did not solely analyze ITSs for programming education, a large proportion of the articles related to ITSs for programming education. This demonstrates that there are multiple quantitative studies reporting the benefits of intelligent tutoring for programming education. The field of programming poses unique problems that make IPTs a distinguishable subfield within ITSs (Crow et al., 2018). For example, most IPTs perform complex tasks like providing hints on the semantics and syntax of a student program. Additionally, IPTs are unique in that they typically require an environment in which to produce and run code.

Existing IPTs have been developed to teach specific languages and concepts. Additionally, IPTs differ in what aspects of feedback and instruction are adapted for the specific user. For example, some IPTs only adapt task recommendation, whereas others provide both navigational support and hints. IPTs incorporate a large range of different types of features in their structure, and each IPT may incorporate a different set of features. Crow et al. (2018) argue that there are two broad categories by which IPTs can be grouped: IPTs that primarily adapt feedback and hints, and IPTs that primarily adapt navigational support. However, these categories do not exactly align with individual features that are found in these existing systems. Additionally, Crow et al. (2018) make note of "supplementary" features in IPTs: features that are not necessarily required to complete interactive programming tasks and features that are not found in all IPTs.

Pillay (2003) published an article detailing a general outline for developing IPTs for novice programmers. The main goal of the article was to propose a generic architecture in which the system is broken down into modules with a generic inter-module communication language so that it can be deployed to tutor different languages in different environments or platforms. This article highlights the gap in the field of IPTs. There are few generic architectures that can be reused, and instead most existing IPTs are custom-built for single implementations.

Le et al. (2013), in their review of AI-Supported tutoring approaches for learning programming, propose a set of a categories for approaches to tutoring: example-based, simulation-based, dialogue-based, program analysis-based, feedback-based, and collaboration-based. These categories are valuable in comparing IPTs, as there is significant variation in what aspect of the tutoring is made intelligent. For reference, the table below describes specific systems that employ one of the aforementioned AI-supported tutoring approaches, as well as the specific techniques used to realize an approach and the overall function of the system.

Table 1: AI-supported Tutoring Approaches (Le et al., 2013)



Type	System	AI Support	
		Techniques	Function
Example	NavEx	concept-based mechanism	adaptive navigation
	ADAPT	Prolog schemata	solution analysis
	Chang's	template-based	solution analysis
Simulation	SICAS	test cases	solution analysis
	OOP-AMIN	test cases	solution analysis
	PESEN	test cases	solution analysis
	WADEin	expression evaluation	solution analysis
Dialogue	PROPL	natural language processing	conversation
Program Analysis	Kumar's	model-based reasoning	solution analysis
Feedback	LISP-Tutor	model-tracing	solution analysis
	JITS	model-tracing	solution analysis
	JavaBugs	machine learning	solution analysis
	PROUST	programming plans & buggy rules	solution analysis
	APROPOS2	algorithms & buggy rules	solution analysis
	ELM-ART	algorithms & buggy, good, sub-optimal rules	solution analysis
	INCOM	semantic table & weighted constraints	solution analysis
Collaboration	HABIPRO	natural language processing	conversation

An example-based approach typically includes explaining example problems and solutions to the student and then prompting him or her to solve similar problems. This allows students to retool learned ways of solving problems to tackle new problems that are similar in nature. Yudelson and Brusilovsky (2005) applied the example-based approach in the development of a web-based tool for exploring programming examples known as NavEx. NavEx provides the student with a programming example annotated with explanations for each important line of code. Students can open the explanation line-by-line or only click on the explanations for lines they struggle to understand. Compared to programming examples provided in a typical textbook, NavEx has a number of advantages. 1) The code for an example can be shown as a block, instead of dissected by comments. This allows students to form their own judgements on the block of code without influence before they can verify their understanding. 2)

Students can access explanations in whichever way best addresses their needs. 3) Students can learn in an exploratory manner rather than reading examples passively. Additionally, NavEx provides adaptive navigational support. Each student is recommended programming examples that correspond with their knowledge level. This adaptive navigation relies on a concept-based mechanism. To apply this technique, a list of programming concepts is identified for each example, with each concept categorized as either pre-requisite concepts or outcome concepts. This separation is supported by the structure of a specific course, defined by the teacher assigning examples in accordance with lecture topics. NavEx's algorithm advances through lectures sequentially until all concepts are effectively covered. Using the pre-requisite concepts and outcome concepts for a specific example and the current state of the user model, the next recommended example can be determined.

Another type of example-based tutoring prompts the user to fill-in-the-blanks of a solution template for a given example. To apply this approach, Gegg-Harrison (1993) developed the ADAPT tutoring system for the programming language Prolog. Utilizing the existence of Prolog schemata, which represent solutions for a class of problems, ADAPT introduces several solution examples and highlights the significant components of the underlying schema. Then, ADAPT prompts the student to solve a similar problem given a solution template with blank lines to be filled. If the student cannot complete the template, ADAPT breaks the template down into its recursive functional components. If the student still cannot complete the template, ADAPT provides a more specific template with some of the solution already filled in. Lastly, if the student cannot complete the template, ADAPT provides and explains the complete solution. The use of a library of Prolog schema allows the system to diagnose errors in students' solutions. First, the system attempts to identify the algorithm implemented in the student solution. "It uses the normal form program for each schema in the schema library to generate a class of representative implementations and then transforms the most appropriate implementation into a structure that best matches the student solution" (Le et al., 2013, p. 3). Following, the systems attempts to explain discrepancies between the student solution and the transformed normal form program produced in the previous phase using a hierarchical bug library.

In addition to a completion strategy, Chang et al. (2000) proposed modification and extension tasks. Modification tasks ask you rewrite a program to simplify or improve it, whereas extension tasks ask you to rewrite a program to account for new criteria or commands. For each problem, this system defines a model program with templates that represent basic programming concepts. Then, the generated model can be used to create modification and extension tasks. Discrepancies between the generated model and the student solution are considered errors. Lastly, the system provides feedback on the student solution.

Simulation-based approaches help students visualize the execution of a piece of code. Such approaches include lowering the abstraction level of programs such that students can program the movements and behavior of concrete entities. This allows students to develop their foundational programming skills for use in more complex environments. Moreover, the visualization of simulation-based approaches can be particularly beneficial to students who struggle to trace code linearly.

Recent research into simulation-based approaches have led to new versions such as engaging visualization, explanatory visualization, and adaptive visualization. Engaging visualization highlights the active role of the student in learning programming. Explanatory visualization annotates visual representations with text explanations. Adaptive visualization adapts visual representations to the knowledge level of the student and the difficulties of the underlying concept. A number of simulation tools employing the first two approaches have been developed to expand on the benefits of basic visualization. One such system, PESEN, aims to teach basic programming concepts like sequential, conditional, and repetition execution (Le et al., 2013). Focusing on algorithm design and implementation, PESEN prompts the student to build a flowchart that represents a solution for a given problem. Then, the algorithm is simulated so that the student can visualize then verify the correctness of his or her algorithm. For each problem there exists one or more algorithms that serve as a solution and some test cases. The test cases allow the student to verify the correctness of his or her algorithm, while the algorithm library allows the system to recognize errors in the student solution.

Another system, WADEin, provides adaptive and engaging visualization (Brusilovsky & Spring, 2004). WADEin was developed to help students explore the process of expression evaluation in the programming language C. Students can type in expressions or select them from a menu, then the system visualizes the execution of each operation. Or students work with a given expression, indicating the order in which operators execute. Then, the system shows the correct order of execution, evaluates the expression and visualizes it.

One way to tutor students is through mixed-initiative dialogues, where both the student and the tutor can initiate questions. Lane and VanLehn (2005) developed PROPL, a virtual tutor which guides students to formulate a natural-language style pseudocode solution to a given problem. The system prompts the student with four types of questions: “1) identifying a programming goal, 2) describing a schema for attaining this goal, 3) suggesting pseudocode steps that achieve the goal, and 4) placing the steps within the pseudocode” (Le et al., 2013, p. 5). Throughout the conversation, the system attempts to correct student errors and misconceptions. For example, if a student does not correctly answer a question, the system will initiate a sub-dialogue to attempt to solicit a more accurate answer. Sub-dialogues include asking students to complete incomplete answers, refine vague answers, or redirect students to more relevant concepts. PROPL depends on a library of Knowledge Construction Dialogues (KCDs) to dynamically understand student answers and communicate using natural language. KCDs "consist of a sequence of tutorial goals, each realized as a question, and a set of acceptable answers to those questions" (Le et al., 2013, p. 5). Each student answer is either classified as the correct response or associated with another KCD that performs correction of errors or misconceptions. Le et al. (2013) describe KCDs as employing a hierarchical structure and recursive, finite-state based approach to dialogue management.

As opposed to the previous approaches that require students to produce a program or algorithm, the program-analysis based approach teaches students through the analysis of an existing program. Kumar (2005) developed an ITS for the C++ programming language that analyzes and debugs C++ code segments. The system prompts the user with different types of tasks such as predicting the result of a program, evaluating an expression step-by-step, identifying bugs in a program, and evaluating the program line-by-line. This system uses model-based reasoning to verify the correctness of a student solution. A model of a C++ domain consisting of the structure and behavior of the domain is used to simulate the correct behavior of an artifact in the domain (i.e., the expected behavior of a program). The system then takes the discrepancies between the behavior predicted by the student and the behavior predicted by a C++ domain model to hypothesize the mental model of student, which is used to generate feedback for the student.

Feedback on student input is an important aspect of any form of instruction. Simple unit testing can verify the correctness of a program, providing binary feedback on whether a solution is correct or incorrect. On the other hand, adaptive feedback is dynamic. In general, adaptive feedback should provide different information for different responses in addition to verifying the correctness of a response. The adaptive feedback of an IPT aims to replicate the dynamic nature of one-on-one human tutoring. Put more simply, different students require different kinds of assistance. Two solutions for a given programming exercises could both be incorrect but for completely different reasons. Adaptive feedback attempts to explain why a given student's response is incorrect by highlighting differences between the correct solution and the student's solution rather than simply providing the results of unit testing.

A number of AI-supported approaches provide adaptive feedback on student solutions as the only means for tutoring. In order to analyze student solutions to provide feedback, systems need a domain model to which error diagnosis techniques can be applied. We will discuss the following AI techniques as they have been applied to feedback-based learning systems: model-tracing, machine learning, libraries of plans and bugs, and a weighted constraint-based model.

The basis of a model-tracing learning system is the cognitive model which consists of "ideal rules" and "buggy rules" (i.e., the correct problem-solving steps of an expert and the typical incorrect problem-solving steps of a student respectively). Following the submission of a student solution, the system generates a set of correct and buggy solution paths. If the student solution path can be identified as belonging to a correct path, the student can move on to the next problem. Otherwise, the system attempts to guide the student toward a correct solution path. The first known Intelligent Programming Tutor (IPT) developed by Anderson and Reiser (1985) for LISP employed this model-tracing technique. The LISP Tutor prompts the student with a problem description containing identifiers for functions and parameters to be used in the solution. The student completes the problems in a structured editor through a sequence of templates. Building on the same approach, Sykes (2006) developed an IPT for Java called JITS. The system dynamically analyzes student solutions to determine the appropriate feedback with respect to the compilation rules of Java. Rather than verifying the semantic requirements specific to a given programming problem, JITS models the grammar of Java to correct compiler errors in student solutions.

To address the time-consuming nature of generating a set of "buggy rules" for a given programming problem, Suarez and Sison (2008) applied a machine learning approach to generate a library of typical Java errors made by novice programmers. The researchers then utilized this bug library to develop a system to examine a small Java program. JavaBugs identifies errors in a student solution in two phases. First, the system compares the student solution to a set of reference programs to identify which one is most similar to the student solution. This allows the system to hypothesize the solution strategy or intention of the student. Following, the system analyzes differences between the reference program and the student solution. Lastly, based on these discrepancies, the system identifies new misconceptions that are in turn used to update the bug library and return feedback to the student.

Some learning systems rely on a library of programming plans and bugs to model domain knowledge and identify student errors. To identify student errors, this class of systems follows similar principles to the model-tracing technique: "1) modeling the domain knowledge using programming plans or algorithms and buggy rules, 2) identifying the student's intention, and 3) detecting errors using buggy rules" (Le et al., 2013, p. 7). In a tutoring system for Pascal called PROUST, a set of programming goals and data objects internally represent a given programming problem (Johnson, 1990). Programming goals are realized by programming plans, which represent a method of implementation that satisfies said programming goals. Other systems such as APROPOS2 rely on the concept of algorithms rather than programming plans to represent the means for solving a given programming problem (Le et al., 2013). Beyond the use of programming plans or algorithms, systems of this class model domain knowledge using common bugs collected from empirical studies. In addition to buggy rules, ELM-ART adds rules of two more types: good rules and sub-optimal rules. Good rules are used to comment on good programs, while sub-optimal rules are used to comment on less-efficient programs with respect to run-time complexity or space complexity.

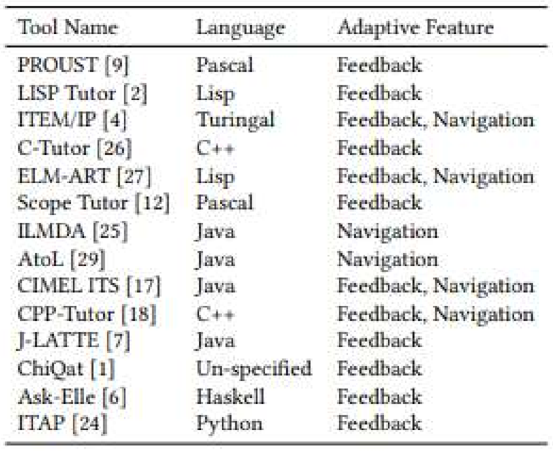
Once the domain model has been defined by either programming plans or algorithms, systems can analyze student solutions by identifying the student intention and detecting specific errors. Through the use of pre-defined programming plans or algorithms, systems can hypothesize the plan or algorithm a student may have implemented to complete a specific problem. If the student solution matches the hypothesized programming plan or algorithm, the student has successfully completed the problem. If the student solution does not match the hypothesized programming plan or algorithm, the database of buggy rules is utilized to explain discrepancies between the student solution and the hypothesized programming plan or algorithm.

Instead of using buggy rules and programming plans or algorithms to model a domain, some systems use a semantic table and a set of weighted constraints to develop a domain model. Le and Menzel (2009) developed a weight constraint-based model for an IPT for Prolog called INCOM. The semantic table models alternative solution strategies as well as represents the general components that comprise each solution strategy. “Constraints are used to check the semantic correctness of a student solution with respect to the requirements specified in the semantic table and to examine general well-formedness conditions for a solution” (Le et al., 2013, p. 8). The weights of the constraints are used in the error diagnosis process. INCOM identifies errors in the student solution in two phases on two different levels. At the strategy level, the system attempts to identify student intention by comparing the student solution to the set of strategies listed in the semantic table. Then, at the implementation variant level, the system hypothesizes the student implementation variant by comparing the specific components of the student solution to those of the selected strategy. The various hypotheses are evaluated based on the aggregated weight value of the violated constraints. This allows the system to determine the student intention by selecting the most plausible solution strategy. All of the techniques employed in a feedback-based approach share one keen similarity: they attempt to identify the student intention based on submitted solutions, then identify errors in those solutions. While we have discussed the ways in which systems dynamically produce feedback on student solutions (model-tracing, machine learning, programming plans and buggy rules, etc.), we have not mentioned the content of that adaptive feedback. Beyond yes or no, what type of feedback do IPTs provide to their users? Le (2016) provides a classification of adaptive feedback in educational systems for programming. In the paper, the author identifies five types of feedback: Yes/No feedback, syntax feedback, layout feedback, and quality feedback. Semantic feedback can be divided into two levels: intention-based and code-based feedback.

The last category of AI-supported tutoring approaches from Le et al. (2013) are those that are collaboration-based. As opposed to previous tutoring approaches that focused on individual learning, a collaboration-based tutoring approach aims to support the learning of several students simultaneously. HABIPRO employs a collaboration-based approach to help students develop healthy programming habits (Vizcaino, 2005). “The system provides four types of exercises: 1) finding a mistake in a program, 2) putting a program in the correct order, 3) predicting the result, and 4) completing a program” (Le et al., 2013, p. 9). If the student group submits an incorrect solution, the system either provides hints on how to solve the problem, displays and explains the solution, shows a similar problem with its solution, or simply displays the solution. HABIPRO has a virtual agent that helps students avoid off-topic situations and actively engage the problem. The system guides the group of students using natural language, as well as models the group of students. “During the collaboration, the simulated agent uses the group model to compare the current state of interaction to these patterns, and proposes actions such as withholding solutions until the students have tried to solve the problem” (Le et al., 2013, p. 9). Additionally, HABIPRO utilizes student models and the group model to determine when to intervene and how to provide suggestions.

Crow et al. (2018) published a systematic review of existing IPTs. As opposed to the review on AI-supported tutoring approaches, the focus of this review is primarily on what supplementary features are present within IPTs (e.g., program plans, quiz questions, lesson materials, reference materials, and worked solutions). In their analysis, Crow et al. (2018) focus on 14 existing IPTs. The IPTs were designed to teach a variety of programming languages. Java, the most common language, is taught by 4 of the IPTs. The name, language, and primary adaptive feature for each IPT is presented in the table below.

Table 2: IPT Languages Taught (Crow et al., 2018)



Tool Name	Language	Adaptive Feature
PROUST [9]	Pascal	Feedback
LISP Tutor [2]	Lisp	Feedback
ITEM/IP [4]	Turingal	Feedback, Navigation
C-Tutor [26]	C++	Feedback
ELM-ART [27]	Lisp	Feedback, Navigation
Scope Tutor [12]	Pascal	Feedback
ILMDA [25]	Java	Navigation
AtoL [29]	Java	Navigation
CIMEL ITS [17]	Java	Feedback, Navigation
CPP-Tutor [18]	C++	Feedback, Navigation
J-LATTE [7]	Java	Feedback
ChiQat [1]	Un-specified	Feedback
Ask-Elle [6]	Haskell	Feedback
ITAP [24]	Python	Feedback

Types of feedback include summative feedback on student produced programs, as well as next step hints and debugging help. Further, feedback can relate to syntax, semantics, or style. Adaptive or intelligent feedback – which attempts to replicate the sort of assistance provided by a human tutor – is a common feature of many IPTs. Crow et al. (2018) note that the clearest distinction between IPTs are those that provide step-based hints to the students and those that only provide summative feedback on student code. In order for a feature to classify as adaptive or intelligent, it must provide some form of feedback beyond what basic unit testing would produce. Out of the 14 IPTs reviewed, only two were identified as providing no adaptive feedback.

Crow et al. (2018) identify six types of supplementary features that were present in some IPTs but not in others. These features represent key parts of the tutoring process that have been automated in a given IPT. For reference, the six supplemental features are listed below:

1. QU – Questions students answer within the IPT; for example, multi-choice, true and false quiz or short-answer questions; these are distinct from programming problems which students answer by producing code.
2. CP – Creation of some form of program plan by students within the IPT.
3. PL – Pre-made or computer generated plans or visualizations of programs used as teaching resources.
4. LE – Lesson materials on programming concepts.
5. RE – Reference materials.
6. SO – Worked solutions supplied as an instructional resource.

Five out of the 14 IPTs included in the review used questions that did not directly ask the student to write code such as multi-choice and true and false questions. For example, ITEM/IP asks students to mentally execute the behavior of a program to predict its output. AtoL includes a course management component that allows instructors to develop questions. It allows students to select between answering simple questions related to programming or complete programming exercises that require the production of code. Additionally, AtoL contains another type of question that is used solely to adapt the tutoring tool rather than evaluate the student on conceptual or procedural knowledge. ELM-ART includes both simple questions related to the execution of a piece of a code and programming exercises. The simple questions are one of five types: yes-no test items, forced-choice test items, multiple-choice test items, free-form test items, and gap-filling test items. As previously mentioned, ELM-ART employs adaptive feedback using a library of buggy rules. However, the system also provides adaptive navigation depending on student's performance on questions and programming exercises. Another IPT called CIMEL ITS employs quiz questions and interactive exercises to reinforce concepts and evaluate the students' understanding. Results from these questions are fed into a student model that is a Bayesian network based on the domain model. The given model for a student is then used to determine the likelihood that a student will be able to effectively understand related concepts. All of the IPTs just discussed utilize results from questions to train the student models linked with a more general domain model. AtoL and CIMEL ITS use Bayesian networks while ELM-ART uses case-based student modeling. The probabilities produced by student models are used to dynamically adapt navigation for students based on the likelihood that they had mastered a concept. The structure of reference material questions provided in these systems mirror the domain model, meaning that not only were the questions used to tutor students but also to develop the student model used to provide adaptive navigational support.

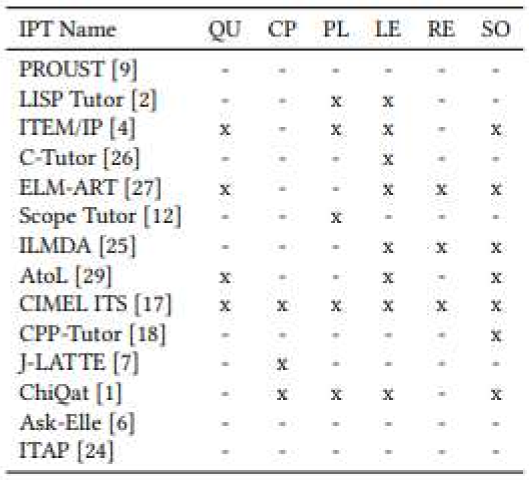
Three out of the 14 IPTs reviewed included exercises in which students had to develop a plan or visualization. In CIMEL ITS, students are required to design a UML diagram representing their solutions before they can implement their code in Java. Because CIMEL ITS was designed to instruct OOP programming, it places a greater emphasis on the use of UML class diagrams. The UML designs produced by the students are fed into the 'Expert Evaluator' component of CIMEL ITS for evaluation. In the ITS for Java J-LATTE, students are also required to design their solutions in a concept mode before they can implement it. Concept abstraction is done at the block or statement level. Students create concepts that can contain a simple statement like an assignment or nested statements like a while loop. This use of abstraction is prevalent in programming languages primarily used for instruction such as Scratch. Such abstraction is particularly helpful for students who struggle more so with syntax rather than semantics. ChiQat also employs user generated designs. This system focuses on both data-structures and the implementation of certain algorithms. Because ChiQat focuses on recursion and linked lists as the primary teaching modules, students are often prompted to generate a visual representation of recursion graphs. This sort of visualization can be more beneficial to students struggling to understand certain data-structures rather than to students struggling with the process of designing traditional sequential programs. The use of plan creation activities by IPTs is rather underdeveloped, as the three existing IPTs that use plan creation focus on a subset of programming topics such as OOP or recursion.

Five of the 14 IPTs in this review use pre-made plans or visualizations in their user interface. LISP Tutor allows students to view a text based plan for a programming exercises if they are struggling. ITEM IP allows students to request a rough plan for a programming exercise as well as hints in the form of a visualization of the program execution. While the plans provided by ITEM IP are text based, the system also provides a basic visualization that allows student to trace code. Scope tutor provides an outline for a programming exercise as well as a visualization of the expected program behavior. CIMEL ITS uses pre-made UML diagrams to scaffold the creation of new diagrams by students before they begin programming an implementation. ChiQat use pre-made visualizations of algorithms and data structures in addition to user-generated visualizations. While pre-made visualizations were not commonly used by the systems included in the review, they can aid students who are struggling to understand specific concepts. ChiQat offers visualizations of how recursion works so that students can generate their own visualizations to implement their own solutions. Scope Tutor provides specific help to students who are struggling to understand how variables change within different scopes.

Some of the IPTs include detail lesson content that provides information about specific programming concepts. Other IPTs only include instructions for tasks and an enhanced programming environment with adaptive feedback. IPTs that lack reference materials "are designed purely to help students with hands-on programming activities when they are assumed to have prior knowledge of programming" (Crow et al., 2018, p. 59). If IPTs are to effectively replace human tutors as a means of one-on-one tutoring, integrating programming exercises that provide either adaptive feedback or navigational support with reference materials would benefit students more than IPTs without integrated reference materials. Simply put, reference materials provide an additional resource for students to look to when they struggle. The organization of reference materials by existing IPTs differs on a case-by-case basis. Some IPTs provide organized reference materials that can be viewed at any time. Other IPTs organize reference materials in a digital textbook with the addition of interactive materials. The summary of which

IPTs provide reference materials as well as which IPTs include the other five supplemental features can be viewed in the table below.

Table 3: IPT Features (Crow et al., 2018)



IPT Name	QU	CP	PL	LE	RE	SO
PROUST [9]	-	-	-	-	-	-
LISP Tutor [2]	-	-	x	x	-	-
ITEM/IP [4]	x	-	x	x	-	x
C-Tutor [26]	-	-	-	x	-	-
ELM-ART [27]	x	-	-	x	x	x
Scope Tutor [12]	-	-	x	-	-	-
ILMDA [25]	-	-	-	x	x	x
AtoL [29]	x	-	-	x	-	x
CIMEL ITS [17]	x	x	x	x	x	x
CPP-Tutor [18]	-	-	-	-	-	x
J-LATTE [7]	-	x	-	-	-	-
ChiQat [1]	-	x	x	x	-	x
Ask-Elle [6]	-	-	-	-	-	-
ITAP [24]	-	-	-	-	-	-

LISP Tutor provides detailed information about the concepts that are being tutored in a given programming exercise. Lessons for specific concepts precede programming exercises that test students' understanding of those concepts. This is in contrast to IPTs that separate lessons and programming exercises, allowing students to select which lesson or exercises they wish to complete. ITEM IP provides an instruction module linked with a domain model in addition to the laboratory in which students complete programming exercises. Domain model concepts and associated instruction are linked with programming exercises in the laboratory section but students can navigate between instruction mode and programming exercises at their own discretion. Reference materials can be introduced at the beginning of a new concept or requested by students at any point. ITEM IP is particularly unique in that it adapts the depth of explanations based on the student level put forth by the student model. C-Tutor does not include reference materials independent of programming exercises but does provide lesson content to students that proceed programming tasks. ELM-ART includes an interactive textbook as well as programming exercises and questions. All lessons, reference materials, questions, and programming exercises are linked by the system. As mentioned in the discussion of AI-supported tutoring approaches, ELM-ART includes a domain model and student model with the latter being updated as students respond to questions and complete programming exercises. The content of the reference materials forms the basis for the domain model in ELM-ART. The system uses the current state of the student model to provide adaptive navigational support within the interactive digital textbook. CIMEL ITS includes comprehensive reference materials. The IPT, linked with the CIMEL interactive courseware, runs within the Eclipse IDE. Like ELM-ART, the content of the reference materials forms the basis for domain modeling and therefore, student modeling. Based on student responses to questions and programming exercises, the student model is updated and adaptive navigational support of the reference material is provided to the student. In CIMEL ITS, the reference material plays a key role in linking the student model to the domain model as well as the intelligent component of the system (adaptive navigational support). The tutoring system ILMDA provides reference materials to students based on their usage history of the system as well as outside factors such as GPA, majors, and courses. The system was custom-built to deliver CS1 course content to students at the University of Nebraska. While ILMDA is the only IPT in the review to not provide an integrated programming environment, the system does provide adaptive navigational support of reference material, lesson content, and programming exercises to students. AtoL and ChiQat include lesson content to teach a variety of programming concepts, but do not include reference material that align with specific tasks. Additionally, these systems to do not use reference material as the basis for an organized domain model and student model. A majority of the IPTs in the review included some form of lesson content, with lessons typically being presented before programming tasks. Additionally, in general lesson content aligned with specific questions and programming exercises.

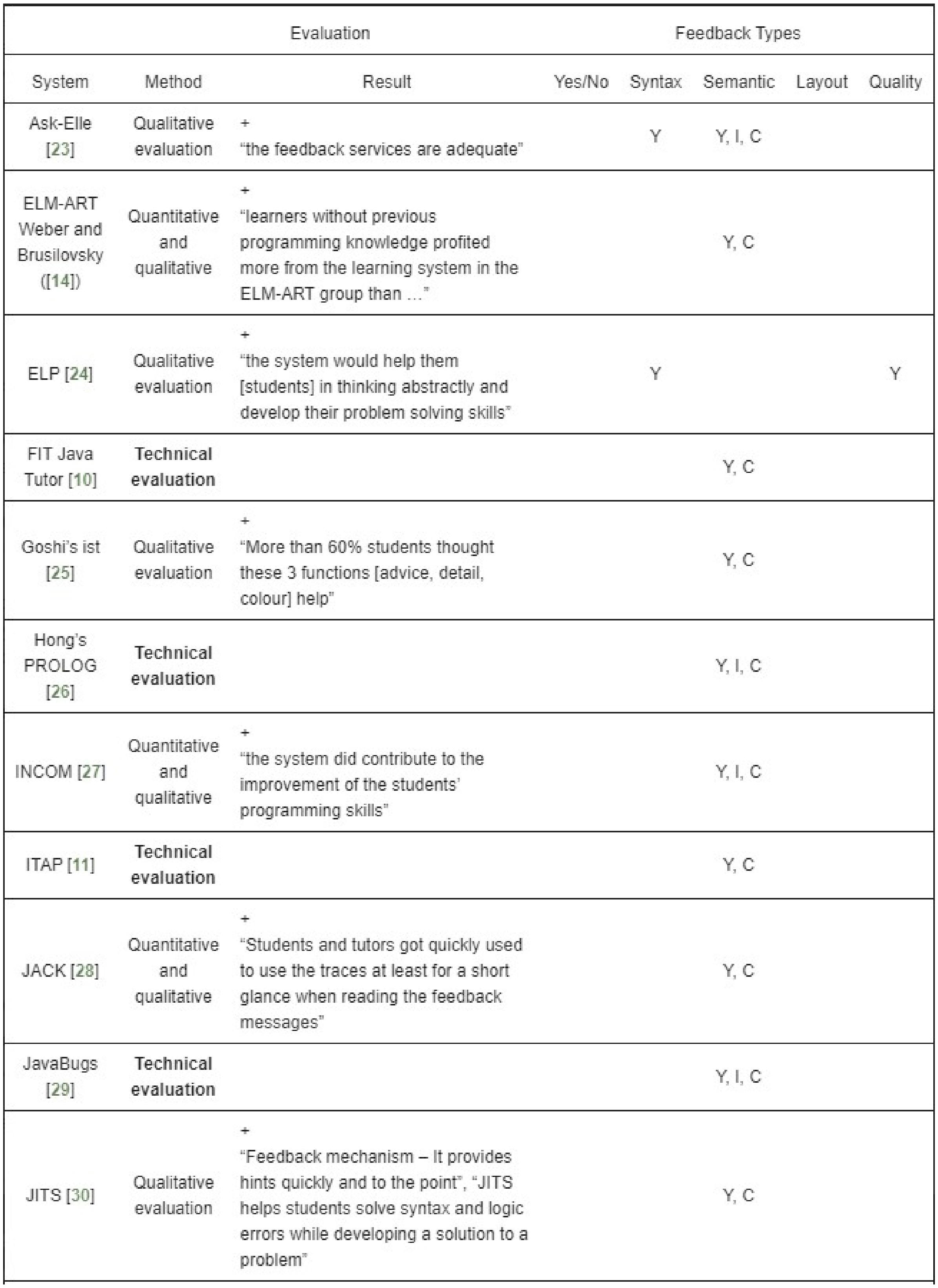
Seven of the 14 IPTs in the review included worked solutions and completed example programs. These are all instances of example-based tutoring as previously discussed. Some IPTs used example programs to explain specific concepts, typically incorporating these examples into lesson content. ELM-ART and CIMEL ITS integrate worked examples into the reference material that form the basis for the domain models in those system. Example solutions are a common feature of both traditional computer science education and IPTs.

At this point, we have discussed the types of AI-supported tutoring approaches employed by existing IPTs, as well as supplemental features that may make a given system unique. As Crow et al. (2018) note, the greatest distinction between IPTs are those that provide adaptive navigational support and those that provide adaptive feedback. Adaptive navigational support is common in IPTs that integrate lesson content and reference material with programming exercises and questions. Based off a student's performance on a specific programming exercise or question, a system will suggest the student either to continue attempting problems associated with same concept or to attempt problems associated with the next concept. On the other hand, adaptive feedback provides different information for different responses in addition to verifying the correctness of a student response to a question or programming exercise. As previously mentioned, Le (2016) presented a classification of adaptive feedback. In addition to providing the classifications for adaptive feedback (Yes/No feedback, syntax feedback, layout feedback, and quality feedback), the author of this paper investigates the impact of adaptive feedback in existing systems on students' development of programming skills. "In general, two classes of evaluation objectives for those systems are distinguished: (1) pedagogical evaluation studies that investigated the impact of systems on learning on cognitive, meta-cognitive and motivational dimensions; and (2) technical evaluation studies that investigated technical properties of a system … and its usability" (Le, 2016, p. 12). Table 4 on the following pages provides the results of the evaluations for the various systems used in the study as well as the types of adaptive feedback included by the system. "Y" indicates that the type of feedback is present in the system. "I" and "C" stand for intention-based feedback and code-based feedback, respectively.

Three systems (QuizJet, QuizPack, and M-PLAT) included Yes/No feedback. QuizJet and QuizPack present an example program to the student in Java and C, respectively, and ask the student for the expected result of a statement or variable. Le (2016) included these two systems because they generate parameterized questions. M-PLAT requires students to submit a whole program in response to a prompt, then the system verifies the correctness of the program. Based on their submission, students will receive one of three responses: (1) the program is correct, (2) the program contains compilation errors, and (3) the program compiles but is incorrect. The intelligent component of M-PLAT adapts the requirements for a given question to the level of the student.

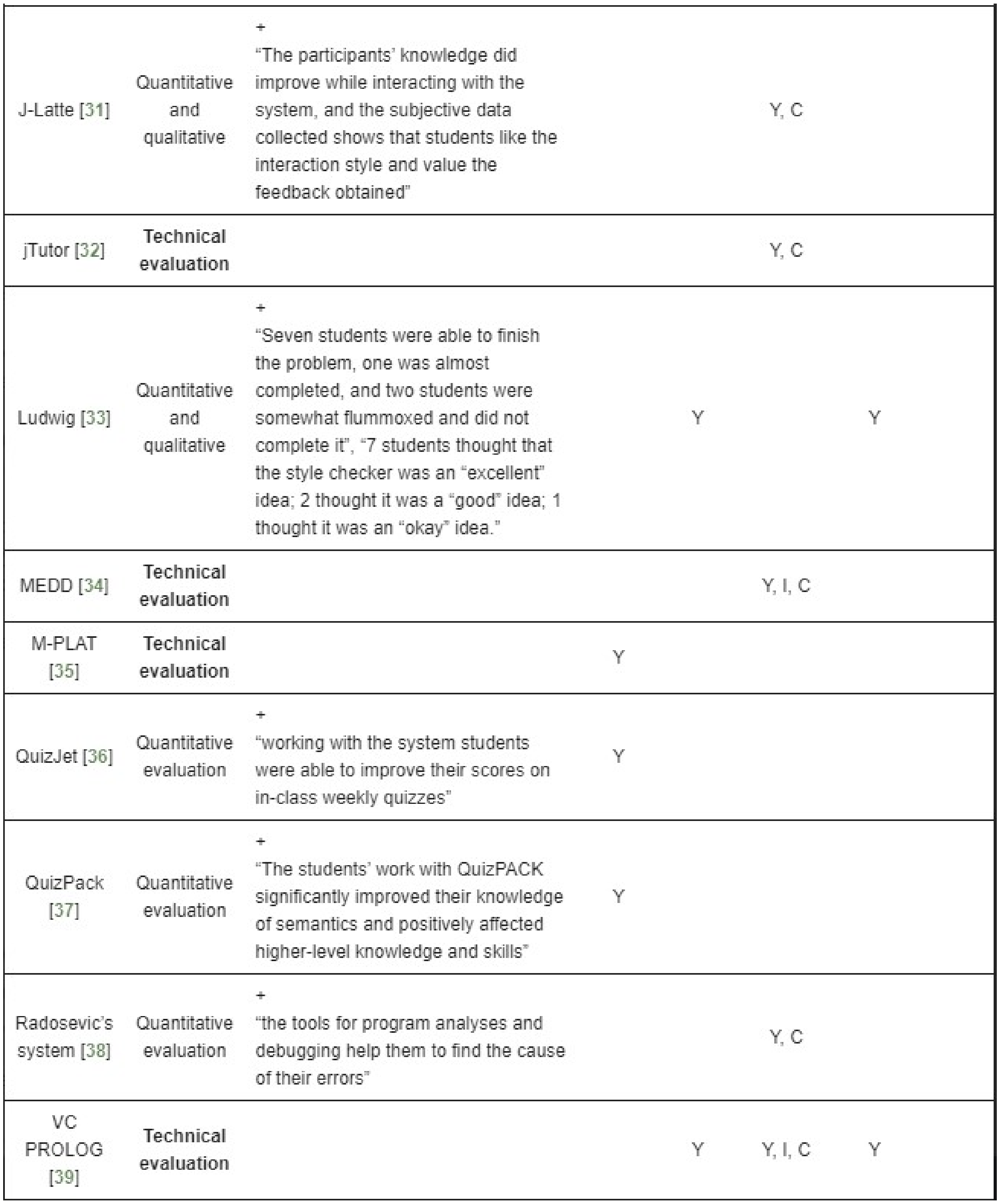
Most systems return the compiler's output as a form of syntax feedback. One system, VC PROLOG, provides syntax feedback for an error by explaining the associated concept as in the defined ontology of Prolog. As previously mentioned however, the student can completely understand the semantics of a given programming exercise without having the knowledge to implement a solution or decipher messages from the compiler. In response to this dilemma, Gross et al. (2013) have proposed a dialogue-based approach to tutoring syntax errors.

Table 4. Impact of the educational systems on students' programming. (Le, 2016).



	Evaluation		Feedback Types				
System	Method	Result	Yes/No	Syntax	Semantic	Layout	Quality
Ask-Elle [23]	Qualitative evaluation	+ "the feedback services are adequate"		Y	Y, I, C		
ELM-ART Weber and Brusilovsky ([14])	Quantitative and qualitative	+ "learners without previous programming knowledge profited more from the learning system in the ELM-ART group than …"			Y, C		
ELP [24]	Qualitative evaluation	+ "the system would help them [students] in thinking abstractly and develop their problem solving skills"		Y			Y
FIT Java Tutor [10]	**Technical evaluation**				Y, C		
Goshi's ist [25]	Qualitative evaluation	+ "More than 60% students thought these 3 functions [advice, detail, colour] help"			Y, C		
Hong's PROLOG [26]	**Technical evaluation**				Y, I, C		
INCOM [27]	Quantitative and qualitative	+ "the system did contribute to the improvement of the students' programming skills"			Y, I, C		
ITAP [11]	**Technical evaluation**				Y, C		
JACK [28]	Quantitative and qualitative	+ "Students and tutors got quickly used to use the traces at least for a short glance when reading the feedback messages"			Y, C		
JavaBugs [29]	**Technical evaluation**				Y, I, C		
JITS [30]	Qualitative evaluation	+ "Feedback mechanism – It provides hints quickly and to the point", "JITS helps students solve syntax and logic errors while developing a solution to a problem"			Y, C		

Table 4. *Cont.* (Le, 2016).



J-Latte [31]	Quantitative and qualitative	+ "The participants' knowledge did improve while interacting with the system, and the subjective data collected shows that students like the interaction style and value the feedback obtained"			Y, C	
jTutor [32]	**Technical evaluation**				Y, C	
Ludwig [33]	Quantitative and qualitative	+ "Seven students were able to finish the problem, one was almost completed, and two students were somewhat flummoxed and did not complete it", "7 students thought that the style checker was an "excellent" idea; 2 thought it was a "good" idea; 1 thought it was an "okay" idea."		Y		Y
MEDD [34]	**Technical evaluation**				Y, I, C	
M-PLAT [35]	**Technical evaluation**		Y			
QuizJet [36]	Quantitative evaluation	+ "working with the system students were able to improve their scores on in-class weekly quizzes"	Y			
QuizPack [37]	Quantitative evaluation	+ "The students' work with QuizPACK significantly improved their knowledge of semantics and positively affected higher-level knowledge and skills"	Y			
Radosevic's system [38]	Quantitative evaluation	+ "the tools for program analyses and debugging help them to find the cause of their errors"			Y, C	
VC PROLOG [39]	**Technical evaluation**			Y	Y, I, C	Y

The third classification of feedback, semantic feedback, relates to errors with respect to completing the requirements of a programming exercise. Feedback at the semantic level can be further divided into two levels: intention analysis and code analysis. Intention analysis aims to determine the intention of the student program by compare it to a set of reference programs. As previously mentioned, techniques such as model-tracing, machine learning, programming plans and buggy rules, etc. allow the system to identify the student's intention. At the code level, semantic feedback can be provided without identifying the student's intention. Using the same techniques, a system employing this type of feedback can recognize differences between the student solution and a reference program to make suggestions about specific lines of code.

Layout feedback aims to improve understandability of students' code according to a specific coding convention. Layout feedback was only present among Ludwig and VC PROLOG. Ludwig performs stylistic analysis on a student's program based on proper indentation, global variables, etc. VC PROLOG compares the structure of a student program with a reference program to provide layout feedback. Layout feedback is not a common feature in IPTs most likely because researchers view issues of semantics and syntax more important than those related to style and layout.

The last classification of feedback is quality feedback. Quality feedback analyzes the runtime complexity and space complexity of a student solution. ELP performs quality analysis in two phases: structural similarity analysis and software engineering analysis. The former identifies pieces of code that are complex, lengthy, or follow poor practices by comparing the student solution to a set of reference programs. The latter similarly identifies piece of complex or poorly written code. JACK provides quality feedback based on a generated trace of the student's program. If the trace of the student's program is longer than that of a reference program, the system will indicate to the student that their solution may be unnecessarily complicated.

At this point, we have discussed the body of literature concerning student misconceptions of programming concepts, as well as literature relating to intelligent tutoring systems for programming education. In the following chapter of the thesis, we will select a set of important and challenging concepts (or misconceptions) from our review that novice programmers should master before learning new concepts. For the purposes of this thesis, we are concerned with challenging concepts that students are expected to learn in any introductory programming regardless of the range of concepts or languages used (i.e., the fundamentals of programming that are challenging to students). Once the set of concepts have been defined, we will demonstrate how features of existing IPTs are particularly useful in tutoring misconceptions related to each of these concepts.

Chapter 4: Handling Misconceptions with Existing Intelligent Programming Tutors

4.1 Selecting the Misconceptions

In the first section of the literature review, we identified the purpose of a concept inventory, as it is used to evaluate pedagogical approaches and identify student misconceptions of programming concepts that have been deemed important and difficult by experts in the field. This idea of a concept inventory for programming education proves to be very important when discussing the effectiveness of various pedagogical approaches or tools. However, there is inherent difficulty in developing a concept inventory for introductory programming due to the variety of languages and concepts that can be taught in the average CS1 course. As Goldman et al. (2008) note in their seminal paper:

> To build learning assessment tools that are sufficiently general to apply to the broad range of curricula and institutions in which computing is taught, it is necessary to identify a representative set of topics for each course that are both undeniably important and sufficiently difficult that the impact of pedagogical improvement can be measured. (p. 259).

For our purposes, we are concerned with the set of topics for introductory programing independent of programming languages and pedagogical approaches. This means that topics such as Object-Oriented Programming (OOP) will not be included in our set despite the fact that they may be both difficult and common in introductory programming classes. By identifying misconceptions that are common throughout introductory programming, we will be able to demonstrate the general effectiveness of IPTs in teaching the fundamentals of programming despite the fact that most IPTs are custom-built for single implementations and are focused on a narrow subset of programming concepts.

Previously, we discussed a number of papers that focus on concept inventories and student misconceptions in introductory programming (Caceffo et al., 2016; Goldman et al., 2008; Kaczmarczyk et al., 2010). While introductory programming concept inventories are developed with the intention of being language-independent, some misconceptions are inextricable from the programming environment in which they manifest. This means that some of the concepts and misconceptions identified by these researchers are not applicable to our circumstances, such as OOP. Additionally, some of the concepts and misconceptions identified are either too narrow or high-level for our purposes. To select the set of introductory programming concepts for our inquiry, we will include concepts that each of the aforementioned agree are important and challenging. Some categories have been deemed to be too narrow, and have instead been combined into another category. The full set of concepts are listed below:

1. Function Parameters and Variables, Use and Scope
2. Memory Model, References, and Pointers
3. Control Flow, Iteration, and Loops
4. Recursion
5. Syntax vs. Conceptual Understanding

Regardless of environment or language, functions and variables are undoubtedly fundamental concepts of introductory programming that every novice is expected to learn. Similarly, the relationship between variables and the underlying memory that represent them is a core concept of computing in general. Control flow, or the order in which individual statements, instructions, or function calls are executed, is a key feature of imperative programming. It is not enough to know the end result of a program; competent programmers can trace the execution of a program from start to finish. Much of programming deals with scanning over a set of inputs to produce some sort of output. There are two primary ways in which this process is represented in programming: iteration and recursion. Iteration refers to repeating of a process while recursion occurs when a thing is defined in terms of itself. Despite typically aiming to accomplish the same goals, recursion and iteration each have different strengths and weaknesses. Our last concept is rather untraditional in that it is unlikely that the average introductory programming course would identify it as a specific concept. This is not due to a lack of importance but rather due to the fact the counterbalance between syntactic understanding and conceptual understanding is present throughout programming at all levels. Our inclusion of this concept serves to highlight the gap that can exist between novice programmers' conceptual understanding and syntactic understanding. It is not uncommon for novice programmers to fully understand introductory programming concepts but lack the syntactic knowledge to implement solutions in programming exercises or answer questions about programming language specifics. However, those who wish to develop their programming skills must actually write code. Bridging the gap between conceptual understanding and designing and implementing code is a key concept in introductory programming.

In the remaining sections of this chapter, we will review the student misconceptions as they relate to each of the six previously identified concepts. For the misconceptions associated with a concept, we will discuss how features of existing IPTs handle those misconceptions including any limitations that may exist.

4.2 Function Parameters and Variables, Use and Scope

In this section, we will discuss how features of existing IPTs can help handle student misconceptions related to understanding the use and scope of function parameters and variables.

In their review of student misconceptions, Caceffo et al. (2016) identified function parameter use and scope as the largest source of misconceptions. More specifically, three distinct misconceptions were identified. The first and most common misconception was a lack of understanding of the difference between call by reference and call by value. In C programming, parameters are call-by-value, meaning that the value of the parameter and not the reference is passed into the function scope. This led to students thinking they could change the content of a formal parameter inside a function, believing this would also change the actual parameter variable outside the function. The second and third misconceptions relate to students failing to understand the initialization of formal parameters and that parameters are local variables. Further, Caceffo et al. (2016) identified similar misconceptions related to the scope of variables. Students do not understand the difference between local and global scope. Variables that are declared within a function are limited to the scope of that function.

Misconceptions related to this programming concept can manifest in a variety of ways. Conversely, a number of different misconceptions related to other concepts can manifest in the same ways as those in this category. Put more simply, it can be difficult to identify misconceptions related to the use and scope of function parameters and variables. However, there are features of existing IPTs that can help students overcome these misconceptions.

As we have mentioned previously, IPTs primarily assess students' understanding of concepts by prompting students to answer questions related to the output of a program or the semantics of a concept, prompting students to produce code to complete a programming exercise, or prompting students to create some form of program plan.

A number of existing IPTs have been designed to perform solution analysis and provide adaptive semantic feedback on student responses to programming exercises such as PROUST.

These systems rely on a library of bugs that comprise the domain model to identify errors in student solutions. To identify the correctness of a solution, both systems first identify the intention of the student solution by comparing it to a library of reference programs and selecting the most similar program. If the selected reference program is incorrect, the system uses the misconceptions associated with that reference program to hypothesize misconceptions associated with the student solution and provide feedback. The key aspect in determining whether or not solution analysis IPTs are effective in handling misconceptions related to the use and scope of variables and function parameters is the content of that adaptive semantic feedback.

One system in particular, JACK, provides feedback in a way that is well-suited to combat misconceptions in this category. JACK uses an automated trace generation to provide feedback. This returns a list of single execution steps in terms of variable values as indicated by the system states. “The authors claimed that reading the trace of a program execution, a student could understand how a program works (i.e., the system state evolves from the given input to the final state). However, students do not need to read a trace from the beginning to the end.” (Le, 2016, p. 10). While the programming exercise a student is completing in JACK may be associated with any number of learning concepts, this trace feedback allows students to closely monitor variable and parameter values as the program enters and exits various subroutines. Instead of simply pointing out incorrect code, JACK encourages students to identify reasons for an erroneous output so that the related program statements can be corrected. While this type of trace can be reproduced by debugging within and IDE, the fact that many novice programmers lack the hands-on experience necessary to effectively debug makes this feedback particularly useful.

Systems such as QuizJet, QuizPack, and jTutor primarily support student learning through quiz questions related to specific concepts. jTutor allows instructor to create quizzes by specifying correct values and hints. The system tests student knowledge by prompting him or her to fill in the correct values, such as the expected output of a program or value of a variable. If the incorrect value is entered, a new hint is provided to the student. And if the student continues to struggle, jTutors will provide a similar quiz or question at a lower level. There is significant opportunity for an intelligent quiz feature in all mediums of computer science education, as seen by its inclusion in many only interactive textbooks such as ZyBooks. Further, quiz questions that are directly designed to test understanding of the scope and use of variables and function parameters can be particularly useful in addressing misconceptions related to that concept. Moreover, the widespread nature of this concept in programming makes the necessity for quiz questions directly addressing parameters and variables more obvious.

4.3 Memory Model, References, and Pointers

Much of the research into student misconceptions points to a significant gap in novice programmers' understanding of memory models, references, and pointers (Caceffo et al., 2016; Goldman et al., 2008; Kaczmarczyk et al., 2010; Miller & Settle, 2019). Regardless of what type of programming one chooses as a focus, the underlying relationship between memory and program statements is foundational to computing. Many novice programmers lack a proper understanding of the difference between references and values and when each should be used. C programming students struggle to return values through pointers and assign values to pointers. Java programming students incorrectly assume that primitive variables without assigned values have no value and that memory is allocated for Objects which have been declared, but not instantiated.

Similar to the last category, quiz questions that are directly designed to test understanding of the underlying relationship between variables and memory as well as the more C programming specific concept of pointers can be particularly useful. QuizPack, which was designed to test student understanding of C programming concepts, is especially useful in testing student's understanding of pointers. By providing quiz questions that ask for expected behavior of a program involving pointers, specific misconceptions can be identified and used to provide hints to the student. Moreover, given the complex nature of pointers, students can begin by proving their understanding of the basics of pointers before exploring more complex situations such as double pointers and 2-D arrays. Further, quiz questions that directly test student understanding of the relationship between variables and memory in Java would promote student understanding of that concept.

We should also note IPTs such as C-Tutor that provide adaptive semantic feedback based on the solution analysis of programming tasks can help address misconceptions in this category. Using a library of reference programs, the system can hypothesize student misconceptions by first identifying the intent of the solution. This intention-based level analysis can reveal a student's incorrect use of pointers in comparison with the reference program. Further, code-level analysis can highlight a student's incorrect use of pointers by identifying syntactically incorrect program statements.

Another important aspect of memory in programming that can be overlooked at the introductory level is understanding how to efficiently use memory. Quality feedback such as that provided by JACK can reveal whether or not a student's solution to a programming exercise is efficiently using memory in comparison to the ideal solution for the exercise. While optimizing memory use may be seen as a higher-level concept, such quality feedback can support student understanding of the relationship between programming and the underlying memory which represents it.

4.4 Control Flow, Iteration, and Loops

Control flow refers to the order in which individual statements, instructions or function calls are executed. Student misconceptions research has revealed that novice programmers cannot trace code linearly. And if novice programmers are unable to trace code linearly, it is very likely their solutions to programming exercises contain errors due to misconceptions about the control flow of their program. Alongside a misconception of control flow comes a misconception of loops and iteration. Many students misunderstand the process of while loops and for loops. Moreover, students struggle to trace code as a program executes a loop, incorrectly assuming the number of times a loop will execute.

A number of systems are well suited to handle misconceptions in this category. As previously mentioned, the automated code trace generated by JACK provides the trace of the student solution to a programming task. This sort of feedback is particularly helpful in handling misconceptions related to control flow, as students are able to verify whether or not the program statements of their solution execute in the order in which they expected.

Some IPTs provide pre-made plans or visualizations to students as they complete programming exercises, which can be particularly helpful to students who are struggling with the control flow of their program. With the system ITEM IP, the student can request to view a rough plan of the program as well as hints in the form of visualization of program actions. Such plans and visualizations can help students understand when certain functions and program statements are supposed to execute. Additionally, it allows the student to understand how the program should behave before actually writing an implementation.

IPTs that employ a simulation-based approach such as PESEN, SICAS, and OOP-AMIN can be particularly useful to students who struggle with control flow. Simulation-based approaches allow students to visualize how a program executes. PESEN was designed to support weak students in learning basic concepts of sequential, conditional and repetition execution. PESEN prompts students to design and implement an algorithm to satisfy the programming exercise. Students build a flowchart that represents a solution, then the solution is simulated, allowing the student verify that their solution works as intended.

Semantic feedback at the code-based level such as used by J-Latte can also benefit students struggling with control flow. Using a pre-specified set of constraints that represent the principles of the Java domain, J-Latte provides feedback on the code-based level. The system can identify errors related to control flow such as students placing a semi-colon at the end of for-loop or while-loop block.

4.5 Recursion

Recursion is widely viewed as one of the more difficult concepts in programming, because it can be challenging to trace the execution of a recursive program mentally. Additionally, it can be difficult to understand the recursive relationship and base case by which something is recursively defined. Regardless, recursion is inherently useful in programming because it can solve big problems by breaking them down into smaller, repetitive problems. Further, recursion allows programmers to define complex problems in simpler terms compared to when using simple iteration. A number of existing IPTs can help handle student misconceptions related to recursion.

As mentioned, it can be difficult to trace the execution of a recursive program. The automated trace generation feedback of JACK can help student's trace the execution of a recursive program. However, traces for larger recursive programs can become burdensome to analyze, which is why understanding a recursive program in terms of the recursive graph that defines it is extremely important. The use of user generated diagrams in ChiQat can be particularly helpful in tutoring students on recursion. ChiQat was designed to focus on algorithms and data structures, namely linked lists and recursion. In modules focused on recursion, Chi-Qat prompts students to complete programming exercises by first generating a recursive graph that represents their solution. By comparing this graph to the ideal algorithm for the programming exercise, the system provides semantic feedback on the student's solution. A feature that is missing amongst the IPTs reviewed is an automatic recursive graph generation for student solutions. By producing the recursive graph for a student's solution, he or she would be able to verify that the recursive relationship and base case defined by their solution is in fact correct.

4.6 Syntax vs. Conceptual Understanding

In their efforts to develop a concept inventory for introductory programming, Caceffo et al. (2016) highlight the difficulty in distinguishing between conceptual misconceptions and highly language-specific syntactic misconceptions. The instructors interviewed in this paper reached a consensus that the nature of programming makes it difficult to elaborate conceptual questions entirely independent of the programming language being taught. In the case where a novice programmer is tasked with producing code, if he or she lacks an understanding of the syntax of the programming language in use, an incorrect solution becomes a foregone conclusion regardless of his or her level of conceptual knowledge. However, there are features of existing IPTs that can help novice programmers bridge the gap between conceptual understanding and syntactic understanding.

The system J-LATTE is particularly well-suited to handle student misconceptions related to the syntax of the Java programming language. In J-LATTE, students first design their Java programs in a concept mode. In this concept mode, abstraction is done at the statement and block level where single Java statements like an assignment or a block containing nested statements like a loop are represented by concept. These blocks are similar to programming languages like Scratch that allow students to focus on understanding concepts without worrying about correct syntax.

JITS, a system that performs solution analysis on programming exercises and provide code-based feedback, is also well-suited to handle errors that occur due to a lack of syntactic understanding. JITS displays incorrect code in specific Java programming exercise. JITS relies on a domain model and student cognitive model to perform both intention-level analysis and code-level analysis. However, the system only provides code-based feedback such as “I think you meant the keyword ‘for’. Is this correct?” After identifying the most similar reference program, JITS uses discrepancies between the reference program and the student solution alongside the domain and student cognitive models to provide adaptive code-based feedback that allows the students to correct syntactic mistakes.

Example-based tutoring approaches can help novice programmers gain a grasp on the syntax of the programming language in use. In addition to providing semantic feedback on programming exercises, the system ADAPT provides several solution examples alongside programming tasks. After introducing the examples, ADAPT prompts the student to solve a similar problem given a solution template with blanks to be filled. Providing examples before asking the student to solve a problem allows the student to verify that they understand the syntax in use before attempting to solve the problem on their own. This can be particularly useful when the example in question includes syntax the student has not seen before or does not understand. NavEx also supports students through an example-based approach and provides even more substantial support to students struggling with syntax. With NavEx, students can learn in an exploratory manner. Students can view the entirety of the example without comments and attempt to understand how it functions. If a student is struggling with specific lines or blocks of code, a textual explanation of the code in question can be accessed.

An obvious but often overlooked feature of effective IPTs is the integration of reference material into programming exercises and questions. Only 3 of the 14 IPTs reviewed by Crow et al. (2018) included in-depth reference material. In addition to typically providing the basis for domain and student models, reference material can be accessed by students in certain systems when adaptive feedback is not enough to address syntactic misconceptions. ELM-ART includes programming exercises and questions alongside an interactive virtual textbook that can be accessed at any time. This inclusion of reference material allows students with little experience in a specific programming concept to read about that concept and try to form their own understanding either before or after attempting programming exercises and questions related to that concept.

4.7 Effective IPT Features

In concluding our chapter on the ability of IPTs to address novice programmers' misconceptions, one theme becomes quite clear: Both the static and intelligent components of IPTs can promote student learning of programming concepts. Apart from the adaptive feedback and adaptive navigational support provided by IPTs, static features present in the traditional classroom setting are still effective in handling some misconceptions. The inclusion of simple true/false questions, example solutions, program planning, pre-made or computer generate plans or visualizations, lesson materials, or reference materials in addition to adaptive feedback and adaptive navigational support can improve the effectiveness of IPTs.

Still, this chapter highlights the positive impact of adaptive feedback and adaptive navigational support on promoting student learning of introductory programming. Without a human instructor or tutor to address the unique needs of his or her pupil(s), intelligent tutoring systems must rely on domain and student cognitive modeling to service the unique needs of its users. Such modeling allows IPTs to track student understanding of learning concepts and identify and address specific misconceptions or add new misconceptions to the domain model as they arise. Moreover, student cognitive modeling allows IPTs to provide adaptive navigational support that static systems lack. This navigational support is extremely important as students often do not know the natural ordering of introductory programming concepts and the relationships between them. We should note that there are inherent limitations in the accuracy of domain and student modeling. However, as more students use IPTs that rely on such modeling, the modeling of those systems becomes more accurate as new misconceptions are revealed and integrated into the domain model.

We have discussed how a number of features of IPTs effectively support introductory programming concepts. In the following chapter, we will discuss what the ideal IPT might look like in terms of structure and features. Additionally, we will highlight the inherent difficulty in developing an IPT for introductory programming independent of language. Regardless, it is our hope that the following chapter provides the basis for the development of language-specific, comprehensive IPTs.

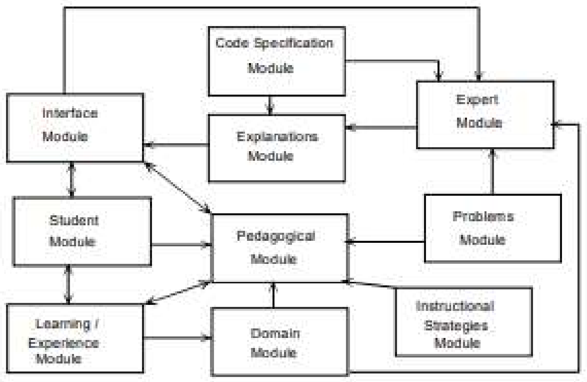
Chapter 5: The Ideal Intelligent Programming Tutor

5.1 Toward a Generic Architecture for IPTs

The difficulty of creating an ideal IPT for novice programmers stems from the fact that introductory programming can be taught in a number of different programming environments, in which mechanics of the programming language can be inextricable from the concepts being taught. Throughout this thesis, we have primarily focused on the features of IPTs rather than the architectures. This is primarily due to the fact the most IPTs are custom-built for single implementations, leading to differences in the architectures of those systems. As Crow et al. (2018) note, there currently exists a gap in the field of generic architectures for IPTs that can be used in programming contexts. The only work we can look to in terms of a generic architecture comes from Pillay (2003). Based on an analysis of the IPTs developed at that point and studies conducted to identify student misconceptions, the essential functions of an IPT are identified and a generic architecture is proposed. It is difficult to determine the impact of the generic architecture proposed in this paper for a number of reasons: (1) many IPTs do not report the architecture implemented in their system, (2) the intelligent component of IPTs differs greatly from system to system, and (3) most IPTs are both language and platform dependent. However, considering that this generic architecture was developed by analyzing existing IPTs, it is possible that IPTs developed after the release of this paper were influenced by the generic architecture described therein.

The essential functions of an IPT according to Pillay (2003) closely align with the features discussed in previous chapters. Adaptive semantic feedback, adaptive navigational support, and supplementary features of IPTs alongside an analysis of existing systems were used to develop a generic architecture for IPTs. The figure below depicts this generic architecture.

Figure 1: Proposed Generic Architecture (Pillay 2003).



Based on this generic architecture and our review of student misconceptions and existing IPTs, we have identified four necessary modules that the ideal IPT for introductory programming must implement: student module, domain module, pedagogical module, and interface module. Put simply, the student module is responsible for tracking the level of student knowledge and student progress, the domain module is responsible for all knowledge related to introductory programming concepts, the pedagogical module is responsible for pedagogical preferences and learning strategies, and the interface module is responsible for providing and adapting the user interface in which students learn. In the following sections, we will discuss in greater detail the functions for which each module is responsible, identifying how various features of IPTs fit into these modules and how modules should communicate with each other.

5.2 Student Module

As mentioned, an extremely important aspect of an effective IPT is the ability to provide individualized instruction. In order to accomplish this, systems must be able to track student progress and the level of student knowledge. In less intelligent systems, this process is done by simply tracking student's completion of tasks related to various programming concepts.

However, such student modeling fails to address the unique needs of novice programmers. The student module proposed in the generic architecture keeps track "of how well a student is performing on material being taught, the misconceptions made by students, the learner's understanding of concepts and the student's acquisition and retention rate" (Pillay, 2003, p. 79). Additionally, Pillay (2003) note that a student's ability to program depends on the student's learning style, meaning the module should also track pedagogical preferences and learning styles.

Regardless of the paradigm in which introductory programming is taught, it is important that effective IPTs maintain a comprehensive student model through which instruction can be individualized. However, beyond tracking student's completion of tasks, to accomplish comprehensive student modeling, this module must be able to monitor student behavior and thought process. As Finlay and Beale note, the traditional approach to knowledge base techniques are inadequate for student modeling (1993). We propose the use of neural networks to expand the level of student modeling beyond data on student progress and level of knowledge. The neural networks receive data regarding student's performance and create a model of the student's knowledge using an inductive algorithm. It's important that the data on student performance is comprehensive (i.e., includes metrics such as time elapsed, use of hints, identified misconceptions, as well as student background, and student goals, etc.), so that the student modeling itself is comprehensive. This allows the student module to apply data on student performance in support of identifying a high level pattern for a student's learning process. This high level modeling allows the student module to communicate to the pedagogical and interface modules recommendations regarding the knowledge of specific topics and effective pedagogical strategies.

It should be noted that there are inherent limitations in the use of neural networks to model student knowledge. As neural networks rely on inductive algorithms to produce models, these algorithms are initially only as good as the data passed into them. It may be difficult to identify high level learning patterns in students when they initially begin using the IPT, but as students complete more programming exercises and questions, the inductive algorithms used to model student knowledge become more accurate. Further, while a student may demonstrate a proclivity for a specific learning style at a high level (e.g., learning by example), he or she may find different pedagogical approaches more effective for different concepts. For that reason, it will be important to track how or if one's learning preferences change when learning different concepts. One way to address these limitations is by incorporating data on student performance and the associated level of student knowledge and learning styles into probabilistic Bayesian networks. Bayesian networks represent a set of random variables and their conditional dependencies in the form of an acyclic directed graph. Maintaining Bayesian networks that model the probabilistic relationship between data on student performance and associated level of knowledge and learning styles can allow the system to provide individualized instruction when there is very little input for the user-specific neural networks. The system must be able to identify when the user-specific neural networks are well developed enough for student modeling such that the student module can provide recommendations to other modules. Additionally, the system may incorporate both throughout a student's learning to provide specific learning recommendations. For example, when a student begins learning recursion, the system might identify that the student prefers to learn by example from his or her specific neural networks, and that most students who have used the system have found the visualization of recursive graphs particularly helpfully, using both preferences to provide recommendations to the pedagogical and interface modules. We should note that these Bayesian networks model general student knowledge, and could be incorporated into the more general domain modeling.

5.3 Interface Module

An essential feature of IPTs and any educational system for that matter is the interface in which students learn. The interface module of an IPT should present the student with the appropriate screen layout for the task at hand and facilitate communication between the student and other components of the IPT (Pillay, 2003). Additionally, the interface of an IPT must clearly indicate where a student is in programming exercises and questions and provide clear feedback on the correctness of an answer, including any misconceptions made by the student. Apart from adaptive feedback, the ideal IPT should also provide adaptive navigational support, providing easier versions of the same problem when students struggle and pointing students to the next concept to be learned.

The interface module of an IPT would also implement a number of supplementary features described in previous chapters. Automated trace generation such as used by system JACK can support novice programmers in debugging their solutions. Apart from feedback, the use of user generated or pre-made plans and visualizations would fall into the interface module. A concept mode such as used by J-LATTE would be an extremely helpful interface instance for students who struggle with the syntax of the programming language being used. Further, the simulation and visualization of programs such as used by NavEx can help students understand and trace the execution of programs.

There are key connections between the interface module and the student module and pedagogical module. As instruction should be individualized, it is important to maintain the student's history and learning style or pedagogical preferences as the system determines what interface should be provided to students. In the ideal IPT, a student who has not demonstrated the ability to produce code would complete programming exercises in a concept mode, whereas a more advanced student would complete programming exercises by some combination of planning and producing code. Additionally, relying on more topic-specific pedagogical recommendations, the interface module may provide different interfaces for different topics.

5.4 Pedagogical Module

Pillay et al. (2003) identify a "pedagogical and instructional strategies modules" (p. 80) as essential to providing individualized instruction to novice programmers. Our definition of this module mirrors that of the generic architecture. As students have a variety of learning styles and pedagogical preferences, it is key for the IPT to be able to identify what approach is best suited for the student and task at hand. Further, there is a great deal of mapping that should be done between the pedagogical module and the domain module. Specifically, there is a need to map the effectiveness of specific pedagogical strategies in tutoring specific concepts to those concepts, as well as mapping specific pedagogical strategies to specific problem types.

The ideal IPT should use input from the student module and the mapping between the pedagogical and domain modules to make decisions regarding what pedagogical strategy to use, frequency of feedback, topic selection, etc. For example, if a student asks for help on a topic more than once, the IPT should vary the content and format of feedback and hints that takes into account the student's past performance. Additionally, Pillay et al. (2003) list the following as low-level questions that the pedagogical module should be able to answer:

1. Which topic to present to the student?

2. Which problem(s) to present to the student?
3. How frequently must the student be provided with feedback such as hints and error listings?

The pedagogical module should use recommendations from the student module as well as topic-specific recommendations from domain module to provide individualized instruction. We should note there is inherent difficulty in populating this module with concrete strategies. IPTs that are language and platform dependent will potentially have significant differences here. Further, to populate this module, an IPT would need to gather data on pedagogical strategies, which is an essentially impossible task to automate.

5.5 Domain Module

The domain module defined in the generic architecture contains the skills and introductory programming concepts on which the student will be tutored. Pillay et al. (2003) identify the following as essential domain knowledge:

1. Procedural knowledge on how to write programs.
2. Knowledge describing errors commonly made by novice programmers.
3. Declarative knowledge on the different programming concepts.

Another aspect that we believe should be included is for the domain module to contain knowledge describing misconceptions commonly held by novice programmers. Additionally, we have incorporated the problem module described in the generic architecture into this module. It is essential for an IPT to maintain a list of problems to be used in tutoring sessions. Problems should be presented with an annotation describing the level of difficulty. Additionally, problems should be associated with specific concepts so that student progress and level of knowledge can be updated in the student model. Problems can come in many forms, namely programming exercises where a student produces code, true/false, fill-in-the-blank, and multiple choice questions that test both procedural knowledge and declarative knowledge on programming concepts.

In addition to maintaining domain knowledge, the domain module should be able to analyze student responses to programming exercises and questions that allows the interface module to provide adaptive feedback to users. The domain module should maintain a bug library, a misconception library, and a reference program library. These libraries are particularly important in providing feedback on programming exercises where students produce code. The reference program library should maintain a mapping of programming exercises to a set of reference programs. The bug and misconception libraries could be populated using a multi-strategy machine learning approach such as used by JavaBugs. When a student submits a solution for a programming exercise, the ideal IPT would first identify the intention of the student program by comparing it to a set of reference programs. After the intention is identified, discrepancies between the student solution and reference program are used to provide feedback and learn new misconceptions, with those misconceptions being mapped to errors in the error library. For example, consider the case in which a student is tasked with writing a recursive function that returns the factorial of some input number. If the student produces the correct solution the first time, the student model would be updated and the student would move on to the next lesson or exercise. However, if the student produces a solution that is correct apart from the base case, the domain module should identify that the student has a misconception about the stopping conditions for a recursive functions. Then, using the recommendations from the student module, the domain module may provide a variety of different feedback types such as a simple text hint (e.g., “Your base case is incorrect.”), a trace of the student’ solution, the recursive graph generated by the student’s code, reference material on recursion, an example solution, among others.

With different programming paradigms and languages comes inevitable differences in domain knowledge and student misconceptions and errors. Within the same programming paradigms and languages comes different division and inclusion of topics as well. The domain knowledge defined in this module is extremely important as it contains what the student is actually trying to learn and forms the basis of student modeling. Within procedural and object-oriented paradigms, there is a need for a consensus of programming concepts that are foundational for introductory programming and language-independent.

5.6 Conclusions and Future Work

This thesis demonstrates the ability of existing IPTs to address specific student misconceptions about fundamental concepts in programming. We have identified five misconception categories that we believe to be fundamental for introductory programming based on literature relating to concept inventories and student misconceptions. This set is not exhaustive, and we understand the inherent difficulty in selecting fundamental concepts for introductory programming that are independent of language..

We discussed and analyzed the features of existing IPTs, namely adaptive semantic feedback, adaptive navigational support, and supplementary features such as integrated reference material, illustrating how these features can address student misconceptions. Existing IPTs were typically custom-built for single implementations, which can be costly and time consuming. There is a gap in the level of research into generic architectures that could be used to implement

IPTs. Generic architectures could lower the costs of developing IPTs and also provide the structure for an ideal IPT that teaches introductory programming. We hypothesize what this ideal IPT would look like in terms of structure and features.

The current landscape of computer science education does not meet the needs of every novice programmer. IPTs are able to provide individualized instruction on par with one-on-one human tutoring. It is our belief that IPTs can revolutionize the field of computer science education. While research into IPTs has continued for decades, the level of research has been relatively low until recent years. There remains much research into student misconceptions, student cognitive modeling, and domain modeling before we are able to develop the ideal IPT for introductory programming.

References

Anderson, J. R., & Reiser, B. (1985). The Lisp Tutor. *Journal Byte*, *10*, 159–175.

Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM*, *26*(9), 677–679.

Bloom, B. S. (1984). The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher*, *13*(6), 4–16.

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, *1*(2), 133–161.

Brown, N. C. C., & Altadmri, A. (2017). Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs. *ACM Transactions on Computing Education*, *17*(2), 1–21.

Brusilovsky, P., & Spring, M. (2004). Adaptive, Engaging, and Explanatory Visualization in a C Programming Course. *World Conference on Educational Multimedia, Hypermedia and Telecommunications (ED-MEDIA)*, 21–26.

Caceffo, R., Wolfman, S., Booth, K. S., & Azevedo, R. (2016). Developing a Computer Science Concept Inventory for Introductory Programming. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE 16*, 364–369.

Chang, K.-E., Chiao, B.-C., Chen, S.-W., & Hsiao, R.-S. (2000). A Programming Learning System for Beginners—A Completion Strategy Approach. *IEEE Transactions on Education*, *43*(2), 211–220.

Crow, T., Luxton-Reilly, A., & Wuensche, B. (2018). Intelligent Tutoring Systems for Programming Education: A Systematic Review. *Proceedings of the 20th Australasian Computing Education Conference*.

Finlay, J. E., & Beale, R. (1993). Pattern Recognition and Classification in Dynamic and Static User Modelling. *ACM SIGCHI Bulletin*.

Gegg-Harrison, T. S. (1993). *Exploiting Program Schemata in a Prolog Tutoring System* [PhD Thesis]. Duke University.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education - SIGCSE 08*.

Gross, S., Strickroth, S., Pinkwart, N., & Le, N.-T. (2013). Towards deeper understanding of syntactic concepts in programming. *In Proceedings of the 16th International Conference on Artificial Intelligence in Education Part 9: The First Workshop on AI-Supported Education for Computer Science (AIEDCS)*.

Guyan, M. (2013). Improving Motivation in eLearning. *ELearn*, *10*.

Hsu, T.-C., & Hwang, G.-J. (2017). Effects of a Structured Resource-based Web Issue-Quest Approach on Students' Learning Performances in Computer Programming Courses. *Journal of Educational Technology & Society*, *20*(3), 82–94.

Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial Intelligence*, *42*(1), 51–97.

Kaczmarczyk, L., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education - SIGCSE 10*, 107–111.

Kulik, J. A., & Fletcher, J. D. (2016). Effectiveness of Intelligent Tutoring Systems. *Review of Educational Research*, *86*(1), 42–78.

Kumar, A. N. (2006). Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors. *Technology, Instruction, Cognition and Learning*, *4*.

Lane, H. C., & VanLehn, K. (2005). Teaching the tacit knowledge of programming to novices with natural language tutoring. *Journal of Computer Science Education*, *15*, 183–201.

Le, N. T., & Menzel, W. (2009). Using Weighted Constraints to Diagnose Errors in Logic Programming—The case of an ill-defined domain. *International Journal of Artificial Intelligence in Education*, *19*(4), 381–400.

Le, N.-T. (2016). A Classification of Adaptive Feedback in Educational Systems for Programming. *Systems*, *4*(2), 22.

Le, N.-T., Strickroth, S., Gross, S., & Pinkwart, N. (2013). A Review of AI-Supported Tutoring Approaches for Learning Programming. *Advanced Computational Methods for Knowledge Engineering*, 267–279.

Miller, C. S., & Settle, A. (2019). Learning to Get Literal: Investigating Reference-Point Difficulties in Novice Programming. *ACM Transactions on Computing Education*, *19*(3), 1–17.

Nesbit, J. C., Adesope, O. O., Liu, Q., & Ma, W. (2014). How Effective are Intelligent Tutoring Systems in Computer Science Education? *2014 IEEE 14th International Conference on Advanced Learning Technologies*, 99–103.

Pillay, N. (2003). Developing Intelligent Programming Tutors for Novice Programmers. *ACM SIGCSE Bulletin*, *35*(2), 78–82.

Plonka, L., Sharp, H., van der Linden, J., & Dittrich, Y. (2015). Knowledge transfer in pair programming: An in-depth analysis. *International Journal of Human-Computer Studies*, *73*, 66–78.

Spohrer, J. C., & Soloway, E. (1986). Alternative to construct-based programming misconceptions. *CHI '86: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 183–191.

Suarez, M., & Sison, R. (2008). Automatic Construction of a Bug Library for Object-Oriented Novice Java Programmer Errors. *International Conference on Intelligent Tutoring Systems*, 184–193.

Sykes, E. R. (2006). Qualitative Evaluation of the Java Intelligent Tutoring System. *Journal of Semantics, Cyber*.

Ullah, Z., Lajis, A., Jamjoom, M., Altalhi, A., Al-Ghamdi, A., & Saleem, F. (2018). The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education*, *26*(6), 2328–2341.

Vizcaino, A. (2005). A Simulated Student Can Improve Collaborative Learning. *Journal of Artificial Intelligence in Education*, *15*, 3–40.

Yudelson, M., & Brusilovsky, P. (2005). NavEx: Providing Navigation Support for Adaptive Browsing of Annotated Code Examples. *The 12th International Conference on AI in Education*, 710–717.

Author Biography

Jack Weakley was born in Dallas, Texas on May 19, 1998. He attended the University of Texas at Austin from 2016 to 2020 where he studied Plan II Honors and Electrical & Computer Engineering. In the spring of 2017, Jack founded Texas Rocket League, a university club for competitive Rocket League players. Following graduation, he will begin a position as a Software Engineer at the Microsoft HQ in Redmond, Washington.